Biblioteca Estándar

- Implementa funcionalidades habitualmente requeridas.
- Cada función se declara en un encabezado (header), junto con las macro que pudieran se necesarias.
- En C23 hay 31 encabezados (lista completa en C23:7.1.2-3) que agrupan funciones relacionadas. Al incluir estos encabezados en nuestro código no es necesario hacerlo en algún orden en particular (C23: 7.1.2-5 Standard headers)
- Identificadores reservados (lista incompleta) (C23: 7.1.3 Reserved identifiers)
  - Los que comiencen con un doble guión bajo o guión bajo seguido de letra mayúscula
  - Los que comiencen con un guión bajo, en ámbito de archivo.
  - Los que terminen con guión bajo t (como en size_t)
  - Los que comiencen con E seguido de dígito u otra mayúscula (códigos de errores)
  - Más info en https://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html

<stdio.h>

- Reúne las funciones de entrada salida.
- Puede ser contra un archivo pero puede ser también una terminal de consola u otro tipo de dispositivo.
- Similar, en alguna medida, a iostream y fstream de c++

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.
- En el encabezado se lo declara como tipo de dato incompleto para segurarse que solo se declaren punteros.
- El estándar agrega al respecto: La dirección del objeto FILE utilizado para controlar una secuencia puede ser significativa, una copia de un objeto FILE puede no servir en lugar del original.

Flujos estándar

- 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
- Devuelve el puntero al FILE creado o NULL si falla

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.
- Al cerrar un archivo se “bajan” los buffers si fuese necesario.
- Si el programa termina normalmente y hay archivos abiertos, los cierra.

Tipos de apertura

En el modo de apertura hay un conjunto de símbolos que puede combinarse, estos símbolos son:	
r	Modo lectura. Si el archivo no existe da error. La posición inicial es al principio del archivo.
w	Modo escritura. Si el archivo no existe se crea, si existe se borra el contenido anterior.
a	Modo agregar. Si el archivo no existe se crea, si existe mantiene el contenido. Cada escritura que se haga será al final del archivo, sin importar si previamente se hizo algún operación de posicionamiento a otra parte del archivo.
b	Modo binario.
+	Modo actualización: lectura y escritura.
x	Modo creación con acceso exclusivo. Si el archivo existe da error, si no existe lo crea en modo que ningún otro proceso pueda acceder al mismo hasta tanto lo libere (si el sistema operativo lo soporta).

Tipos de apertura

Combinaciones aceptadas	
Lectura	
r	Abrir archivo de texto para leer.
r+	Abrir archivo de texto para actualización (lectura y escritura).
rb	Abrir archivo binario para leer.
r+b o rb+	Abrir archivo binario para actualización (lectura y escritura).
Agregado	
a	Abrir o crear un archivo de texto para escribir al final del archivo
a+	Abrir o crear un archivo de texto para actualizar, escribiendo al final del archivo
ab	Abrir o crear un archivo binario para escribir al final del archivo
a+b o ab+	Abrir o crear un archivo binario para actualizar, escribir al final del archivo

Tipos de apertura

Combinaciones aceptadas	
Escritura	
w	Truncar a longitud cero o crear un archivo de texto para escribir.
w+	Truncar a longitud cero o crear un archivo de texto para actualizar.
wb	Truncar a longitud cero o crear un archivo binario para escribir.
w+b o wb+	Truncar a longitud cero o crear un archivo binario para actualizar.
wx	Crear archivo de texto para escribir en modo exclusivo.
w+x	Crear archivo de texto para actualizar en modo exclusivo.
wbx	Crear un archivo binario para escribir en modo exclusivo.
w+bx o wb+x	Crear un archivo binario para actualizar en modo exclusivo.

