



# Argumentos del main

PROFESOR: WEBER  
FEDERICO

# Construcción de programas



- La entrada es la “información” o datos que queremos analizar o utilizar.
- Luego el programa realiza una secuencia de pasos y operaciones sobre estos datos de entrada.
- Al finalizar el proceso, obtenemos un resultado.

•En general, hasta ahora usábamos como entrada lo que ingresábamos por teclado. Pero también puede ser un archivo, datos de red, o información que le envía el SO al programa cuando lo ejecutamos.

•**Esto último** es lo que vamos a ver ahora.

# Función main ()

- En C, `main` puede definirse con o sin parámetros. Cuando se incluyen parámetros, la definición convencional es:

```
int main(int argc, char *argv[])
```

- Estas dos variables nos permiten acceder a los argumentos de la línea de comandos. Por convención se les nombra `argc` y `argv`.
- Antes de seguir, ¿desde donde se le envían parámetros a `main`?
  - Desde el SO, por ejemplo desde Windows:

```
C:\>miProg.exe arg1 arg2 "arg con espacios" arg4
```

En este caso los parámetros los envió el usuario que ejecuto el programa, podría ser un script.

## argc y argv

```
int main(int argc, char *argv[])
```

- **argc** (abreviatura de *argument count*, "recuento de argumentos"): es un entero (int) que indica cuántos argumentos se pasaron al programa en la línea de comandos.
  - Este conteo *incluye* el nombre del programa en sí mismo como primer argumento.
  - En otras palabras, argc es normalmente al menos 1, porque **siempre** cuenta la invocación del programa
  - Por ejemplo, si ejecutamos el programa sin argumentos adicionales, argc será 1 (solo el propio nombre).
  - Y en este caso:

```
C:\>miProg.exe arg1 arg2 "arg con espacios" arg4
```

argc = 5 (nombre de programa + 4 arg)

# argc y argv

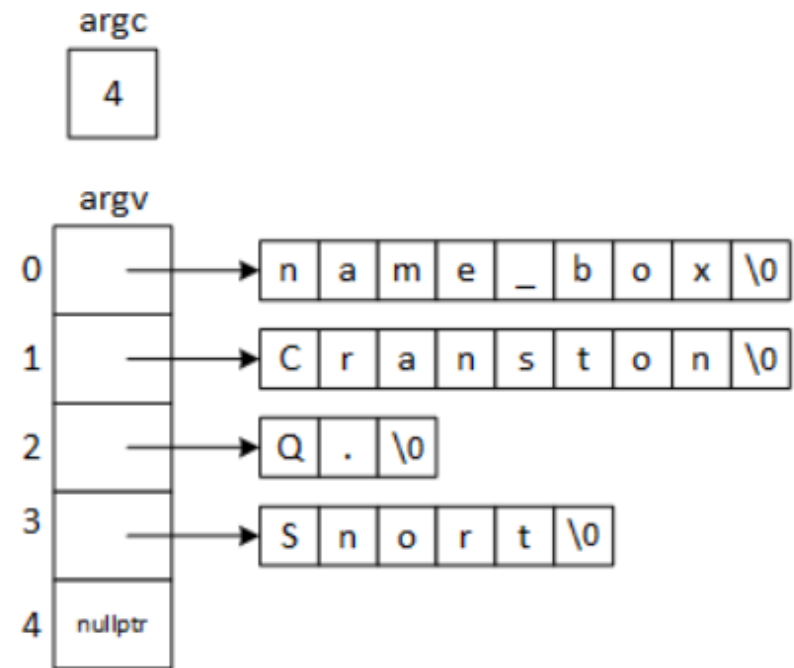
```
int main(int argc, char *argv[])
```

- argv (abreviatura de argument vector, "vector de argumentos"): es un vector de strings que contiene los argumentos proporcionados.
  - En C, esto se representa como `char *argv[]` (vector de punteros a char) o equivalente `char **argv` (puntero a punteros a char).
  - Cada elemento `argv[i]` es un string C (es decir, una secuencia de caracteres terminada en `'\\0'`) con uno de los argumentos.
  - El vector tiene `argc` elementos válidos indexados de 0 a `argc-1`. Por convenio:
    - `argv[0]` es el nombre con el que se ejecutó el programa (puede ser el nombre del ejecutable, por ejemplo "miPrograma", o una ruta como `"/miPrograma"` dependiendo del sistema)
    - `argv[1]` es el primer argumento "real" pasado al programa, `argv[2]` el segundo, y así sucesivamente.
    - `argv[argc]` está definido por el estándar C/C++ como un puntero nulo (NULL), que actúa como centinela para marcar el fin del vector de argumentos. Aunque no se suele usar explícitamente en C (ya que conocemos `argc`), esta posición nula existe.

