

Unidad 1

Manipulación de archivos

Jesús Torres

Vamos a aprender a manipular archivos utilizando las llamadas al sistema del sistema operativo mientras desarrollamos un programa que copia archivos entre diferentes rutas en el sistema de archivos.

Contenidos

1. Actividad 1	3
1.1. Objetivo	3
1.2. Línea de comandos y funcionamiento básico	3
1.3. Manejo de errores	4
1.4. Implementación	4
1.4.1. Comprobar los argumentos de la línea de comandos	4
1.4.2. Comprobar si DESTINO es un directorio	5
1.4.3. Obtener el nombre del archivo desde una ruta	5
1.4.4. Copiar el archivo	6
1.5. Comprobación	6
1.5.1. Casos de prueba adicionales	8
A. Operaciones con archivos con la API POSIX	9
A.1. Descriptores de archivo	9
A.1.1. Identificación de recursos	9
A.1.2. Identificadores según el sistema operativo	10
A.1.3. Descriptores estándar	11
A.1.4. Descriptores y otros recursos del sistema	11
A.2. Gestión de errores en la API POSIX	12
A.2.1. Convención de valores de retorno	12
A.2.2. Valores de <code>errno</code>	13
A.2.3. Lectura inmediata de <code>errno</code>	13
A.3. Abrir archivos	15
A.3.1. Opciones de apertura	15
A.3.2. Permisos de los nuevos archivos	16
A.3.3. Permisos y máscara umask	16
A.4. Cerrar descriptores de archivo	17

A.5. Leer datos de archivos	17
A.5.1. Valor de retorno	17
A.5.2. Consideraciones importantes	18
A.6. Escribir datos en archivos	19
A.6.1. Valor de retorno	19
A.7. Acceder a los atributos de un archivo	20
A.7.1. Comprobar si dos archivos son el mismo	21
A.7.2. Acceso a los permisos y al tipo de archivo	21
B. Gestión de errores en C++	22
B.1. Propagación de errores con <code>std::expected</code>	23
B.1.1. Funciones con valor de retorno	23
B.1.2. Funciones sin valor de retorno	24
B.2. Información de error enriquecida	25
B.3. Forzar la comprobación de errores con <code>[[nodiscard]]</code>	27

1. Actividad 1

1.1. Objetivo

Vamos a crear un programa llamado `copy` que copia el contenido de un archivo en otra ubicación. El programa debe cumplir con los siguientes requisitos:

- La ruta del archivo de origen y la ruta del archivo de destino se indicarán como argumentos en la línea de comandos al ejecutar el programa.
- Podrá copiar archivos de cualquier tamaño, incluyendo archivos muy grandes –aunque no quepan completamente en memoria– o muy pequeños.
- Para manipular los archivos usará, exclusivamente, las llamadas al sistema `open()`, `read()`, `write()`, `stat()`, `close()` y similares, tal y como se explica en el Apéndice A.
- Antes de finalizar, el programa cerrará todos los descriptores de archivo abiertos, liberará correctamente cualquier otro recurso reservado y finalizará con el código `EXIT_SUCCESS` (0).

! Importante

Para manipular los archivos no se pueden usar funciones de más alto nivel como `std::ifstream`, `std::ofstream`, `std::fstream`, `std::FILE`, `fopen()`, `fread()`, `fwrite()`, `fclose()` o similares.

1.2. Línea de comandos y funcionamiento básico

El programa `copy` acepta los siguientes argumentos de línea de comandos:

```
copy ORIGEN DESTINO
```

Cuando se ejecuta con estos dos argumentos:

- El programa debe copiar el contenido del archivo en la ruta `ORIGEN` a la ruta del archivo `DESTINO`.
- Si `DESTINO` ya existe y es un archivo, debe sobrescribirse.
- Si `DESTINO` ya existe y es un directorio, el programa debe copiar `ORIGEN` dentro de ese directorio.
- Si `DESTINO` no existe, debe crearse el archivo para hacer la copia.

Si no se ejecuta indicando estos dos argumentos, el programa debe mostrar un mensaje de error indicando cómo debe usarse el programa y terminar con un código de salida de error:

```
$ copy /tmp/testfile.txt
copy: se deben indicar los archivos ORIGEN y DESTINO
```

1.3. Manejo de errores

La mayor parte de las funciones que sirven servicios y recursos del sistema pueden fallar por diversos motivos, por lo que debemos comprobar y tratar cualquier condición de error posible.

En todas las situaciones de error:

- El programa debe mostrar un mensaje de error por la **salida de error**.
- El programa debe terminar con un código de salida distinto de `EXIT_SUCCESS (0)`.
- Las funciones que implementemos deben retornar la condición de error a quien las llamó, en lugar de mostrar ellas el mensaje, pues de esta forma son más reutilizables. Como regla general, los mensajes de error deben ser enviados a imprimir en `main()` o lo más cerca posible de `main()`, antes de terminar el programa.
- La terminación debe ocurrir retornando desde `main()` con `return`, nunca usando `std::exit()`, `exit()` ni funciones similares, para seguir las buenas prácticas recomendadas en C++.

1.4. Implementación

El flujo de ejecución del programa es muy sencillo:

```
result = check_args(argc, argv)
if has_error(result):
    raise error("Argumentos inválidos")

if is_directory(destino):
    destino = destino + "/" + get_filename(origen)

result = copy_file(origen, destino)
if has_error(result):
    raise error("Error al copiar archivo")
```

En este pseudocódigo se indican las funciones que se deben implementar y cuándo se deben llamar. Cuando se indica `raise error("...")`, significa que se aplicará el manejo de errores descrito en la Sección 1.3.

Vamos a ver cada uno de estos pasos y las funciones indicadas con más detalle.

1.4.1. Comprobar los argumentos de la línea de comandos

Para comprobar las condiciones sobre el número de argumentos indicados en la línea de comandos, implementa la siguiente función:

```
bool check_args(int argc, char* argv[]);
```

A la que se le pasan los parámetros `argc` y `argv` recibidos en `main()`. Esta función debe devolver `true` si los argumentos son correctos y `false` en caso contrario.

