

# Minishell

## OBJETIVO DEL PROYECTO

Construir una shell funcional que:

- Ejecute comandos básicos con historial y manejo de señales.
- Gestione redirecciones (`<`, `>`, `>>`, `<<`), pipes (`|`), variables de entorno (`$`, `$?`), y builtins (`cd`, `echo`, `pwd`, `export`, `unset`, `env`, `exit`).
- Sea robusta: sin *segfaults*, *leaks*, ni errores de norma.

## OBJETIVO GENERAL DEL PROYECTO **minishell**

Crear una **réplica simplificada de `bash`**, tu propio *shell interactivo*, desde cero, con un comportamiento lo más similar posible al de un shell real, pero limitado a los requisitos del subject.

Este proyecto es **mucho más que una simulación de comandos**: es una **inmersión profunda en el corazón del sistema operativo**. Aprenderás cómo se comunican los programas, cómo se manejan procesos, cómo redirigir entrada y salida, y cómo controlar el entorno de ejecución como lo hace un sistema Unix real.

## ¿Qué aprenderán e implementarán concretamente?

### 1. Entrada interactiva (Readline y el ciclo del shell)

- Aprenderán cómo funciona un *prompt* que se mantiene en espera de instrucciones.
- Usarán `readline()` para capturar entrada del usuario, manejar el historial y facilitar edición.
- Manejarán los diferentes modos de entrada del usuario: vacío, ctrl-D, ctrl-C, etc.

#### Interiorizan:

- Lectura estándar.
- Comportamiento de interfaces interactivas (ciclo `read → parse → execute → repeat`).
- Cómo gestionar entrada/salida correctamente.

### 2. Tokenización (Lexer)

- Descompondrán la línea ingresada por el usuario en *tokens* significativos: comandos, argumentos, pipes, redirecciones, comillas, etc.
- Aprenderán a **no romper** lo que está entre comillas o a tratar `$VAR` como un único token.

#### Interiorizan:

- Cómo separar una línea de texto en unidades gramaticales.
- Cómo tratar los distintos metacaracteres (`|`, `>`, `<`, `"` y `'`) como el shell real.
- Lógica detrás del lenguaje shell.

### 3. Parser y estructuras de comandos

- Transformarán esos tokens en **estructuras de datos** organizadas (ej. árboles o listas) que representen comandos encadenados, sus redirecciones y argumentos.

Ejemplo:

```
cat file.txt | grep hola > salida.txt
```

→ implica varios procesos y flujos conectados.

#### ➡ Interiorizan:

- Estructuras de datos complejas.
- Cómo interpretar correctamente la intención del usuario en una línea de comandos.
- Validación de sintaxis (comillas mal cerradas, pipes consecutivos, etc.).

### 4. Ejecución de comandos (Process management)

- Aprenderán a usar `fork()` para crear procesos hijos y `execve()` para reemplazar su código con el binario a ejecutar.
- Usarán `pipe()`, `dup()`, `dup2()` para redirigir flujos estándar de entrada y salida.
- Implementarán ejecución **encadenada con pipes** (`cmd1 | cmd2 | cmd3`).

#### ➡ Interiorizan:

- Cómo se crean, gestionan y sincronizan procesos.
- Cómo funcionan las tuberías (`pipe`) en Unix.
- Qué es un file descriptor y cómo se manipulan.

### 5. Redirecciones ( `<` , `>` , `>>` , `<<` )

- Aprenderán a **redirigir la entrada o salida** de un proceso desde/hacia archivos o inputs temporales (`heredoc`).

#### ➡ Interiorizan:

- Manejo de archivos (`open`, `close`, `read`, `write`).
- Redirección estándar (`stdin`, `stdout`, `stderr`).
- Comportamiento avanzado del shell.

### 6. Built-ins (Comandos internos)

Implementarán comandos internos como:

- `cd`, `pwd`, `echo`, `env`, `export`, `unset`, `exit`

#### ➡ Interiorizan:

- Diferencia entre comandos externos (ejecutables) y comandos que deben modificarse *dentro del propio proceso* (como cambiar de directorio).

- Manipulación del entorno ( `environ` , variables).
- 

## 7. Expansión de variables ( `$VAR` , `$?` )

- Interpretarán expresiones como `$USER` , `$?` , y las sustituirán por su valor actual.

### Interiorizan:

- Entorno del proceso (variables y cómo se transmiten).
  - Cómo el shell interpreta `$` en distintos contextos ( `"..."` vs `!..!` ).
- 

## 8. Manejo de señales ( `ctrl-C` , `ctrl-D` , `ctrl-\` )

- Capturarán y reaccionarán correctamente a señales como `SIGINT` , `SIGQUIT` , `EOF` .

### Interiorizan:

- El ciclo de vida de un proceso.
  - Cómo se interrumpe o termina un programa correctamente.
  - Gestión de entrada "limpia" cuando el usuario interrumpe algo.
- 

## 9. Control de errores y recursos

- Detectarán errores de memoria (*leaks*), errores lógicos (comandos inválidos), y errores de uso (pipes mal escritos).
- Liberarán correctamente toda la memoria dinámica.

