Apuntes minishell

https://github.com/lgandarillas/minishell.git

1. Git Setup inicial

Commits

- **Add: File.txt - Blabla…**
- **Fix: File.txt - Blabla…**
- **Update: File.txt - Blabla…**

Ramas

- **main** = Rama principal con todo correcto (testeado) y listo para entregar.
- **develop** = Copia de la rama **master**.
  Cada vez que voy a ponerme a tocar código, tengo que hacer pull de **develop** a mi rama personal.
- **lgandari** = Rama personal.
- **aquinter** = Rama personal.

Notas

- Los cambios en local no pertenecen a ninguna rama. Solo cambio de rama cuando **git status** diga que está todo subido y ok para evitar hacer **commit** en la rama equivocada.

Setup inicial

```
Unset
-- Crear rarama, estando posicionado en git(main)
git checkout -b develop

-- Hacer push de la rama
git push --set-upstream origin develop

-- Ver ramas
git branch

-- Moverse entre ramas
git checkout develop

-- Crear ramas propias
git checkout -b lgandari
git push --set-upstream origin lgandari
git checkout -b aquinter
git push --set-upstream origin aquinter
```

Hacer pull desde otra rama.

- **Pull de rama “develop” a mi rama “lgandari”.**
- Lo hago antes de programar.

```
Unset
git pull origin develop
```

Hago commit a mi rama

```
Unset
git add .
git commit -m "Msg"
git push
```

Me voy a la rama a la que quiero aplicar los cambios, “develop”

- Cada vez que me muevo a una rama hago pull de esa rama **“git pull”**
- Merge de otra rama: **“git merge lgandari”** y **“git push”**

```
Unset
git checkout develop
git pull
git merge lgandari
git push
```

2. Fork de libft_v2

Fork:

- Ir al repositorio original de github (lgandarillas).
- Hacer click en el botón fork. Esto generará una copia del proyecto en el repositorio interesado (alejandquintero).

Clonar repositorio fork en máquina local:

```
Unset
git clone https://github.com/alejandquintero/libft_v2.git
```

Configurar el repositorio original como remoto. De esta forma podremos subir cambios tanto al repositorio propio, como solicitar pull request al repositorio original:

```
Unset
cd libft_v2
git remote add upstream https://github.com/lgandarillas/libft_v2.git
```

Creamos nueva rama en local:

```
Unset
git checkout -b alejandquintero
```

Realizamos cambios y commiteamos:

```
Unset
git add .
git commit -m "Descripción de cambios"
```

Realizar push de los cambios realizados al repo propio forkeado (alejandquintero)

```
Unset
git push origin mi-nueva-rama
```

En este punto, ya debemos tener los cambios guardados en nuestro repo de github.

Ahora, para poder tener estos cambios en el repo original, debemos hacer pull request:

- Ir al fork en GitHub.
- Hacer click en "Compare & pull request".
- Rellenar el formulario de solicitud de extracción (pull request) con una descripción de los cambios realizados
- Envía el pull request. Esto creará una solicitud que lgandarillas podrá revisar y aprobar.

3. Hacer pull request con cliente de github

- Instalar cliente de github

```
Unset
sudo apt install gh # En ubuntu
brew install gh     # En mac
```

- Loguearse al cliente (terminal interactiva)

```
Unset
gh auth login
```

- Realizar commit, hacer commit y push a la rama

- Crear pull request
    - --base main especifica la rama base a la que deseas fusionar tu PR (en este caso, main).
    - --head mi-nueva-rama especifica la rama de origen que estás fusionando (tu nueva rama).
    - --title y --body son el título y la descripción de tu PR, respectivamente.

```
Unset
gh pr create --base main --head mi-nueva-rama --title "Título del Pull Request"
--body "Descripción detallada del Pull Request"

// En este caso
gh pr create --base main --head alejandquintero:alejandquintero --title "update:
README" --body "Remove test line"
```

