

# Tema 9: Excepciones. Patrón RAII.

## Definicion.

- Las *excepciones* se emplean en situaciones *excepcionales* que se producen en tiempo de ejecución y que no se pueden conocer en tiempo de compilación, cuando estamos desarrollando nuestra aplicación o biblioteca.
- Por tanto no se deben confundir con *errores en la lógica* de nuestro código, estos son *otra cosa totalmente distinta*.

Simply put, an Error is a bug in the application. An Exception is a bug in the input to the application. The former is not recoverable, the latter is.

— Walter Bright, D Language creator.

## Tratamiento clásico de los errores (asertos).

### Errores en la lógica del código.

- La detección de errores de lógica en nuestro código ya la conocemos de *Programación I*, consiste en el uso de `assert`.
- `assert(condition)` es una macro que evalúa un predicado, si este es `cierto`, el código sigue ejecutándose, pero si es `falso` se aborta la ejecución de la aplicación inmediatamente en ese punto, informándonos de *dónde* ha fallado la aserción y por qué motivo.
- Para hacer uso de `assert` desde **C++** incluiremos la cabecera `cassert`.

```
#include <iostream>
#include <cassert>

double fastSqrt (double x) {
```

```
}
```

## Pero todo esto se puede hacer con sentencias `if` , ¿no?

- Por ejemplo, así:

```
#include <iostream>

double fastSqrt (double x) {
    if (not (x >= 0.0) ) ... ;
    ...
    if (not (x == (result*result)) ... ;
    return result;
}
```

- ¿Dónde está la ventaja entonces de `assert` ?
- Podemos deshabilitar fácilmente el chequeo de asertos en código libre de errores de lógica definiendo al compilar la macro `NDEBUG` , p.e. así: `g++ -DNDEBUG example.cc` .

## Tratamiento de errores mediante el uso de excepciones.

- Son la alternativa al tratamiento de errores consistente en devolver un simple código de error (valor numérico).
- Aportan, por tanto, más información al punto de nuestro código donde se recibe el error.
- Pero, sobre todo en el paradigma de POO, donde podemos tener varios objetos vivos en el punto donde se produce el error y que por tanto deberían desaparecer liberando toda la memoria que pudieran tener reservada, nos permite automatizar este proceso.
- Para ello nos permiten deshacer la pila de llamadas, invocando los destructores de los objetos implicados; y ya sabemos lo que podemos/debemos hacer en el destructor de un

- **C++** (y lenguajes como *Java*, *C#*, *Vala*, *D*, etc...) emplean una sintáxis similar.
- En **C++** se emplean las palabras reservadas:
  - `try`
  - `throw`
  - `catch`
- Un ejemplo muy sencillo:

```
#include <iostream>

double f(double d) {
    if (d > 1e7)
        throw std::overflow_error("too big");
    else
        return d;
}

int main() {
    try {
        std::cout << f(1e10) << '\n';
    } catch (const std::overflow_error& e) {
        std::cout << e.what() << '\n';
    }
}
```

## Características especiales de las excepciones en C++.

- Se permite lanzar excepciones de tipos *base* del lenguaje, p.e., `char` , `int` , etc...
- Se puede indicar la lista de excepciones que puede lanzar una función en **C++** pre C++14:  
`void f() throw(int);`

- Aquí tienes más información sobre [noexcept](#).
- Si se lanza una excepción no especificada en la lista de posibles excepciones, se llama a `std::unexpected()` .
- Por defecto `unexpected` llama a `std::terminate` y el programa acaba.
- Se puede cambiar la función a la que llama, p.e.:

```
#include <exception>      // std::set_unexpected
void myunexpected () {
    std::cerr << "unexpected called\n";
}
int main() {
    std::set_unexpected(myunexpected);
    ...
}
```

- Si una función no especifica una lista vacía de excepciones o `noexcept` , se asume que puede lanzar cualquier excepción.
- A un bloque *try* le podemos asociar varios bloques *catch*:

```
int main (void) {
    std::set_unexpected (myunexpected);
    try {
        myfunction();
    }
    catch (int) { std::cerr << "caught int\n"; }
    catch (...) { std::cerr << "caught some other exception type\n"; }
    return 0;
}
```