# Visualización de cómo se almacenan estos argumentos en memoria

```
C:\>name_box Cranston Q. Snort
```

- Los argumentos en **argv** son proporcionados al programa por el **sistema operativo** en el momento la ejecución del programa.
- De hecho, cuando el SO invoca nuestro programa, llama a **main** y le pasa estos dos parámetros (**argc** y **argv**) con la información de la línea de comandos.
- Esto permite que nuestros programas reaccionen a información externa al momento de iniciarse (por ejemplo, pasando un nombre de archivo a procesar, opciones, etc.), en lugar de tener valores fijos o pedir todos los datos por entrada estándar después de iniciar.



# Equivalencia de `char *argv[]` y `char **argv`:

- Ambas notaciones en la declaración de `main` significan lo mismo.
- En C, declarar un parámetro como `char *argv[]` (vector de punteros a char) es equivalente a `char **argv` (puntero a puntero a char), ya que los vectores como parámetros decaen a punteros.
- Es puramente cuestión de estilo; veremos ambas formas en código de distintos autores.

```
int main(int argc, char **argv)
```

```
int main(int argc, char *argv[])
```

son firmas idénticas en la práctica (el estándar C99 y C11 permiten ambas). Lo importante es que `argv` finalmente se usa como si fuera un vector indexable de strings.

# Variaciones validas de la función main:

- El estándar ISO C define dos firmas principales para main en un programa

```
int main()
```

```
int main(void)
```

```
int main(int argc, char **argv)
```

```
int main(int argc, char *argv[])
```

- En todos los casos, **main** debe devolver un **int**, que corresponde al **código de salida** del programa (convención: retornar 0 indica terminación exitosa, cualquier otro valor indica algún error o código particular).
- Si el programa llega al final de **main** sin un **return**, desde el estándar C99+ implica un **return 0** automático.



## Ejemplo 01:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Numero de argumentos (argc): %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

## Ejemplo 01:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Numero de argumentos (argc): %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

EN QT PROBAR ARGUMENTOS SIMPLES , ARGUMENTOS CON ESPACIOS, ARGUMENTOS VACIOS  
COMPARAR QT, WINDOWS, UNIX (GIT BASH)

# Validación de argumentos y buenas practicas:

- Siempre que el programa espere un número específico de argumentos, es importante verificar argc.
- Si argc no coincide con lo esperado (por ejemplo, faltan argumentos obligatorios), se debe notificar al usuario e idealmente mostrar uso correcto del programa.
- Por ejemplo, si un programa espera 2 argumentos (además del nombre), argc debería ser 3. Podemos hacer:

```
if (argc < 3) {  
    printf ("Se esperaban 3 argumentos y se recibieron %d\n", argc);  
    return 1;  
}
```

# Validación de argumentos usando atoi :

- atoi – ASCII to Integer
  - Función simple para convertir una cadena (char \*) en un número entero (int).
  - Declarada en la biblioteca <stdlib.h>.
  - No detecta errores: si la cadena no es un número válido, devuelve 0 sin avisar.
  - Es la que mas usamos en INFO2, pero no recomendada en programas críticos.

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Uso: %s <numero>\n", argv[0]);
        return 1;
    }

    int valor = atoi(argv[1]);

    printf("[atoi] Valor convertido: %d\n", valor);
    return 0;
}
```

Ir al QT. Ejemplo02. Mostrar caso bueno, y caso malo.

# Validación de argumentos – usando strtol :

## strtol – String to Long

- Convierte una cadena a número, pero con más control que atoi.
- Declarada en <stdlib.h>.
- Permite detectar errores de conversión (con errno y endptr).
- Soporta distintas bases: decimal, hexadecimal, octal, etc.

