# Programación de aplicaciones — Consejos al programar en C++

**Sistemas Operativos 2024/2025**

Jesús Torres

En esta práctica vamos a desarrollar una herramienta utilizando directamente algunas de las funciones de la librería del sistema. Estas funciones suelen estar escritas en C, pero no estamos obligados a usar este lenguaje para emplearlas. En su lugar, veremos cómo podemos usar C++ para desarrollar nuestro programa de forma más sencilla y segura.

## Contenidos

# 1. Introducción

Vamos a repasar algunas recomendaciones y consejos generales que pueden ayudar en el desarrollo de la práctica. Te recomendamos que los tengas presentes mientras lees y resuelves las distintas partes del guion.

Verás que el guion está lleno de referencias a estos consejos, por lo que te fácil recordarlos y aplicarlos.

# 2. Opciones recomendadas del compilador

El compilador de C++ dispone de una serie de opciones que permiten detectar errores y problemas en el código de nuestros programas. Estas opciones son especialmente útiles durante el desarrollo del software, ya que permiten detectar problemas en el código y corregirlos antes de que se conviertan en errores durante la ejecución.

## 2.1. Opciones para avisos y errores de compilación

Con g++ recomendamos utilizar al menos las siguientes opciones de diagnóstico:

- `-Wall`. Activa los avisos más comunes.
- `-Wextra`. Activa avisos adicionales. Cuando se usa junto con `-Wall`, se cubren la mayoría de los problemas de código.
- `-Werror`. Convierte los avisos en errores, haciendo que el compilador termine con un error si se detecta alguno.
- `-Wpedantic`. Activa avisos sobre el uso de extensiones no estándares del lenguaje.

Por tanto, ejecutaríamos el compilador así:

```
$ g++ -std=c++23 -Wall -Wextra -Werror -Wpedantic -o ejemplo ejemplo.cpp
```

Además de las opciones mínimas anteriores, es recomendable activar también las siguientes:

- `-Wshadow`. Activa avisos sobre variables locales que ocultan variables de ámbito superior.
- `-Wnon-virtual-dtor`. Activa avisos sobre clases con funciones virtuales que tienen un destructor no virtual.
- `-Wold-style-cast`. Activa avisos sobre el uso de *typecasts* de estilo-C.
- `-Wcast-align`. Activa avisos sobre *typecasts* que pueden causar problemas de alineamiento, lo que puede tener impacto en el rendimiento.

- `-Wunused`. Activa avisos sobre variables, funciones o tipos que no se usan.
- `-Woverloaded-virtual`. Activa avisos sobre funciones virtuales que se sobrecargan en lugar de sobreescribir.
- `-Wconversion`. Activa avisos sobre posibles pérdidas de datos en conversiones implícitas.
- `-Wsign-conversion`. Activa avisos sobre posibles pérdidas de datos en conversiones implícitas de tipos con signo.
- `-Wnull-dereference`. Activa avisos sobre posibles desreferencias de punteros nulos.
- `-Wdouble-promotion`. Activa avisos sobre posibles pérdidas de precisión en conversiones implícitas de `float` a `double`.
- `-Wformat=2`. Activa avisos sobre posibles problemas de seguridad en funciones de formateo de texto, como `printf()`.
- `-Wmisleading-indentation`. Activa avisos sobre indentaciones que parecen indicar que el código pertenece a un bloque pero tal bloque no existe.
- `-Wduplicated-cond`. Activa avisos sobre condiciones duplicadas en cadenas de `if/else`.
- `-Wduplicated-branches`. Activa avisos sobre código duplicado en cadenas de `if/else`.
- `-Wlogical-op`. Activa avisos sobre operadores lógicos que parecen indicar que se querían usar operadores a nivel de bits.
- `-Wuseless-cast`. Activa avisos sobre *typecasts* que no cambian el tipo de la expresión porque ya es del tipo al que se quiere convertir.

Para lo que tendríamos que iniciar la compilación así:

```
$ g++ -std=c++23 -Wall -Wextra -Werror -Wpedantic \
      -Wshadow -Wnon-virtual-dtor -Wold-style-cast \
      -Wcast-align -Wunused -Woverloaded-virtual \
      -Wconversion -Wsign-conversion -Wnull-dereference \
      -Wdouble-promotion -Wformat=2 -Wmisleading-indentation \
      -Wduplicated-cond -Wduplicated-branches -Wlogical-op \
      -Wuseless-cast -o ejemplo ejemplo.cpp
```

## 2.2. Opciones para sanear el código

Con g++ también recomendamos utilizar la opción `-fsanitize=address,undefined,leak` para *sanear* el código durante el desarrollo de la práctica:

```
$ g++ -std=c++23 -Wall -Wextra -Werror -Wpedantic \
      -Wshadow -Wnon-virtual-dtor -Wold-style-cast \
      -Wcast-align -Wunused -Woverloaded-virtual \
      -Wconversion -Wsign-conversion -Wnull-dereference \
      -Wdouble-promotion -Wformat=2 -Wmisleading-indentation \
      -Wduplicated-cond -Wduplicated-branches -Wlogical-op \
```

```
    -Wuseless-cast -fsanitize=address,undefined,leak \
    -o ejemplo ejemplo.cpp
```

Sanear el código quiere decir que el compilador inserta código adicional en el programa para detectar problemas en tiempo de ejecución.

