



Minishell

[minishell.pdf](#)

Guía

[Chapter5-WritingYourOwnShell.pdf](#)

Introducción

El capítulo pasado cubrió cómo usar un programa shell usando comandos UNIX. El shell es un programa que interactúa con el usuario a través de un terminal o toma la entrada de un archivo y ejecuta una secuencia de comandos que se pasan al Sistema Operativo. En este capítulo aprenderás a escribir tu propio programa shell.

Programas Shell

Un programa shell es una aplicación que permite interactuar con el ordenador. En un shell el usuario puede ejecutar programas y también redirigir la entrada para que provenga de un archivo y la salida para que provenga de un archivo. Los programas shell también proporcionan construcciones de programación como if, for, while, funciones, variables etc. Además, los programas shell ofrecen funciones como edición de líneas, historial, completado de archivos, comodines, expansión de variables de entorno y construcciones de programación. He aquí una lista de los programas shell más populares en UNIX:

sh	Shell Program. The original shell program in UNIX.
csh	C Shell. An improved version of sh.
tcsh	A version of Csh that has line editing.
ksh	Korn Shell. The father of all advanced shells.
bash	The GNU shell. Takes the best of all shell programs. It is currently the most common shell program.

Además de los shells de línea de comandos, también existen shells gráficos como el Windows Mac OS Finder, o Linux Gnome y KDE, que simplifican el uso de los ordenadores a la mayoría de los usuarios. Sin embargo, estos shells gráficos no sustituyen a los shells de línea de comandos para los usuarios avanzados que quieren ejecutar secuencias complejas de comandos repetidamente o con parámetros no disponibles en los amigables pero limitados diálogos y controles gráficos.

Partes de un programa shell

La implementación del shell se divide en tres partes: El Parser, El Ejecutor, y Shell Subsistemas.

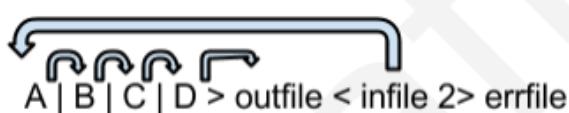
El parser

El Parser es el componente de software que lee la línea de comandos como "ls al" y lo pone en una estructura de datos llamada Tabla de Comandos que almacenará los comandos que serán ejecutarán.

El ejecutor

El ejecutor tomará la tabla de comandos generada por el parser y por cada SimpleCommand de la matriz creará un nuevo proceso. Si es necesario, también creará tuberías para comunicar la salida de un proceso a la entrada del siguiente. Además redirigirá la entrada estándar, la salida estándar y el error estándar si hay alguna redirección.

La siguiente figura muestra una línea de comandos "A | B | C | D". Si hay una redirección como "< infile" detectada por el analizador sintáctico, la entrada del primer SimpleCommand A se redirige desde infile. Si hay una redirección de salida como "> outfile", redirige la salida del último SimpleCommand (D) a outfile.



Si hay una redirección a errfile como ">& errfile" el stderr de todos los procesos de SimpleCommand será redirigido a errfile.

Subsistemas shell

Otros subsistemas que completan tu shell son:

- **Variables de entorno:** Las expresiones de la forma \${VAR} se expanden con la variable de entorno correspondiente. También el shell debe ser capaz de establecer, expandir e imprimir vars de entorno.
- **Comodines:** Los argumentos de la forma a*a se expanden a todos los archivos que coincidan con ellos en el directorio local y en múltiples directorios .
- **Subconjuntos:** Los argumentos entre `` (backticks) se ejecutan y la salida se envía como entrada a la shell.

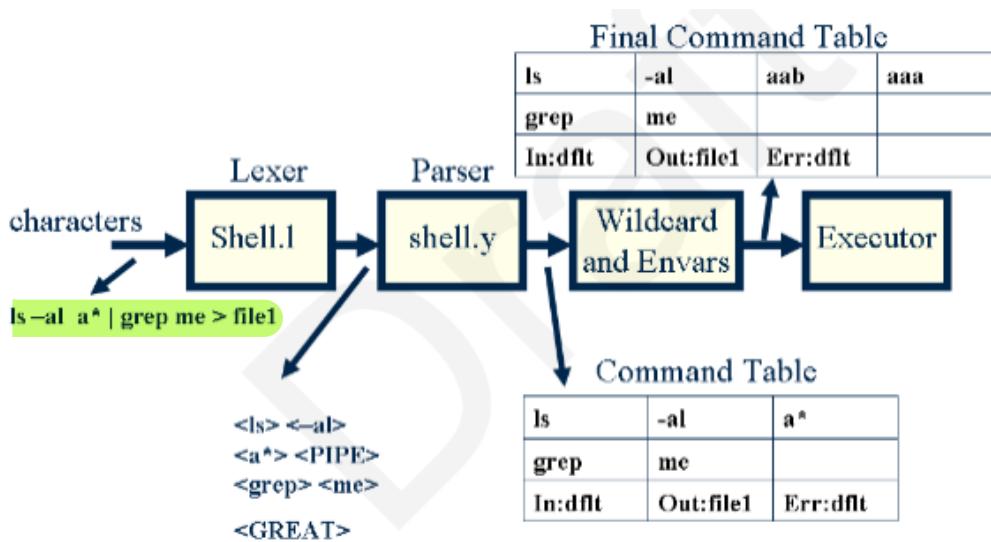
Le recomendamos encarecidamente que implemente su propio shell siguiendo los pasos de este capítulo. Implementar tu propio shell te dará una muy buena comprensión de cómo interactúan el shell y el sistema operativo. Además, será un buen proyecto para mostrar durante tu entrevista de trabajo a futuros empleadores.

Usando Lex y Yacc para implementar el parser

Utilizará dos herramientas UNIX para implementar su analizador sintáctico: Lex y Yacc. Estas herramientas se utilizan para implementar compiladores, intérpretes y preprocesadores. Usted no necesita saber compilador para usar estas herramientas. Todo lo que necesita saber sobre estas herramientas se explicará en este capítulo.

Un analizador sintáctico se divide en dos partes: un Analizador Léxico o Lexer toma los caracteres de entrada y los agrupa en palabras llamadas tokens, y un analizador sintáctico que procesa los tokens de acuerdo a una gramática y construye la tabla de comandos.

Aquí tienes un diagrama del Shell con el Lexer, el Parser y los demás componentes.



Los tokens se describen en un archivo shell.l utilizando expresiones regulares. El archivo shell.l se procesado con un programa llamado lex que genera el analizador léxico.

Las reglas gramaticales utilizadas por el analizador sintáctico se describen en un archivo llamado shell.y utilizando expresiones de sintaxis que describimos a continuación.

shell.y se procesa con un programa llamado yacc que genera un programa analizador sintáctico que genera un programa analizador sintáctico. Tanto lex como yacc son comandos estándar en UNIX. Estos comandos pueden utilizarse para implementar compiladores muy complejos. Para el shell utilizaremos un subconjunto de Lex y Yacc para construir la tabla de comandos que necesita el shell. Necesitas implementar la siguiente gramática en shell.l y shell.y para hacer que nuestro parser interprete las líneas de comandos y proporcione a nuestro ejecutor la información correcta.

```
cmd [arg]* [ | cmd [arg]* ]*
[ [> filename] [< filename] [>& filename] [>>& filename] ]* [&]
```

Fig 4: Shell Grammar in Backus-Naur Form

Esta gramática está escrita en un formato llamado "BackusNaur Form". Por ejemplo cmd [arg]* significa un comando, cmd, seguido de 0 o más argumentos, arg. La expresión [| cmd [arg]*]* representa los subcomandos opcionales de la tubería, donde puede haber 0 o más de ellos. La expresión [>nombredearchivo] significa que puede haber 0 o 1 redirecciones >nombredearchivo. La expresión [&] al final significa que el carácter & es opcional.

Ejemplos de comandos aceptados por esta gramática son:

```
ls -al
ls -al > out
ls -al | sort >& out
awk -f x.awk | sort -u < infile > outfile &
```

La tabla de comandos

La tabla de comandos es una matriz de estructuras SimpleCommand. Una estructura SimpleCommand contiene miembros para el comando y los argumentos de una única entrada en el canal. El analizador analizará también la línea de comandos y determinará si hay alguna entrada o salida en función de los símbolos presentes en el comando (es decir, < infile, o > outfile).

Aquí hay un ejemplo de un comando y de la tabla de comandos que genera:



command

ls -al | grep me > file1

Command Table

SimpleCommand array:

0:	ls	-al	NULL
1:	grep	me	NULL

IO Redirection:

in: default	out: file1	err: default
--------------------	-------------------	---------------------

Para representar la tabla de comando vamos a usar las siguientes clases: Command y SimpleCommand

```

// Command Data Structure

// Describes a simple command and arguments
struct SimpleCommand {
    // Available space for arguments currently preallocated
    int _numberOfAvailableArguments;

    // Number of arguments
    int _numberOfArguments;

    // Array of arguments
    char ** _arguments;

    SimpleCommand();
    void insertArgument( char * argument );
};

// Describes a complete command with the multiple pipes if any
// and input/output redirection if any.
struct Command {
    int _numberOfAvailableSimpleCommands;
    int _numberOfSimpleCommands;
    SimpleCommand ** _simpleCommands;
    char * _outFile;
    char * _inputFile;
    char * _errFile;
    int _background;

    void prompt();
    void print();
    void execute();
    void clear();

    Command();
    void insertSimpleCommand( SimpleCommand * simpleCommand );

    static Command _currentCommand;
    static SimpleCommand *_currentSimpleCommand;
};

```

El constructor SimpleCommand::SimpleCommand construye un comando simple vacío. El método SimpleCommand::insertArgument(char * argument) inserta un nuevo argumento en el SimpleCommand y amplía la matriz _arguments si es necesario. También se asegura que el último elemento es NULL, ya que es necesario para la llamada al sistema exec().

El constructor Command::Command() construye un comando vacío que será rellenará con el método Command::insertSimpleCommand(SimpleCommand * simpleCommand). insertSimpleCommand también amplía la matriz _simpleCommands si es necesario. si es necesario. Las variables _outFile, _inputFile, _errFile serán NULL si no se ha realizado ninguna o el nombre del fichero al que se redirigen.

Las variables _comandoactual y _comandoactual son variables estáticas, es decir sólo hay una para toda la clase. Estas variables se utilizan para construir el Comando y Comando simple durante el análisis sintáctico del comando.

Las clases Command y SimpleCommand implementan la estructura de datos principal que usaremos en el shell.

Implementando el analizador léxico

El analizador léxico separa la entrada en tokens. Leerá los caracteres uno a uno desde la entrada estándar y formará un token que pasará al analizador sintáctico. El analizador léxico utiliza un archivo shell.l que contiene expresiones regulares que describen cada uno de los tokens. El analizador léxico leerá la entrada carácter por carácter e intentará hacer coincidir la entrada con cada una de las expresiones regulares. Cuando una cadena de la entrada coincide con una de las expresiones regulares, ejecutará el código ejecutará el código {...} a la derecha de la expresión regular. La siguiente es una versión simplificada simplificada de shell.l que usará su shell.

```
/*
 * shell.l: simple lexical analyzer for the shell.
 */

#include <string.h>
#include "y.tab.h"

%%
\n      {
        return NEWLINE;
```

```

        }

[ \t]   {
        /* Discard spaces and tabs */
    }

">"   {
        return GREAT;
    }

"<"   {
        return LESS;
    }

">>>"  {
        return GREATGREAT;
    }

">>&"  {
        return GREATAMPERSAND;
    }

"|"   {
        return PIPE;
    }

"&"   {
        return AMPERSAND;
    }

[^ \t\n][^ \t\n]* {
    /* Assume that file names have only alpha chars */
    yyval.string_val = strdup(yytext);
    return WORD;
}

/* Add more tokens here */

.   {
    /* Invalid character in input */
    return NOTOKEN;
}

```

%%

El archivo shell.l se pasa a través de lex para generar un archivo C llamado lex.yy.c. Este archivo implementa el escáner que utilizará el analizador sintáctico para traducir caracteres en tokens.