Para que los argumentos sean correctos, deben cumplirse las siguientes condiciones:

- Deben indicarse exactamente dos argumentos (además del nombre del programa en `argv[0]`).
- Los archivos de `ORIGEN` y `DESTINO` no deben ser el mismo archivo. Para comprobar si dos archivos son el mismo, se usará la técnica descrita en la Sección A.7.1.

Como esta función es muy específica de este programa, puede mostrar directamente el mensaje de error adecuado en caso de que los argumentos no sean correctos.

1.4.2. Comprobar si DESTINO es un directorio

Para comprobar si el archivo indicado en `DESTINO` es un directorio, implementa la siguiente función:

```
bool is_directory(const std::string& path);
```

Esta función debe devolver `true` si el archivo en la ruta `path` es un directorio y `false` en caso contrario. Para comprobarlo se puede usar la función `stat()` descrita en la Sección A.7.

1.4.3. Obtener el nombre del archivo desde una ruta

Necesitamos obtener el nombre del archivo desde una ruta completa para construir la ruta de destino cuando `DESTINO` es un directorio. Es decir, si `ORIGEN` es `/home/user/document.txt` y `DESTINO` es `/tmp/`, debemos construir la ruta `/tmp/document.txt` para copiar el archivo dentro del directorio de destino.

Para obtener el nombre del archivo desde una ruta completa, implementa la siguiente función:

```
std::string get_filename(const std::string& path);
```

Puedes hacerlo buscando la última ocurrencia de `/` en la ruta y retornando la subcadena posterior, o usando la función `basename()`.

Advertencia

La función `basename()` de la biblioteca estándar de C modifica el argumento que se le pasa. Es recomendable utilizar `std::string::copy()` para trabajar con una copia del

string path.

1.4.4. Copiar el archivo

Para copiar el archivo de ORIGEN a DESTINO, implementa la siguiente función:

```
std::expected<void, std::system_error> copy_file(
    const std::string& src_path, const std::string& dest_path,
    mode_t dst_perms=0);
```

Esta función debe abrir el archivo en `src_path` para lectura y el archivo en `dest_path` para escritura –creándolo si no existe con los permisos `dst_perms` o sobrescribiéndolo si ya existe– y debe copiar todo el contenido del archivo de origen al archivo de destino. Para ello, debe usar un buffer de tamaño fijo para leer y escribir los datos en bloques de 64 KiB (65 536 bytes) hasta que todo el archivo haya sido copiado, tal y como se ilustra en la Figura 1.

El uso de las funciones `open()`, `read()`, `write()` y `close()` está explicado en el Apéndice A.

La función debe retornar `std::expected<void, std::system_error>` para indicar si la operación de copia fue exitosa o si ocurrió algún error. En caso de error, debe devolver un objeto `std::system_error` con el código de error de `errno`. El uso de `std::expected` está descrito en el Apéndice B.

1.5. Comprobación

Para comprobar que `copy` funciona correctamente, podemos crear archivos con datos aleatorios generados desde el dispositivo `/dev/urandom`. Por ejemplo, usando el comando `dd` se puede crear un archivo de nombre `testfile.dat` con 16 MiB de datos aleatorios:

```
$ dd if=/dev/urandom of=testfile.dat bs=1M count=16 iflag=fullblock
```

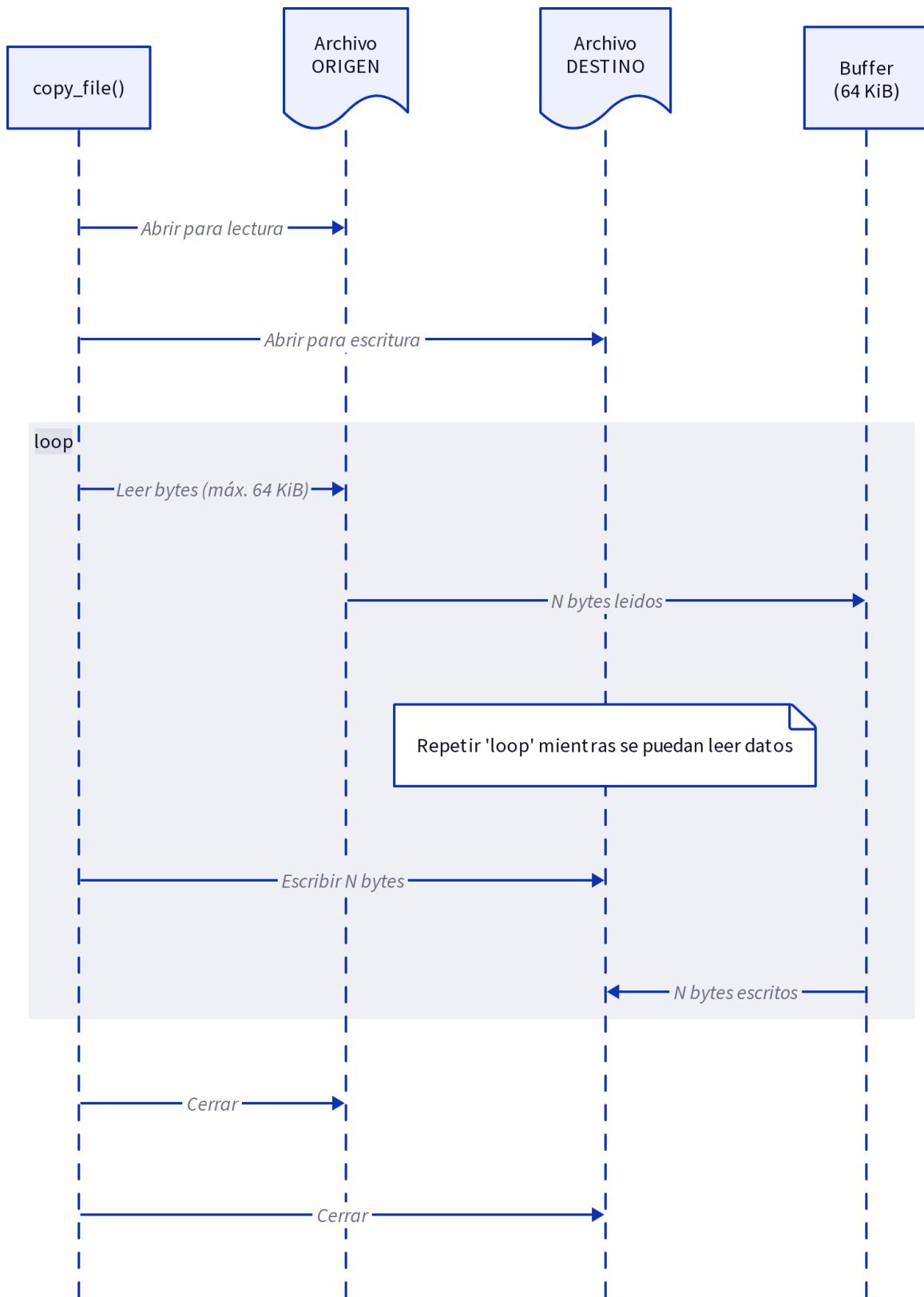
Para comprobar que la copia funciona, primero usamos nuestro programa `copy` para copiar `testfile.dat` a `testfile2.dat`:

```
$ copy testfile.dat testfile2.dat
```