### 2.2.1. *Undefined behaviors* en C++

C++ es un lenguaje de programación conocido por su capacidad para escribir software de alto rendimiento. En parte por eso, el estándar de C++ no define el comportamiento esperado del programa en determinados casos, dejando libertad a los compiladores para que optimicen el código de forma agresiva, con el objetivo de conseguir el mejor rendimiento.

Estos casos de comportamiento indefinido se conocen en el estándar de C++ como *undefined behavior* (UB).

Los *undefined behavior* son asunto de mucho debate porque, pese a sus ventajas respecto al rendimiento, pueden dar lugar a comportamientos inesperados del programa, obligando a los programadores a tener mucho cuidado al escribir código en C++.

Un ejemplo es el acceso a un vector fuera de sus límites.

```
#include <vector>

int main(int argc, char* argv[])
{
    std::vector<int> v(5);
    v[5] = 1;                                                    ①

    return EXIT_SUCCESS;
}
```

① El acceso a un elemento fuera de los límites de un vector es un *undefined behavior*.

Pese a lo que se suele creer, el acceso a un elemento fuera de los límites de un vector no es definido por el estándar de C++ como un error, porque si fuera así, el código generado por el compilador debería detectarlo, comunicar el error y terminar el programa, como ocurre en otros lenguajes de programación.

En su lugar es un *undefined behavior*, de forma que el compilador no tiene que hacer ninguna comprobación en cada intento de acceso, mejorando el rendimiento del código generado. Sin embargo, a cambio, el programador debe ser consciente de que el programa puede comportarse de forma inesperada si se accede a un elemento fuera de los límites del vector.

Según el caso, el programa puede terminar con un error o no, puede mostrar un resultado incorrecto o no, o puede abrir una puerta a la ejecución de código malicioso.

### 2.2.2. Flujo de trabajo recomendado para evitar *undefined behaviors*

Durante el desarrollo de software en C++, una práctica recomendada es activar opciones específicas en el compilador que ayudan a detectar *undefined behaviors* y otros problemas potenciales en el código.

Algunos de estos problemas se pueden detectar en tiempo de compilación, activando opciones adicionales de diagnóstico que emiten mensajes de advertencia o error.

Sin embargo, muchos de los posibles problemas solo se pueden detectar en tiempo de ejecución, por lo que se utilizan opciones de compilación que insertan código adicional en el programa para detectar estos problemas durante la ejecución.

Estos son los *saneadores* (*sanitizers*) del compilador. Con el código adicional que introducen, se consigue que el programa termine con un error en caso de que se detecte un problema, en lugar de que se comporte de forma inesperada.

Por ejemplo, la opción `-fsanitize=address,undefined,leak`, el ejemplo anterior que accedía fuera de los límites de un vector falla así durante su ejecución, en lugar de comportarse de forma inesperada:

```
$ ./ejemplo
=================================================================
==1==ERROR: AddressSanitizer: heap-buffer-overflow on address...          ①
WRITE of size 4 at 0x503000000054 thread T0
    #0 0x401394 in main /app/example.cpp:6                                ②
...
```

① El error es un *buffer overflow*.
② El error ha sido causado por la línea 6 del código fuente en `example.cpp`.

Como este código adicional puede afectar al rendimiento, los programas siempre se prueban con estas opciones activadas durante le desarrollo, permitiendo detectar problemas en el código y corregirlos. Pero cuando el programa está listo para su distribución, se puede optar por compilar sin estas opciones, para entregar un ejecutable con el mejor rendimiento posible.

## 2.3. Pasar opciones al compilador

Obviamente, escribir todas estas opciones cada vez que se compila un programa no resulta muy cómodo:

```
$ g++ -std=c++23 -Wall -Wextra -Werror -Wpedantic \
      -Wshadow -Wnon-virtual-dtor -Wold-style-cast \
      -Wcast-align -Wunused -Woverloaded-virtual \
      -Wconversion -Wsign-conversion -Wnull-dereference \
      -Wdouble-promotion -Wformat=2 -Wmisleading-indentation \
```

```
      -Wduplicated-cond -Wduplicated-branches -Wlogical-op \
      -Wuseless-cast -fsanitize=address,undefined,leak \
      -o ejemplo ejemplo.cpp
```

Por eso recomendamos usar un *script* de BASH, `Makefile` o proyecto de CMake para automatizar la compilación de los programas.

Otra opción es definir una variable como `CXXFLAGS` con las opciones que interesen y luego indicar esta variable al compilador en la línea de comandos. Por ejemplo, si se define `CXXFLAGS` así:

```
$ CXXFLAGS="-std=c++23 -Wall -Wextra -Werror -Wpedantic \
    -Wshadow -Wnon-virtual-dtor -Wold-style-cast \
    -Wcast-align -Wunused -Woverloaded-virtual \
    -Wconversion -Wsign-conversion -Wnull-dereference \
    -Wdouble-promotion -Wformat=2 -Wmisleading-indentation \
    -Wduplicated-cond -Wduplicated-branches -Wlogical-op \
    -Wuseless-cast -fsanitize=address,undefined,leak"
```

Simplemente tenemos que ejecutar el compilador de la siguiente manera:

```
$ g++ $CXXFLAGS -o ejemplo ejemplo.cpp
```

## 3. Aprovecha las ventajas de C++ todo lo posible

Nuestro objetivo es aprender cómo funciona el sistema operativo. Así que usaremos, necesariamente, la interfaz de programación de aplicaciones del sistema operativo, que está escrita para C, pero no es necesario desarrollar toda la práctica en C. Al contrario, es recomendable crear clases y métodos que envuelvan esa interfaz de bajo nivel, permitiéndonos acceder los servicios del sistema operativo de forma más sencilla desde C++.