Aquí está el comando empleado para usar lex:

```

bash% lex shell.l
bash% ls
lex.yy.c

```

El archivo lex.yy.c es un archivo C que implementa el lexer para separar los tokens descritos en shell.l.

Hay dos partes en shell.l. La parte superior tiene este aspecto:

```
%{  
#include <string.h>  
#include "y.tab.h"  
%}
```

Esta es una porción que será insertada en la parte superior del archivo lex.yy.c directamente sin modificación que incluye los archivos de cabecera y las definiciones de variables que utilizará en el analizador. Ahí es donde puedes declarar las variables que usarás en tu lexer.

La segunda porción delimitada por %% es similar a:

```
%%  
\n {  
    return NEWLINE;  
}  
[ \t ] {  
    /* Discard spaces and tabs */  
}  
>" {  
    return GREAT;  
}  
[^ \t\n][^ \t\n]* {  
    /* Assume that file names have only alpha chars */  

```

Esta parte contiene las expresiones regulares que definen los tokens formados al tomar los de la entrada estándar. Una vez formado un token, será devuelto, o en algunos casos descartado. Cada regla que define un token tiene también dos partes:

```
regular-expression {  
    action  
}
```

E.g.

```
\n {  
    return NEWLINE;  
}
```

La primera parte es una expresión regular que describe el token que esperamos que coincida. La acción es un fragmento de código C que el programador añade y que se ejecuta una vez

que el token coincide con la expresión regular. En el ejemplo anterior, cuando se encuentra el carácter newline, lex devolverá la constante NEWLINE. Describiremos más adelante dónde se definen las constantes NEWLINE están definidas.

Aquí hay un token más complejo que describe una WORD. Una WORD puede ser un argumento para un comando o el comando en sí mismo.

```
[^ \t\n][^ \t\n]* {  
    /* Assume that file names have only alpha chars */  
    yyval.string_val = strdup(yytext);  
    return WORD;  
}
```

La expresión en [...] coincide con cualquier carácter que esté dentro de los corchetes. La expresión [^...] coincide con cualquier carácter que no esté dentro de los corchetes. Por lo tanto, [^ \t\n][^ \t\n]* describe un que comienza con un carácter que no es un espacio, un tabulador o una nueva línea y es seguido por cero o más caracteres que no son espacios, tabuladores o nuevas líneas o más caracteres que no son espacios, tabulaciones o nuevas líneas. El token coincidente se encuentra en una variable llamada yytext. Una vez que se ha encontrado una palabra, se asigna un duplicado del token encontrado a yyval.string_val en la siguiente sentencia:

```
yyval.string_val = strdup(yytext);
```

esta es la forma en que el valor del token se pasa al analizador sintáctico. Por último, la constante WORD devuelta al analizador sintáctico.

Añadiendo nuevos tokens a shell.l

El shell.l descrito anteriormente admite actualmente un número reducido de fichas. Como primer paso en el desarrollo de su shell necesitará añadir más tokens a la nueva gramática que no están actualmente en shell.l . Consulte la gramática de la Figura 4 para ver qué tokens faltan y deben añadirse a shell.l. A continuación se muestran algunos de estos tokens:

```
">>" { return GREATGREAT; }  
"!" { return PIPE; }  
"&" { return AMPERSAND; }  
Etc.
```

Añadiendo nuevos tokens a shell.y

Añadirá los nombres de los tokens que creó en el paso anterior en shell.y en la sección %token sección:

readline()

La función `readline` en C es una función de la biblioteca estándar que se utiliza para leer una línea de entrada del usuario desde la consola. Esta función se utiliza comúnmente en aplicaciones de línea de comandos para leer la entrada del usuario y procesarla.

La sintaxis de la función `readline` es la siguiente:

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

Esta función toma como argumento una cadena que se utiliza como el mensaje de solicitud de entrada para el usuario. Devuelve una cadena que contiene la línea de entrada leída desde la consola.

Aquí hay un ejemplo sencillo de cómo utilizar la función `readline` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {
    char *input;

    input = readline("Introduce un número: ");

    if (input == NULL) {
        printf("No se ha introducido ningún número\n");
        exit(EXIT_FAILURE);
    }

    printf("Has introducido el número %s\n", input);

    free(input);

    return 0;
}
```

En este ejemplo, se utiliza la función `readline` para solicitar al usuario que introduzca un número. La función `readline` espera a que el usuario introduzca una línea de texto y la almacena en una variable de tipo `char *`. Si el usuario pulsa Ctrl-D (o introduce EOF), la función `readline` devuelve un valor `NULL`. Por lo tanto, se comprueba si el valor devuelto es `NULL` para manejar el caso en el que el usuario no ha introducido nada.

Una vez que se ha leído la entrada del usuario, se imprime un mensaje que indica el número introducido y se libera la memoria utilizada por la cadena devuelta por la función `readline` mediante la función `free`.

rl_clear_history()

La función `rl_clear_history` es una función de la biblioteca readline que se utiliza para eliminar todo el historial de comandos almacenado en la memoria por la biblioteca readline. Esta función es útil cuando se desea eliminar todos los comandos anteriores que han sido almacenados en el historial.

La sintaxis de la función `rl_clear_history` es la siguiente:

```
void rl_clear_history(void);
```

Esta función no toma argumentos y no devuelve ningún valor. Simplemente elimina todos los comandos almacenados en el historial.

Aquí hay un ejemplo sencillo de cómo utilizar la función `rl_clear_history` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {
    char *input;

    // Añadir algunos comandos al historial
    add_history("comando1");
    add_history("comando2");
    add_history("comando3");

    printf("Historial antes de borrar:\n");
    HIST_ENTRY **history_list = history_list();
    for (int i = 0; history_list[i]; i++) {
        printf("%s\n", history_list[i]->line);
    }

    // Borrar todo el historial
    rl_clear_history();

    printf("Historial después de borrar:\n");
    history_list = history_list();
    for (int i = 0; history_list[i]; i++) {
        printf("%s\n", history_list[i]->line);
    }

    return 0;
}
```

En este ejemplo, se utiliza la función `add_history` para añadir tres comandos al historial. Luego, se utiliza la función `history_list` para obtener una lista de todos los comandos en el historial y se imprimen en la consola.

A continuación, se utiliza la función `rl_clear_history` para borrar todo el historial. Se vuelve a utilizar la función `history_list` para obtener la lista actualizada de comandos en el historial y se imprime en la consola. En este caso, la lista debe estar vacía, ya que todos los comandos anteriores han sido eliminados por la función `rl_clear_history`.

rl_on_new_line()

La función `rl_on_new_line` es una función de la biblioteca readline que se utiliza para mover el cursor a una nueva línea en la consola. Esta función es útil cuando se desea mover el cursor a una nueva línea antes de imprimir más texto en la consola.

La sintaxis de la función `rl_on_new_line` es la siguiente:

```
void rl_on_new_line(void);
```

Esta función no toma argumentos y no devuelve ningún valor. Simplemente mueve el cursor a una nueva línea en la consola.

Aquí hay un ejemplo sencillo de cómo utilizar la función `rl_on_new_line` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {
    char *input;

    input = readline("Introduce un número: ");

    if (input == NULL) {
        printf("No se ha introducido ningún número\n");
        exit(EXIT_FAILURE);
    }

    // Mover el cursor a una nueva línea
    rl_on_new_line();

    printf("Has introducido el número %s\n", input);

    free(input);

    return 0;
}
```

En este ejemplo, se utiliza la función `readline` para solicitar al usuario que introduzca un número. Después de leer la entrada del usuario, se utiliza la función `rl_on_new_line` para mover el cursor a una nueva línea antes de imprimir el mensaje que indica el número introducido.

Este ejemplo es un caso simple de cómo utilizar la función `rl_on_new_line`, pero se puede utilizar en conjunto con otras funciones readline para realizar tareas más complejas. Por ejemplo, se puede utilizar la función `rl_redisplay` para volver a imprimir una línea de entrada previa del usuario después de mover el cursor a una nueva línea con `rl_on_new_line`.

rl_replace_line()

La función `rl_replace_line` es una función de la biblioteca readline que se utiliza para reemplazar la línea actual de entrada del usuario en el buffer de entrada. Esta función es útil cuando se desea modificar la línea de entrada del usuario antes de enviarla al programa para su procesamiento.

La sintaxis de la función `rl_replace_line` es la siguiente:

```
void rl_replace_line(const char *text, int clear_undo);
```

Esta función toma dos argumentos: `text` es un puntero a una cadena de caracteres que representa la nueva línea de entrada del usuario, y `clear_undo` es un valor entero que indica si se debe borrar el historial de edición de la línea anterior.

Aquí hay un ejemplo sencillo de cómo utilizar la función `rl_replace_line` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {
    char *input;

    input = readline("Introduce una palabra: ");

    if (input == NULL) {
        printf("No se ha introducido ninguna palabra\n");
        exit(EXIT_FAILURE);
    }

    // Modificar la línea de entrada
    char *new_input = malloc(strlen(input) + 2);
    sprintf(new_input, "%s!", input);
    rl_replace_line(new_input, 0);
    rl_redisplay();

    // Procesar la línea de entrada modificada
    printf("Has introducido la palabra %s\n", new_input);

    free(input);
    free(new_input);

    return 0;
}
```

En este ejemplo, se utiliza la función `readline` para solicitar al usuario que introduzca una palabra. Después de leer la entrada del usuario, se utiliza la función `rl_replace_line` para reemplazar la línea de entrada actual con una nueva línea que termina con un signo de exclamación.

Se utiliza la función `rl_redisplay` para volver a imprimir la nueva línea de entrada modificada en la consola.

Por último, se procesa la línea de entrada modificada y se imprime en la consola. Este ejemplo es un caso simple de cómo utilizar la función `rl_replace_line`, pero se puede utilizar en conjunto con otras funciones readline para realizar tareas más complejas, como la edición de la línea de entrada del usuario antes de enviarla al programa para su procesamiento.

rl_redisplay()

La función `rl_redisplay` es una función de la biblioteca readline que se utiliza para volver a imprimir la línea actual de entrada del usuario en la consola. Esta función es útil cuando se desea volver a imprimir la línea de entrada después de haberla modificado mediante otra función readline, como `rl_replace_line`.

La sintaxis de la función `rl_redisplay` es la siguiente:

```
void rl_redisplay(void);
```

Esta función no toma argumentos y no devuelve ningún valor. Simplemente vuelve a imprimir la línea actual de entrada del usuario en la consola.