- Ver si tengo un pull request en un repositorio

```
Unset
gh pr list
```

- Como aceptar un pull request

```
Unset
gh pr merge NUMERO_DEL_PR
// Create a merge commit
// Yes
// Submit
```

4. Submódulo de libft - comandos útiles

Creación de submódulo:

```
Unset
git submodule add https://github.com/lgandarillas/libft_v2.git libft_v2
```

Clonar repos con submódulos:

```
Unset
git clone --recurse-submodules <url-del-repositorio-principal>
```

Inicializar los submódulos si ya hemos clonado:

```
Unset
git submodule update --init --recursive
```

Actualizar los submódulos

```
Unset
git submodule update --remote libft_v2
```

Manejo entre ramas para traerse el submódulo

Nos cambiamos a la rama donde estamos trabajando:

```
Unset
git checkout feature-branch
git merge main # combina el historial de dos ramas. Más fácil de usar
ó
git rebase main # reaplica los commits de la rama actual sobre otra base, puede
                ser un poco mas complejo a la hora de resolver conflictos
```

Actualizamos el submódulo:

```
Unset
git submodule update --init --recursive
```

Commitear y pushear a la rama que hayamos creado:

```
Unset
git add libft
git commit -m "Update submodule libft in feature-branch"
git push origin feature-branch
```

Pasos para solucionar submódulo libft

Para asegurarnos de que el submódulo está en la rama correcta y no en un commit específico:

Cambiamos a la rama correcta en el submódulo estando en la rama main de la minishell:

```
Unset
cd libft
git checkout main  # O la rama correcta del submódulo
git pull origin main  # Nos aseguramos de que esté actualizado
cd ..
```

Agregamos y hacemos commit:

```
Unset
git add libft
git commit -m "Update: submodule libft to point to the main branch"
```

Pusheamos los cambios al repo de la minishell:

```
Unset
git push origin main
```

5. Funciones permitidas

readline() - <stdio.h>

- **char *readline(const char *prompt);**
- La función muestra un prompt y espera que el usuario introduzca una línea de texto, que puede editar y una vez presione ENTER devuelve la línea introducida.

rl_clear_history() - <readline/readline.h>, <readline/history.h>

- **void rl_clear_history(void);**
- La función ***rl_clear_history()*** borra el historial de comandos, eliminando todas las entradas previamente agregadas. Esto es útil para reiniciar el historial en una sesión de comandos.

rl_on_new_line() - <readline/readline.h>, <readline/history.h>

- **void rl_on_new_line(void);**
- La función rl_on_new_line() notifica a Readline que se está en una nueva línea de entrada, preparando la biblioteca para la siguiente operación de entrada/salida. Es útil cuando se hacen cambios manuales en el prompt o en la línea de entrada y se quiere asegurar que Readline maneje correctamente la nueva línea.

rl_replace_line() - <readline/readline.h>, <readline/history.h>

- **void rl_replace_line(const char *text, int clear_undo);**
- Cambia el contenido de la línea de entrada actual por el texto proporcionado. Si el segundo parámetro es 1, se borra el historial de deshacer, eliminando cualquier acción previa que podría ser revertida. Esto es útil para actualizar dinámicamente el contenido de la línea de entrada en aplicaciones interactivas.

rl_redisplay() - <readline/readline.h>, <readline/history.h>

- **void rl_redisplay(void);**
- La función rl_redisplay() redibuja la línea de entrada actual en el terminal, mostrando cualquier cambio que se haya hecho. Esto es especialmente útil después de modificar la línea de entrada usando funciones como rl_replace_line(), asegurando que el usuario vea la versión actualizada de la línea de entrada.

add_history() - <readline/readline.h>, <readline/history.h>

- **void add_history(const char *line);**
- La función add_history() agrega la línea proporcionada al historial de comandos, lo que permite que la línea sea accesible en futuras sesiones de edición de línea. Esto es útil para mantener un registro de los comandos ingresados por el usuario, permitiendo navegar y reutilizar comandos anteriores.