```
long int strtol(const char *nptr, char **endptr, int base);
```

## Parámetros:

- **\*nptr**: puntero a la cadena de caracteres que querés convertir.
- **\*\*endptr**: puntero a puntero donde strtol va a guardar la posición del primer carácter no válido (se puede pasar NULL si no te interesa).
- **base**: la base numérica para la conversión (ej. 10 para decimal, 16 para hexadecimal).

# Validación de argumentos usando strtol :

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

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Falta un número como argumento.\n");
        return 1;    }
    char *fin;
    long valor = strtol(argv[1], &fin, 10);

    if (*fin != '\0') {
        printf("El texto '%s' no es un numero valido.\n", argv[1]);
    } else {
        printf("Numero convertido: %ld\n", valor);
    }
    return 0;
}
```

Fin es un puntero a carácter.

La función strtol esta esperando un \*\*char.

Le envio la dirección de mi puntero fin. Eso le permite a strtol modificar el valor del puntero, y hacer que apunte al carácter posterior al número interpretado.

El contenido de lo apuntado por fin

Ir al QT. Ejemplo03. Mostrar caso bueno, y caso malo.

# Validación de argumentos usando atof :

## atof – ASCII to Float

- Función simple para convertir una cadena (char \*) en un número decimal (float).
- Declarada en la biblioteca <stdlib.h>.
- No detecta errores: si la cadena no es un número válido, devuelve 0 sin avisar.
- Es la que mas usamos en INF02, pero no recomendada en programas críticos.

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Error: falta un numero.\n");
        return 1;
    }

    float valor = atof(argv[1]); // Convierte sin verificar
    printf("[atof] Valor convertido: %.2f\n", valor);

    return 0;
}
```

Ir al QT. Ejemplo04. Mostrar caso bueno, y caso malo.

# Validación de argumentos – usando strtod :

## strtod – String to Double

- Convierte una cadena a número, pero con más control que atof.
- Declarada en <stdlib.h>.
- Permite detectar errores de conversión (con errno y endptr).

## Parámetros:

- **\*nptr**: puntero a la cadena de caracteres que querés convertir.
- **\*\*endptr**: puntero a puntero donde strtod va a guardar la posición del primer carácter no válido (se puede pasar NULL si no te interesa).

```
double strtod(const char *nptr , char **endptr);
```



# Validación de argumentos usando strtod:

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Error: falta un número.\n");
        return 1;
    }

    char *fin;
    float valor = strtod(argv[1], &fin);

    if (*fin != '\0') {
        printf("Error: '%s' no es un numero valido.\n", argv[1]);
        return 1;
    }

    printf("[strtod] Valor convertido: %.2f\n", valor);
    return 0;
}
```

Ir al QT. Ejemplo05. Mostrar caso bueno, y caso malo.

# Tabla comparativa

---

Función	Prototipo completo	Valida errores	Uso principal
atoi	int atoi (const char *nptr);	No	Convertir texto a entero
strtol	long int strtol (const char *nptr , char** endptr , int base);	Sí	Conversión robusta a entero
atof	double atof (const char *nptr);	No	Convertir texto a decimal
strtod	float strtod (const char *nptr , char** endptr);	Sí	Conversión robusta a float

- Si bien no lo vamos a usar en este curso es importante destacar que **atoi** y **atof** no modifican la variable **errno**.
- **errno** es una variable global definida en **<errno.h>**, que indica si ocurrió un error en algunas funciones de la biblioteca estándar, como **strtol**, **strtod**, **fopen**, **malloc**, entre otras.

# Variables de entorno y su uso en C

---

- Además de argc/argv, los programas en C pueden obtener información de **variables de entorno** del sistema operativo.
- Las variables de entorno son pares clave=valor que el entorno (generalmente la shell o el SO) mantiene y que se heredan por los procesos hijos.
- A diferencia de los argumentos de comando, las variables de entorno no pasan por argv (a menos que se haga explícitamente mediante extensiones); se accede a ellas a través de funciones específicas.

## ¿Qué son las variables de entorno?

- Son valores nombrados que existen en el SO antes de lanzar el programa, y que el SO pone a disposición del programa.
- Por ejemplo, en sistemas UNIX es típico tener variables como HOME (directorio de inicio del usuario), PATH (rutas de búsqueda de ejecutables), LANG (configuración de idioma), etc.
- En Windows existen SystemRoot, USERNAME, TEMP, etc.
- Todas son cadenas de caracteres (texto).
- El programa padre (por ejemplo, la shell) provee estas variables al iniciar el programa hijo. Por lo tanto, **desde la perspectiva de C, son datos globales que podemos consultar en cualquier momento durante la ejecución.**