C++ implementa una extensa librería estándar que es de gran ayuda para los programadores. Utilizarla todo lo posible para desarrollar la lógica del programa, con toda seguridad nos quitará mucho trabajo y nos evitará muchos errores.

### 3.1. Cadenas de caracteres

Por ejemplo, la función de la librería del sistema que devuelve el valor de una variable de entorno en el contexto del proceso actual tiene este prototipo:

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

Pero usar y manipular cadenas `char*` de C puede ser bastante tedioso. Por eso es mucho mejor envolver la función anterior con una versión que trabaje con `std::string`, para así aprovechar más fácilmente las comodidades de las cadenas de C++:

```
std::string getenv(const std::string& name)
{
    char* value = getenv(name.c_str());
    if (value) {
        return std::string(value);
    }
    else {
        return std::string();
    }
}
```

Al llamar a esta nueva versión, se obtiene una cadena con el valor de la variable, si la variable existe y tiene un valor. En caso contrario, devuelve una cadena vacía.

## 3.2. Buffers

Muchas de las funciones de la librería del sistema necesitan que se les indiquen la dirección y el tamaño de un buffer donde almacenar el resultado de una operación u obtener los datos necesarios para realizarla. Por ejemplo, en las funciones `read()` y `write()`, que se utilizan para leer y escribir en archivos:

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

El argumento `buf` debe usarse para indicar un puntero al buffer de memoria donde se guardarán los bytes leídos o se tomarán los bytes que deben ser escritos. Mientras que `count` es el tamaño del buffer, en bytes. Es decir, el máximo de bytes que se pueden leer del archivo y guardar en el buffer –cuando la operación es `read()`– o el número de bytes del buffer que se quieren escribir en el archivo –si la operación es `write()`–.

En C++ la forma recomendada de crear un buffer de tamaño fijo es con `std::array`:

```
std::array<uint8_t, 240> buffer;
ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
if (bytes_read < 0)
{
    // Manejar error en read()...
}
```

Sin embargo, debemos tener presente que los buffers `std::array` se crean en la **pila** del hilo y que esta tiene un tamaño limitado –en muchos sistemas, 8 MiB–.

> **Nota**
>
> En sistemas Linux puedes conocer el tamaño por defecto de la pila, ejecutando el comando `ulimit -s`. El valor devuelto está en KiB, no en bytes.

Para reservar cantidades dinámicamente más grandes de memoria en el **montón** o cuando nos conviene que el buffer tenga un tamaño variable, es mejor usar `std::vector`:

```
std::vector<uint8_t> buffer(16ul * 1024 * 1024);
ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
if (bytes_read < 0)
{
    // Manejar error en read()...
}
buffer.resize(bytes_read);                                          (1)
```

(1) Reducir el tamaño del vector a la cantidad de bytes que `read()` ha leído en realidad, para que solo contenga los bytes válidos.

Como en el caso de `getenv()`, podemos envolver la función `read()` con una versión que trabaje con `std::vector`, para así aprovechar más fácilmente las comodidades de los vectores de C++ en el resto de nuestro código:

```
int read_file(int fd, std::vector<uint8_t>& buffer)
{
    ssize_t bytes_read = read(fd, buffer.data(), buffer.size());
    if (bytes_read < 0)
    {
        // Manejar error en read()...
    }
    buffer.resize(bytes_read);
    return 0;
}
```

> **Nota**
>
> Observa que la función `read_file()` recibe un `std::vector<uint8_t>` para almacenar los bytes leídos del archivo, pero podemos pasarle un `std::vector` de cualquier otro tipo de dato o simplemente una estructura que queramos leer del archivo. Lo importante es que la función `read()` necesita la dirección de la memoria donde almacenar los datos leídos y el tamaño del buffer.

## 4. Descriptores de archivo y otros recursos del sistema

En sistemas POSIX, los descriptores de archivo son enteros que identifican un recurso del sistema operativo, como un archivo, un socket o un dispositivo. Por ejemplo, la función `open()` abre un archivo y devuelve un descriptor de archivo que se puede usar para leer o escribir en él:

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

Este descriptor de archivo es el que se pasa a las funciones `read()` y `write()` a través de su primer argumento para leer y escribir en dicho archivo.

El descriptor de archivo es un recurso del sistema operativo que debe ser liberado –generalmente usando la función `close()`– cuando ya no se necesite, por lo que es recomendable envolverlo en una clase que se encargue de cerrarlo automáticamente cuando el objeto que lo contiene se destruya.

```
class SafeFD
{
public:
    explicit SafeFD(int fd) noexcept
        : fd_{fd}
    {}

    ~SafeFD() noexcept
    {
        if (fd_ >= 0)
        {
            close(fd_);
        }
    }

    [[nodiscard]] int get() const noexcept
    {
        return fd_;
    }

private:
    int fd_;
};
```

Con esta clase `SafeFD`, podemos definir nuestra función `open_file()` mediante `open()` así:

```
SafeFD open_file(const std::string& path, int flags, mode_t mode = 0)
{
    int fd = open(path.c_str(), flags, mode);
    if (fd < 0)
    {
        // Manejar error en open()...
    }
    return SafeFD{fd};
}
```

Y la función `read_file()` que hemos visto antes, la podemos modificar para que reciba un `SafeFD` en lugar de un `int`:

```
int read_file(const SafeFD& fd, std::vector<uint8_t>& buffer)
{
    ssize_t bytes_read = read(fd.get(), buffer.data(), buffer.size());
    if (bytes_read < 0)
    {
        // Manejar error en read()...
    }
    buffer.resize(bytes_read);
    return 0;
}
```

Con la ventaja de que cuando la variable de tipo `SafeFD` se destruya, el descriptor de archivo se cerrará automáticamente.

