

# **Excepciones en C++**

Antonio Espín Herranz

# Excepciones

- C++ incorpora un soporte para poder manejar situaciones anómalas durante la ejecución de un programa.
- Con las excepciones podemos verificar errores de una forma limpia.
- El código será algo así:

```
try {  
    // Código de la aplicación  
} catch (excepcion1 e){  
    // Tratamiento ...  
} catch (excepcion2 e){  
    // Tratamiento ...  
}
```

# Excepciones

- El manejo de excepciones reduce la complejidad del código.
- Los métodos que invocan a otros no necesitan comprobar valores de retorno.
- Tener en cuenta una división por cero NO es estándar en C++. Hay que programarla.

# Excepciones en C++

- **bad\_alloc** en <new>, lanzada por new cuando no hay memoria suficiente.
- **bad\_cast** en <typeinfo>, lanzada por dynamic\_cast cuando no es posible realizar la conversión.
- **bad\_typeid** en <typeinfo>, lanzada por typeid cuando su argumento es 0 o una dirección no válida.
- **bad\_exception** en <exception> lanzada cuando ocurre un error no esperado.
- Estas son excepciones estándar lanzadas por el lenguaje.

# Excepciones en C++

- **out\_of\_range**: en `<stdexcept>` lanzada para informar que el valor de un argumento está fuera del rango.
- **invalid\_argument**: en `<stdexcept>` para informar de un argumento no válido.
- **overflow\_error**: en `<stdexcept>` para informar de un desbordamiento aritmético.

# Excepciones

- Las excepciones en C++ son objetos de clases derivadas de la clase **exception** definida en el espacio de nombres **std**.
- Con **using namespace std**;
  - Podemos utilizar directamente **exception**.
  - Sin using → `std::exception`
- O pueden ser clases definidas por el usuario.
- Nosotros vamos a poder definir nuestras propias clases.

# Excepciones

- Las excepciones implementan un método **what()** que devuelve un string indicando un mensaje del error producido.

```
try {  
    // Código a evaluar ...  
} catch (bad_alloc& e){  
    cout << "ERROR: " << e.what() << endl;  
}
```

# Manejar excepciones

- Cuando un método encuentra alguna anomalía lo lógico es que **lance (throw)** una excepción, esperando que el método que lo llamó directa o indirectamente lo **capture (try / catch)**.
- Lanzar una excepción implica crear un objeto de la clase de la excepción.
- Se pueden lanzar excepciones de cualquier clase incluso de datos primitivos.



# Excepciones derivadas

- Un bloque try puede estar seguido de varios bloques catch ,tantos como excepciones diferentes tengamos que manejar.
- Cada **catch** tendrá un parámetro de la clase exception, de alguna clase derivada de esta o de alguna clase definida por el usuario.

# Orden de la excepciones

- Tenemos que tener en cuenta el orden de colocación de las excepciones en los bloques catch.
- Sabemos por la relación de herencia que de forma implícita un objeto de la clase derivada puede ser convertido en un objeto de la clase base.
- Esto mismo ocurre con las excepciones.

# Orden de la excepciones 2

- Teniendo en cuenta esto si tenemos varios catch y en el primero colocamos un parámetro de la clase exception **no se alcanzará ningún bloque catch que haga referencia a alguna de las clases derivadas de exception.**
- Por lo que si tenemos que indicar varios bloques catch tendremos que colocar primero lo mas particular y después lo mas general, es decir, la clase exception debería estar en última posición.

# Ejemplo

```
try {  
    // ...  
} catch (out_of_range& e){  
    // Manejar la excepción ...  
  
} catch (logic_error& e){  
    // Manejar esta exception.. .  
  
} catch (exception& e){  
    // ...  
}
```

# Capturar cualquier excepcion

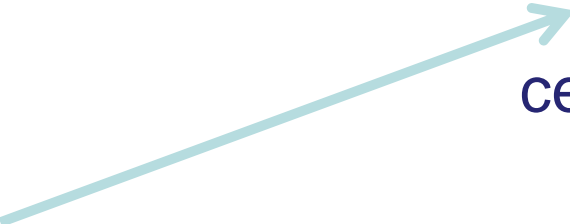
- Tres puntos suspensivos indica cualquier parámetro.
- `catch(...)` → indica capturar cualquier exception.
  - *En este caso podríamos capturar excepciones del tipo:*
  - *throw “un mensaje”*
  - *throw 23;*

```
try {  
  
} catch (exception& e){  
    // ...  
  
} catch (...) {  
    //...  
}
```

# Lanzar excepciones de otros tipos

```
#include <iostream>
using namespace std;
void foo()
{
    int i;
    i = -15;
    throw i;
}
```

```
int main(){
    try {
        foo();
    } catch(int n) {
        cerr << "exception caught\n"
              << n << endl; }
}
```



# Relanzar una exception

- Otras veces puede interesarnos relanzar una exception.
- Lo haremos con el parámetro throw; sin indicar parámetros.
- Por ejemplo que no podamos tratar el error en ese momento, nos puede interesar relanzarla para que sea capturada por otro método en la pila de llamadas.

