

TEMA 5

GESTIÓN DE ERRORES

Cristina Cachero, Pedro J. Ponce de León

1 Sesión (1.5 horas)

Versión 0.5



Gestión Errores

Objetivos



- **Saber utilizar `try`, `throw` y `catch` para observar, indicar y manejar excepciones, respectivamente.**
- **Comprender la jerarquía de excepciones estándar.**
- **Ser capaz de crear excepciones personalizadas**
- **Ser capaz de procesar las excepciones no atrapadas y las inesperadas.**

Gestión de Errores

Motivación



- Realizar un programa que solicite dos números y visualice la división de ambos.

```
int main()
{
    float dividendo, divisor, resultado;
    cout << "PROGRAMA DIVISOR" << endl;
    cout << "Introduce Dividendo : " ;
    cin >> dividendo;
    cout << "Introduce Divisor : " ;
    cin >> divisor;
    resultado = dividendo / divisor;
    cout << dividendo << "/" << divisor << "=" << resultado;
    return (0);
}
```

- ¿Qué ocurre si el usuario introduce un divisor=0?

Gestión de Errores

Motivación



- Esto obliga a definir un esquema de programación similar a :
 - Llevar a cabo tarea 1
 - *Si se produce error*
 - *Llevar a cabo procesamiento de errores*
 - Llevar a cabo tarea 2
 - *Si se produce error*
 - *Llevar a cabo procesamiento de errores*
- ¿Qué problemas podéis detectar en este esquema?
 - Entremezcla la lógica del programa con la del tratamiento de errores (disminuye legibilidad)
 - Puede degradar el rendimiento del sistema
- ¿Qué podemos hacer en lugar de crear código *spaguetti*?:
 - Abortar el programa
 - ¿Y si el programa es crítico?
 - Usar indicadores de error (Ej Una función devuelve un código de error.)
 - ¿Se comprueban siempre?
 - USAR EXCEPCIONES

Gestión de Errores

Excepciones: Concepto



- Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias
- Muchas clases de errores pueden utilizar excepciones -- desde serios problemas de hardware, como la avería de un disco duro, a los simples errores de programación
 - División por cero
 - Acceso a un índice incorrecto de un tipo de las STL
 - Fallo en la reserva de memoria del *new*
 - ...
- Las excepciones pueden ser ignoradas si conviene.

Gestión de Errores

Excepciones: Sintaxis



C++

```
try
{
    // Código de ejecución normal
    throw Tipo1();
}
catch (Tipo1 &ex)
{
    // Gestión de excep tipo 1
}
catch (Tipo2 &ex)
{
    // Gestión de excep tipo 2
}
catch (...)
{
    /* Gestión de cualquier excep no
       capturada mediante los catch
       anteriores*/
}
//Continuación del código
```

JAVA

```
try
{
    // Código de ejecución normal
    throw new Tipo1();
}
catch(Tipo1 ex)
{
    // Gestión de excep tipo 1
}
catch(Tipo2 ex)
{
    // Gestión de excep tipo 2
}
finally{
    // se ejecuta siempre
}
```

Gestión de Errores

Excepciones: Sintaxis



- En C++ y en Java:
 - El bloque **try** contiene el código que forma parte del funcionamiento normal del programa
 - El bloque **catch** contiene el código que gestiona los diversos errores que se puedan producir

- Sólo en JAVA:
 - El bloque **finally** de Java proporciona un mecanismo que permite a sus métodos limpiarse a sí mismos sin importar lo que sucede dentro del bloque **try**. Se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema. El bloque finally se puede ejecutar (1) tras finalizar el bloque try o (2) después de las cláusulas catch.

Gestión de Errores

Excepciones: Sintaxis



- Funcionamiento:

- 📁 Ejecutar instrucciones try

- 👉 Si hay error, interrumpir el bloque try e ir a bloque catch correspondiente (*)

- 📁 Continuar la ejecución después de los bloques catch

- La excepción es capturada por el bloque-**catch** cuyo argumento coincida con el tipo de objeto lanzado por la sentencia **throw**. La búsqueda de coincidencia se realiza sucesivamente sobre los bloques **catch** en el orden en que aparecen en el código hasta que aparece la primera concordancia.