### Interiorizan:

- Manejo responsable de recursos.
  - Debugging con herramientas como `valgrind` .
  - Robustez y limpieza del código.
- 

## 10. Normas, colaboración, y organización

- Escribirán el código en C siguiendo la **Norma de 42**.
  - Usarán `Makefile` , git, revisiones cruzadas, buena estructura modular.
  - Aprenderán a dividir el trabajo de forma efectiva.
- 

## Conclusión: ¿qué sabrán hacer al terminar?


Después de este proyecto, sabrán:

- ✓ Entender cómo funciona un shell internamente
- ✓ Gestionar múltiples procesos y redirecciones
- ✓ Manipular memoria y descriptores de archivos como pros
- ✓ Aplicar diseño estructurado en C con calidad
- ✓ Trabajar en equipo en proyectos grandes y técnicos
- ✓ Construir herramientas que interactúan directamente con el sistema operativo

## 1. Organización del equipo y enfoque de trabajo

Roles generales sugeridos (intercambiables por etapas):

Persona 1	Persona 2
Parser & lexer	Ejecutador & builtins
Manejo de señales y pipes	Redirecciones y heredocs
Tests, leaks y norma	Makefile y entorno git




 Revisan el código del otro antes de integrar al main branch.

## 2. Estructura de carpetas

```
minishell/
├── include/           # Headers
│   ├── minishell.h
│   └── parser.h
├── src/
│   ├── main.c
│   ├── prompt.c      # readline y entrada
│   ├── lexer/        # tokenización
│   ├── parser/        # parseo de tokens
│   ├── executor/      # ejecución de comandos
│   ├── builtins/      # funciones echo, cd, etc.
│   ├── signals/       # manejo de señales
│   ├── redirections/  # <, >, <<, >>
│   └── utils/
├── libft/             # (si usan su libft)
├── Makefile
└── README.md
```

## 3. Plan de desarrollo por etapas (con división de tareas)

### Semana 1: Preparación y estructura base

- ☐  Estructura de carpetas y Makefile funcional.
- ☐  Programa base: muestra prompt, lee línea, imprime lo que escribe el usuario.
- ☐  Implementar control-C, control-D, control-\ básicos.

- ☐ 🛠 Variable global para señales (única permitida).

| Reunión diaria breve (15-30 min) para sincronizar avances y dudas.

## ✅ Semana 2: Lexer, Parser y estructura de comandos

- ☐ ✂ Lexer: dividir input en tokens (`echo`, `|`, `"texto"`, etc.).
- ☐ 🧩 Parser: armar estructura de comandos conectados (pipeline, redirecciones).
- ☐ 🚫 Manejo correcto de comillas `'` y `"` y casos inválidos.

| 🔄 Ej: `echo "hola" | grep h > file.txt`

## ✅ Semana 3: Ejecución y built-ins

- ☐ 🔄 Implementar ejecución básica (fork + execve).
- ☐ 🧬 Manejo de PATH y comandos con rutas.
- ☐ 📦 Implementar built-ins (`echo`, `cd`, `pwd`, `env`, `export`, `unset`, `exit`).
- ☐ 🪄 Probar cada comando con argumentos, con y sin comillas.

## ✅ Semana 4: Redirecciones y Pipes

- ☐ ➡ Redirecciones `>`, `>>`, `<`.
- ☐ 📦 Pipes `cmd1 | cmd2 | cmd3`.
- ☐ 📖 Heredoc `<<`.

| Validar combinaciones: `cat < file | grep hola > salida.txt`

## ✅ Semana 5: Variables de entorno y estado \$?

- ☐ 🧬 Expansión de `$VARIABLE`, `$?`.
- ☐ 📖 Manejo correcto en comillas dobles y simples.
- ☐ 📅 Implementar actualización de exit status tras cada comando.

## ✅ Semana 6: Limpieza y validación

- ☐ 🧪 Comprobación de leaks con `valgrind`.
- ☐ 📖 Revisión estricta de norma.
- ☐ 🔄 Pruebas manuales con hoja de evaluación.
- ☐ 🪄 Test de errores: entradas inválidas, comandos que fallan, múltiples pipes/redirecciones.
- ☐ 🐛 Últimos bugs, mejoras en control de señales y errores.