# Ejemplo

```
#include <iostream>

using namespace std;

void myFunction() {

try {
    throw "hello";
} catch(const char *) {
    cout << "Caught char * inside myFunction\n";
    throw ;
}
}
```

```
int main(){
    cout << "Start\n";

    try{
        myFunction();
    }
    catch(const char *) {
        cout << "Caught char * inside main\n";
    }

    cout << "End";

    return 0;
}
```



# Crear excepciones

- C++ nos proporciona la posibilidad de crear nuestras propias excepciones.
- Las nuevas excepciones que creemos no tienen porque derivar de la clase exception.
- Mi excepción tendrá un atributo string para indicar el tipo de error producido y un método what() para poder obtener el mensaje.
- Podemos crear nuestras propias jerarquías de excepciones.

# Definición de la clase exception

- En la STL de C++ la clase exception se define:

```
class exception {  
    public: exception() throw() { }  
    virtual ~exception() throw();  
    virtual const char* what() const throw();  
};
```

# Ejemplo: sobrescribir what()

// Demostración: sobrescritura del método what()

```
#include <iostream>
#include <cstdlib>
#include <exception>
using namespace std;
```

```
class div_cero : public exception {
    public: const char* what() const throw() {
        return "Error: división por cero...";
    }
};
```

```
int main(int argc, char *argv[]) {
    double N, D;
    cout << "Probando división" << endl;
    cout << "Ingrese el numerador :";
    cin >> N;
    cin.clear();
    cout << "Ingrese el denominador :";
    cin >> D;
    cin.clear();
    try {
        if (D == 0) throw div_cero();
        cout << N << " / " << D << " = " <<
            N/D << endl;
    } catch(exception& e) {
        cout << e.what() << endl;
    }
    return 0;
}
```

# Ejemplo: Crear Exception

```
class EValorNoValido {  
    private:  
        string mensaje;  
    public:  
        EValorNoValido(string msg = "..."){  
            mensaje = msg;  
        }  
  
        string what(){ return mensaje; }  
}
```

# Ejemplo: Lanzar Exception

- En alguna parte del código tendremos algo así:

```
if ( condición)
```

```
    throw EValorNoValido("ERROR: se ha producido  
    ...");
```

- En otro lugar de nuestro código tendremos el try  
{ } catch (EValorNoValido& e) para dar un  
tratamiento a nuestra exception.

# Especificar excepciones

- Si una función lanza excepciones puede declararlo.
- Para que otros usuarios puedan saber que excepcion pueden saltar.
- `void miFuncion() throw(EValorNoValido);`
- Así indicamos que la función puede lanzar excepciones del tipo: `EValorNoValido` o de sus clases derivadas.

# Declaraciones

- `void f();` // La función puede lanzar cualquier excepción.
- `void f() throw ();` // La función no lanza excepciones.
- `void f() throw(X);` // La función sólo lanza excepciones del tipo X.
- `void f() throw(X, Y);` // La función sólo lanza excepciones del tipo X e Y.

# Excepciones no esperadas

- ¿Qué ocurre cuando se lanza una excepción imprevista?
- En este caso se invoca a la función **std::unexpected** que a su vez invoca a la función especificada en **set\_unexpected** que por omisión invoca `std::terminate` → abort.
- Podemos modificar el tratamiento de `unexpected`.



# Ejemplo

- Se fuerza a unexpected que lance bad\_exception que la podremos capturar y tratar.

- Se indica en la función:

```
void f() throw (EValorNoValido, bad_exception);
```

```
void exception_inesperada(){  
    throw std::bad_exception();  
}
```

```
void f() throw (EValorNoValido, bad_exception){  
    // Modificamos el comportamiento de la exception unexpected.  
    set_unexpected(exception_inesperada);
```

```
    // El código de mi función puede lanzar excepciones.  
    throw EValorNoValido(" mensaje ");
```

```
    throw "excepción inesperada";  
}
```

PRACTICA  
EXCEPTION