Y luego el comando `cmp` nos dirá si ambos archivos son iguales o diferentes:

```
$ cmp testfile.dat testfile2.dat
```

Si los archivos son iguales, `cmp` no mostrará nada y terminará con código de salida 0.

Figura 1: Diagrama de secuencia del funcionamiento de la función `copy_file()`.

! Importante

Si los archivos tienen diferente tamaño, es porque seguramente el programa `copy` no ha copiado todo el contenido del archivo origen porque no ha manejado correctamente la situación en la que `read()` lee menos bytes de los solicitados (ver la Sección A.5.2).

1.5.1. Casos de prueba adicionales

Adicionalmente, es importante probar los siguientes escenarios:

- Intentar copiar un archivo sin indicar el destino:

```
$ copy testfile.dat  
copy: se deben indicar los archivos ORIGEN y DESTINO
```

- Intentar copiar un archivo a sí mismo:

```
$ copy testfile.dat testfile.dat  
copy: el archivo ORIGEN y DESTINO no pueden ser el mismo
```

- Intentar copiar un archivo que no existe:

```
$ copy noexiste.dat destino.dat  
copy: error al abrir el archivo de origen: No such file or directory
```

- Intentar copiar un archivo a un directorio sin permisos de escritura:

```
$ copy testfile.dat /  
copy: error al abrir el archivo de destino: Permission denied
```

- Intentar copiar un archivo sin permisos de lectura:

```
$ chmod 000 testfile.dat  
$ copy testfile.dat destino.dat  
copy: error al leer el archivo de origen: Permission denied
```

A. Operaciones con archivos con la API POSIX

Los archivos son uno de los mecanismos fundamentales para el almacenamiento persistente de información en los sistemas operativos. La API POSIX define un conjunto de funciones de bajo nivel que permiten a los programas interactuar con el sistema de archivos de manera estandarizada: abrir, leer, escribir y cerrar archivos, así como consultar y modificar sus atributos. En esta sección exploraremos estas funciones básicas y aprenderemos cómo los procesos solicitan recursos mediante descriptores de archivo, cómo gestionar los errores y cómo desarrollar programas robustos.

A.1. Descriptores de archivo

Antes de poder realizar cualquier operación sobre un archivo –como leer su contenido o escribir datos en él–, es necesario comprender el mecanismo mediante el cual los procesos identifican y acceden a los recursos del sistema operativo.

A.1.1. Identificación de recursos

Los procesos necesitan solicitar recursos al sistema operativo para realizar su trabajo: acceder a archivos, comunicarse con otros procesos, utilizar dispositivos de hardware, etc. Sin embargo, por razones de seguridad y gestión eficiente, **los procesos no pueden acceder directamente a la mayoría de estos recursos**.

En su lugar, el sistema operativo actúa como intermediario:

1. Cuando un proceso solicita un recurso (por ejemplo, abrir un archivo), el sistema operativo reserva y prepara ese recurso.
2. En lugar de dar acceso directo al recurso, el sistema operativo devuelve al proceso un **identificador** único que representa ese recurso.
3. Cada vez que el proceso quiere realizar una operación sobre el recurso (leer, escribir, cerrar, etc.), debe proporcionar ese identificador al sistema operativo para indicar sobre qué recurso desea actuar.
4. El sistema operativo verifica que el proceso tiene permisos para realizar la operación y la ejecuta en su nombre.
5. Cuando el proceso termina de usar el recurso, debe notificarlo al sistema operativo –generalmente cerrando el recurso mediante su identificador– para que pueda liberarlo.

i Nota

Este modelo sería similar a si en programación orientada a objetos, en lugar de llamar directamente a los métodos de un objeto, tuviéramos que pedirle a un **intermediario** que lo haga por nosotros. En ese caso, el intermediario necesitaría saber qué objeto queremos usar (el identificador) y qué método queremos invocar (la operación).

A.1.2. Identificadores según el sistema operativo

Diferentes sistemas operativos usan diferentes tipos de identificadores para representar recursos:

En sistemas POSIX (Linux, macOS, BSD, etc.) Los identificadores se llaman **descriptores de archivo** y son números enteros no negativos (**int**). Por ejemplo: 0, 1, 2, 3, 4, etc.

En sistemas Windows Los identificadores se llaman **handles** y son valores de tipo **HANDLE**, que internamente son punteros opacos. El proceso no puede acceder a lo que apunta el **HANDLE**; solo puede pasarlo a las funciones de la Windows API para identificar el recurso.

Pero en ambos casos, el principio es el mismo:

1. El proceso solicita un recurso y recibe un identificador.
2. El proceso usa ese identificador en todas las operaciones sobre el recurso.
3. Cuando termina, el proceso libera el recurso mediante el identificador.