- Implica que **el orden de colocación de los bloques catch es determinante**. Por ejemplo: si se incluye un manejador universal, éste debería ser el último.

- En caso de no existir un manejador adecuado a una excepción determinada, se desencadena un protocolo que, por defecto, produce sin más la finalización del programa

Gestión de Errores

Excepciones: lanzamiento



- La cláusula throw lanza la excepción como un objeto
- Ejemplo

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

```
class miExcepcion {
    string queHaPasado() const
    {return "Ocurrió mi excepción"; }
};
```

```
int main() {
    try{ throw miExcepcion(); }
    catch(miExcepcion &e){ cout<<e.queHaPasado() <<endl; }
    return (0);
}
```

¿Qué ocurriría si catch
recibiese el parámetro
por valor?



Excepciones: especificación de soporte

- En C++ existe una opción denominada **especificación de excepción** que permite señalar que tipo de excepciones puede lanzar una función directa o indirectamente (en funciones invocadas desde ella). Este especificador se utiliza en forma de sufijo en la declaración de la función y tiene la siguiente **sintaxis**:

```
throw (<lista-de-tipos>) // lista-de-tipos es opcional
```

- La ausencia de especificador indica que la función puede lanzar cualquier excepción.

Gestión de Errores

Excepciones estándares en C++



- Todas las excepciones lanzadas por componentes de la Librería Estándar de C++ son excepciones derivada de la superclase `exception`, definida en la cabecera **<exception>**, que tiene la siguiente interfaz:

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

- Esta clase tiene cinco métodos públicos, ninguno de los cuales puede lanzar una excepción (cláusula `throw()`).
- Además, la clase `exception` proporciona una función miembro virtual **what()**, que devuelve un `const char *` con un mensaje verbal (dependiente del compilador que estéis utilizando) que refleja el tipo concreto de excepción.
 - Este mensaje puede ser sobrescrito en clases derivadas para contener una descripción personalizada de la excepción.

Gestión de Errores

Excepciones estándares en C++



■ Tipos de excepciones (I):

■ logic_error:

- domain_error
- invalid_argument
- length_error
- out_of_range

■ runtime_error

- range_error
- overflow_error
- underflow_error

```
class logic_error : public exception {  
    public:  
        explicit logic_error (const string& what_arg);  
};  
class domain_error : public logic_error {  
    public:  
        explicit domain_error (const string& what_arg);  
};
```

```
class runtime_error : public exception {  
    public:  
        explicit runtime_error (const string& what_arg);  
};  
class range_error : public runtime_error {  
    public:  
        explicit range_error (const string& what_arg);  
};
```

Gestión de Errores

Excepciones estándares en C++



- Tipos de excepciones (II):
 - Otros tipos (que también heredan de `exception`):
 - **bad_alloc**: lanzada por el operador **new** si hay un error de reserva de memoria
 - **bad_cast**: lanzada por el operador **dynamic_cast** si no puede manejar un tipo referenciado
 - **bad_exception**: excepción genérica lanzada cuando un tipo de excepción no está permitida en una función determinada
 - El tipo de excepciones permitidas se especifica con la cláusula `throw()`.
 - **bad_typeid**: lanzado por el operador **typeid** a una expresión nula
 - **ios_base::failure**: lanzado por las funciones en la librería ***iostream***

Gestión de Errores

Excepciones estándares en C++



- Ubicación de las excepciones:
 - **std::exception** está definida en la cabecera `<exception>`.
 - **std::bad_alloc** está definida en la cabecera `<new>`.
 - **std::bad_cast** está definida en la cabecera `<typeinfo>`.
 - El resto de excepciones están definidas en la cabecera `<stdexcept>`.
- Todas las excepciones estándares pueden ser lanzadas implícitamente por el sistema o de manera explícita por el programador
 - P. ej.

```
throw out_of_range("límite por debajo array");
```