Ver C23:7.23.5.3-3

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, por ejemplo complemento a dos (con el endianness del CPU) o 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)

E/S con formato

- Las funciones printf y scanf (y sus variantes) permiten una entrada salida formateada.
- El primer parámetro de las funciones indica el formato y para poder intercalar variables (direcciones en el caso de scanf) se usa el símbolo % como secuencia de escape, seguida del indicador de formato
- Cada secuencia de escape se reemplaza por el valor del parámetro adicional

Secuencias de escape en E/S

Secuencia	Uso
d,i, u	Números enteros en base 10 (en scanf i trata de detectar la base), u es para unsigned.
o	Enteros en base octal
x,X	Enteros en base Hexadecimal X usa letras mayúsculas, x usa minúsculas
b,B	Enteros en binario, con 0b o 0B como prefijo
c	Caracter
s	String
f	double en printf (**float en scanf**) imprime con . decimal
e,E	Flotantes pero con “notación científica”
p	Para punteros.
%	Para poder mostrar un % en la salida
Modificadores	
l	para long (por ej: ld para long int)
L	para long double (se usa Lf)

Ver C23:7.23.6 Formatted input/output functions

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*`

Alternativas

- Las funciones `printf` y scanf tienen alternativas, ya vimos fprintf y fscanf, que si se usan con stdin y stdout equivalen a printf y scanf.
- La función sprintf lleva como primer parámetro un puntero a una cadena de caracteres, que es a donde ser realiza la salida. Permite armar un buffer con un texto formateado.
- La función snprintf es similar pero con seguridad de no rebalsar el buffer, su segundo parámetro es la cantidad máxima de caracteres a escribir, incluyendo el '\0' final.

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 peligroso 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). El estándar recomienda usar fgets en su lugar.
- 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 E0F 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. Se implementa como macro, por lo tanto no podemos pasar parámetros que tengan efectos de lado ni obtener la dirección de la función.

```
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 E0F
- Tenemos dos tipos de funciones, las del estándar usan long int para indicar desplazamientos. Es conveniente usar las que agregan al nombre una **o** al final, indicando que usan parámetros de tipo `off_t` que suelen ser una extensión habitual (XOPEN).

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

<ctype.h>

- Funciones que devuelven si un carácter pertenece o no a un grupo. Las firmas son del tipo: int isxxx(int c)
  - isalpha, islower, isupper
  - isdigit, isxdigit
  - isalnum
  - isspace (espacio en blanco y en general \n, \r, \t, \f, \v)
  - ispunct (imprimible y no es espacio o alfanumérico)
- Funciones de conversión
  - tolower
  - toupper

<stdlib.h>

- Utilidades varias, entre ellas:
  - Manejo de memoria
    - malloc y free (similar new y delete en c++)
  - Aleatorios
    - rand y srand
  - Conversiones varias de string a nros
    - Strtod, atof, etc.
  - Búsqueda y ordenamiento
    - bsearch y qsort
  - Funciones con enteros
    - abs

Cadenas <string.h>

- Lenguaje C no maneja cadenas (strings) como un elemento del lenguaje, sino que lo asimila a secuencia de caracteres con un “centinela” (`'\0'`) que indica el final de la secuencia.
- Reservar memoria para las cadenas es tarea del programador. Los métodos habituales son declarar un vector de caracteres o usar la función `malloc()`.
- La lógica para manipular cadenas es recorrerlas y operarlas carácter a carácter.
- La biblioteca estándar string.h provee varias funciones para operar con cadenas.

Definir como vector

- Si declaro como un vector
  - `char palabra[10];`
    - Permite guardar cadenas de hasta 9 caracteres, ya que debemos contemplar el `'\0'` final.
  - `char frase[] = "Hola mundo";`
    - El vector frase se dimensiona automáticamente de 11 posiciones (10 caracteres + `'\0'` final)
  - `char frase[100] = "Hola mundo";`
    - Similar al caso anterior, con las posiciones no inicializadas explícitamente en cero, como cualquier vector.
    - **OJO**: El estándar no permite inicializar con cadena de más elementos que la dimensión del vector. Si pongo un literal del mismo largo que la dimensión del vector, **NO** pone el `'\0'` final.

Definir como vector

- El uso de literal cadena es equivalente a haber inicializado cada posición :

```
char palabra[5] = "Hola";
char palabra[5] = {'H', 'o', 'l', 'a', '\0'};
```

- Como con cualquier vector **solo sirve para inicializar**, la siguiente secuencia es **incorrecta**
  - `char palabra[10];`
  - `palabra = "Hola";`
- El modo **correcto** sería
  - `char palabra[10];`
  - `strcpy(palabra, "Hola"); //strcpy en string.h`

Definir con punteros

- Puedo simplemente definir un puntero a char, pero eso no reserva memoria para los datos, debo hacerlo aparte