Por ejemplo, en POSIX:

```
int fd = open("archivo.txt", O_RDONLY);          (1)
read(fd, buffer, tamaño);
close(fd);                                     (2)
                                              (3)
```

- (1) Obtener un descriptor para identificar el archivo abierto (**fd**).
- (2) Usar el descriptor para leer datos del archivo.
- (3) Cerrar el descriptor para liberar el recurso.

Y en Windows API:

```
HANDLE hFile = CreateFile("archivo.txt", ...);      (1)
ReadFile(hFile, buffer, tamaño, ...);                (2)
CloseHandle(hFile);                                (3)
```

- (1) Obtener un *handle* para identificar el archivo abierto (**hFile**).
- (2) Usar el handle para leer datos del archivo.
- (3) Cerrar el handle para liberar el recurso.

i Nota

En esta asignatura nos centraremos en la API POSIX, por lo que utilizaremos **descriptores de archivo**.

A.1.3. Descriptores estándar

Todo proceso se inicia con tres descriptores que están siempre disponibles y no es necesario abrirlos explícitamente:

- 0 (**STDIN_FILENO**) Entrada estándar (*standard input*). Por defecto, conectado al teclado.
Se puede leer de él usando `read()`.
- 1 (**STDOUT_FILENO**) Salida estándar (*standard output*). Por defecto, conectado a la terminal. Se puede escribir en él usando `write()`.
- 2 (**STDERR_FILENO**) Salida de error estándar (*standard error*). Por defecto, también conectado a la terminal. Se usa para escribir mensajes de error y diagnóstico.

i Nota

En C y C++, las macros `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO` están definidas en el archivo de cabecera `<unistd.h>`.

Estos descriptores permiten al proceso interactuar con el entorno (terminal, consola, etc.) y se pueden usar como cualquier otro descriptor. Por ejemplo:

```
int main()
{
    const char* mensaje = "Hola, mundo!\n";
    write(STDOUT_FILENO, mensaje, strlen(mensaje));           ①
    return 0;
}
```

- ① Escribe el mensaje en la salida estándar (terminal).

A.1.4. Descriptores y otros recursos del sistema

Aunque los ejemplos anteriores muestran operaciones con archivos, el mismo concepto de identificadores se aplica a otros recursos del sistema operativo, como:

- **Tuberías (pipes)**: Creadas con `pipe()`.
- **Sockets**: Creados con `socket()`.
- **Dispositivos**: Archivos especiales en `/dev/` que representan hardware.

Todos estos recursos se manipulan usando descriptores de archivo y las mismas funciones como `read()`, `write()` y `close()`, aunque el sistema operativo también ofrece otras funciones para realizar operaciones específicas según el tipo de recurso.

A.2. Gestión de errores en la API POSIX

Las funciones del sistema operativo pueden fallar por múltiples razones: permisos insuficientes, archivos inexistentes, falta de espacio en disco, interrupciones del sistema o memoria insuficiente, entre otras. Por ello, el manejo correcto de errores es fundamental para desarrollar programas robustos y fiables.

A.2.1. Convención de valores de retorno

Muchas funciones de las API de los sistemas operativos siguen convenciones similares para indicar éxito o error:

1. Terminan devolviendo un valor específico para indicar que ocurrió un error, como: `false`, `NULL`, `-1`, etc.
2. Proveen un mecanismo para obtener información detallada sobre el error ocurrido, como una variable global o una función que devuelve un mensaje descriptivo. En los sistemas POSIX, este mecanismo es el valor de `errno`, mientras que en Windows API se usa la función `GetLastError()`. En ambos casos, lo que se obtiene es un código numérico que identifica el tipo de error.

Entre las funciones de más bajo nivel de la API POSIX, la convención más común es que **devuelvan -1 para indicar un error**. Por ejemplo:

```
// Ejemplo de manejo de error al abrir un archivo
int fd = open("archivo.txt", O_RDONLY);
if (fd == -1)
{
    // Ha ocurrido un error al abrir el archivo
    std::println(stderr, "Error ({} ) al abrir el archivo: {}",
        errno,
        std::strerror(errno))(1)(2);
}
```

- (1) `errno` contiene el código numérico del error ocurrido.
 (2) `strerror(errno)` devuelve un mensaje descriptivo del error.

! Importante

Es fundamental verificar el valor de retorno de todas las llamadas al sistema.
 No comprobar errores es una de las causas más comunes de bugs difíciles de diagnosticar y de vulnerabilidades de seguridad.

A.2.2. Valores de errno

errno se define en el archivo de cabecera `<errno.h>` (o `<cerrno>` en C++) y contiene un valor de tipo `int`. Los valores posibles de `errno` son constantes definidas en `<errno.h>`, como:

- ENOENT (*No such file or directory*)** El archivo o directorio especificado no existe.
- EACCES (*Permission denied*)** Permisos insuficientes para realizar la operación.
- EEXIST (*File exists*)** El archivo ya existe (usado con `O_CREAT | O_EXCL`).
- EISDIR (*Is a directory*)** Se intentó abrir un directorio con `O_WRONLY` o `O_RDWR`.
- ENOSPC (*No space left on device*)** No hay espacio disponible en el dispositivo.
- EMFILE (*Too many open files*)** El proceso ha alcanzado su límite de descriptores abiertos.
- EINTR (*Interrupted system call*)** La llamada fue interrumpida por una señal.

Cada función –como `open()` o `read()`– documenta en su página del manual los posibles códigos de error que puede generar.

Por tanto, el patrón estándar para verificar errores en llamadas al sistema POSIX es:

```
int result = read(...);
if (result == -1)    // Verificar si hubo error
{
    // En este punto, errno contiene el código de error

    if (errno == ENOENT)
    {
        // Manejar error de archivo no encontrado
    }
    else if (errno == EACCES)
    {
        // Manejar error de permisos insuficientes
    }
    // ... otros errores específicos

    // Acción genérica en caso de error
    // ...
}

// Continuar con el código normal si no hubo error
```

A.2.3. Lectura inmediata de errno

Un detalle importante es que **resulta crucial leer el valor de `errno` inmediatamente después de detectar un error**, porque:

1. **errno solo se establece cuando una función falla.** Si una función tiene éxito, no modifica `errno`, por lo que este puede contener el valor de un error anterior.
2. **Cualquier llamada posterior a funciones del sistema puede modificar errno,** incluso si esas funciones tienen éxito.