printf() - <stdio.h>

- **int printf(const char *format, …);**

malloc() - <stdlib.h>

- **void *malloc(size_t size);**

free() - <stdlib.h>

- **void free(void *ptr);**

write() - <unistd.h>

- **ssize_t write(int fd, const void *buf, size_t count);**

access() - <unistd.h>

- **int access(const char *path, int mode);**
- La función access() se utiliza para verificar los permisos de acceso a un archivo específico.
- The mode is specified using constants such as R_OK, W_OK, and X_OK, which represent read, write, and execute permissions, respectively.

open() - <fcntl.h>

- **int open(const char *pathname, int flags);**

read() - <unistd.h>

- **size_t read(int fd, void *buf, size_t count);**

close - <unistd.h>

- **int close(fd);**

fork() - <unistd.h>

- **pid_t fork(void);**

wait() - <sys/wait.h>

- **pid_t wait(int *status);**
- Suspende la ejecución del proceso actual hasta que uno de sus procesos hijos termine. Hace que el proceso padre se bloquee en la llamada wait() a la espera de que uno o más de sus procesos hijo terminen su ejecución.
- Cuando el hijo termina, devuelve un “exit status” que se almacena en una variable int status, aunque no es necesario.
- Ejemplo, ***wait(NULL); ó wait(&status);***
- Macros relacionadas:
  WIFEXITED(status)
  - Devuelve verdadero si el hijo terminó normalmente.

  WEXITSTATUS(status)
  - Devuelve el código de salida del hijo si terminó normalmente.

waitpid() - <sys/wait.h>

- **pid_t waitpid(pid_t pid, int *status, int options);**
- Se utiliza para esperar a que termine la ejecución de un proceso hijo en concreto.
- Si **waitpid**() se ejecuta correctamente, devuelve el **PID** del proceso hijo que terminó. Si hay un error, devuelve **-1**.
- Ejemplo, ***waitpid(pid1, NULL, 0); ó waitpid(pid, &status, 0);***

wait3() - <sys/wait.h>

- **pid_t wait3(int *status, int options, struct rusage *rusage);**
- La función wait3() es similar a waitpid(), pero además proporciona información sobre el uso de recursos del proceso hijo, a través del parámetro rusage.

wait4() - <sys/wait.h>

- **pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);**
- La función wait4() es similar a waitpid(), pero añade la capacidad de obtener estadísticas detalladas sobre el uso de recursos del proceso hijo a través de la estructura rusage.

signal() - <signal.h>

- **void (*signal(int signum, void (*handler)(int)))(int);**
- La función signal() establece un manejador de señales para una señal específica (signum). Un manejador de señales es una función que se ejecutará cuando se reciba la señal especificada. Esta función se utiliza para controlar el comportamiento del programa cuando se recibe una señal determinada, como SIGINT (generada por Ctrl+C) o SIGTERM (utilizada para terminar el proceso).

sigaction() - <signal.h>

- **int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);**
- La función sigaction() se utiliza para establecer y modificar el manejo de señales para una señal específica (signum). A diferencia de signal(), sigaction() proporciona un control más preciso sobre cómo se manejan las señales, incluyendo la capacidad de especificar un conjunto de señales que deben ser bloqueadas durante la ejecución del manejador de señales.

kill() - <signal.h>

- **int kill(pid_t pid, int sig);**
- La función kill() envía una señal sig al proceso o grupo de procesos especificado por pid.

exit() - <stdlib.h>

- **void exit(int status);**

getcwd() - <unistd.h>

- **char *getcwd(char *buf, size_t size);**
- Obtiene el pathname del directorio de trabajo actual y lo guarda en el buffer buf. El tamaño del buffer debe ser especificado por size, el cual debe ser lo suficientemente grande para almacenar el pathname completo del directorio de trabajo, incluyendo el carácter nulo final.

chdir() - <unistd.h>

- **int chdir(const char *path);**
- La función chdir() cambia el directorio de trabajo actual del proceso a aquel especificado por path.

