

# Cómo hacer una consola interactiva

Post extraído de [Readline: La biblioteca de GNU que hace devs felices](#)  (autor: Tomás Ferraro)

Nunca se preguntaron ¿cómo hacen programas como `bash` para generar una consola cómoda? Uno podría pensar que forma parte de la implementación, pero cuando empezás a pensarlo un poco mejor, ¿no es raro que cada consola interactiva de cada lenguaje que pueda existir, implemente *tooodo* ese trabajo por sí misma? Pues claro, la verdad de la milanesa **es que no lo hace**.

Resulta que, si uno se pone a indagar, existen múltiples bibliotecas que nos proveen esas funcionalidades para evitar reinventar la rueda. En este caso particular, me voy a centrar en `readline`, una biblioteca del proyecto GNU, que es justamente la que implementa la consola de `bash`. ¿Porqué elegí `readline`? Principalmente por lo anterior.

## Leyendo líneas

---


La función `readline` es nuestro plato principal. La misma esta firma:

```
char *readline(const char *prompt)
```

c

Pasándole un literal, bajo el nombre de `prompt`, `readline` te promete mostrar lo que le pasaste por parámetro en la consola al inicio de cada línea, y devuelve una porción de memoria con lo que ingresó el usuario, sin agregar el `\n` al final.

¿Cuál es la diferencia de leer de `stdin`? *La magia empieza ahora*.

`Readline` interfiere un poco con lo que devuelve, especialmente con teclas como las flechas, `supr`, `inicio`, `fin` y **algunas más** . ¡Es exactamente lo que estás pensando!

Cada vez que detecta que el caracter ingresado, es un caracter especial dentro de los que reconoce, en vez de mostrar el valor en el retorno, genera el comportamiento que estaríamos esperando (por ejemplo, moverse para atrás cuando se aprieta `←`). Más fácil imposible.

No solo `readline` posee estas funcionalidades, sino que también se encarga de reservar la memoria necesaria para almacenar la línea, dejándonos la responsabilidad de **liberar dicha memoria**. Pero es mucho mejor que tener que estimar un máximo de tamaño para la línea a leer, ¿no?

Veamos como quedaría en código c:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <readline/readline.h>
5
6  void main() {
7      char *linea;
8      while (1) {
9          linea = readline(">");
10
11          if (!linea) {
12              break;
13          }
14          printf("%s\n", linea);
15          free(linea);
16      }
17  }

```

c

Noten que si `readline` devuelve `null`, corta el bucle. Esto es porque cuando `readline` detecta un EOF (la representación del fin de archivo, en este caso con `stdin`, es cuando terminamos de escribir una línea y apretamos enter) y la línea leída hasta ese punto está vacía, retorna `null` para evitarnos un chequeo extra.

## Manejar el historial

Hasta ahora venimos bien, ya tenemos `readline` funcionando, pero todavía falta más. Notarán que si bien no se escriben en pantalla ni en lo que recibimos, las flechas `↑` y `↓` no funcionan. Esto es porque para ello es necesario **mantener un historial de los comandos ingresados**. La biblioteca nos provee la función `add_history(linea)` justamente para eso.

El funcionamiento de la *biblioteca de Historial* no se va a cubrir en detalle en este post, pero si pecan de curiosos pueden revisar [la documentación en línea](#).

A grandes rasgos, para agregar las líneas ingresadas al historial, tendríamos que agregar el `add_history()` a nuestro ejemplo anterior de la siguiente manera:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <readline/readline.h>
6  #include <readline/history.h>
7
8  void main() {
9      char *linea;
10     while (1) {
11         linea = readline(">");
12         if (linea) {
13             add_history(linea);
14         }
15         if (!strncmp(linea, "exit", 4)) {
16             free(linea);
17             break;
18         }
19         printf("%s\n", linea);
20         free(linea);
21     }
22 }

```

c

Ahora, cada vez que se ingresa una línea *que no está vacía*, la agrega en el historial. Para salir, evitamos comparar contra la línea vacía (justamente para mostrar lo anterior) y en cuanto detecta que se ingresó **exit** termina el bucle.

## Auto completar

Ya tenemos casi todo, pero nos falta la frutilla del postre, el famoso **auto completar de la tecla TAB**. Si bien hasta ahora, la cosa venía bastante fácil, auto completar los comandos no es algo relativamente sencillo, ya que hay que generar una función que complete los comandos que nos interesan en nuestra aplicación en específico.

Dado que generar un auto completar es un tema bastante largo para cubrir, dejo la implementación particular **para otro post más avanzado** sobre Readline. Tal como dije antes, si quieren implementar un auto completar de todas maneras, les dejo un **ejemplo** [🔗](#) sobre una implementación simple en la documentación de la biblioteca.

## Compilar con readline

Está el código listo, pero eso *¿con qué se come?*

Primero que nada, debemos asegurarnos que **la biblioteca de readline esté instalada**. Como dije antes, la usa bash, así que, ¿no debería estar? Bueno sí, la mayoría de las veces, **debería estar**. Normalmente, *lo que falta es la contraparte para desarrolladores*. Por las dudas, dejo los nombres de los paquetes de ambos:

```
sudo apt-get install libreadline8 libreadline-dev
```

sh

Finalmente, solo resta compilar nuestro ejemplo usando el flag para bibliotecas de gcc, como `-lreadline`.

```
gcc example.c -o example -lreadline
```

sh