# Acceso a variables de entorno en C

- La forma estándar de obtener el valor de una variable de entorno es usando la función **getenv** (definida en `<stdlib.h>`).
- Esta función recibe el nombre de la variable (una cadena) y devuelve un puntero a la cadena de valor correspondiente, o NULL si no existe tal variable. Ejemplo:

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

int main(int argc, char *argv[]) {
    char *user = getenv("USERNAME");
    if (user != NULL) {
        printf("La variable USERNAME es: %s\n", user);
    } else {
        printf("USERNAME no está definida.\n");
    }
    return 0;
}
```

```
>argmain.exe
La variable USERNAME es: Fede
```

# Acceso a variables de entorno en C

- Muchos programas usan `getenv` para leer configuraciones como `HOME` (por ejemplo, para saber en qué directorio crear un archivo de config del usuario) u opciones como `TZ` (zona horaria), etc.
- Es útil pensar en las variables de entorno como un segundo canal de entrada además de `argv`. De hecho, el POSIX estándar (acuerdo de paz en sistemas UNIX, como Linux y MAC) sugiere que hay dos mecanismos de pasar información al nuevo proceso: los argumentos (`argv`) y el entorno. Ambos se configuran en el momento de la llamada del SO para lanzar un nuevo programa.

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

int main(int argc, char *argv[]) {

    char *verbose = getenv("VERBOSE");
    if (verbose != NULL) {
        printf("La variable VERBOSE es: %s\n", verbose);
    } else {
        printf("VERBOSE no esta definida.\n");
    }
    return 0;
}
```

Ir al QT. Ejemplo07. Mostrar sin verbose y con verbose.

# Uso del tercer parámetro envp

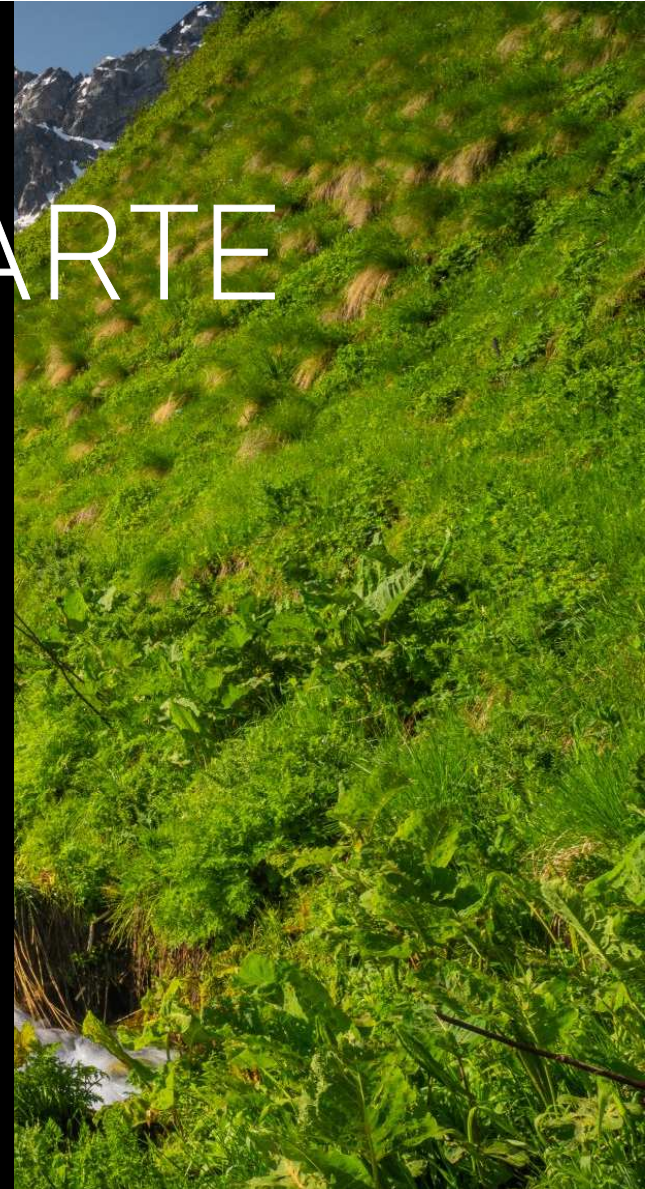
---

- : Algunas implementaciones permiten declarar main con tres parámetros:
  - `int main(int argc, char *argv[], char *envp[])`.
- Aquí **envp** es un vector de strings similar a argv, que contiene todas las variables de entorno en formato "NOMBRE=valor". Termina con un elemento NULL también.
- Esto no es parte del estándar ISO C, pero es una extensión común en Unix (GCC lo soporta).
- Por ejemplo, podríamos iterar sobre envp para imprimir todas las variables. Sin embargo, dado que getenv existe, no es necesario depender de envp.
- De hecho, getenv es más portátil (funciona en Windows y Unix), mientras que envp en main puede que no esté disponible en ciertos compiladores.