stat() - <sys/stat.h>

- **int stat(const char *path, struct stat *buf);**
- La función stat() obtiene la información detallada sobre un archivo especificado por path y guarda esta información en la estructura struct stat apuntada por buf.

lstat() - <sys/stat.h>

- **int lstat(const char *path, struct stat *buf);**
- La función lstat() obtiene la información detallada sobre un archivo especificado por path, incluyendo el tipo de archivo y sus atributos. A diferencia de stat(), lstat() no sigue el enlace simbólico y proporciona información sobre el enlace simbólico mismo, en lugar del archivo o directorio al que apunta.

fstat() - <sys/stat.h>

- **int fstat(int fd, struct stat *buf);**
- The fstat() function obtains information about the file referred to by the file descriptor fd and stores this information in the structure pointed to by buf.

unlink() - <unistd.h>

- **int unlink(const char *pathname);**
- Se utiliza para eliminar un archivo existente del sistema de archivos. El parámetro pathname es una cadena que especifica el nombre y la ruta del archivo que se desea eliminar. Devuelve 0 si el archivo se ha eliminado correctamente, -1 en caso contrario.

execve() - *<unistd.h>*

- **int execve(const char *pathname, char *const argv[], char *const envp[]);**
- V = “Vector”, E = “Environment”
- Se utiliza para ejecutar un nuevo programa. Al reemplazar la imagen del proceso anterior, lo que esté después del exec no se ejecutará. Permite especificar los argumentos del programa y el entorno del nuevo programa de forma explícita.
- Ejemplo: ***execve("/usr/bin/python3", args, env);***

dup() - *<unistd.h>*

- **int dup(int oldfd);**
- La función dup() crea una copia (duplica) el descriptor de archivo oldfd.
- Devuelve un nuevo descriptor de archivo que es el número entero más bajo disponible que no se está utilizando actualmente y apunta al mismo archivo o recurso. En caso de error devuelve -1.
- Ejemplo, ***fd2 = dup(fd1);***

dup2() - *<unistd.h>*

- **int dup2(int oldfd, int newfd);**
- La función dup2() duplica el fd de archivo oldfd y lo asigna al descriptor de archivo newfd. Si newfd ya está abierto, primero se cierra antes de asignarle el nuevo fd duplicado. Esto significa que dup2() asegura que newfd apunte al mismo archivo o recurso que oldfd.
- Ejemplo, ***fd2 = dup2(fd1, 5);***
- **Redireccionar la entrada estándar (stdin) a un fd:**
  Se redirecciona la entrada estándar (stdin) de un programa de un fd, de forma que en lugar de tomar la entrada por consola la toma por un fd (pipe, archivo, …).
  dup2(fd, 0);
- **Redireccionar la salida estándar (stdout):**

Se redirecciona la salida estándar (stdout) de un programa hacia un archivo (fd) en lugar de la consola.
dup2(fd, 1);

pipe() - <*unistd.h*>

- **int pipe(int pipefd[2]);**
- Las pipes se utilizan para crear 2 fd's interconectados que se utilizan para comunicar procesos. Tras la llamada a pipe() se abren ambos fd's.
- Una pipe tiene un extremo de lectura (fd[0]) y uno de escritura (fd[1])
- Los pipes se crean en el proceso padre antes de hacer fork().
- En cada proceso, hay que cerrar todos los fd's siempre antes de que termine la ejecución.
- Cuando la función pipe() funciona correctamente, devuelve un valor entero que es igual a 0. Cuando pipe() falla, devuelve un valor -1.

opendir() - <dirent.h>

- **dir *opendir(const char *dirname);**
- Se abrir un directorio específico y obtener un descriptor de directorio que puede ser utilizado para acceder a la información sobre los archivos contenidos en ese directorio.

readdir() - <dirent.h>

- **struct dirent *readdir(DIR *dir);**
- Lee la siguiente entrada del directorio apuntado por el puntero dir y devuelve un puntero a una estructura struct dirent. Esta estructura contiene información sobre el archivo o subdirectorio leído, incluyendo su nombre y otros atributos.