Aquí hay un ejemplo sencillo de cómo utilizar la función `rl_redisplay` en conjunto con `rl_replace_line` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {
    char *input;

    input = readline("Introduce una palabra: ");

    if (input == NULL) {
        printf("No se ha introducido ninguna palabra\n");
        exit(EXIT_FAILURE);
    }

    // Modificar la línea de entrada
    char *new_input = malloc(strlen(input) + 2);
    sprintf(new_input, "%s!", input);
    rl_replace_line(new_input, 0);

    // Volver a imprimir la linea de entrada modificada
    rl_redisplay();

    // Procesar la línea de entrada modificada
    printf("Has introducido la palabra %s\n", new_input);

    free(input);
    free(new_input);

    return 0;
}
```

En este ejemplo, se utiliza la función `readline` para solicitar al usuario que introduzca una palabra. Después de leer la entrada del usuario, se utiliza la función `rl_replace_line` para reemplazar la línea de entrada actual con una nueva línea que termina con un signo de exclamación.

Después de modificar la línea de entrada, se utiliza la función `rl_redisplay` para volver a imprimir la línea de entrada modificada en la consola.

Por último, se procesa la línea de entrada modificada y se imprime en la consola. Este ejemplo es un caso simple de cómo utilizar la función `rl_redisplay`, pero se puede utilizar en conjunto con otras funciones `readline` para realizar tareas más complejas, como la edición de la línea de entrada del usuario antes de enviarla al programa para su procesamiento.

add_history()

La función `add_history` es una función de la biblioteca `readline` que se utiliza para añadir una línea de entrada a la historia de comandos. La historia de comandos es una lista de todas las líneas de entrada que ha introducido el usuario durante la sesión actual del programa.

La sintaxis de la función `add_history` es la siguiente:

```
void add_history(const char *line);
```

Esta función toma un argumento: `line` es un puntero a una cadena de caracteres que representa la línea de entrada que se va a añadir a la historia de comandos.

Aquí hay un ejemplo sencillo de cómo utilizar la función `add_history` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {
    char *input;

    while ((input = readline("Introduce una palabra: ")) != NULL) {
        // Añadir la línea de entrada a la historia de comandos
        add_history(input);

        // Procesar la línea de entrada
        printf("Has introducido la palabra %s\n", input);

        free(input);
    }

    return 0;
}
```

En este ejemplo, se utiliza un bucle `while` para solicitar al usuario que introduzca una palabra. Después de leer la entrada del usuario, se utiliza la función `add_history` para añadir la línea de

entrada a la historia de comandos.

Por último, se procesa la línea de entrada y se imprime en la consola. Este ejemplo es un caso simple de cómo utilizar la función `add_history`, pero se puede utilizar en conjunto con otras funciones readline para realizar tareas más complejas, como la edición de la línea de entrada del usuario antes de enviarla al programa para su procesamiento.

access()

La función `access` es una función de la biblioteca estándar de C que se utiliza para verificar si el proceso actual puede acceder a un archivo o directorio en el sistema de archivos.

La sintaxis de la función `access` es la siguiente:

```
int access(const char *path, int mode);
```

Esta función toma dos argumentos:

- `path` es una cadena de caracteres que representa la ruta del archivo o directorio que se va a verificar.
- `mode` es un valor entero que representa el tipo de acceso que se va a verificar. Puede ser `F_OK` para verificar la existencia del archivo, `R_OK` para verificar si el archivo tiene permisos de lectura, `W_OK` para verificar si el archivo tiene permisos de escritura, y `X_OK` para verificar si el archivo tiene permisos de ejecución.

La función `access` devuelve un valor entero que indica si el acceso se pudo verificar o no. Si la función devuelve `0`, significa que el acceso se pudo verificar correctamente. Si la función devuelve `-1`, significa que se produjo un error y se debe consultar la variable `errno` para obtener más información sobre el tipo de error que se produjo.

Aquí hay un ejemplo sencillo de cómo utilizar la función `access` en un programa de C:

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

int main() {
    char *path = "/etc/passwd";
    int ret;

    // Verificar si el archivo existe
    ret = access(path, F_OK);
    if (ret == 0) {
        printf("El archivo %s existe\n", path);
    } else {
        perror("access");
        exit(EXIT_FAILURE);
    }

    // Verificar si el archivo tiene permisos de lectura
    ret = access(path, R_OK);
    if (ret == 0) {
```

```

        printf("El archivo %s tiene permisos de lectura\n", path);
    } else {
        perror("access");
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

En este ejemplo, se utiliza la función `access` para verificar si el archivo `/etc/passwd` existe y si tiene permisos de lectura. Se utilizan las constantes `F_OK` y `R_OK` para especificar el tipo de acceso que se va a verificar.

Si la función `access` devuelve `0`, se imprime un mensaje indicando que el acceso se pudo verificar correctamente. Si la función devuelve `-1`, se utiliza la función `perror` para imprimir un mensaje de error que indica el tipo de error que se produjo.

open()

La función `open` es una función de la biblioteca estándar de C que se utiliza para abrir un archivo en el sistema de archivos y obtener un descriptor de archivo, que se utiliza para realizar operaciones de lectura y escritura en el archivo.

La sintaxis de la función `open` es la siguiente:

```
int open(const char *path, int flags);
```

Esta función toma dos argumentos:

- `path` es una cadena de caracteres que representa la ruta del archivo que se va a abrir.
- `flags` es un valor entero que representa los indicadores de modo de apertura del archivo.

Puede ser `O_RDONLY` para abrir el archivo en modo de sólo lectura, `O_WRONLY` para abrir el archivo en modo de sólo escritura, o `O_RDWR` para abrir el archivo en modo de lectura y escritura. Además, se pueden combinar otros indicadores con estos, como `O_CREAT` para crear el archivo si no existe, `O_TRUNC` para truncar el archivo a cero si ya existe, y `O_APPEND` para añadir datos al final del archivo.

La función `open` devuelve un valor entero que representa el descriptor de archivo del archivo abierto. Si la función devuelve `-1`, significa que se produjo un error y se debe consultar la variable `errno` para obtener más información sobre el tipo de error que se produjo.

Aquí hay un ejemplo sencillo de cómo utilizar la función `open` en un programa de C:

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    char *path = "archivo.txt";

```

```

int fd;

// Abrir el archivo en modo de sólo escritura y crearlo si no existe
fd = open(path, O_WRONLY | O_CREAT, 0666);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

// Escribir datos en el archivo
write(fd, "Hola, mundo!", 12);

// Cerrar el archivo
close(fd);

return 0;
}

```

En este ejemplo, se utiliza la función `open` para abrir el archivo `archivo.txt` en modo de sólo escritura y crearlo si no existe. Se utilizan los indicadores `O_WRONLY` y `O_CREAT` para especificar el modo de apertura y se utiliza el valor `0666` para establecer los permisos del archivo.

A continuación, se utiliza la función `write` para escribir los datos `"Hola, mundo!"` en el archivo. Finalmente, se cierra el archivo utilizando la función `close`.

read()

La función `read` es una función de la biblioteca estándar de C que se utiliza para leer datos de un archivo o de otro descriptor de archivo en el sistema operativo.

La sintaxis de la función `read` es la siguiente:

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

Esta función toma tres argumentos:

- `fd` es un valor entero que representa el descriptor de archivo del archivo o dispositivo desde el cual se leerán los datos.
- `buf` es un puntero a un área de memoria donde se almacenarán los datos leídos.
- `count` es un valor entero que representa el número de bytes que se leerán.

La función `read` devuelve un valor entero que representa el número de bytes leídos, o `-1` si se produjo un error. Si la función devuelve `-1`, se debe consultar la variable `errno` para obtener más información sobre el tipo de error que se produjo.

Aquí hay un ejemplo sencillo de cómo utilizar la función `read` en un programa de C:

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

```

#define BUFFER_SIZE 1024

int main() {
    char *path = "archivo.txt";
    int fd;
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    // Abrir el archivo en modo de sólo lectura
    fd = open(path, O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Leer datos del archivo
    bytes_read = read(fd, buffer, BUFFER_SIZE);
    if (bytes_read == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    // Mostrar los datos leídos por pantalla
    printf("Datos leídos: %.s\n", (int)bytes_read, buffer);

    // Cerrar el archivo
    close(fd);

    return 0;
}

```

En este ejemplo, se utiliza la función `open` para abrir el archivo `archivo.txt` en modo de sólo lectura. Se utiliza la constante `BUFFER_SIZE` para especificar el tamaño del búfer de lectura.

A continuación, se utiliza la función `read` para leer los datos del archivo y almacenarlos en el búfer de lectura. Si la función devuelve `-1`, se muestra un mensaje de error utilizando la función `perror`.

Finalmente, se muestra por pantalla los datos leídos utilizando la función `printf`. La cadena de formato `%.s` se utiliza para imprimir una cadena de caracteres con una longitud variable, que se especifica utilizando el valor entero `(int)bytes_read`.

Por último, se cierra el archivo utilizando la función `close`.

close()

La función `close` es una función de la biblioteca estándar de C que se utiliza para cerrar un descriptor de archivo. Cuando se termina de utilizar un archivo o un dispositivo en un programa, es importante cerrar el descriptor de archivo asociado para liberar los recursos del sistema operativo.

La sintaxis de la función `close` es la siguiente:

```
int close(int fd);
```

Esta función toma un argumento:

- `fd` es un valor entero que representa el descriptor de archivo que se va a cerrar.

La función `close` devuelve un valor entero que indica si se cerró correctamente el descriptor de archivo. Si la función devuelve `-1`, se debe consultar la variable `errno` para obtener más información sobre el tipo de error que se produjo.

Aquí hay un ejemplo sencillo de cómo utilizar la función `close` en un programa de C:

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

int main() {
    char *path = "archivo.txt";
    int fd;

    // Abrir el archivo en modo de sólo lectura
    fd = open(path, O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Hacer algo con el archivo...

    // Cerrar el archivo
    if (close(fd) == -1) {
        perror("close");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

En este ejemplo, se utiliza la función `open` para abrir el archivo `archivo.txt` en modo de sólo lectura. Después de hacer algo con el archivo, se utiliza la función `close` para cerrar el descriptor de archivo.

Si la función `close` devuelve `-1`, se muestra un mensaje de error utilizando la función `perror`.

fork()

La función `fork` es una llamada al sistema en Unix y sistemas operativos similares que se utiliza para crear un nuevo proceso, que es una copia exacta del proceso que llama (el proceso padre). El nuevo proceso creado se llama proceso hijo y tiene su propio espacio de direcciones, su propia tabla de archivos abiertos y su propio identificador de proceso (PID).

La sintaxis de la función `fork` es la siguiente:

```
#include <unistd.h>

pid_t fork(void);
```

Esta función no toma argumentos y devuelve un valor entero. Si la función devuelve 0, se está ejecutando en el proceso hijo. Si la función devuelve un valor positivo, se está ejecutando en el proceso padre y el valor de retorno es el PID del proceso hijo. Si la función devuelve un valor negativo, se produjo un error al crear el proceso hijo.

Aquí hay un ejemplo sencillo de cómo utilizar la función `fork` en un programa de C:

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

int main() {
    pid_t pid;

    // Crear un nuevo proceso
    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Estamos en el proceso hijo
        printf("Hola, soy el proceso hijo!\n");
    } else {
        // Estamos en el proceso padre
        printf("Hola, soy el proceso padre!\n");
    }

    return 0;
}
```

En este ejemplo, se llama a la función `fork` para crear un nuevo proceso. El proceso padre imprime "Hola, soy el proceso padre!" en la salida estándar, mientras que el proceso hijo imprime "Hola, soy el proceso hijo!".

Es importante tener en cuenta que después de llamar a `fork`, ambos procesos (padre e hijo) continúan ejecutando el mismo código a partir de la línea siguiente a la llamada a `fork`. La única diferencia es el valor devuelto por la función `fork`, que permite al programa saber en qué proceso está actualmente ejecutándose. En el ejemplo anterior, se utilizó una declaración `if-else` para imprimir un mensaje diferente en función del valor devuelto por `fork`.

wait()

La función `wait` es una llamada al sistema en Unix y sistemas operativos similares que se utiliza para esperar a que un proceso hijo termine y obtener información sobre su estado de salida. Esta función suspende la ejecución del proceso padre hasta que uno de sus procesos hijos termine.