# FLUJOS PRIMERA PARTE (STREAMS)

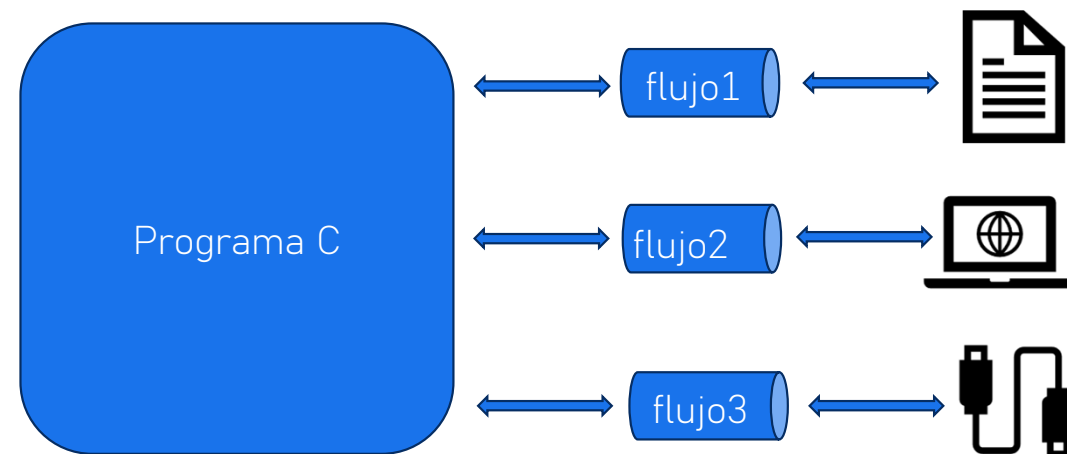
---

INFO II – PROF: WEBER



# ¿Qué es un *stream* en C?

En C, un **stream** (flujo) es una **abstracción** que representa una fuente o un destino de datos. Es como un "buffer" por donde entran o salen datos, sin que tengamos que preocuparnos por cómo se transmiten realmente.



- El motivo por el que existen los *streams* es que permiten al programador **leer o escribir datos** sin preocuparse por el origen o destino real de esos datos.
- Ya sea un archivo, la consola, un socket o un dispositivo, el código puede mantenerse igual, porque el acceso se hace a través de una interfaz común (*fopen*, *fscanf*, *fprintf*, etc.).



# Tipos de streams estándar en C

Nombre	Tipo	Significado común
stdin	Entrada	Teclado (lectura)
stdout	Salida	Pantalla (escritura normal)
stderr	Salida	Pantalla (pero para errores)

- Los streams son variables globales que representan un puntero a una estructura FILE.
- O sea, stdin, stdout, stderr son del tipo FILE \*.
- Estas variables apuntan a estructuras FILE internas que fueron creadas y abiertas automáticamente por el sistema cuando el programa arranca. No debemos hacer nada. Están ahí.

## ¿Y qué es FILE?

Es una estructura (struct) que representa un flujo de datos.

Esta estructura está oculta al usuario: no vamos a acceder a sus campos directamente. Trabajamos siempre a través de funciones como fgetc, fgets, **fprintf**, fread, etc.