Por ejemplo, este código es **incorrecto**:

```
int fd = open(filename, O_RDONLY);
if (fd == -1)
{
    // Hacer otras operaciones...
    std::println("Archivo: {}", filename); (1)

    // Para cuando llegamos aquí, errno puede haber cambiado
    if (errno == ENOENT) (2)
    {
        std::println(stderr, "El archivo no existe");
    }
}
```

- ① La llamada a `std::println` puede llamar internamente a funciones del sistema que modifiquen `errno`.
- ② Por tanto, al llegar a esta línea, `errno` puede no contener el código de error original de `open()`.

El código correcto debe guardar `errno` inmediatamente:

```
int fd = open(filename, O_RDONLY);
if (fd == -1)
{
    int error_code = errno; (1)

    // Ahora podemos hacer otras operaciones de forma segura
    std::println("Error al abrir archivo: {}", filename);

    if (error_code == ENOENT) (2)
    {
        std::println(stderr, "El archivo no existe");
    }
}
```

- ① Guardar el valor de `errno` en una variable local inmediatamente después de detectar el error.
- ② Usar la copia guardada para verificar el tipo de error.

A.3. Abrir archivos

La función `open()` abre el archivo cuya ruta se pasa como primer argumento (`pathname`) y devuelve, en caso de éxito, un descriptor de archivo (ver la Sección A.1) que puede ser usado posteriormente para leer, escribir o realizar otras operaciones sobre el archivo abierto.

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Si la llamada falla, `open()` retorna `-1` y `errno` contiene el código de error (ver la Sección A.2 para más detalles sobre el manejo de errores). Los posibles códigos de error que puede devolver `open()` están documentados en la [página del manual de `open\(2\)`](#).

A.3.1. Opciones de apertura

El segundo argumento (`flags`) es un conjunto de opciones que indican cómo se va a abrir el archivo. Algunas de estas opciones son:

- `O_RDONLY` abre el archivo para lectura.
- `O_WRONLY` abre el archivo para escritura.
- `O_RDWR`: abre el archivo para lectura y escritura.
- `O_TRUNC` trunca el archivo a 0 bytes al abrirlo.
- `O_APPEND` abre el archivo para escritura y coloca el puntero de escritura al final del archivo. Esto es útil cuando se quieren añadir datos al final del archivo.
- `O_CREAT` crea el archivo si no existe. Si no se indica esta opción, y el archivo no existe, la función `open()` fallará con `ENOENT`.
- `O_EXCL` en combinación con `O_CREAT`, si el archivo ya existe, la llamada a `open()` falla con `EEXIST`

Se pueden combinar varias de estas opciones usando el operador '`|`' –or bit a bit–:

```
int fd = open("ruta/del/archivo", O_RDONLY | O_CREAT);
if (fd == -1)
{
    // Error al abrir el archivo...
}
```

A.3.2. Permisos de los nuevos archivos

El tercer argumento (`mode`) indica los permisos del nuevo archivo, si es que va a ser creado porque no existía previamente. Por tanto, esta opción solo tiene sentido si se pasa la opción `O_CREAT` en `flags` porque, de lo contrario, el archivo abierto ya existe y conserva sus permisos originales.

Si se crea un archivo nuevo y no se indica `mode`, el archivo se creará con permisos 0000, es decir, sin permisos para nadie. Así que lo adecuado es indicar siempre los permisos que se quiere que tenga el archivo, para lo que se puede usar la misma notación octal que se utiliza con el comando `chmod` en la *shell*:

```
int fd = open("ruta/del/archivo", O_RDONLY | O_CREAT, 0666);
if (fd == -1)
{
    // Error al abrir el archivo...
}
```

En el ejemplo anterior, el nuevo archivo hereda del proceso que llama a `open()` el propietario y el grupo. Respecto a los permisos, el deseo del programador es que el archivo tenga permisos de lectura y escritura para el propietario, y permisos de lectura para el grupo y para otros.

A.3.3. Permisos y máscara umask

Los permisos indicados en el argumento `mode` de `open()` son modificados por la máscara `umask` del proceso antes ser aplicados al archivo, de tal forma que los permisos reales del archivo serán `mode & ~umask`.

La `umask` puede ser configurada por cada usuario mediante el comando `umask` de la shell y se hereda por todos los procesos lanzados por ese usuario, lo que permite a los usuarios controlar los permisos que tendrán los archivos que creen los programas que ejecuten. Además, cada proceso puede cambiar su propia `umask` mediante la función `umask()`:

Por ejemplo, para establecer la `umask` a 0022 durante una sesión, se puede usar el siguiente comando en la *shell*:

```
$ umask 0022
```

El comando `umask` también se puede usar para comprobar el valor actual de la máscara:

```
$ umask
0022
```

Si un programa crea un nuevo archivo con `open(..., 0666)`, pero la `umask` heredada por el proceso desde la *shell* del usuario es 0022, el archivo tendrá permisos $0666 \& \sim 0022 = 0644$. Es decir, con esa máscara, por defecto, los nuevos archivos no tendrán permisos de escritura ni para el grupo y ni para otros.

i Nota

No suele tener sentido indicar permisos de ejecución en `mode` –por ejemplo, 0777– a menos que se esté desarrollando un compilador y se sepa que el archivo que se va a crear va a ser ejecutable.

A.4. Cerrar descriptores de archivo

Cuando un archivo ya no es necesario, es importante cerrarlo para liberar los recursos del sistema asociados a ese archivo. Los descriptores de archivo se cierran usando la función `close()`.

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

El motivo de error más común al llamar a `close()` es intentar cerrar un descriptor inválido o que ya ha sido cerrado.

A.5. Leer datos de archivos

La función `read()` permite leer datos de un archivo abierto y avanzar el puntero de lectura.