### 4.1. `SafeFD` más seguro

La clase `SafeFD` que hemos definido es muy básica y no es muy segura. Por ejemplo, si hubiéramos definido `read_file()` así:

```
int read_file(SafeFD fd, std::vector<uint8_t>& buffer)                    ①
{
    // ...
}
```

① Se pasa el objeto `SafeFD` por valor, creando una copia del objeto original indicado como argumento.

Y la invocamos así:

```
SafeFD testFd = open_file("test.txt", O_RDONLY);
read_file(fd, buffer);                                                    ①
```

① Al salir de `read_file`, el descriptor de archivo en `testFd` estará cerrado y ya no será válido.

El `SafeFD` se pasaría por valor a `read_file()`, pero eso haría que el descriptor de archivo que contiene se cierre antes de salir de `read_file()` y destruir el objeto `fd`. Como ese descriptor de archivo es el mismo que el del objeto `testFd` –pues `fd`es una copia de `testFd`– al volver de `read_file()` el descriptor de archivo en `testFd` ya no sería válido.

Por eso, es mejor pasar `SafeFD` siempre por referencia. Para asegurar que nunca intentaremos copiarlo, se puede eliminar el constructor de copia y el operador de asignación de la clase `SafeFD` anterior:

```
class SafeFD
{
public:
    explicit SafeFD(int fd) noexcept
        : fd_{fd}
    {}

    SafeFD(const SafeFD&) = delete;                                    ①
    SafeFD& operator=(const SafeFD&) = delete;                         ②

    ~SafeFD() noexcept
    {
        if (fd_ >= 0)
        {
            close(fd_);
        }
    }

    [[nodiscard]] int get() const noexcept
    {
        return fd_;
    }

private:
    int fd_;
};
```

① Eliminar el constructor de copia. Ya no se podrá pasar un `SafeFD` por valor.
② Eliminar el operador de asignación. Ya no se podrá asignar un `SafeFD` a otro usando el operador '`=`'.

Lo que sí puede hacerse es permitir que un objeto `SafeFD` se mueva a otro, para lo que se puede definir un constructor de movimiento y un operador de asignación de movimiento:

```cpp
class SafeFD
{
public:
    explicit SafeFD(int fd) noexcept
        : fd_{fd}
    {}

    explicit SafeFD() noexcept                                          ①
        : fd_{-1}
    {}

    SafeFD(const SafeFD&) = delete;
    SafeFD& operator=(const SafeFD&) = delete;

    SafeFD(SafeFD&& other) noexcept                                     ②
        : fd_{other.fd_}
    {
        other.fd_ = -1;
    }

    SafeFD& operator=(SafeFD&& other) noexcept                          ③
    {
        if (this != &other && fd_ != other.fd_)
        {
            // Cerrar el descriptor de archivo actual
            close(fd_);

            // Mover el descriptor de archivo de 'other' a este objeto
            fd_ = other.fd_;
            other.fd_ = -1;
        }
        return *this;
    }

    ~SafeFD() noexcept
    {
        if (fd_ >= 0)
        {
            close(fd_);
        }
    }

    [[nodiscard]] bool is_valid() const noexcept                        ④
    {
        return fd_ >= 0;
```

```
    }

    [[nodiscard]] int get() const noexcept
    {
        return fd_;
    }

private:
    int fd_;
};
```

① Constructor por defecto que inicializa el descriptor de archivo a -1. Por tanto, cualquier objecto `SafeFD` creado de esta manera contiene un descriptor inválido.
② Constructor de movimiento. Construye un objeto `SafeFD` a partir de otro, moviendo el descriptor de archivo de ese a este. El otro objeto queda con un descriptor de archivo inválido.
③ Operador de asignación de movimiento. Mueve el descriptor de archivo de otro objeto a este, cerrando el descriptor de archivo actual en este objeto y dejando un descriptor de archivo inválido en el otro objeto.
④ Método para comprobar si el descriptor de archivo es válido (mayor que 0).

Con estos cambios, es posible mover un objeto `SafeFD` a otro para, por ejemplo, permitir el siguiente caso:

```
SafeFD testFd;                                                    ①

if (condition)
{
    testFd = open_file("test.txt", O_RDONLY);                     ②
}

// ...

read_file(testFd, buffer);
```

① Objeto `SafeFD` por defecto, con descriptor de archivo inválido (-1).
② Mover el descriptor de archivo devuelto por `open_file()` a `testFd`.

## 5. Manejo de errores

En los ejemplos que hemos visto con `getenv()` y `read()` siempre comprobamos el valor devuelto para detectar si se ha producido algún error.

> **Importante**
>
> La mayor parte de las funciones que sirven servicios y recursos del sistema pueden fallar por diversos motivos, por lo que debemos comprobar esta condición antes de continuar y tratar de usar su resultado.