## 🪄 4. Tests sugeridos (según hoja de evaluación)

Verifica con checklist como:

- `ctrl-C` en línea vacía o con texto
- `echo "hola mundo"` y `echo '$USER'`
- `cat << EOF` seguido de contenido
- `ls | grep archivo > salida.txt`
- `export VAR=42` , `echo $VAR` , `unset VAR`
- Comando inexistente `jddajdja`
- Heredoc con EOF, comillas dobles, comillas simples

## 5. Git y colaboración

- Crear rama `main` protegida
- Usar ramas tipo `feat/parser` , `fix/signals` , etc.
- Hacer *merge request* con revisión del otro
- Usar issues para bugs/tareas pendientes
- Git log claro: "fix: redirection parsing crash", "feat: implement echo builtin"

## SEMANA 1 — Setup & Prompt

Tarea	Responsable	Comentarios
Crear estructura de carpetas	Ambos	<code>include/</code> , <code>src/</code> , <code>libft/</code> si aplica
Crear Makefile básico	Persona 1	Con flags <code>-Wall -Werror -Wextra</code>
Implementar prompt con readline	Persona 2	Mostrar <code>minishell\$</code>
Manejo de Ctrl+D (salida)	Persona 2	<code>readline</code> devuelve NULL
Control básico de señales (Ctrl+C, Ctrl+)	Persona 1	Usar <code>sigaction()</code>
Preparar <code>.gitignore</code>	Ambos	Ignorar <code>a.out</code> , <code>*o</code> , <code>*dSYM</code> , etc.

## SEMANA 2 — Lexer & Parser

Tarea	Responsable	Comentarios
Diseñar struct para tokens	Persona 1	Tipos: CMD, ARG, PIPE, REDIR_OUT, etc.
Crear lexer que convierte línea a tokens	Persona 1	Manejar comillas, espacios, pipes
Crear parser que convierte tokens en comandos ejecutables	Persona 2	Comandos en pipes, redirecciones
Detectar errores de sintaxis	Persona 2	Pipes sin comandos, comillas mal cerradas

## SEMANA 3 — Execve & Builtins

Tarea	Responsable	Comentarios
Implementar ejecución con <code>execve</code> y <code>PATH</code>	Persona 2	Comandos externos
Implementar <code>echo [-n]</code>	Persona 1	Usar <code>ft_printf</code> o <code>write</code>
Implementar <code>cd</code> con manejo de errores	Persona 1	<code>chdir()</code>
Implementar <code>pwd</code> , <code>env</code>	Persona 1	<code>getcwd()</code> y entorno
Implementar <code>exit</code> , <code>unset</code> , <code>export</code>	Persona 2	Controlar argumentos

## SEMANA 4 — Redirecciones y Pipes

Tarea	Responsable	Comentarios
Implementar redirección <code>&gt;</code> y <code>&gt;&gt;</code>	Persona 1	<code>open()</code> con flags
Implementar redirección <code>&lt;</code>	Persona 2	<code>open()</code> + <code>dup2()</code>
Implementar heredoc <code>&lt;&lt;</code>	Persona 2	Leer hasta delimitador
Implementar pipe entre comandos	Persona 1	<code>pipe()</code> , <code>fork()</code> , <code>dup2()</code>
Soporte para combinaciones	Ambos	Ej: <code>`cat &lt; f</code>

## SEMANA 5 — Expansión & Status

Tarea	Responsable	Comentarios
Implementar expansión de <code>\$VAR</code> , <code>\$?</code>	Persona 2	<code>getenv()</code> + seguimiento del <code>status</code>
Expansión dentro de comillas dobles	Persona 2	No expandir dentro de <code>'</code>
Integrar todo: lexer, parser, executor	Ambos	Refactor para limpiar funciones
Historial de readline con <code>add_history()</code>	Persona 1	

## SEMANA 6 — Norma, Tests y Defensa

Tarea	Responsable	Comentarios
Comprobar norma y eliminar warnings	Ambos	<code>norminette</code> , sin <code>-Wall</code> warnings
Valgrind y comprobación de leaks	Persona 1	Comprobar en cada comando
Simulacro de evaluación	Ambos	Usar hoja de evaluación oficial
Documentar proyecto y README.md	Persona 2	Cómo compilar, usar y organizar
Preparar defensa y dividir preguntas	Ambos	Cada uno domina al menos la mitad

### 1. Estructura inicial del repositorio

Crea la siguiente estructura de carpetas y archivos (puedes copiar este contenido en un script o crear a mano):

```
minishell/
|
├── include/
│   └── minishell.h    # Header principal
|
```

```

├── src/
│   ├── main.c
│   ├── prompt.c
│   ├── lexer/
│   ├── parser/
│   ├── executor/
│   ├── builtins/
│   ├── redirections/
│   ├── signals/
│   └── utils/
├── libft/          # Solo si usan libft
├── Makefile
└── README.md

```

## 2. Makefile básico (estructura mínima)

Asegúrense de que cumpla con los flags y targets requeridos:

```

NAME = minishell

CC = cc
CFLAGS = -Wall -Wextra -Werror

SRCS = src/main.c \
      src/prompt.c \
      # Agrega aquí los .c que vayan apareciendo

OBJS = $(SRCS:.c=.o)

all: $(NAME)$(NAME): $(OBJS)
    $(CC) $(CFLAGS) -o $(NAME) $(OBJS) -lreadline

clean:
    rm -f $(OBJS)

fclean: clean
    rm -f $(NAME)

re: fclean all

.PHONY: all clean fclean re

```