La sintaxis de la función `wait` es la siguiente:

```
#include <sys/wait.h>

pid_t wait(int *status);
```

Esta función toma un argumento:

- `status` es un puntero a un entero que se utilizará para almacenar información sobre el estado de salida del proceso hijo.

La función `wait` devuelve el PID del proceso hijo que ha terminado. Si se produjo un error, la función devuelve `-1`.

Aquí hay un ejemplo sencillo de cómo utilizar la función `wait` en un programa de C:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;

    // Crear un nuevo proceso
    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Estamos en el proceso hijo
        printf("Hola, soy el proceso hijo!\n");
        exit(EXIT_SUCCESS);
    } else {
        // Estamos en el proceso padre
        printf("Hola, soy el proceso padre!\n");

        // Esperar a que el proceso hijo termine
        if (wait(&status) == -1) {
            perror("wait");
            exit(EXIT_FAILURE);
        }

        if (WIFEXITED(status)) {
            printf("El proceso hijo terminó con estado de salida %d\n", WEXITSTATUS(status));
        } else {
            printf("El proceso hijo terminó anormalmente\n");
        }
    }
}

return 0;
}
```

En este ejemplo, se llama a la función `fork` para crear un nuevo proceso. El proceso hijo imprime "Hola, soy el proceso hijo!" y sale con un estado de salida `EXIT_SUCCESS`. El proceso padre imprime "Hola, soy el proceso padre!", espera a que el proceso hijo termine utilizando la función `wait`, y luego imprime información sobre el estado de salida del proceso hijo utilizando las macros de la biblioteca `sys/wait.h`.

Es importante tener en cuenta que la función `wait` solo espera a que un proceso hijo termine. Si el proceso padre tiene varios procesos hijos, debe llamar a `wait` varias veces para esperar a que terminen todos los procesos hijos.

waitpid()

La función `waitpid` es una llamada al sistema en Unix y sistemas operativos similares que se utiliza para esperar a que un proceso hijo específico termine y obtener información sobre su estado de salida. Esta función es similar a la función `wait`, pero proporciona una mayor flexibilidad en la selección del proceso hijo que se espera.

La sintaxis de la función `waitpid` es la siguiente:

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Esta función toma tres argumentos:

- `pid` es el PID del proceso hijo que se espera. Si se establece en `-1`, la función espera a cualquier proceso hijo.
- `status` es un puntero a un entero que se utilizará para almacenar información sobre el estado de salida del proceso hijo.
- `options` son opciones adicionales para controlar el comportamiento de la función `waitpid`.

La función `waitpid` devuelve el PID del proceso hijo que ha terminado. Si se produjo un error, la función devuelve `-1`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;

    // Crear un proceso hijo
    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
```

```

// Estamos en el proceso hijo
printf("Hola, soy el proceso hijo!\n");
exit(EXIT_SUCCESS);
}

// Estamos en el proceso padre
printf("Hola, soy el proceso padre!\n");

// Esperar a que el proceso hijo termine
if (waitpid(pid, &status, 0) == -1) {
    perror("waitpid");
    exit(EXIT_FAILURE);
}

if (WIFEXITED(status)) {
    printf("El proceso hijo terminó con estado de salida %d\n", WEXITSTATUS(status));
} else {
    printf("El proceso hijo terminó anormalmente\n");
}

return 0;
}

```

En este ejemplo, se llama a la función `fork` para crear un proceso hijo. El proceso hijo imprime un mensaje y sale con un estado de éxito (`EXIT_SUCCESS`). El proceso padre espera a que el proceso hijo termine utilizando la función `waitpid`, y luego imprime un mensaje indicando el estado de salida del proceso hijo.

En este caso, estamos utilizando `waitpid` con los siguientes argumentos:

- `pid`: el ID del proceso hijo que queremos esperar.
- `status`: un puntero a un entero que utilizaremos para almacenar información sobre el estado de salida del proceso hijo.
- `0`: una máscara de bits que especifica que queremos esperar a que el proceso hijo termine de forma normal.

wait3()

La función `wait3` en C es similar a la función `waitpid`, pero proporciona más información sobre el proceso hijo. Aquí te explico en qué consiste y te muestro un ejemplo de su uso:

La función `wait3` es una función de bajo nivel que espera a que un proceso hijo termine y proporciona información adicional sobre su estado. A diferencia de `waitpid`, la función `wait3` proporciona información adicional sobre el uso de recursos del proceso hijo, como la cantidad de tiempo de CPU que utilizó y la cantidad de memoria que consumió.

La función `wait3` tiene la siguiente declaración:

```
pid_t wait3(int *status, int options, struct rusage *rusage);
```

- `status` : un puntero a un entero que almacenará información sobre el estado de salida del proceso hijo.
- `options` : una máscara de bits que controla el comportamiento de la función `wait3`.
- `rusage` : un puntero a una estructura `rusage` que se utiliza para almacenar información sobre los recursos utilizados por el proceso hijo.

Aquí tienes un ejemplo de cómo utilizar la función `wait3` en C:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/resource.h>

int main() {
    pid_t pid;
    int status;
    struct rusage usage;

    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        printf("Hola, soy el proceso hijo!\n");
        sleep(2);
        exit(EXIT_SUCCESS);
    }

    printf("Hola, soy el proceso padre!\n");

    if (wait3(&status, 0, &usage) == -1) {
        perror("wait3");
        exit(EXIT_FAILURE);
    }

    if (WIFEXITED(status)) {
        printf("El proceso hijo terminó con estado de salida %d\n", WEXITSTATUS(status));
        printf("El proceso hijo utilizó %ld segundos de tiempo de CPU y %ld kilobytes de memoria\n",
               usage.ru_utime.tv_sec, usage.ru_maxrss);
    } else {
        printf("El proceso hijo terminó anormalmente\n");
    }

    return 0;
}
```

En este ejemplo, el proceso padre crea un proceso hijo utilizando la función `fork`. El proceso hijo imprime un mensaje y duerme durante dos segundos antes de salir con un estado de éxito. El proceso padre espera a que el proceso hijo termine utilizando la función `wait3`, y luego imprime información adicional sobre los recursos que utilizó el proceso hijo.

En este caso, estamos utilizando `wait3` con los siguientes argumentos:

- `status` : un puntero a un entero que almacenará información sobre el estado de salida del proceso hijo.
- `0` : una máscara de bits que especifica que queremos esperar a que el proceso hijo termine de forma normal.
- `&usage` : un puntero a una estructura `rusage` que utilizaremos para almacenar información sobre el uso de recursos del proceso hijo.

Espero que esto te ayude a entender cómo funciona la función `wait3` en C.

wait4()

La función `wait4` es una variante de la función `waitpid` que proporciona información adicional sobre los procesos hijos. Esta función es similar a `wait3`, pero es más flexible en cuanto a las opciones que se pueden pasar y es más adecuada para su uso en sistemas con múltiples procesadores. Aquí te explico en qué consiste y te muestro un ejemplo de su uso:

La función `wait4` es una función de bajo nivel que espera a que un proceso hijo termine y proporciona información adicional sobre su estado y uso de recursos. A diferencia de `waitpid`, la función `wait4` permite pasar opciones adicionales que afectan su comportamiento.

La función `wait4` tiene la siguiente declaración:

```
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

- `pid` : el PID del proceso hijo que se desea esperar. Si se establece en -1, la función esperará a cualquier proceso hijo.
- `status` : un puntero a un entero que almacenará información sobre el estado de salida del proceso hijo.
- `options` : una máscara de bits que controla el comportamiento de la función `wait4`.
- `rusage` : un puntero a una estructura `rusage` que se utiliza para almacenar información sobre los recursos utilizados por el proceso hijo.

La función `wait4()` en C se utiliza para suspender la ejecución del proceso actual hasta que alguno de sus procesos secundarios termine. La función también proporciona información sobre el proceso hijo que ha terminado.

Aquí hay un ejemplo que muestra el uso de la función `wait4()` en C:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;
```

```

int options = 0;
struct rusage usage;

pid = fork();

if (pid < 0) {
    printf("Fork failed\n");
    exit(1);
}
else if (pid == 0) {
    printf("Child process running\n");
    sleep(3);
    printf("Child process exiting\n");
    exit(0);
}
else {
    printf("Parent process waiting for child to finish...\n");
    pid_t child_pid = wait4(pid, &status, options, &usage);
    printf("Child with pid %d has finished\n", child_pid);
    printf("Exit status: %d\n", WEXITSTATUS(status));
    printf("User time used: %ld\n", usage.ru_utime.tv_usec);
    printf("System time used: %ld\n", usage.ru_stime.tv_usec);
}
return 0;
}

```

En este ejemplo, el proceso padre crea un proceso hijo utilizando la función `fork()`. El proceso hijo espera 3 segundos y luego sale. El proceso padre espera a que el proceso hijo termine usando la función `wait4()`. El código también proporciona información sobre el uso de recursos del proceso hijo a través del parámetro `rusage`.

La salida del programa sería algo así:

```

Parent process waiting for child to finish...
Child process running
Child process exiting
Child with pid 12345 has finished
Exit status: 0
User time used: 0
System time used: 0

```

En este ejemplo, "12345" es el pid del proceso hijo. El estado de salida del proceso hijo es 0 (lo que significa que terminó sin errores). Y la estructura `rusage` proporciona información sobre el tiempo de CPU utilizado por el proceso hijo.

signal()

La función `signal()` en C se utiliza para manipular las señales del sistema. Las señales son notificaciones asíncronas enviadas a un proceso para indicar ciertas condiciones, como la terminación de un proceso, la interrupción del usuario o la ocurrencia de un error fatal. La función `signal()` se utiliza para configurar la acción que se tomará cuando se reciba una señal.

Aquí hay un ejemplo que muestra el uso de la función `signal()` en C:

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int signal_num) {
    printf("Signal %d received\n", signal_num);
    exit(signal_num);
}

int main() {
    signal(SIGINT, signal_handler);

    while (1) {
        printf("Waiting for signal...\n");
        sleep(1);
    }

    return 0;
}

```

En este ejemplo, el proceso principal utiliza la función `signal()` para establecer un controlador de señal personalizado para la señal SIGINT (que es la señal enviada al proceso cuando se presiona Ctrl+C en la terminal). El controlador de señal personalizado es una función llamada `signal_handler()` que simplemente imprime un mensaje de que se ha recibido una señal y sale del programa con el número de señal recibido.

El programa entra en un bucle infinito y espera a que se reciba la señal SIGINT. Cuando se presiona Ctrl+C en la terminal, se envía la señal SIGINT al proceso y el controlador de señal personalizado se activa, imprimiendo un mensaje y saliendo del programa con el número de señal 2 (que es el número de señal de la señal SIGINT).

La salida del programa sería algo así:

```

Waiting for signal...
Waiting for signal...
^CSignal 2 received

```

En este ejemplo, "^AC" representa el carácter Ctrl+C que se ha presionado en la terminal para enviar la señal SIGINT.

sigaction()

La función `sigaction()` en C se utiliza para configurar y manipular la acción que se tomará cuando se reciba una señal en un programa. Permite un mayor control sobre el manejo de señales que la función `signal()`, ya que se pueden especificar opciones adicionales y se puede obtener información sobre el comportamiento previo de la señal.