En los sistemas POSIX, la mayor parte de las funciones de la librería del sistema devuelven un valor negativo en caso de error, y un valor no negativo en caso de éxito. En el caso de `read()`, el valor devuelto es el número de bytes leídos, o -1 en caso de error. Por ejemplo, observa la comprobación sobre el valor devuelto por `read()` en la línea 4 de nuestra función:

```
1  int read_file(const SafeFD& fd, std::vector<uint8_t>& buffer)
2  {
3      ssize_t bytes_read = read(fd.get(), buffer.data(), buffer.size());
4      if (bytes_read < 0)                                                    ①
5      {
6          std::println( std::cerr, "Error ({}) en read(): {}", errno,        ②
7              std::strerror(errno));
8          return errno;
9      }
10     buffer.resize(bytes_read);
11     return 0;
12 }
```

① Comprobación de error de la operación `read()`.
② Mostrar un mensaje de error con el código de error de `errno` y su descripción con `std::strerror()`.

El valor devuelto por `read()` y otras funciones en caso de error no indica el tipo de error que ha ocurrido. Para obtener un código de error que nos indique el tipo, debemos consultar la macro `errno` (véase la línea 6 del ejemplo anterior). Este código nos ayudará a nosotros o al usuario que ha invocado el programa a identificar el origen del problema. Especialmente con ayuda de la función `strerror()` –o `std::strerror()` de C++– que nos permite obtener un mensaje de texto descriptivo para cada valor posible de `errno`.

> **Nota**
>
> Esta forma de gestionar los errores es muy común en API, librerías y programas en C. Por ejemplo, Windows API define una función `GetLastError()` para obtener el código de error de la última operación fallida. Las funciones de Windows API devuelven NULL o un valor negativo en caso de error al realizar una operación.

El valor de `errno` puede ser cambiado por cualquier función de la librería del sistema. Muchas de las funciones de la librería estándar del lenguaje o de otras librerías pueden hacer uso funciones de la librería del sistema sin que nosotros lo sepamos. Acciones tan sencillas como

añadir un elemento a un vector o imprimir un mensaje por la salida estándar, pueden usar la librería del sistema. Por eso, **lo primero que hay que hacer tras detectar que una función de la librería del sistema ha fallado, es guardar el valor actual de `errno`** para preservar el código del error de cambios posteriores.

### 5.0.1. Propagación de códigos de error

En el ejemplo anterior, la función `read_file()` muestra un mensaje de error y devuelve el código de error de `errno` en caso de fallo. Sin embargo, no deberíamos imprimir mensajes de error en la función `read_file()` porque no sabemos si la función que la invoca quiere hacerlo o no. Igualmente, podríamos hacer que `read_file()` terminase unilateralmente el programa en caso de error, pero eso, generalmente, **es una mala idea**. La función `read_file()` no sabe si en el contexto en el que está siendo llamada, el programa debe terminar o tiene que hacer otra cosa en caso de error.

Por ejemplo, si se usase `read_file()` para implementar una función `copy_file()` que copia un archivo a una ruta de destino, lo recomendable es que `read_file()` devuelva el error para que `copy_file()` pueda comprobar el valor devuelto, manejar adecuadamente la situación y, probablemente, propagar a su vez la condición de error a la función que la invocó a ella. Es decir, si `read_file()` falla, la operación `copy_file()` también fallará y esa condición, así como el motivo del error, debe ser comunicado a la función que invocó a `copy_file()` en primer lugar.

En general, no es buena idea que una función haga terminar unilateralmente el programa en caso de detectar un error. Ni tampoco que imprima mensajes de error. Es más flexible que comunique la situación a quien la invocó, para que así sea más fácil rehusar el código en distintas partes de este o en otros proyectos.

> **!** Importante
>
> En el caso de C++, **todo programa debería terminar con un `return` en `main()`** –con un 0 o un valor distinto de 0, en función de si el programa terminó con éxito o no, respectivamente– lo que nos obliga a propagar los errores hasta la función principal del programa.

La necesidad de gestionar y propagar códigos de error de distintas API y librerías es tan común, que C++ ofrece `std::error_code`[1] para facilitar el trabajo.

Por ejemplo, en la siguiente versión de nuestra función `read_file()`, se devuelve un objeto `std::error_code` de la categoría `std::system_category()`, con 0 en caso de éxito o el código de error de `errno` en caso de error:

```
std::error_code read_file(const SafeFD& fd, std::vector<uint8_t>& buffer)
{
    ssize_t bytes_read = read(fd.get, buffer.data(), buffer.size());
```

[1] `std::error_code` se declara en `<system_error>`.

```
    if (bytes_read < 0)
    {
        return std::error_code(errno, std::system_category());          ①
    }
    buffer.resize(bytes_read);
    return std::error_code(0, std::system_category());                  ②
}
```

① Retornar un objeto `std::error_code` con el código del error de `read()`.
② Retornar un objeto `std::error_code` con el valor 0, para indicar el éxito de la operación.

Cada objeto `std::error_code` almacena un código de error y una categoría de errores. El concepto de categoría es importante porque permite usar el mismo valor de error en tipos de error diferente, si están en distintas categorías. Por ejemplo, una librería de comunicaciones en red puede usar los mismos valores de error que la librería del sistema, si define su propia categoría de errores.

El valor devuelto por `std::system_category()` corresponde a la categoría de *errores del sistema*, donde podemos englobar los errores notificados por la librería del sistema.