# Con stdin ya hemos trabajado

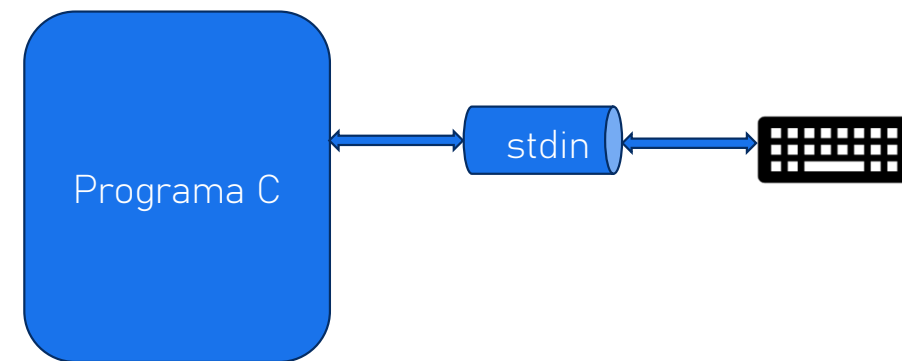
```
#include <stdio.h>
```

```
int main() {  
    int edad;  
    char tipo;  
  
    printf("Edad: ");  
    scanf("%d", &edad);  
  
    printf("Tipo (A/B): ");  
    scanf("%c", &tipo);  
  
    printf("Edad ingresada: %d\n", edad);  
    printf("Tipo ingresado: %c\n", tipo);  
  
    return 0;  
}
```

¿Qué pasó?

- `scanf("%d", &edad)` lee el número 18 pero deja el '\n' en el buffer de stdin.
- `scanf("%c", &tipo)` lo encuentra inmediatamente y lee eso como si fuera la entrada del nombre.

```
Edad: 18  
Tipo (A/B): Edad ingresada: 18  
Tipo ingresado:  
  
Press <RETURN> to close this window...
```



# Con stdin ya hemos trabajado

```
#include <stdio.h>
```

```
int main() {
```

```
    int edad;
```

```
    char tipo;
```

```
    printf("Edad: ");
```

```
    scanf("%d", &edad);
```

```
    fflush (stdin);
```

```
    printf("Tipo (A/B): ");
```

```
    scanf("%c", &tipo);
```

```
    printf("Edad ingresada: %d\n", edad);
```

```
    printf("Tipo ingresado: %c\n", tipo);
```

```
    return 0;
```

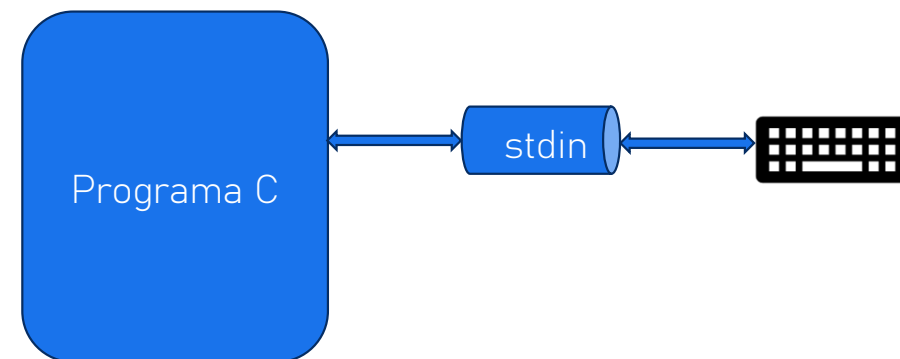
```
Edad: 18
Tipo (A/B): A
Edad ingresada: 18
Tipo ingresado: A
Press <RETURN> to close this wi
```

```
Edad: 18
Tipo (A/B): Edad ingresada: 18
Tipo ingresado:

Press <RETURN> to close this window...
```

¿Cómo lo arreglabamos?

- Usando fflush (stdin)



# Uso adecuado del fflush

---

- fflush es una función declarada en stdio.h
- Su prototipo completo es:
  - `int fflush (FILE *stream );`
- ¿Cuál es su función según el estándar?
  - Vaciar (forzar) el **buffer de salida** asociado a un stream de salida.
  - Esto significa: si tenés datos en el buffer que aún no se han enviado al destino final (por ejemplo, a pantalla o a un archivo), **fflush** los envía inmediatamente.
- stdin .... ¿Es un flujo de salida o de entrada?
  - De entrada
  - ¿Entonces...?
    - En el estándar C, fflush(stdin) es comportamiento indefinido.
    - En sistemas como Windows con MSVC (Visual Studio), fflush(stdin) fue implementado como extensión para descartar el contenido pendiente en el buffer de entrada, es decir, vaciar lo que queda en stdin.
    - Por eso, en esos entornos, parece “arreglar” problemas como el salto de línea que queda tras un scanf.
    - Pero ese comportamiento no es portátil, no funciona igual en GCC, Linux o compiladores estándar, y no es la mejor practica.