Aquí hay un ejemplo que muestra cómo usar la función `sigaction()` para manejar la señal SIGINT:

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void sigint_handler(int signal_num) {
    printf("Caught SIGINT signal\n");
    exit(signal_num);
}

int main() {
    struct sigaction sa;

    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while (1) {
        printf("Waiting for signal...\n");
        sleep(1);
    }

    return 0;
}

```

En este ejemplo, se utiliza la función `sigaction()` para establecer un controlador de señal personalizado para la señal SIGINT. La estructura `struct sigaction` se utiliza para especificar la acción que se tomará cuando se reciba la señal. En este caso, el controlador de señal personalizado es una función llamada `sigint_handler()` que simplemente imprime un mensaje de que se ha recibido una señal y sale del programa con el número de señal recibido.

El programa entra en un bucle infinito y espera a que se reciba la señal SIGINT. Cuando se presiona Ctrl+C en la terminal, se envía la señal SIGINT al proceso y el controlador de señal personalizado se activa, imprimiendo un mensaje y saliendo del programa con el número de señal 2 (que es el número de señal de la señal SIGINT).

La salida del programa sería algo así:

```

Waiting for signal...
Waiting for signal...
^CCaught SIGINT signal

```

En este ejemplo, "^AC" representa el carácter Ctrl+C que se ha presionado en la terminal para enviar la señal SIGINT.

kill()

La función `kill()` en C se utiliza para enviar una señal a un proceso específico o a un grupo de procesos. La señal puede ser una de las señales predefinidas en el sistema operativo, como SIGINT (Ctrl+C) o SIGKILL (matar).

Aquí hay un ejemplo que muestra cómo usar la función `kill()` para enviar la señal SIGUSR1 a otro proceso:

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

int main() {
    pid_t pid;

    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(1);
    }
    else if (pid == 0) {
        // Child process
        printf("Child process waiting...\n");
        pause();
        printf("Child process resumed\n");
        exit(0);
    }
    else {
        // Parent process
        sleep(1);
        printf("Sending SIGUSR1 signal to child process\n");
        kill(pid, SIGUSR1);
        printf("Sent SIGUSR1 signal to child process\n");
    }

    return 0;
}
```

En este ejemplo, se utiliza la función `kill()` para enviar la señal SIGUSR1 al proceso hijo creado con `fork()`. El proceso hijo está en un bucle de espera infinita hasta que recibe la señal, momento en el que el programa continúa su ejecución y finaliza. El proceso padre espera 1 segundo antes de enviar la señal al proceso hijo utilizando la función `kill()`.

La salida del programa sería algo así:

```
Child process waiting...
Sending SIGUSR1 signal to child process
Sent SIGUSR1 signal to child process
Child process resumed
```

En este ejemplo, el proceso hijo entra en un bucle de espera infinita después de imprimir "Child process waiting...". Cuando se recibe la señal SIGUSR1 del proceso padre, el programa continúa su ejecución, imprimiendo "Child process resumed" y saliendo del programa con un

código de salida 0. El proceso padre espera 1 segundo antes de enviar la señal SIGUSR1 al proceso hijo utilizando la función `kill()`. Cuando se envía la señal, el proceso hijo recibe la señal y continúa su ejecución.

exit()

La función `exit()` en C se utiliza para terminar la ejecución del programa y devolver un valor de salida al sistema operativo. El valor de salida puede ser utilizado por otros programas que ejecuten el programa terminado, para determinar si la ejecución fue exitosa o no.

Aquí hay un ejemplo que muestra cómo usar la función `exit()` para terminar la ejecución del programa:

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

int main() {
    printf("El programa ha comenzado\n");

    // Realiza algunas operaciones aquí...

    printf("El programa ha terminado con éxito\n");
    exit(0);
}
```

En este ejemplo, el programa imprime "El programa ha comenzado" y luego realiza algunas operaciones que no se muestran en el código. Luego, el programa imprime "El programa ha terminado con éxito" y llama a la función `exit()` con un valor de salida de 0. Esto indica que el programa ha terminado con éxito.

La salida del programa sería algo así:

```
El programa ha comenzado
El programa ha terminado con éxito
```

En este ejemplo, el programa termina con éxito y devuelve un valor de salida de 0 al sistema operativo. Si el programa encontrara algún error o no pudiera realizar las operaciones necesarias, se podría llamar a la función `exit()` con un valor diferente de 0 para indicar que el programa terminó de manera anormal o con errores.

getcwd()

La función `getcwd()` en C se utiliza para obtener el nombre del directorio de trabajo actual. Esta función toma como argumento un búfer de caracteres donde se almacenará el nombre del directorio actual, y un tamaño máximo para ese búfer.

Aquí hay un ejemplo que muestra cómo utilizar la función `getcwd()` para obtener el directorio de trabajo actual:

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

int main() {
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("El directorio de trabajo actual es: %s\n", cwd);
    } else {
        perror("getcwd() error");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

En este ejemplo, primero se declara un búfer de caracteres llamado `cwd` con un tamaño de 1024 caracteres. Luego, se llama a la función `getcwd()` para obtener el directorio actual y se almacena en el búfer `cwd`.

Si `getcwd()` tiene éxito, la función imprime el nombre del directorio de trabajo actual utilizando la función `printf()`. Si `getcwd()` falla, la función imprime un mensaje de error utilizando la función `perror()` y termina el programa con la función `exit()`.

La salida del programa sería algo así:

```
El directorio de trabajo actual es: /home/usuario/
```

En este ejemplo, el programa muestra el directorio de trabajo actual en la salida estándar. La ruta del directorio mostrada en la salida puede ser diferente en función del sistema operativo y del directorio actual del usuario en el momento de la ejecución del programa.

chdir()

La función `chdir()` en C se utiliza para cambiar el directorio de trabajo actual a un directorio especificado por la ruta proporcionada como argumento.

Aquí hay un ejemplo que muestra cómo utilizar la función `chdir()` para cambiar el directorio actual a un directorio especificado:

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

int main() {
    char *newdir = "/home/usuario/mi_directorio";
    if (chdir(newdir) != 0) {
        perror("chdir() error");
        exit(EXIT_FAILURE);
    }
}
```

```

    }
    printf("El directorio de trabajo actual es: %s\n", getcwd(NULL, 0));
    return 0;
}

```

En este ejemplo, primero se declara una variable `newdir` que contiene la ruta del directorio al que se quiere cambiar. Luego, se llama a la función `chdir()` con `newdir` como argumento.

Si `chdir()` tiene éxito, la función `getcwd()` se llama para obtener el nombre del directorio actual y mostrarlo en la salida estándar utilizando `printf()`. Si `chdir()` falla, la función imprime un mensaje de error utilizando la función `perror()` y termina el programa con la función `exit()`.

La salida del programa sería algo así:

```
El directorio de trabajo actual es: /home/usuario/mi_directorio
```

En este ejemplo, el programa cambia el directorio de trabajo actual a `/home/usuario/mi_directorio` y luego muestra el directorio de trabajo actual utilizando la función `getcwd()`. La ruta del directorio mostrada en la salida puede ser diferente en función del sistema operativo y del directorio especificado en la variable `newdir`.

stat()

La función `stat()` en C se utiliza para obtener información sobre un archivo, como sus permisos, tamaño, fecha de última modificación, etc. La información se guarda en una estructura llamada `struct stat`.

Aquí hay un ejemplo que muestra cómo utilizar la función `stat()` para obtener información sobre un archivo especificado:

```

#include <stdio.h>
#include <sys/stat.h>

int main() {
    char *archivo = "archivo.txt";
    struct stat info;

    if (stat(archivo, &info) != 0) {
        perror("stat() error");
        return -1;
    }

    printf("El archivo %s tiene un tamaño de %lld bytes\n", archivo, info.st_size);
    printf("Los permisos del archivo son %o\n", info.st_mode & 0777);

    return 0;
}

```

En este ejemplo, se declara una variable `archivo` que contiene el nombre del archivo del que se desea obtener información. A continuación, se declara una estructura `struct stat` llamada `info`

que se utilizará para almacenar la información del archivo.

La función `stat()` se llama con `archivo` como primer argumento y la dirección de `info` como segundo argumento. Si `stat()` tiene éxito, la información del archivo se almacena en la estructura `info`.

En este ejemplo, se imprimen dos elementos de la estructura `info`. El primero es el tamaño del archivo, que se encuentra en el miembro `st_size`. El segundo es los permisos del archivo, que se encuentran en el miembro `st_mode`.

Si `stat()` falla, la función `perror()` se utiliza para imprimir un mensaje de error.

La salida del programa sería algo así:

```
El archivo archivo.txt tiene un tamaño de 100 bytes
Los permisos del archivo son 644
```

En este ejemplo, el programa utiliza la función `stat()` para obtener información sobre el archivo `archivo.txt`, como su tamaño y permisos. La salida mostrada puede variar según el sistema operativo y el archivo utilizado en el ejemplo.

Istat()

La función `lstat()` en C es similar a la función `stat()`, pero se utiliza para obtener información sobre un enlace simbólico en lugar del archivo al que está enlazado. La información se guarda en una estructura llamada `struct stat`.

Aquí hay un ejemplo que muestra cómo utilizar la función `lstat()` para obtener información sobre un enlace simbólico especificado:

```
#include <stdio.h>
#include <sys/stat.h>

int main() {
    char *enlace_simbolico = "enlace_simbolico.txt";
    struct stat info;

    if (lstat(enlace_simbolico, &info) != 0) {
        perror("lstat() error");
        return -1;
    }

    printf("El enlace simbólico %s tiene un tamaño de %lld bytes\n", enlace_simbolico, info.st_size);
    printf("Los permisos del enlace simbólico son %o\n", info.st_mode & 0777);

    return 0;
}
```

En este ejemplo, se declara una variable `enlace_simbolico` que contiene el nombre del enlace simbólico del que se desea obtener información. A continuación, se declara una estructura `struct stat` llamada `info` que se utilizará para almacenar la información del enlace simbólico.

La función `lstat()` se llama con `enlace_simbolico` como primer argumento y la dirección de `info` como segundo argumento. Si `lstat()` tiene éxito, la información del enlace simbólico se almacena en la estructura `info`.

En este ejemplo, se imprimen dos elementos de la estructura `info`. El primero es el tamaño del enlace simbólico, que se encuentra en el miembro `st_size`. El segundo es los permisos del enlace simbólico, que se encuentran en el miembro `st_mode`.

Si `lstat()` falla, la función `perror()` se utiliza para imprimir un mensaje de error.

La salida del programa sería algo así:

```
El enlace simbolico enlace_simbolico.txt tiene un tamaño de 12 bytes
Los permisos del enlace simbolico son 777
```

En este ejemplo, el programa utiliza la función `lstat()` para obtener información sobre el enlace simbólico `enlace_simbolico.txt`, como su tamaño y permisos. La salida mostrada puede variar según el sistema operativo y el enlace simbólico utilizado en el ejemplo.

fstat()

La función `fstat()` en C se utiliza para obtener información sobre un archivo abierto a través de un descriptor de archivo. Esta función es similar a la función `stat()`, pero en lugar de tomar el nombre del archivo como argumento, toma un descriptor de archivo.

La sintaxis de la función es la siguiente:

```
int fstat(int fd, struct stat *buf);
```

Donde `fd` es el descriptor de archivo y `buf` es un puntero a una estructura `stat` donde se almacenará la información del archivo.