Al invocar a `read_file()` es muy sencillo comprobar si ha habido algún error, ya que el objeto `std::error_code` tiene un operador de conversión a `bool` que devuelve `true` si el código de error es 0 y `false` en caso contrario:

```
std::error_code copy_file(const std::string& src_path,
    const std::string& dst_path)
{
    SafeFD src_fd = open_file(src_path, /* ... */);

    // ...

    std::error_code error = read_file(src_fd, buffer);                  ①
    if (error)                                                          ②
    {
        // Manejar error en read_file()...                              ③
        return error;                                                   ④
    }

    // ...

    return std::error_code(0, std::generic_category());                 ⑤
}
```

① Llamar a `read_file()` para leer del archivo.
② Comprobar si `read_file()` terminó con éxito comprobado el objeto `std::error_code` devuelto.

③ En caso de error, es necesario manejarlo. Por ejemplo, cerrando y liberando recursos reservados dentro de `copy_file()`, que ya no van a ser necesarios debido a que estamos a punto de salir de la función. En este ejemplo no hace falta porque `SafeFD` se encarga de cerrar el descriptor de archivo al destruirse.
④ Propagar el error al invocador de `copy_file()` usando el valor de retorno. El invocador tendrá que hacer su propio manejo del error y continuar con la propagación de este por la pila de llamadas.
⑤ Si todo ha ido bien, cerrar y librear recursos y terminar indicado que se ha tenido éxito.

Además, el objeto `std::error_code` tiene un método `message()` que devuelve un mensaje de texto descriptivo del error –que obtiene llamando a `std::strerror`– y un método `value()` que devuelve el código de error –es decir, el valor de `errno` cuando se creó el objeto–.

```
std::error_code error = read_file(fd, buffer);
if (error)
{
    std::println( std::cerr, "Error ({}): {}", error.value(),          ①
        error.message());
}
```

① Usar los métodos `value()` y `message` del objeto `std::error_code` para componer un mensaje de error.

## 5.1. Obligar a comprobar los errores

En C es muy común gestionar los errores mediante el retorno de códigos de error y su propagación a través del retorno de las funciones hacia `main()`.

Algunos lenguajes modernos –como Rust o Go– también han optado por esta solución y en C++ hay cierto interés en mejorar su soporte introduciendo algunas ayudas adicionales. El motivo es que obliga a los programadores a tratar los errores de forma explícita, en el punto donde se producen.

En los lenguajes más modernos, incluso se puede impedir que el programador ignore el código de error devuelto, obligando a añadir el código necesario para gestionarlo. En C++ se puede indicar al compilador que muestre un *warning* si el programador olvida leer el código de error devuelto por una función, especificando el atributo `nodiscard` al declararla:

```
[[nodiscard]]
std::error_code read_file(const SafeFD& fd, std::vector<uint8_t>& buffer)
{
    // ...
}
```

> **Nota**
>
> Por claridad, en los guiones de prácticas no usaremos el atributo `nodiscard`. Sin embargo, **recomendamos usarlo** para resolver la práctica, con el objeto de que nos recuerde que **siempre tenemos que comprobar los errores devueltos por las funciones** y manejarlos adecuadamente.

## 5.2. Propagación de errores con `std::expected`

En nuestra función `read_file()` los datos leídos del archivo se devuelve mediante un argumento `buffer` pasado por referencia. Por tanto, no tenemos problema en usar el valor de retorno para devolver el código de error usando un `std::error_code`. Sin embargo, ¿qué podemos hacer cuando queremos retornar algún valor en caso de éxito y un código de error en caso de error?.

Por ejemplo, la función `open_file()` que desarrollamos anteriormente se declara así:

```
SafeFD open_file(const std::string& path, int flags, mode_t mode = 0);
```

Esta función devuelve un descriptor de archivo en caso de éxito pero necesitamos que también devuelva un código de error `std::error_code` en caso de fallo.

Para poder devolver uno de los dos valores, según el caso, C++ ofrece `std::expected`[2]. Esta clase se utiliza como se muestra en la siguiente función `open_file()`:

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

    return SafeFD{fd};                                                    ③
}
```

① La función retorna `std::expected<SafeFD, std::error_code>`. El primer parámetro de `std::expected` debe ser el tipo del objeto a retornar en caso de éxito, mientras que el segundo es el tipo del objeto para devolver el error.

② En caso de error, devolvemos un objeto `std::expected`, pero para que este señale un error, se crea con `std::unexpected()`, pasándole el objeto con el código de error.

[2] `std::expected` está disponible desde gcc-12 (C++23) y se declara en `<expected>`.

③ 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` dentro de un objeto `SafeFD`.

Al volver de `open_file()`, es muy sencillo comprobar si ha habido algún error, ya que el objeto `std::expected` tiene un método `has_value()` que devuelve `true` si no contiene un error:

```
std::expected<SafeFD, std::error_code> result = open_file("test.txt",      ①
    flags, mode = 0);
if (! result.has_value())                                                  ②
{
    // Manejar error en open_file()...                                     ③
    return result.error();
}