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

Los argumentos de `read()` son:

- fd** Descriptor del archivo abierto del que se quieren leer datos. Este descriptor se puede haber obtenido previamente con `open()`, `pipe()`, `dup()` o cualquier otra función que genere un descriptor de archivo.
- buf** Puntero a la región de memoria donde se almacenarán los datos leídos. Es responsabilidad del programador reservar la región de memoria y asegurarse de que tiene suficiente espacio para almacenar los datos.
- count** Número máximo de bytes que se desean leer del archivo. La función intentará leer como máximo hasta este número de bytes, pero puede leer menos.

A.5.1. Valor de retorno

Si la llamada tiene éxito, `read()` retorna el número de bytes realmente leídos, que puede ser:

- **Mayor que 0**, si se han leído datos correctamente. El número retornado indica cuántos bytes se han almacenado en `buf`.
- **Igual a 0**, si se ha alcanzado el final del archivo (*end-of-file*), por lo que no hay más datos que leer.

Si la llamada falla, `read()` retorna `-1` y `errno` contiene el código de error (ver la Sección A.2 para más detalles sobre el manejo de errores). Los posibles códigos de error que puede devolver `read()` están documentados en la [página del manual de `read\(2\)`](#).

A.5.2. Consideraciones importantes

La función `read()` no garantiza leer todos los bytes solicitados, incluso cuando el archivo contiene suficientes datos. El número de bytes leídos puede ser menor que `count` por diversos motivos:

- Se ha alcanzado el final del archivo antes de leer `count` bytes.
- La lectura fue interrumpida por una señal.
- En el caso de descriptores de archivos de *sockets* o tuberías, puede haber menos datos disponibles de los solicitados.

Por tanto, **es frecuente tener que llamar a `read()` múltiples veces en un bucle** hasta que se hayan leído todos los datos necesarios o se alcance el final del archivo.

Por ejemplo:

```
void read_data(int fd, std::vector<uint8_t>& buffer, size_t bytes_needed)
{
    ssize_t bytes_read;
    ssize_t total_read = 0;

    buffer.resize(bytes_needed);

    while (total_read < bytes_needed)
    {
        bytes_read = read(fd, buffer.data() + total_read,
                          bytes_needed - total_read);

        if (bytes_read == -1)
        {
            // Error al leer...
            break;
        }
        else if (bytes_read == 0)
        {
            // Final del archivo alcanzado
            break;
        }

        total_read += bytes_read;
    }
}
```

```
    buffer.resize(total_read);  
}
```

(1)

- ① Ajustar el tamaño del buffer al número real de bytes leídos.

A.6. Escribir datos en archivos

La función `write()` permite escribir datos en un archivo abierto y avanzar el puntero de escritura.

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

Los argumentos de `write()` son:

- fd** Descriptor del archivo abierto en el que se quieren escribir datos. Este descriptor se puede haber obtenido previamente con `open()`, `pipe()`, `dup()` o cualquier otra función que genere un descriptor de archivo con permisos de escritura.
- buf** Puntero a la región de memoria que contiene los datos que se desean escribir. Los datos se escribirán en el archivo a partir de la posición actual del puntero de escritura del descriptor.
- count** Número de bytes que se desean escribir en el archivo. La función intentará escribir este número de bytes desde `buf`.

A.6.1. Valor de retorno

Si la llamada tiene éxito, `write()` retorna el número de bytes realmente escritos.

Si la llamada falla, `write` retorna `-1` y `errno` contiene el código de error (ver la Sección A.2 para más detalles sobre el manejo de errores). Los posibles códigos de error que puede devolver `write()` están documentados en la [página del manual de write\(2\)](#).

Hay varios motivos por los que la función `write()` puede escribir menos bytes de los que se le han indicado, cuando se usa para escribir en un *socket* o en una tubería. Sin embargo, en el caso de un archivo, lo más común es que nunca ocurra. Si pasa, normalmente se debe a que no queda espacio suficiente en el dispositivo.

Lo correcto es comprobar el valor de retorno de `write()` y si es menor que el tamaño del buffer indicado, volver a llamar a `write()` con el resto de bytes que quedan por escribir. Si el problema se repite, entonces `write()` retornará un `-1` y el valor de `errno` indicará el error —por ejemplo, `ENOSPC` si el espacio en el dispositivo no es suficiente—.

A.7. Acceder a los atributos de un archivo

Las funciones `stat()` y `fstat()` ofrecen una manera de comprobar la existencia de un archivo y de obtener acceso a sus atributos para obtener información como:

- Tipo de archivo.
- Tamaño y número de bloques ocupados.
- Número de inodo.
- Propietario y grupo.
- Permisos.
- Fechas de acceso y modificación.

```
int stat(const char* file_name, struct stat* buf);
int fstat(int filedes, struct stat* buf);
```

Si `stat()` o `fstat()` fallan con el error `ENOENT`, es que el archivo por el que se pregunta no existe.

Ambas versiones de `stat()` reciben un puntero a una estructura `stat`, que se rellena con la información del archivo al volver de la llamada. La única diferencia entre `stat()` y `fstat()` es que `stat()` recibe la ruta del archivo y `fstat()` recibe el descriptor de archivo que se debe haber abierto previamente con `open()`.

La estructura `stat` contiene información sobre el archivo, como el número de inodo, los permisos, el tamaño, el número de enlaces, el propietario, el grupo o las fechas de acceso y modificación:

```
struct stat {
    dev_t          st_dev;      /* dispositivo */
    ino_t          st_ino;      /* inodo */
    mode_t         st_mode;     /* protección y tipo */
    nlink_t        st_nlink;    /* número de enlaces físicos */
    uid_t          st_uid;      /* ID del usuario propietario */
    gid_t          st_gid;      /* ID del grupo propietario */
    dev_t          st_rdev;     /* tipo dispositivo (si es
                                dispositivo inodo) */
    off_t          st_size;     /* tamaño total, en bytes */
    blksize_t      st_blksize;   /* tamaño de bloque para el
                                sistema de ficheros de E/S */
    blkcnt_t       st_blocks;    /* número de bloques asignados */
    time_t         st_atime;    /* hora último acceso */
    time_t         st_mtime;    /* hora última modificación */
    time_t         st_ctime;    /* hora último cambio */
};
```

A.7.1. Comprobar si dos archivos son el mismo

Cada archivo tiene un número de inodo único que lo identifica en el sistema de archivos. Un mismo archivo puede ser accesible por medio de varias rutas, por lo que se pueden comparar los números de inodos de archivos en rutas diferentes para determinar si son el mismo archivo.