Aquí hay un ejemplo de cómo usar la función `fstat()` para obtener información sobre un archivo:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
    struct stat buf;

    // Abrir el archivo de ejemplo
    fd = open("archivo.txt", O_RDONLY);

    // Verificar si hubo un error al abrir el archivo
    if (fd == -1) {
```

```

        perror("open");
        exit(EXIT_FAILURE);
    }

    // Obtener la información del archivo
    if (fstat(fd, &buf) == -1) {
        perror("fstat");
        exit(EXIT_FAILURE);
    }

    // Imprimir la información del archivo
    printf("Tamaño del archivo: %ld bytes\n", buf.st_size);
    printf("Número de inodo: %ld\n", buf.st_ino);
    printf("Último acceso: %s", ctime(&buf.st_atime));
    printf("Última modificación: %s", ctime(&buf.st_mtime));
    printf("Último cambio de estado: %s", ctime(&buf.st_ctime));

    // Cerrar el archivo
    close(fd);

    return 0;
}

```

En este ejemplo, se abre el archivo `"archivo.txt"` y se utiliza la función `fstat()` para obtener su información. Luego, se imprime la información del archivo, incluyendo su tamaño, número de inodo y fechas de acceso, modificación y cambio de estado. Finalmente, se cierra el archivo.

unlink()

La función `unlink()` en C se utiliza para eliminar un archivo del sistema de archivos. La función toma como argumento el nombre del archivo a eliminar y devuelve 0 en caso de éxito, o -1 si hay algún error.

La sintaxis de la función es la siguiente:

```
int unlink(const char *path);
```

Donde `path` es el nombre del archivo a eliminar.

Aquí hay un ejemplo de cómo usar la función `unlink()` para eliminar un archivo:

```

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

int main() {
    // Crear un archivo de ejemplo
    FILE *archivo = fopen("ejemplo.txt", "w");
    fprintf(archivo, "Este es un archivo de ejemplo\n");
    fclose(archivo);

    // Eliminar el archivo
    if (unlink("ejemplo.txt") == -1) {
        perror("unlink");
        exit(EXIT_FAILURE);
    }
}

```

```

    }

    printf("El archivo ha sido eliminado\n");

    return 0;
}

```

En este ejemplo, se crea un archivo llamado `"ejemplo.txt"` con un contenido de ejemplo. Luego, se utiliza la función `unlink()` para eliminar el archivo. Si la función devuelve -1, se imprime un mensaje de error y se sale del programa. Si la función tiene éxito, se imprime un mensaje de confirmación de que el archivo ha sido eliminado.

execve

La función `execve()` en C se utiliza para ejecutar un programa en el mismo espacio de proceso que el programa que llama. La función toma como argumentos el nombre del archivo a ejecutar, una matriz de argumentos de línea de comando y un conjunto de variables de entorno para el nuevo programa.

La sintaxis de la función es la siguiente:

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

Donde `path` es el nombre del archivo a ejecutar, `argv` es una matriz de argumentos de línea de comando y `envp` es una matriz de variables de entorno.

Aquí hay un ejemplo de cómo usar la función `execve()` para ejecutar un programa en el mismo espacio de proceso que el programa que llama:

```

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

int main() {
    char *const argv[] = {"ls", "-l", NULL};
    char *const envp[] = {NULL};
    if (execve("/bin/ls", argv, envp) == -1) {
        perror("execve");
        exit(EXIT_FAILURE);
    }
    return 0;
}

```

En este ejemplo, se utiliza la función `execve()` para ejecutar el programa `/bin/ls` con los argumentos de línea de comando `{"ls", "-l", NULL}` y un conjunto vacío de variables de entorno. Si la función devuelve -1, se imprime un mensaje de error y se sale del programa. Si la función tiene éxito, el programa llamador será reemplazado por el programa ejecutado por `execve()`.

dup()

La función `dup()` en C se utiliza para duplicar un descriptor de archivo. Esto significa que crea una copia del descriptor de archivo existente y devuelve un nuevo descriptor de archivo que se puede utilizar para acceder al mismo archivo.

La sintaxis de la función es la siguiente:

```
int dup(int oldfd);
```

Donde `oldfd` es el descriptor de archivo que se va a duplicar.

Aquí hay un ejemplo de cómo usar la función `dup()` para duplicar el descriptor de archivo estándar de entrada (`stdin`):

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

int main() {
    int stdin_copy = dup(STDIN_FILENO);
    if (stdin_copy == -1) {
        perror("dup");
        return 1;
    }
    printf("El descriptor de archivo original de stdin es: %d\n", STDIN_FILENO);
    printf("El nuevo descriptor de archivo para stdin es: %d\n", stdin_copy);
    return 0;
}
```

En este ejemplo, se utiliza la función `dup()` para duplicar el descriptor de archivo estándar de entrada (`stdin`). El resultado de la función `dup()` es el nuevo descriptor de archivo que se ha creado. El nuevo descriptor de archivo se almacena en la variable `stdin_copy`. Se comprueba si la función `dup()` se ha ejecutado correctamente y se imprimen los valores de los descriptores de archivo originales y duplicados. El descriptor de archivo original de `stdin` es 0, que se puede acceder a través de la macro `STDIN_FILENO`.

dup2()

La función `dup2()` en C se utiliza para duplicar un descriptor de archivo y reasignarlo a otro número de descriptor de archivo específico. Si el nuevo descriptor de archivo ya está en uso, se cierra antes de duplicar el descriptor de archivo.

La sintaxis de la función es la siguiente:

```
int dup2(int oldfd, int newfd);
```

Donde `oldfd` es el descriptor de archivo que se va a duplicar y `newfd` es el número de descriptor de archivo específico al que se desea asignar el nuevo descriptor de archivo.

Aquí hay un ejemplo de cómo usar la función `dup2()` para duplicar el descriptor de archivo estándar de entrada (`stdin`) y asignarlo al número de descriptor de archivo 10:

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

int main() {
    int newfd = 10;
    int stdin_copy = dup2(STDIN_FILENO, newfd);
    if (stdin_copy == -1) {
        perror("dup2");
        return 1;
    }
    printf("El descriptor de archivo original de stdin es: %d\n", STDIN_FILENO);
    printf("El nuevo descriptor de archivo para stdin es: %d\n", stdin_copy);
    printf("El descriptor de archivo para stdin asignado a %d\n", newfd);
    return 0;
}
```

En este ejemplo, se utiliza la función `dup2()` para duplicar el descriptor de archivo estándar de entrada (`stdin`) y asignarlo al número de descriptor de archivo 10. El resultado de la función `dup2()` es el nuevo descriptor de archivo que se ha creado y se ha asignado al número de descriptor de archivo 10. Si el número de descriptor de archivo 10 ya está en uso, se cerrará antes de duplicar el descriptor de archivo. Se comprueba si la función `dup2()` se ha ejecutado correctamente y se imprimen los valores de los descriptores de archivo originales y duplicados, así como el número de descriptor de archivo al que se ha asignado el nuevo descriptor de archivo.

pipe()

La función `pipe` en C se utiliza para crear una tubería (pipe) que se puede utilizar para la comunicación entre procesos. Una tubería es un mecanismo que permite la comunicación entre procesos, donde uno de los procesos es el escritor y el otro es el lector. Los datos escritos en un extremo de la tubería pueden ser leídos desde el otro extremo.

La función `pipe` crea un par de descriptores de archivo (file descriptors) que representan los extremos de la tubería. El descriptor de archivo `fd[0]` es el extremo de lectura (read end) de la tubería, y el descriptor de archivo `fd[1]` es el extremo de escritura (write end) de la tubería.

La sintaxis de la función `pipe` es la siguiente:

```
int pipe(int fd[2]);
```

Donde `fd` es un array de dos enteros que contendrá los descriptores de archivo del extremo de lectura y del extremo de escritura de la tubería.

Aquí hay un ejemplo de cómo se utiliza la función `pipe` en C:

```

#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2];
    char buffer[20];

    // Crear la tubería
    if (pipe(fd) == -1) {
        perror("pipe");
        return 1;
    }

    // Escribir datos en la tubería
    write(fd[1], "Hola mundo", 10);

    // Leer datos de la tubería
    read(fd[0], buffer, 10);

    printf("Mensaje recibido: %s\n", buffer);

    return 0;
}

```

En este ejemplo, se crea una tubería usando la función `pipe`. Luego, se escribe la cadena "Hola mundo" en el extremo de escritura de la tubería (`fd[1]`) usando la función `write`. Finalmente, se lee la misma cadena desde el extremo de lectura de la tubería (`fd[0]`) usando la función `read`, y se imprime en la pantalla.

opendir()

La función `opendir()` en C se utiliza para abrir un directorio y leer su contenido. Esta función devuelve un puntero al directorio abierto que se utilizará posteriormente en otras llamadas de función para leer su contenido.

La función `opendir()` tiene la siguiente sintaxis:

```

#include <dirent.h>

DIR *opendir(const char *name);

```

- `name`: es una cadena de caracteres que representa el nombre del directorio que se quiere abrir.
- La función devuelve un puntero de tipo `DIR` que apunta al directorio abierto.

A continuación, se muestra un ejemplo que utiliza la función `opendir()` para abrir un directorio y leer su contenido:

```

#include <stdio.h>
#include <dirent.h>

```

```

int main() {
    DIR *dir;
    struct dirent *ent;

    // abrir el directorio actual
    dir = opendir(".");

    // verificar si el directorio se ha abierto correctamente
    if (dir == NULL) {
        printf("No se pudo abrir el directorio");
        return 1;
    }

    // leer y mostrar el contenido del directorio
    while ((ent = readdir(dir)) != NULL) {
        printf("%s\n", ent->d_name);
    }

    // cerrar el directorio
    closedir(dir);

    return 0;
}

```

En este ejemplo, se utiliza la función `opendir()` para abrir el directorio actual ".", que es el directorio en el que se está ejecutando el programa. Luego, se verifica si el directorio se ha abierto correctamente y se utiliza un bucle `while` y la función `readdir()` para leer y mostrar el contenido del directorio. Finalmente, se utiliza la función `closedir()` para cerrar el directorio.

readdir()

La función `readdir` en C se utiliza para leer el siguiente elemento de un directorio. Toma como argumento un puntero al directorio que se va a leer y devuelve un puntero a una estructura `dirent` que contiene información sobre el elemento leído.

La sintaxis de la función `readdir` es la siguiente:

```

#include <dirent.h>

struct dirent *readdir(DIR *dirp);

```

donde `dirp` es un puntero al directorio abierto previamente utilizando la función `opendir`. La estructura `dirent` se define en la cabecera `dirent.h` y contiene los siguientes campos:

```

struct dirent {
    ino_t         d_ino;      // número de inodo
    off_t         d_off;      // desplazamiento en el directorio
    unsigned short d_reclen;  // longitud del registro
    unsigned char  d_type;    // tipo de archivo
    char          d_name[256]; // nombre del archivo
};

```

El campo `d_name` contiene el nombre del archivo, mientras que el campo `d_type` indica el tipo de archivo. Los tipos de archivo que se pueden encontrar son los siguientes:

- `DT_BLK` : dispositivo de bloques
- `DT_CHR` : dispositivo de caracteres
- `DT_DIR` : directorio
- `DT_FIFO` : canalización FIFO
- `DT_LNK` : enlace simbólico
- `DT_REG` : archivo regular
- `DT_SOCK` : socket
- `DT_UNKNOWN` : tipo desconocido

A continuación se muestra un ejemplo de cómo utilizar la función `readdir` para leer los archivos de un directorio:

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

int main(void) {
    DIR *dir;
    struct dirent *ent;

    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    while ((ent = readdir(dir)) != NULL) {
        printf("%s\n", ent->d_name);
    }

    closedir(dir);
    return 0;
}
```

Este programa lee los archivos del directorio actual y los imprime en la pantalla. El programa abre el directorio utilizando la función `opendir` y comprueba si se abrió correctamente. Luego, utiliza un bucle `while` para llamar a la función `readdir` y leer cada uno de los archivos en el directorio. El nombre del archivo se imprime utilizando `printf`. Finalmente, el programa cierra el directorio utilizando la función `closedir`.