- **IMPORTANTE:** Los bloques *catch* se evalúan por orden, por tanto si el tipo de un *catch* concuerda con el tipo de la excepción lanzada, entonces se trata en ese *catch*.

```
f (); // throws ExceptionDerived
}
catch (ExceptionBase& eb) {...} // <-- se captura aquí
catch (ExceptionDerived& ed) {...}
```

- Los bloques *catch* con elipsis (...): `catch (...)` capturan cualquier tipo de excepción, por eso se ponen al final.
- Una excepción se puede relanzar:

```
try {
    f (); // throws ExceptionDerived
}
catch (ExceptionBase& eb) { throw; } // <-- se relanza aquí
```

## Jerarquía de clases de excepciones.

- **C++** nos proporciona una serie de clases predefinidas que sirven para representar tipos de error en nuestro código.
- Debemos incluir la cabecera `<stdexcept>` (la cual incluye a `<exception>`).
- De este modo tenemos acceso a las clases `exception` y derivadas de ella como `logic_error`, `length_error`, etc...
- Todas ellas tienen un constructor a partir de una cadena, p.e.:  
`runtime_error( const std::string& what_arg );`
- En esta cadena podemos aprovechar para poner una descripción del error.
- Esta cadena puede ser consultada con el método heredado de la clase `exception`:  
`virtual const char* what() const noexcept;`

## Patrón RAI.

```

class file {                                     // https://es.wikipedia.org/wiki/RAII
public:
    file (const char* filename) : file_ (std::fopen(filename, "w+")) {
        if (!file_) {
            throw std::runtime_error ("Error al abrir un archivo.");
        }
    }

    ~file() {
        if (std::fclose(file_)) {
            throw std::runtime_error ("Error al cerrar un archivo.");
        }
    }

    void write (const char* str) {...}
}

```

```

void main () {
    //////////////////////////////////////
    // Abrimos el archivo, estamos adquiriendo un recurso //
    //////////////////////////////////////
    file logfile ("registro_de_incidentes.txt");

    logfile.write ("¡Hola, registro!");
} // <- Al salir de ámbito, en el destructor de logfile se cierra el fichero

```

## Patrón RAII y punteros inteligentes.

- El uso del patrón RAII ha dado lugar a la introducción en **C++** de los llamados [punteros inteligentes](#) o *smart pointers*.
- Ya no será necesario ocuparse de la liberación de memoria reservada dinámicamente, juzga tú mismo:

```
// compilar con C++11
```

```
// creeras lo que ven tus ojos...  
std::unique_ptr<int> pe (new int[1024]);  
// compara con una línea como esta:  
// int* pe = new int[1024];  
  
return 0;  
}
```

## Patrón RAII y excepciones.

- Si echas un vistazo a cómo otros LOO implementan el tratamiento de excepciones, verás que se parece mucho al de C++ (java, C#, Vala, D).
- En casi todos ellos observarás que aparece una palabra reservada nueva: `finally`.
- La introdujo [Java](#) al no ser posible implementar de forma correcta el patrón RAII en este lenguaje ya que las variables que representan objetos siempre son *referencias* y no objetos completos.
- Los bloques *finally* se ejecutan siempre, tanto si se sale del bloque *try* de forma correcta o por que se ha lanzado una excepción y hemos entrado en un *catch*.

```
try { ... }  
catch (et1) {...}  
catch (et2) {...}  
finally { ... }
```

## ¿Existe algún otro modo de tratar los errores?

- Sí, claro. Determinados lenguajes de programación han elegido un modelo diferente de tratamiento de errores.
- Veamos el caso de [Rust](#).
- En *Rust* existe este enumerado:

- Podemos ver `Ok(T)` y `Err(E)` como sendas funciones que crean un dato de tipo `Result` a partir de un dato de tipo:
  - `T` si *hay resultado*, `T` es el tipo del resultado
  - `E` si *se produce* un error, `E` es el tipo del error

## Ejemplo de tratamiento de errores en Rust.

```
fn mayor_o_menor_0 (n: i32) -> Result<bool, &'static str> {
    if n < 0 || n > 0 {
        Ok(true)
    } else {
        Err("n es 0")
    }
}

fn main() {
    let r = mayor_o_menor_0(0);

    assert!(r.is_err())
}
```

Puedes ampliar tus conocimientos sobre este tema consultando este [artículo](#).

## Aclaraciones.

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la [página web de la ficha de la asignatura](#) y en la web propia de la asignatura.



En esta página >	Sinopsis
	de otros LOO.