El inconveniente es que el número de inodo solo es único dentro del mismo sistema de archivos, es decir, que dos archivos pueden tener el mismo número de inodo en sistemas de archivos diferentes. Por eso se necesita comparar tanto `st_dev` –que identifica el dispositivo donde se encuentra el archivo– como `st_ino` –que identifica el archivo dentro del dispositivo– de la estructura `stat`, para saber si dos rutas conducen al mismo archivo.

A.7.2. Acceso a los permisos y al tipo de archivo

Tanto los permisos como el tipo del archivo se almacenan en el campo `st_mode` de la estructura `stat`. Para comprobar si un archivo es de un tipo en particular, se puede utilizar alguna de las siguientes macros:

- `S_ISLNK(m)`: ¿Es un enlace simbólico?
- `S_ISREG(m)`: ¿En un fichero regular?
- `S_ISDIR(m)`: ¿Es un directorio?
- `S_ISCHR(m)`: ¿Es un dispositivo de caracteres?
- `S_ISBLK(m)`: ¿Es un dispositivo de bloques?
- `S_ISFIFO(m)`: ¿Es una tubería con nombre (FIFO)?
- `S_ISSOCK(m)`: ¿Es un *socket* de dominio UNIX con nombre?

Por ejemplo:

```
stat st;
if (stat(filepath, &st) == -1)
{
    // Error al obtener los atributos del archivo...
}
else
{
    if (S_ISLNK(st.st_mode))
    {
        // 'filepath' es un enlace simbólico
    }
    else {
        // 'filepath' no es un enlace simbólico
    }
}
```

La macro `S_IFMT` contiene una máscara con todos bits que sirven para indicar el tipo de archivo a 1. Es decir, que se puede usar `S_IFMT` para extraer por separado los bits de tipo y los permisos del archivo:

```
mode_t file_type = st.st_mode & S_IFMT;
mode_t file_permissions = st.st_mode & ~S_IFMT;
```

B. Gestión de errores en C++

La gestión adecuada de errores es fundamental en cualquier aplicación robusta. En C++ moderno, disponemos de varias estrategias para manejar errores, desde el retorno de códigos de error hasta el uso de excepciones, siendo cada una apropiada para diferentes situaciones. En esta sección vamos a centrarnos específicamente en cómo gestionar errores usando `std::expected`, introducida en C++23 y declarada en `<expected>`.

`std::expected<T, E>` es una clase *template* introducida en C++23 que puede contener un valor de tipo T en caso de éxito o un error de tipo E en caso de fallo. Es decir, permite que una función retorne un valor exitoso, si tuvo éxito, o un objeto con información del error, si la operación falló. En caso de error, el objeto que contiene la información del error, a su vez, puede ser propagado a las funciones que realizaron la llamada, sucesivamente, hasta que se maneje el error en un nivel superior.

En la actualidad, se recomienda usar `std::expected` en la mayoría de situaciones de manejo de errores en C++ moderno, especialmente para la mayoría de los errores predecibles y relativamente comunes que forman parte del flujo normal de la aplicación, como:

- Archivo no encontrado
- Permisos insuficientes
- Validación de entrada fallida
- Formato de datos incorrecto
- Validación de la línea de comandos

Mientras que las excepciones deben reservarse para situaciones verdaderamente excepcionales que interrumpen el flujo normal del programa y de las que es complicado recuperarse, como:

- Errores de lectura/escritura en hardware
- Falta de memoria
- Violaciones de invariantes críticas
- Condiciones que no deberían ocurrir en sistemas correctamente configurados

B.1. Propagación de errores con std::expected

B.1.1. Funciones con valor de retorno

Cuando una función necesita retornar un valor en caso de éxito y un código de error en caso de fallo, `std::expected` es ideal. Por ejemplo, consideremos una función `open_file()` que envuelve la llamada al sistema `open()` y retorna su descriptor:

```
std::expected<int, int> open_file(const std::string& path,           ①
    int flags, mode_t mode = 0)
{
    int fd = open(path.c_str(), flags, mode);
    if (fd == -1)                                ②
    {
        return std::unexpected(errno);            ③
    }

    return fd;                                    ④
}
```

- ① La función retorna `std::expected<int, int>`. El primer parámetro de `std::expected` es el tipo del objeto a retornar en caso de éxito (`int`, por tratarse de un descriptor de archivo), mientras que el segundo es el tipo del objeto para devolver el error (`int`, en este caso, que representa el código de error).
- ② La función `open()` retorna `-1` en caso de error.
- ③ En caso de error, devolvemos un objeto `std::unexpected()` que es un objeto `std::expected` que señala que ocurrió un error y contiene información sobre dicho error (el valor de `errno`).
- ④ En caso de éxito, también se devuelve un objeto `std::expected` creado con el valor que queremos que retorne la función en caso de éxito, que en este caso es el descriptor de archivos `fd`.

El uso de esta función es directo y al volver de la llamada se puede verificar fácilmente el resultado:

```
auto result = open_file("datos.txt", O_RDONLY);          ①
if (!result.has_value())                                 ②
{
    int error_code = result.error();                    ③
    std::println(stderr, "Error ({} ) al abrir archivo: {}",
                 error_code, std::strerror(error_code));
    // Manejar error...
}

int fd = result.value();                                ④
// Usar el descriptor de archivo...
```

- ① Llamar a `open_file()` para abrir el archivo.
- ② Comprobar si `open_file()` terminó con éxito verificando si el objeto `std::expected` devuelto contiene un error. `has_value()` retorna `true` si la operación fue exitosa y `false` si hubo un error.
- ③ En caso de error, se puede recuperar el código de error llamando al método `error()` del objeto `std::expected` y se maneja el error según sea necesario. Esto puede implicar registrar el error, notificar al usuario o realizar alguna otra acción correctiva.
- ④ Si todo ha ido bien, se puede usar el método `value()` para acceder al descriptor de archivo almacenado en el objeto `std::expected`. El descriptor de archivos se puede usar para leer o escribir en el archivo, entre otras operaciones.