closedir() - <dirent.h>

- La función **closedir()** cierra el directorio representado por el puntero dir, liberando los recursos asociados con el descriptor de directorio.

strerror() - <string.h>

- **char *strerror(int errnum);**
- La función strerror() toma un número de error (errnum) como argumento y devuelve una cadena de caracteres que describe el error correspondiente.

perror() - <*errno.h*>

- **void perror(const char *s);**
- Imprime un mensaje de error en el flujo de error estándar (stderr). Este mensaje incluye una cadena de caracteres proporcionada por el programador (s) seguida por : y la descripción del error correspondiente al valor actual de errno.

isatty() - <unistd.h>

- **int isatty(int fd);**
- La función isatty() comprueba si el descriptor de archivo fd está asociado a un terminal interactivo. Devuelve 1 si lo es, 0 si no lo es y -1 en caso de error.

ttyname() - <unistd.h>

- **char *ttyname(int fd);**
- Devuelve un puntero a una cadena de caracteres que representa el nombre del dispositivo terminal asociado al descriptor de archivo fd.

ttyslot() - <unistd.h>

- **int ttyslot(void);**
- Devuelve el índice del terminal asociado al proceso actual en la tabla de terminales del sistema.

ioctl() - <sys/ioctl.h>

- **int ioctl(int fd, unsigned long request, …);**
- La función ioctl() realiza operaciones de control sobre el descriptor de archivo fd especificado, usando el comando request y, opcionalmente, argumentos adicionales dependiendo del comando..

getenv() - <stdlib.h>

- **char *getenv(const char *name);**
- Busca el valor de la variable de entorno cuyo nombre es especificado por name.

tcsetattr() - <termios.h>

- **int tcsetattr(int fd, int optional_actions, const struct termios *termios_p);**
- La función tcsetattr() establece los parámetros del terminal asociados al descriptor de archivo fd desde la estructura termios apuntada por termios_p.

tcgetattr() - <termios.h>

- **int tcgetattr(int fd, struct termios *termios_p);**
- La función tcgetattr() obtiene los parámetros del terminal asociados al descriptor de archivo fd y los almacena en la estructura termios apuntada por termios_p.

tgetent() - <termcap.h>

- **int tgetent(char *bp, const char *name);**
- La función tgetent() carga la entrada de la base de datos de terminales para el terminal especificado por name y la almacena en el buffer apuntado por bp.

tgetflag() - <termcap.h>

- **int tgetflag(const char *id);**
- La función tgetflag() busca la flag especificada por id en la entrada de la base de datos de terminales cargada y devuelve si está establecida.

tgetnum() - <termcap.h>

- **int tgetnum(const char *id);**
- La función tgetnum() busca el valor numérico de la capacidad del terminal especificada por id en la entrada de la base de datos de terminales cargada y devuelve dicho valor.

tgetstr() - <termcap.h>

- **char *tgetstr(const char *id, char **area);**

- La función tgetstr() busca el valor de la capacidad de cadena del terminal especificada por id en la entrada de la base de datos de terminales cargada y devuelve un puntero a dicha cadena. La cadena se almacena en el área apuntada por area.

tgoto() - <termcap.h>

- **char *tgoto(const char *cap, int col, int row);**
- La función tgoto() se utiliza para generar una secuencia de control de posición del cursor basada en una capacidad del terminal obtenida previamente, como la cadena de posición del cursor (por ejemplo, obtenida con tgetstr("cm", &area)). Esta función toma la cadena de capacidad del terminal y los valores específicos de columna y fila, y devuelve una cadena que puede ser enviada al terminal para mover el cursor a la posición deseada.

tputs() - <termcap.h>

- **int tputs(const char *str, int affcnt, int (*putc)(int));**
- La función tputs() se utiliza para enviar secuencias de control al terminal. Estas secuencias de control pueden ser cadenas obtenidas previamente con funciones como tgetstr(). tputs() maneja adecuadamente el retraso requerido por algunos terminales para ciertas operaciones y permite especificar una función para enviar los caracteres al terminal.