# Otros métodos de vaciar el flujo stdin

```
#include <stdio.h>
```

```
int main() {  
    int edad;  
    char tipo,c;  
  
    printf("Edad: ");  
    scanf("%d", &edad);  
    while ((c = getchar()) != '\n');  
    printf("Tipo (A/B): ");  
    scanf("%c", &tipo);  
    while ((c = getchar()) != '\n');  
    printf("Edad ingresada: %d\n", edad);  
    printf("Tipo ingresado: %c\n", tipo);  
  
    return 0;  
}
```

Otro método alternativo es usar siempre la función **fgets**. Pero la veremos mas adelante.

Lo importante es recordar que la solución de fflush (stdin) NO es portable y NO es una buena practica.

# Flujo redirigido desde stdout a un archivo mediante consola

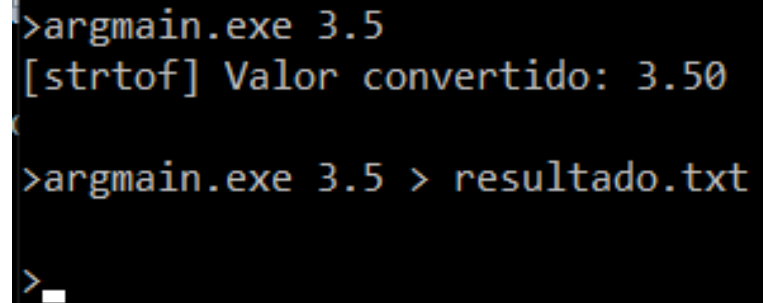
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Error: falta un número.\n");
        return 1;
    }

    char *fin;
    float valor = strttof(argv[1], &fin);

    if (*fin != '\0') {
        printf("Error: '%s' no es un numero válido.\n", argv[1]);
        return 1;
    }

    printf("[strttof] Valor convertido: %.2f\n", valor);
    return 0;
}
```



A terminal window with a black background and green text. It shows the execution of a program named 'argmain.exe'. The first command is 'argmain.exe 3.5', which outputs '[strttof] Valor convertido: 3.50'. The second command is 'argmain.exe 3.5 > resultado.txt', which redirects the output to a file named 'resultado.txt'. The prompt '>' is shown on the next line.

Lo que hace el símbolo ">" es redirigir todo lo que va al stream stdout al archivo resultado.txt

# Otra forma de ver el printf

- Prototipo completo de `printf`:
  - `int printf (const char *format , ...);`
- En este punto sabemos muy bien usar `printf`.
- Lo queremos remarcar es que `printf` interactúa directamente con el stream `stdout`.
  
- Prototipo completo de `fprintf`:
  - `int fprintf (FILE *stream , const char *format, ...);`
- Es igual a `printf`, pero debemos especificar con que flujo vamos a interactuar.
- Por lo tanto:
  - `fprintf (stdout, "Hola mundo") ;`
  - `printf ("Hola mundo");`
  - Son idénticas.

Ir al QT. Ejemplo06. Mostrar cuando se va por stout y cuando por stderr

# Ejercicios de argumentos del main y variables del sistema

1. Realizar un programa que muestre la cantidad de argumentos del main que recibe y los enumere.
2. Realizar un programa que reciba un único argumento desde la línea de comandos. Si el argumento es un número entero válido y positivo, imprimirlo por stdout. Si el argumento no es válido o es negativo, imprimir un mensaje de error por stderr. Redirigir el resultado a desde el Shell a un archivo "resultado.txt"
3. Realizar un programa que muestre todas las variables de "enviroment" del sistema.
4. Realizar un programa que funcione como una calculadora por argumentos del main. Asi:  
calc operacion numero1 numero2 ...
5. (versión UNIX)  
En Bash, el shell que funciona en las terminales de linux las variables de enviroment se generan asi:  
`export VAR="contenido de la variable"`  
(versión Windows)  
En CMD, el shell que funciona en las terminales de windows las variables de enviroment se generan asi:  
`set VAR="contenido de la variable"`  
(para ambos) Realizar un programa que muestre el contenido de una variable MIVAR.
6. Realizar un programa que devuelva la suma de sus argumentos a traves del return .  
Probarlo de esta forma en consola:  
(versión UNIX)  
`/suma 3 6`  
`echo La suma de 3 y 6 es $? [El resultado del ultimo programa queda guardado en la variable $? ]`  
(versión Windows)  
`suma.exe 3 6`  
`echo %ERRORLEVEL% [El resultado del ultimo programa queda guardado en la variable %ERRORLEVEL% ]`



FIN