B.1.2. Funciones sin valor de retorno

Para funciones que no necesitan retornar un valor en caso de éxito, podemos usar `std::expected<void, E>`. Esto es especialmente útil para operaciones que solo pueden fallar o tener éxito:

```
std::expected<void, int> write_data(int fd,
                                     const std::vector<uint8_t>& data)
{
    size_t total_written = 0;

    while (total_written < data.size())
    {
        ssize_t written = write(fd,                         (1)
                               data.data() + total_written,
                               data.size() - total_written);

        if (written == -1)                                (2)
        {
            return std::unexpected(errno);                (3)
        }

        total_written += static_cast<size_t>(written);
    }

    return {};
}
```

- ① Llamada a `write()` para escribir datos en el archivo.
- ② La función `write()` retorna -1 en caso de error.
- ③ En caso de error, devolvemos un objeto `std::unexpected()` con el código de error (`errno`).
- ④ En caso de éxito, devolvemos un objeto `std::expected` vacío, indicando que la operación se completó sin errores.

Esta función se usaría de manera similar:

```
std::vector<uint8_t> datos = {/* ... */};
auto result = write_data(fd, datos);

if (!result.has_value())
{
    std::println(stderr, "Error ({} ) al escribir: {}",
        result.error(), std::strerror(result.error()));
    // Manejar error...
}

// Continuar con la siguiente operación...
```

B.2. Información de error enriquecida

En los ejemplos anteriores, usamos `int` como tipo de error para representar códigos de error del sistema (`errno`), pero podemos definir nuestros propios tipos de error para proporcionar información más rica y específica sobre los errores que pueden ocurrir en nuestra aplicación. Por ejemplo, podríamos definir un `enum` para representar diferentes tipos de errores o una clase que encapsule información adicional sobre el error, como el tipo de error, un mensaje descriptivo, la línea de código donde ocurrió el error, etc.

C++ ya proporciona varias clases de error estándar que también se pueden utilizar como excepciones, como: `std::runtime_error`, `std::system_error`, `std::logic_error`, entre otras.

En particular, `std::system_error` es útil para representar errores relacionados con llamadas al sistema y puede encapsular códigos de error del sistema junto con mensajes descriptivos:

```
std::expected<void, std::system_error> write_data(int fd,           ①
    const std::vector<uint8_t>& data)
{
    size_t total_written = 0;

    while (total_written < data.size())
    {
        ssize_t written = write(fd,
            data.data() + total_written,
            data.size() - total_written);

        if (written == -1)
        {
            return std::unexpected(
                std::system_error(          ②

```

```

        errno,
        std::system_category(),
        "Error al escribir datos")
    );
}

total_written += static_cast<size_t>(written);
}

return {};
}

```

- ① La función retorna `std::expected<void, std::system_error>`, indicando que en caso de error se retornará un objeto `std::system_error`.
- ② Creamos un objeto `std::system_error` para encapsular información detallada sobre el error.
- ③ El primero de los parámetros de `std::system_error` es el código de error del sistema (`errno`).
- ④ El segundo parámetro es la categoría del error, que en los errores del sistema es `std::system_category()`.
- ⑤ El tercer parámetro es un mensaje descriptivo personalizado que proporciona contexto adicional sobre el error.

El uso de esta función sería similar al ejemplo anterior, pero ahora podemos acceder al objeto `std::system_error` para obtener tanto el código de error como el mensaje descriptivo:

```

std::vector<uint8_t> datos = {/* ... */};
auto result = write_data(fd, datos);

if (!result.has_value())
{
    std::println(std::cerr, "Error ({}) al escribir: {}",                (1)
                result.error().code().value(),                                (1)
                result.error().what());                                         (2)
    // Manejar error...
}

// Continuar con la siguiente operación...

```

- ① `result.error().code().value()` retorna el valor de `errno` encapsulado en el objeto `std::system_error`.
- ② `result.error().what()` retorna el mensaje descriptivo del error, que incluye tanto el mensaje personalizado como la descripción del código de error que proporciona el sistema.

B.3. Forzar la comprobación de errores con [[nodiscard]]

Una de las principales ventajas de usar `std::expected` es que podemos obligar al programador a comprobar si la operación tuvo éxito antes de usar el resultado. Esto se logra usando el atributo `[[nodiscard]]` en la declaración de la función:

```
[[nodiscard]]
std::expected<int, std::system_error> open_file(const std::string& path,
    int flags, mode_t mode = 0)
{
    int fd = open(path.c_str(), flags, mode);
    if (fd == -1)
    {
        std::string error_message =
            std::format("Error al abrir '{}'", path);
        return std::unexpected(std::system_error(errno,
            std::system_category(), error_message));
    }

    return fd;
}
```

Con `[[nodiscard]]`, el compilador emitirá una advertencia si intentamos llamar a la función sin usar su valor de retorno:

```
open_file("archivo.txt", O_RDONLY);                                ①

// ...

auto result = open_file("archivo.txt", O_RDONLY);                  ②
if (!result.has_value())
{
    // Manejar error...
}
```

- ① En este caso, el compilador emitirá una advertencia porque el valor de retorno de `open_file()` se está ignorando.
- ② Aquí, el valor de retorno se almacena en `result` y se comprueba correctamente, por lo que no habrá advertencia.

Esto es especialmente útil para funciones que pueden fallar, ya que obliga a los programadores a manejar las condiciones de error de manera explícita. Si realmente queremos ignorar el resultado, debemos hacerlo de forma explícita:

```
// Ignorar explícitamente el resultado (no recomendado)
[[maybe_unused]] auto result = open_file("archivo.txt", O_RDONLY);

// O usando std::ignore
std::ignore = open_file("archivo.txt", O_RDONLY);
```

i Nota

El atributo `[[nodiscard]]` es especialmente valioso durante el desarrollo, ya que ayuda a detectar errores donde olvidamos comprobar si una operación tuvo éxito. Recomendamos usar este atributo en todas las funciones que retornen `std::expected`.