`closedir()`

La función `closedir()` en C se utiliza para cerrar un directorio abierto con la función `opendir()`. Esta función libera la memoria asignada al directorio y cierra el descriptor de archivo.

Aquí está la sintaxis de la función `closedir()`:

```
int closedir(DIR *dirp);
```

La función devuelve un valor entero. Si la operación se realiza con éxito, devuelve 0. De lo contrario, devuelve -1.

Aquí hay un ejemplo que muestra cómo se puede usar la función `closedir()`:

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

int main() {
    DIR *dir = opendir(".");
    struct dirent *ent;

    if (dir == NULL) {
        perror("No se puede abrir el directorio");
        return -1;
    }

    while ((ent = readdir(dir)) != NULL) {
        printf("%s\n", ent->d_name);
    }

    closedir(dir);
    return 0;
}
```

En este ejemplo, se abre el directorio actual utilizando la función `opendir()`. A continuación, se imprime el nombre de todos los archivos y directorios en el directorio utilizando la función `readdir()`. Finalmente, se cierra el directorio utilizando la función `closedir()`.

strerror()

La función `strerror` en C devuelve una cadena de caracteres que describe el último error producido por una función del sistema.

La firma de la función es la siguiente:

```
#include <string.h>

char *strerror(int errnum);
```

El parámetro `errnum` es el código de error a describir. Si se proporciona un valor de -1, la función utilizará el valor de la variable `errno` para describir el error.

Aquí hay un ejemplo que ilustra el uso de la función `strerror` para describir el último error producido por una función del sistema:

```

#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(void) {
    FILE *fp = fopen("no_file.txt", "r");

    if (fp == NULL) {
        printf("Error al abrir el archivo: %s\n", strerror(errno));
    } else {
        printf("El archivo se ha abierto correctamente.\n");
        fclose(fp);
    }

    return 0;
}

```

En este ejemplo, se intenta abrir un archivo que no existe y se comprueba si la operación fue exitosa. Si la operación falla, se utiliza la función `strerror` para imprimir un mensaje de error descriptivo. La variable `errno` se establece automáticamente por la función que falla, en este caso, `fopen`.

perror()

La función `perror` en C se utiliza para imprimir un mensaje de error en el flujo de salida de errores estándar `stderr`. Este mensaje está compuesto por el mensaje de error correspondiente al valor de `errno` actual y una cadena adicional especificada por el usuario.

La sintaxis de la función `perror` es la siguiente:

```
void perror(const char *s);
```

donde `s` es una cadena de caracteres que se imprimirá antes del mensaje de error.

Un ejemplo de uso de la función `perror` podría ser el siguiente:

```

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

int main(void) {
    FILE *file;

    file = fopen("no_existe.txt", "r");
    if (file == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

En este ejemplo, se intenta abrir un archivo que no existe utilizando la función `fopen`. Como resultado, la variable `errno` se establece en un valor que indica el error ocurrido. La función `perror` se utiliza para imprimir un mensaje de error que indica el tipo de error que ocurrió al abrir el archivo, junto con el mensaje "fopen" que se especifica en el argumento de la función. Luego, el programa termina con una llamada a la función `exit` indicando que la operación no tuvo éxito.

isatty()

La función `isatty` en C es utilizada para determinar si un descriptor de archivo se refiere a un terminal.

Su prototipo es el siguiente:

```
int isatty(int fd);
```

donde `fd` es el descriptor de archivo que se va a verificar.

La función devuelve 1 si `fd` es un terminal, y 0 si no lo es.

Aquí hay un ejemplo que muestra cómo se puede utilizar `isatty`:

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

int main()
{
    if (isatty(STDOUT_FILENO))
        printf("stdout es un terminal\n");
    else
        printf("stdout no es un terminal\n");

    return 0;
}
```

En este ejemplo, se utiliza `isatty` para determinar si `STDOUT_FILENO` (el descriptor de archivo para la salida estándar) se refiere a un terminal. Si es así, el programa imprimirá "stdout es un terminal"; de lo contrario, imprimirá "stdout no es un terminal".

ttyname()

La función `ttyname` en C se utiliza para obtener el nombre del dispositivo de terminal asociado con un descriptor de archivo. Su prototipo es el siguiente:

```
char *ttyname(int fd);
```

donde `fd` es el descriptor de archivo que se desea consultar. Si `fd` no se refiere a un dispositivo de terminal, `ttyname` devolverá un puntero nulo.

Aquí hay un ejemplo de cómo se podría usar `ttyname`:

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

int main() {
    int fd;
    char *name;

    fd = fileno(stdout); // Obtener el descriptor de archivo de stdout

    name = ttyname(fd); // Obtener el nombre del dispositivo de terminal asociado con stdout

    if (name != NULL) {
        printf("El nombre del dispositivo de terminal es %s\n", name);
    } else {
        perror("No se pudo obtener el nombre del dispositivo de terminal");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

En este ejemplo, se obtiene el descriptor de archivo de `stdout` utilizando la función `fileno`. Luego, se llama a `ttyname` para obtener el nombre del dispositivo de terminal asociado con `stdout`. Si se pudo obtener el nombre, se imprimirá en la pantalla. Si `ttyname` devolvió un puntero nulo, se imprimirá un mensaje de error utilizando la función `perror`.

ttyslot()

La función `ttyslot()` en C devuelve el número de la entrada en la tabla `termcap` que corresponde a la terminal asociada al descriptor de archivo dado. El número de entrada se puede usar para buscar información adicional sobre la terminal en la tabla `termcap`.

La función tiene la siguiente declaración:

```
int ttyslot(void);
```

El valor de retorno es el número de entrada de la tabla `termcap` si se encuentra la entrada correspondiente, y 0 si no se encuentra.

Aquí hay un ejemplo que muestra cómo usar la función `ttyslot()`:

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

int main() {
```

```

int slot = ttyslot();
if (slot != 0) {
    char *term = getenv("TERM");
    if (term == NULL) {
        perror("getenv");
        return 1;
    }
    char *cap = tgetstr("co", &term);
    printf("La terminal %s tiene %d columnas.\n", term, tgetnum("co"));
} else {
    printf("No se pudo determinar la terminal.\n");
}
return 0;
}

```

En este ejemplo, se llama a `ttyslot()` para obtener el número de entrada en la tabla `termcap` para la terminal actual. Luego, se obtiene el nombre de la terminal actual de la variable de entorno `TERM`. Se llama a `tgetstr()` para obtener la cadena de capacidad para la capacidad de la columna (`co`) y `tgetnum()` para obtener el número de columnas que tiene la terminal. Finalmente, se imprime el nombre de la terminal y el número de columnas.

ioctl()

La función `ioctl()` en C se utiliza para realizar una operación de control en un archivo o dispositivo abierto. El propósito principal de la función es cambiar las características del dispositivo y obtener información sobre el mismo. El nombre "ioctl" significa "Entrada / salida controlada".

La función tiene la siguiente declaración:

```
int ioctl(int fd, unsigned long request, ...);
```

donde `fd` es el descriptor de archivo o dispositivo, `request` es un código que indica la operación de control a realizar y `...` son argumentos adicionales, dependiendo de la operación de control.

Un ejemplo de uso de la función `ioctl()` es para obtener el número de columnas en la pantalla del terminal:

```

#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>

int main() {
    struct winsize w;
    ioctl( STDOUT_FILENO, TIOCGWINSZ, &w );
    printf("Número de columnas: %d\n", w.ws_col );
    return 0;
}

```

En este ejemplo, se utiliza la estructura `winsize` para obtener información sobre el tamaño de la ventana del terminal. Luego, se llama a la función `ioctl()` con el descriptor de archivo `STDOUT_FILENO`, el código de operación `TIOCGWINSZ` y un puntero a la estructura `winsize` para almacenar la información de salida. Finalmente, se imprime el número de columnas de la ventana en la pantalla.

getenv()

La función `getenv` en C se utiliza para obtener el valor de una variable de entorno. Una variable de entorno es una variable de sistema que contiene información específica del usuario y del sistema operativo, como la ruta de acceso de un archivo o el nombre de usuario actual.

La sintaxis de la función `getenv` es la siguiente:

```
char *getenv(const char *name);
```

Donde `name` es el nombre de la variable de entorno que se desea obtener.

La función devuelve un puntero a una cadena de caracteres que contiene el valor de la variable de entorno, o `NULL` si la variable no está definida.

Aquí hay un ejemplo que utiliza `getenv` para obtener el valor de la variable de entorno `HOME` y mostrarlo por pantalla:

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

int main() {
    char *home_dir = getenv("HOME");
    if (home_dir == NULL) {
        printf("La variable de entorno HOME no está definida.\n");
        return 1;
    } else {
        printf("El directorio HOME es: %s\n", home_dir);
        return 0;
    }
}
```

En este ejemplo, se utiliza `getenv` para obtener el valor de la variable de entorno `HOME`. Si la variable no está definida, se imprime un mensaje de error. Si la variable está definida, se imprime su valor.

tcsetattr()

La función `tcsetattr` en C es una función que establece los atributos del terminal asociado con el descriptor de archivo dado. La función toma tres argumentos: el descriptor de archivo del terminal, una bandera que indica cuándo aplicar los cambios, y una estructura que especifica los atributos.

La estructura `termios` se utiliza para especificar los atributos del terminal y se define en la biblioteca `<termios.h>`. La estructura contiene varios miembros que se utilizan para configurar el comportamiento del terminal, como la velocidad de transmisión, el modo de entrada, el modo de salida, etc.

Aquí hay un ejemplo de cómo usar `tcsetattr` en C para establecer los atributos del terminal:

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

int main() {
    struct termios attr;

    // obtener los atributos actuales del terminal
    if (tcgetattr(STDIN_FILENO, &attr) != 0) {
        perror("Error en tcgetattr");
        exit(EXIT_FAILURE);
    }

    // establecer nuevos atributos
    attr.c_lflag &= ~(ICANON | ECHO); // desactivar el modo canónico y eco
    attr.c_cc[VMIN] = 1; // establecer el número mínimo de caracteres
    attr.c_cc[VTIME] = 0; // establecer el tiempo de espera

    // aplicar los cambios
    if (tcsetattr(STDIN_FILENO, TCSANOW, &attr) != 0) {
        perror("Error en tcsetattr");
        exit(EXIT_FAILURE);
    }

    printf("Escriba algo: ");

    char buffer[256];
    int n = read(STDIN_FILENO, buffer, 256);

    printf("\nUsted escribió: %s\n", buffer);

    // restaurar los atributos originales
    if (tcsetattr(STDIN_FILENO, TCSANOW, &attr) != 0) {
        perror("Error en tcsetattr");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Este programa obtiene los atributos actuales del terminal con `tcgetattr`, luego establece nuevos atributos con `attr.c_lflag &= ~(ICANON | ECHO)` y `attr.c_cc[VMIN] = 1` para desactivar el modo canónico y eco, y establecer el número mínimo de caracteres. Luego, aplica los cambios con `tcsetattr` usando la bandera `TCSANOW`. El programa lee la entrada del usuario usando `read` y finalmente restaura los atributos originales del terminal usando `tcsetattr`.

tcgetattr()

La función `tcgetattr()` es una función de la biblioteca estándar de C que se utiliza para obtener los parámetros de configuración de un terminal. Esta función toma dos argumentos: el primero es un descriptor de archivo que se refiere al terminal del que se desean obtener los parámetros de configuración, y el segundo es un puntero a una estructura `termios` en la que se almacenarán los parámetros.

Aquí hay un ejemplo de cómo usar la función `tcgetattr()` para obtener los parámetros de configuración del terminal estándar y luego imprimir algunos de ellos en la pantalla:

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

int main() {
    struct termios tty_attr;

    if (tcgetattr(STDIN_FILENO, &tty_attr) == -1) {
        perror("Error getting tty attributes");
        return 1;
    }

    printf("Terminal speed is %d\n", cfgetispeed(&tty_attr));
    printf("Echo is %s\n", (tty_attr.c_lflag & ECHO) ? "on" : "off");

    return 0;
}
```