SafeFD fd = std::move(result.value());                                     ④

std::vector<uint8_t> buffer(1024);
std::error_code error = read_file(fd, buffer);                             ⑤
```

① Llamar a `open_file()` para abrir el archivo.
② Comprobar si `open_file()` terminó con éxito comprobado si objeto `std::expected` devuelto contiene un error.
③ En caso de error, es necesario manejarlo, por ejemplo, cerrando y liberando recursos reservados. Después se propaga el código de error –el objeto `std::error_code` en `std::expected`– al invocador de la función. Este objeto se puede obtener llamado al método `error()` del objeto `std::expected`.
④ Si todo ha ido bien, se puede usar el método `value()` para acceder al descriptor de archivo `SafeFD` almacenado en el objeto `std::expected`. Al hacerlo **es importante usar `std::move()` para forzar que el objeto `SafeFD` dentro del objeto `std::expected` se mueva a la variable `fd`**, puesto que hemos prohibido las copias de `SafeFD`. Después de esta operación, el objeto `SafeFD` al que se tiene acceso mediante `result.value()` contendrá un descriptor de archivo inválido.
⑤ El descriptor de archivos se puede usar para leer o escribir en el archivo, entre otras operaciones.

Como los tipos de retorno de las funciones con `std::expected` pueden tener nombres muy largos que dificultan la legibilidad, puede ser conveniente crear alias:

```
using open_file_result = std::expected<SafeFD, std::error_code>;          ①

open_file_result open_file(const std::string& path, int flags,            ②
    mode_t mode = 0);
{
    // ...
```

```
}

// ...

open_file_result result = open_file("test.txt", flags);
if (result)
{
    SafeFD fd = std::move(result.value());                    ③

    // ...
}
```

① Definir un alias para el tipo retornado por `open_file()`.
② Usar el alias para definir la función `open_file()` y para crear un variable en la que guardar el objeto `std::expected` retornado.
③ Es importante usar `std::move()` para forzar que se mueva el descriptor de archivo dentro de `result` a `fd`. Después de esta operación, el objeto `SafeFD` al que se tiene acceso mediante `result.value()` contendrá un descriptor de archivoea inválido.

O, mejor, añadimos el atributo `nodiscard` para que no nos olvidemos de guardar el resultado de la función:

```
using open_file_result = std::expected<SafeFD, std::error_code>;

[[nodiscard]]
open_file_result open_file(const std::string& path, int flags,
    mode_t mode = 0);
```

## 5.3. Excepciones de C++

En C++ y otros lenguajes con orientación a objetos, la forma más común de propagar errores es mediante excepciones. Si estás familiarizado con este concepto y lo prefieres, **puedes utilizar excepciones para gestionar los errores en la práctica**. En caso contrario, puedes ignorar el resto de este apartado.

Para utilizar excepciones solo necesitas saber que podemos lanzar una excepción para un código de error de `errno` concreto:

```
throw std::system_error(errno, std::system_category(),
    std::string("__FILE__") + "#" + std::to_string(__LINE__));
```

Las excepciones `std::system_error`[3] guardar el código de error `errno` en el objeto y permiten obtener un mensaje de texto descriptivo del error con el método `what()`. Este mensaje descriptivo es equivalente al que se obtiene con `std::strerror(errno)`.

[3] `std::system_error` se declara en `<system_error>`.

El tercer argumento del constructor de `std::system_error` es un mensaje de texto que precede al mensaje descriptivo del código de error. En el ejemplo se construye con el nombre del fichero de código fuente –usando el valor de la macro `__FILE__`– y el número de línea –usando el valor de `__LINE__`– donde se ha producido el error. Esto puede ser muy útil para localizar rápidamente los errores al depurar el programa.

Para obtener el mensaje descriptivo completo del error, podemos usar el método `what()` del objeto de la excepción al capturarla:

```
try
{
    read_file(int fd, buffer);
}
catch (std::system_error& e)
{
    std::cerr << "Error: " << e.what() << '\n';
}
```

## 6. Argumentos de la línea de comandos

Como ya sabemos, los argumentos de la línea de comandos se reciben en `main()` mediante `argv`, un array de punteros a `char`:

```
int main(int argc, char* argv[])
{
    // ...
}
```

Para facilitar su procesamiento, `argv` se puede convertir en un vector de `std::string_view`:

```
int main(int argc, char* argv[])
{
    std::vector<std::string_view> args(argv + 1, argv + argc);

    // ...
}
```

Usamos `std::string_view` porque nos ofrece una funcionalidad similar a la de `std::string` pero sin hacer una copia de cada cadena. Es decir, los argumentos de la línea de comandos están en `argv`, pero accedemos a ellos a través de los objetos `std::string_view` en `args`

Con este vector, es muy sencillo iterar para procesar los argumentos de la línea de comandos uno tras otro:

```
for (auto it = args.begin(), end = args.end(); it != end; ++it)
{
    if (*it == "-h" || *it == "--help")
    {
        show_help = true;
    }

    if (arg == "-o" || arg == "--output")
    {
        if (++it != end)
        {
            output_filename = *it;
        }
        else
        {
            // Error por falta de argumento...
        }
    }

    // ...
}
```

> **Nota**
>
> Obviamente, es mucho más sencillo usar un *for-range* de la forma `for (auto& arg: args)`, pero nos dará problemas si tenemos opciones que van seguidas de un argumento, como es el caso de `-o` en el ejemplo anterior.

## 6.1. Función `parse_args()`

Para separar mejor la responsabilidad, se puede meter el código anterior en una función `parse_args()` que procese los argumentos de la línea de comandos y devuelva una estructura con los valores de las opciones encontradas:

```
enum class parse_args_errors                                    ①
{
    missing_argument,
    unknown_option,
    // ...
};

struct program_options                                          ②
{
    bool show_help = false;
```

