

# Programación de aplicaciones — Excepciones

Sistemas Operativos 2024/2025

Jesús Torres

Vamos a ver como manejar los errores de las librerías del sistema usando excepciones en C++.

## Contenidos

|   |   |
|---|---|
| 1. Cuándo usar excepciones              | 1 |
| 2. Manejo de errores con excepciones    | 2 |
| 3. Patrón <code>protected_main()</code> | 3 |

## 1. Cuándo usar excepciones

En C++ y otros lenguajes con orientación a objetos, una forma común de gestionar errores es mediante excepciones. Si embargo, en C++, actualmente, se suele preferir que las excepciones se utilicen solo para gestionar situaciones realmente excepcionales, es decir, para errores realmente serios, poco frecuentes o inesperados.

Por ejemplo, no poder abrir un fichero porque no existe o no poder conectar con un servidor remoto mediante *sockets*, no son situaciones excepcionales, sino errores comunes, que se pueden prever y gestionar de forma más sencilla. Por tanto, la recomendación es que estos errores se gestionen retornando clases de error con la ayuda de `std::expected`.

Sin embargo, no poder leer un fichero porque se ha producido un error de lectura o no poder escribir porque no hay suficiente espacio de almacenamiento disponible, si son situaciones excepcionales, que no suelen ocurrir en los sistemas modernos. Por tanto, en estos casos, se puede utilizar excepciones para gestionar los errores.

En todo caso, la decisión de usar excepciones o no, es una cuestión de estilo y preferencia del equipo de desarrollo. Por nuestra parte, si estás familiarizado con las excepciones y lo prefieres, **puedes utilizarlas para gestionar los errores en la práctica.**

## 2. Manejo de errores con excepciones

Para utilizar excepciones cuando se gestionan los errores de las librerías del sistema, solo necesitas saber que puedes lanzar una excepción para un código de error de `errno` concreto así:

```
throw std::system_error(errno, std::system_category(), "Error en foo()");
```

Por ejemplo, en `read_file()` se puede lanzar una excepción de tipo `std::system_error` con el valor de `errno` cuando `read()` falla:

```
ssize_t read_file(const SafeFD& fd, std::vector<uint8_t>& buffer,
    size_t max_size)
{
    // ...

    ssize_t bytes_read = read(fd.get(), buffer.data(), max_size);
    if (bytes_read < 0)
    {
        throw std::system_error(errno, std::system_category(),
            "Error en read()");
    }

    // ...
}
```

Esta excepción se puede capturar en la función que llama a `read_file()` y manejarla adecuadamente:

```
try
{
    read_file(fd, buffer);
}
catch (std::system_error& e)
{
    std::println(std::cerr, "Error ({}): {}", e.code().value(), e.what());
}
```

Las excepciones `std::system_error`<sup>1</sup> guardan el código de error `errno` en el objeto. Llamando al método `code()` del objeto de la excepción –la variable `e` en el ejemplo anterior– se obtiene el objeto `std::error_code` que contiene el código de error con el que se lanzó la excepción. Dicho código se puede obtener con el método `value()` del objeto `std::error_code` devuelto por `code()`:

---

<sup>1</sup>`std::system_error` se declara en `<system_error>`.

```
try
{
    read_file(fd, buffer);
}
catch (std::system_error& e)
{
    int errno_value = e.code().value();
    std::println(std::cerr, "Código de error {}", errno_value);
}
```

- ① Obtener el valor de `errno` con el que se ha lanzado la excepción.

Además, el objeto de la excepción `e` guarda un mensaje de texto descriptivo del error que se puede obtener con el método `what()`. Este mensaje descriptivo es equivalente al que se obtiene con `std::strerror(errno)`, pero al que se le añade la cadena indicada en el tercer argumento del constructor de `std::system_error`.

Este tercer argumento puede ser útil para construir un mensaje donde se indique la línea y el fichero donde se ha producido el error, lo que puede facilitar la localización del problema durante el desarrollo:

```
throw std::system_error(errno, std::system_category(),
    std::format("Lanzado desde '{}' línea {}", __FILE__, __LINE__));
```

- ① Crear un mensaje de error más informativo para la excepción, con el nombre del fichero –usando el valor de la macro `__FILE__`– y el número de línea –usando el valor de `__LINE__`– donde se ha lanzado la excepción.

```
try
{
    read_file(fd, buffer);
}
catch (std::system_error& e)
{
    std::println(std::cerr, "Error ({}): {}",
        e.code().value(), e.what());
}
```

- ① Imprimir el mensaje de error de la excepción. En esta caso la cadena devuelta por `what()` incluye el nombre del fichero y el número de línea donde se ha lanzado la excepción.

### 3. Patrón `protected_main()`

Un patrón muy común para hacer el código más legible es separar el manejo de excepciones del resto del código de la función `main()`. Básicamente, consiste en extraer el có-

odigo de la aplicación que puede lanzar excepciones y ponerlo en una función llamada `protected_main()`:

```
int protected_main(int argc, char* argv[])
{
    // Código de la aplicación...

    read_file(int fd, buffer);

    // Más código de la aplicación...

    return EXIT_SUCCESS;
}
```

Y luego crear una función `main()` con el bloque `try-catch` que llame a `protected_main()` dentro de `try` y gestione las excepciones en `catch`:

```
int main(int argc, char* argv[])
{
    try
    {
        return protected_main();
    }
    catch(std::system_error& e)
    {
        std::println(std::cerr,
            "Error ({}): {}", e.code().value(), e.what());
    }
    catch(std::exception& e)
    {
        std::println(std::cerr,
            "Error: Excepción: {}", e.what());
    }

    return EXIT_FAILURE;
}
```

Esto permite programar libremente en `protected_main()` las tareas y funcionalidades de nuestra aplicación, sabiendo que cualquier excepción será interceptada y tratada adecuadamente en `main()`, antes de salir del programa.