```
char *cadena;
cadena = malloc(20);
strcpy(cadena, "Hola");
```

- Si la zona de memoria ya está reservada puede apuntarla directamente

```
char buffer[512];
char *cadena = buffer;
char *saludo = "Hola";
```

- El tipo de dato de un literal cadena es arreglo de caracteres, que se degrada a `const char*`. Puede que no sea (físicamente) modificable.

Funciones de string.h

- `size_t strlen(const char *s);`
  - Devuelve la cantidad de caracteres en la cadena s (sin incluir el `'\0'`)
- Ejemplo

```
char pais[20] = "Argentina";
char *p = malloc(strlen(pais) + 1);
strcpy(p, pais);
```

- Más simple:

```
char pais[20] = "Argentina";
char *p = strdup(pais);
//strdup a partir de C23, antes solo
//posix
```

Duplicar cadena

- `char *strdup(const char *s);`
  - Duplica la cadena s, pide memoria con malloc y copia la cadena. Devuelve puntero a la cadena duplicada o NULL si no hay suficiente memoria
- `char *strndup(const char *s, size_t n);`
  - Similar a la anterior pero copia a lo sumo n caracteres y luego agrega `'\0'`
- Usar `malloc` y `strcpy` si `strdup` no está disponible (p.ej: en mingw)

Copiar cadena

- `char *strcpy(char * restrict s1, const char * restrict s2);`
  - Copia la cadena apuntada por s2 en la memoria apuntada por s1. Si al copiar hay solapamiento el resultado es indefinido. Devuelve el valor de s1
- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`
  - Similar a la anterior pero copiando a lo sumo n caracteres. Esto implica que puede **NO** copiar el `'\0'` final!!!. Si largo de la cadena copiada es menor a n, el resto de las posiciones se llenan con `'\0'`.

Concatenar cadena

- ```
  char *strcat(char * restrict s1,
               const char * restrict s2);
  ```
  - Agrega una copia de la cadena apuntada por s2 al final de la cadena s1. Si hay solapamiento el resultado es indefinido. Devuelve el valor de s1
- ```
  char *strncat(char * restrict s1,
                const char * restrict s2,
                size_t n);
  ```
  - Similar a la anterior pero agrega a los sumo n caracteres de s2. Si al copiar n caracteres el último NO es un `'\0'`, lo **agrega**. Significa que la cadena resultante puede llegar a ser de largo strlen(s1)+n+1

Comparar cadenas

- `int strcmp(const char *s1, const char *s2);`
  - Compara las cadenas apuntadas por s1 y s2. El entero devuelto será:
    - `> 0` si `s1 > s2`
    - `< 0` si `s1 < s2`
    - `= 0` si `s1 = s2`
- `int strncmp(const char *s1, const char *s2, size_t n);`
  - Similar al anterior pero compara solo hasta el carácter n, o menos si alguna de las dos cadenas termina antes.

Ejemplo a

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

int main()
{
        char s1[30] = "Hola";
        char *s2 =", que tal!";
        if (strlen(s1) + strlen(s2) < sizeof(s1))
                strcat(s1, s2);
        printf("%s\n", s1);
        //Imprime: Hola, que tal!
        return EXIT_SUCCESS;
}
```

Ejemplo b

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

char *strmin(const char* str1, const char*str2)
{
        if (strcmp(str1, str2) < 0) {
                //cuando está disponible posix
                return strdup(str1);
        } else {
                // Cuando posix no está disponible
                char *temp = malloc(strlen(str2) + 1);
                strcpy(temp, str2);
                return temp;
        }
}
```

Ejemplo b (continuación)

```c
int main()
{
        char *smin;
        char cand1[50];
        char cand2[50];
        printf("Nombre del primer candidato: ");
        scanf("%[^\n]%*c",cand1);
        printf("Nombre del segundo candidato: ");
        scanf("%[^\n]",cand2);

        smin = strmin(cand1, cand2);
        printf("Debe listarse primero a: %s\n", smin);
        free(smin);
        return EXIT_SUCCESS;
}
```

Licencia