```
    std::string output_filename;
    // ...
};

std::expected<program_options, parse_args_errors>                          ③
parse_args(int argc, char* argv[])
{
    std::vector<std::string_view> args(argv + 1, argv + argc);
    program_options options;

    for (auto it = args.begin(), end = args.end(); it != end; ++it)
    {
        if (*it == "-h" || *it == "--help")
        {
            options.show_help = true;
        }
        else if (*it  == "-o" || *it  == "--output")
        {
            if (++it != end)
            {
                options.output_filename = *it;
            }
            else
            {
                return std::unexpected(
                    parse_args_errors::missing_argument);                  ④
            }
        }
        // Opciones adicionales...
        else
        {
            return std::unexpected(
                parse_args_errors::unknown_option);                        ⑤
        }
    }

    return options;                                                        ⑥
}
```

① Enumeración con los posibles errores al procesar los argumentos de la línea de comandos.
② Estructura con las opciones admitidas por el programa.
③ La función `parse_args()` devuelve un `std::expected` que puede contener una estructura `program_options` en caso de éxito o un `parse_args_errors` con el motivo del error en caso de fallo.
④ En caso de que se indique la opción `-o` o `--output` pero no se indique el nombre del

archivo de salida, se devuelve `parse_args_errors::missing_argument` para notificar el error a la función que ha invocado a `parse_args()`. Este valor se marca como `std::unexpected` para indicar que se ha producido un error.

⑤ Si se encuentra una opción desconocida, se devuelve el código de error `parse_args_errors::unknown_option`. Este valor se marca como `std::unexpected` para indicar que se ha producido un error.

⑥ En caso de éxito, se devuelve la estructura `program_options` con las opciones encontradas y sus argumentos

La función `parse_args()` devuelve un `std::expected` para poder devolver la estructura de datos `program_options` en caso de éxito o un *enum* `parse_args_errors` con el código de error en caso de fallo.

Este función se puede invocar fácilmente desde `main()`:

```
int main(int argc, char* argv[])
{
    auto options = parse_args(argc, argv);                                    ①
    if (! options.has_value())                                                ②
    {
        // Usar options.error() para comprobar el motivo del error...
        if (options.error() == parse_args_errors::missing_argument)           ③
        {
            // Mostrar mensaje de error por falta de argumento...
        }
        else if (options.error() == parse_args_errors::unknown_option)
        {
            // Mostrar mensaje de error por opción desconocida...
        }

        // ...

        return EXIT_FAILURE;                                                  ④
    }

    // Usar options.value() para acceder a las opciones...
    if (options.value().show_help)                                            ⑤
    {
        print_usage();
    }

    // ...

    return EXIT_SUCCESS;
}
```

① Llamar a `parse_args()` para procesar los argumentos de línea de comandos en `argc` y `argv`. El objeto devuelto en `options` es de tipo `std::expected<program_options, parse_args_errors>`
② Comprobar si `parse_args()` terminó con éxito, comprobado el objeto `std::expected` devuelto. Si `options.has_value()` devuelve `false`, es que ha habido un error y lo que guardar `options` no es un `program_options` sino un `parse_args_errors`.
③ Comprobar el motivo del error para mostrar un mensaje adecuado.
④ En caso de error, terminar el programa con un código de salida diferente de 0.
⑤ En caso de éxito al procesar los argumentos de la línea de comandos, se accede a la estructura `program_options` con `options.value()`. Como se ilustra en el ejemplo, así se puede acceder a `program_options::show_help` para comprobar si el usuario indicó que quería leer la ayuda del programa.

## 7. Evita el mal uso de los espacios de nombres

Aún hoy en día es frecuente encontrar en libros, blogs o en webs, como Stack Overflow, ejemplos similares al siguiente, en cuanto al uso de los espacios de nombres:

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;                                                    ①
5
6  int main(int argc, char* argv[])
7  {
8      cout << "¡Hola, mundo!\n";
9      return EXIT_SUCCESS;
10 }
```

① Indicar de forma global que no queremos que haga falta poner `std::` para acceder a elementos de la librería estándar de C++.

El uso de `using namespace std` de forma global –tal y como se puede observar en la línea 4 del ejemplo anterior– es una mala práctica, según la comunidad de desarrolladores de C++.

Los espacios de nombre están para evitar la colisión de nombres entre clases y funciones. Cuanto más complejo es nuestro programa, más probable es que estas colisiones ocurran, de formar que lo mejor es usar simplemente `std::` –y otros espacios de nombre– donde sea necesario:

```
#include <cstdlib>
#include <iostream>

int main(int argc, char* argv[])
```

```
{
    std::cout << "¡Hola, mundo!\n";
    return EXIT_SUCCESS;
}
```

Un ejemplo de este problema lo ilustran perfectamente la función de la librería de sistema `bind()` –en sistemas que soportan *sockets*– y la función de la librería estándar de C++ `std::bind()`. Cuando se usa `using namespace std` y se invoca a `bind()` en alguna parte del código, el compilador puede acabar llamando a una función diferente a la que nos interesaba. Por eso es preferible ser explícitos con los espacios de nombres.

> **Nota**
>
> En todo caso, puede ser buena idea poner tu código en un espacio de nombres propio, para evitar conflictos con las funciones de otras librerías. Por ejemplo, puedes meter tu propio código en el espacio de nombres `exercise` o `project`.