En este ejemplo, se llama a `tcgetattr()` para obtener los parámetros de configuración del terminal estándar. Si la llamada a `tcgetattr()` falla, se imprime un mensaje de error con `perror()` y el programa sale con un código de error. Si la llamada a `tcgetattr()` tiene éxito, algunos de los parámetros se imprimen en la pantalla con `printf()`. En este caso, la velocidad del terminal se obtiene con `cfgetispeed()` y se imprime junto con el estado del eco (encendido o apagado).

tgetent()

La función `tgetent()` es una función de la biblioteca `curses` en C, que se utiliza para obtener la entrada de la base de datos terminfo. La base de datos terminfo es una base de datos que contiene información sobre los diversos tipos de terminales que existen, y se utiliza para determinar las capacidades del terminal actual en uso.

La función `tgetent()` toma un puntero a un buffer de caracteres que contiene el nombre de la base de datos terminfo a utilizar, y devuelve un valor entero que indica si la entrada de la base de datos se ha leído correctamente o no.

Aquí hay un ejemplo de cómo se puede utilizar la función `tgetent()`:

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

int main() {
```

```

char term_buffer[1024];
int result = tgetent(term_buffer, "xterm");

if (result == 1) {
    printf("Entrada de terminfo leída correctamente.\n");
} else if (result == 0) {
    printf("No se encontró la entrada de terminfo para el terminal especificado.\n");
} else {
    printf("Error al leer la entrada de terminfo.\n");
}

return 0;
}

```

En este ejemplo, se utiliza la función `tgetent()` para leer la entrada de terminfo para el terminal "xterm". Si la función devuelve un valor de 1, esto significa que la entrada de terminfo se ha leído correctamente y se puede utilizar para determinar las capacidades del terminal. Si devuelve un valor de 0, significa que no se encontró la entrada de terminfo para el terminal especificado, y si devuelve un valor negativo, indica que se produjo un error al leer la entrada de terminfo.

tgetflag()

La función `tgetflag` es parte de la biblioteca `curses.h` en C y se utiliza para obtener el valor de una bandera de capacidad en una entrada terminfo. La función devuelve el valor de la capacidad indicada, si está presente en la entrada terminfo y si tiene un valor definido.

Aquí hay un ejemplo de uso de `tgetflag`:

```

#include <curses.h>
#include <term.h>
#include <stdio.h>

int main() {
    char *termtype = getenv("TERM");
    int cursor;

    if (tgetent(NULL, termtype) < 0) {
        fprintf(stderr, "Cannot access terminal database\n");
        return 1;
    }

    if ((cursor = tgetflag("cr")) == -1) {
        fprintf(stderr, "Terminal entry does not have cursor move capability\n");
        return 1;
    }

    if (cursor) {
        printf("Terminal cursor move capability present\n");
    } else {
        printf("Terminal cursor move capability not present\n");
    }

    return 0;
}

```

Este programa obtiene el valor de la variable de entorno `TERM`, que indica el tipo de terminal que se está utilizando. Luego, utiliza la función `tgetent` para obtener la entrada terminfo correspondiente a ese tipo de terminal. A continuación, utiliza la función `tgetflag` para comprobar si la capacidad de mover el cursor (`cr`) está presente en la entrada terminfo. Si la capacidad está presente, el programa imprimirá "Terminal cursor move capability present", de lo contrario imprimirá "Terminal cursor move capability not present".

tgetnum()

La función `tgetnum` es una función de la biblioteca de C `curses.h` que se utiliza para obtener el valor numérico de una capacidad de terminal. La capacidad se especifica por su nombre y se devuelve su valor entero correspondiente. Si la capacidad no está disponible en la terminal, la función devuelve -1.

Aquí hay un ejemplo:

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

int main() {
    int num_cols;
    char *termtype = getenv("TERM");
    if (termtype == NULL) {
        printf("La variable de entorno TERM no está definida.\n");
        return 1;
    }
    if (tgetent(NULL, termtype) != 1) {
        printf("No se pudo acceder a la base de datos terminfo.\n");
        return 1;
    }
    num_cols = tgetnum("cols");
    if (num_cols == -1) {
        printf("La capacidad de terminal 'cols' no está disponible.\n");
        return 1;
    }
    printf("El ancho de la terminal es %d.\n", num_cols);
    return 0;
}
```

Este programa obtiene el nombre de la terminal de la variable de entorno `TERM`, luego usa `tgetent` para cargar la base de datos terminfo. A continuación, utiliza la función `tgetnum` para obtener el número de columnas de la terminal y lo muestra en la salida. Si la capacidad no está disponible, se imprime un mensaje de error.

tgetstr()

La función `tgetstr` en C es parte de la biblioteca `curses` y se utiliza para obtener una cadena de escape para una determinada capacidad del terminal.

La sintaxis de la función es la siguiente:

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

Donde `id` es el identificador de la capacidad del terminal que se desea obtener, y `area` es un puntero a un área de memoria que se utilizará para almacenar la cadena de escape.

Un ejemplo de uso de `tgetstr` sería el siguiente:

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

int main() {
    char *clear_screen;

    // Inicializamos la biblioteca curses
    initscr();

    // Obtenemos la cadena de escape para limpiar la pantalla
    clear_screen = tgetstr("clear", NULL);

    // Enviamos la cadena de escape al terminal
    tputs(clear_screen, 1, putchar);

    // Terminamos la biblioteca curses
    endwin();

    return 0;
}
```

En este ejemplo, inicializamos la biblioteca `curses` con la función `initscr()`. Luego, utilizamos la función `tgetstr` para obtener la cadena de escape correspondiente a la capacidad "clear", que es utilizada para limpiar la pantalla. Finalmente, enviamos la cadena de escape al terminal utilizando la función `tputs` y terminamos la biblioteca `curses` con la función `endwin()`.

tgoto()

La función `tgoto` se utiliza en la biblioteca C `curses.h` para generar una secuencia de control que lleva el cursor a una posición específica en la pantalla. Su prototipo es el siguiente:

```
char *tgoto(const char *cap, int col, int row);
```

Donde `cap` es la capacidad de la terminal que se está utilizando (por ejemplo, obtenida a través de `tgetent`), `col` y `row` son las coordenadas en las que se desea colocar el cursor.

La función devuelve una cadena que contiene la secuencia de control generada para colocar el cursor en la posición deseada.

Aquí hay un ejemplo de cómo usar `tgoto` para mover el cursor a la posición (10, 5):

```

#include <curses.h>

int main() {
    // inicializar curses
    initscr();

    // obtener la capacidad de la terminal
    char termcap[1024];
    char *termtype = getenv("TERM");
    tgetent(termcap, termtype);

    // obtener la secuencia de control para mover el cursor
    char *cursor_move = tgoto(tgetstr("cm", NULL), 10, 5);

    // imprimir la secuencia de control
    printf("Moviendo cursor a (10, 5)\n");
    printf("%s\n", cursor_move);

    // finalizar curses
    endwin();

    return 0;
}

```

Este ejemplo obtiene la capacidad de la terminal, luego usa `tgetstr` para obtener la secuencia de control para mover el cursor (`cm`). Luego, usa `tgoto` para generar la secuencia de control para mover el cursor a la posición (10, 5). Finalmente, se imprime la secuencia de control generada.

Tenga en cuenta que este ejemplo solo imprime la secuencia de control generada en la consola. Para que se muestre correctamente en la pantalla, se deben usar las funciones de la biblioteca `curses.h` para controlar la pantalla.

tputs()

La función `tputs()` en C es una función que es utilizada para enviar una cadena de caracteres a la terminal en formato de control de caracteres. Esta función es útil para manipular la salida de la terminal.

La sintaxis de la función `tputs()` es la siguiente:

```
int tpsets(const char *str, int affcnt, int (*putc)(int));
```

Donde:

- `str`: es un puntero a la cadena de caracteres a imprimir.
- `affcnt`: es el número de líneas afectadas por la cadena de caracteres. Si este valor es negativo, la cadena afecta a todas las líneas.
- `putc`: es un puntero a una función que es utilizada para imprimir los caracteres.

La función `tputs()` devuelve un valor negativo si hay un error al imprimir la cadena, y un valor positivo en caso contrario.

Aquí hay un ejemplo de cómo usar la función `tputs()` para imprimir el texto "Hello, world!" en la terminal:

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

int main() {
    char *text = "Hello, world!";
    char *clear = tgetstr("cl", NULL);
    char *move = tgetstr("cm", NULL);

    tputs(clear, 1, putchar);
    tputs(move, 1, putchar);
    tputs(text, 1, putchar);

    return 0;
}
```

En este ejemplo, primero se obtienen las cadenas de caracteres para borrar la pantalla y mover el cursor a la posición (0, 0) utilizando las funciones `tgetstr()`. Luego, se llama a la función `tputs()` para imprimir la cadena "Hello, world!" en la posición (0, 0) utilizando la función `putchar()`.

ENLACES

<https://baulderasec.wordpress.com/programando-2/programacion-con-linux/5-terminales/detectar-pulsaciones-de-las-teclas/>

<https://baulderasec.wordpress.com/programando-2/programacion-con-linux/5-terminales/detectar-pulsaciones-de-las-teclas/>

PASOS

1- Comprobar cadena válida

Función separar

```
char*** separateArray(char **array) {

    int i = 0;
    int j = 0;
    int dim = 0;
    int k = 0;
    char*** matrix;

    while (array[dim] != NULL)
    {
        // printf("%s\n", array[dim]);
        ++dim;
    }

    matrix = malloc(dim * sizeof(char**));
    if (!matrix)
    {
        perror("reservar");
        return (NULL);
    }

    while (i < dim)
    {
        matrix[i] = malloc(2 * sizeof(char*));
        if (!matrix[i])
        {
            perror("reservar");
            return (NULL);
        }
        ++i;
    }
    // printf("dim %i ", dim);
    i = 0;
    while (i < dim)
    {
        j = 0;
        k = 0;
        while (array[i][k] != '=' && array[i][k] != '\0')
        {
            printf("LLEGA\n");
            matrix[i][0][j] = array[i][k];
            ++j;
            ++k;
        }
        matrix[i][0][j] = '\0';
        printf("LLEGA2\n");
        j = 0;
        while (array[i][k] != '\0')
        {
            matrix[i][1][j] = array[i][k];
            ++j;
            ++k;
        }
        matrix[i][1][j] = '\0';
        ++i;
    }
}
```

```
for (int q = 0; q < dim; ++q)
{
    printf("cadena 1: %s, cadena 2: %s\n", matrix[q][0], matrix[q][1]);
}

return (matrix);

}
```