Gestión de Errores

Excepciones estándares en C++: reserva memoria



```
int main()
{
    double *ptr[50];
    for (int i= 0 ; i < 50; i++) {
        ptr[i] = new double[500000000];
        cout << "Reservo memoria elemento " << i << endl;
    }
    return (0);
}
```

- **¿Qué ocurre si me quedo sin memoria disponible?**
 - **El programa aborta.**

Gestión de Errores

Excepciones estándares en C++: reserva memoria



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

int main()
{
    double *ptr[50];
    try {
        for (int i= 0 ; i < 50; i++) {
            ptr[i] = new double[500000000];
            cout << "Reservando memoria para elemento " << i << endl;
        }
    }
    catch (bad_alloc &ex){
        cout << "Ocurrio un error de tipo " << ex.what() << endl;
    }
    cout<<"Termino programa normalmente"<<endl;
    return (0);
}
```


Gestión de Errores

Excepciones predefinidas en C++: reserva memoria



```
Reservando memoria para elemento 0
Reservando memoria para elemento 1
Reservando memoria para elemento 2
Reservando memoria para elemento 3
Ocurrio un error de tipo bad allocation
Termino programa normalmente
Press any key to continue
```

Gestión de Errores

Excepciones estándares en C++: error fichero



```
int main()
{
    fstream fic1;

    fic1.open ("lucas.txt",ios::in);
    fic1.close();

    cout<<"Termino programa normalmente"<<endl;
    return (0);
}
```

- **Si el fichero no existe no ocurre nada. Pero podemos utilizar excepciones con ficheros.**

Gestión de Errores

Excepciones estándares en C++: error fichero



```
#include <fstream>
#include <iostream>
#include <exception>
using namespace std;
int main()
{
    fstream fic1;
    fic1.exceptions(ios::failbit); //los fich por defecto no lanzan excepciones
    try{
        fic1.open ("lucas.txt",ios::in); //LANZA ios_base::failure
    }
    catch (ios_base::failure &ex){
        cout<<"Error al abrir el fichero"<<endl;
    }
    try{
        fic1.close();
    }
    catch(ios_base::failure &ex){
        cout<<"Error al cerrar fichero"<<endl;
    }
    cout<<"Termino programa normalmente"<<endl;
    return (0);
}
```

Si puedo utilizar un
'if' no son necesarias
las excepciones

Gestión de Errores

Excepciones estándares en C++: otros errores



```
int main () {
    try
    {
        /* ERROR ENTRADA/SALIDA CIN*/
        cin.exceptions(ios::failbit);
        cout << "Mete lo primero que se te ocurra, distinto de entero: " << endl;
        int entero; cin >> entero;
        /* ERROR SEGMENTATION FAULT: NO ES CAPTURABLE SALVO QUE TRABAJEMOS CON STL
        char *p=NULL;
        cout<<*p<<endl; */
    }
    catch (ios_base::failure &ex){
        cout<<"Error de I/O, mensaje: "<<ex.what()<<endl;
    }
    catch (std::exception& stdexc)
    {
        cout << "Error general, mensaje: " << stdexc.what() << endl;
    }
    catch (...)
    {
        cout << "Error general no derivado de exception" << endl;
    }
    cout<<"Termino el programa normalmente"<<endl;
    return (0);
}
```

Gestión de Errores

Excepciones de Usuario en C++



- Los programadores pueden definir el conjunto de excepciones más acorde al programa que están desarrollando.
- Cada tipo de excepción será una CLASE que hereda, **opcionalmente**, de la clase predefinida **exception**
- **Ejemplo:**

```
class ExcepcionNoFichero : public exception {  
private:  
    string nom_fic;  
public :  
    ExcepcionNoFichero(string nombre): exception(), nom_fic(nombre) {}  
    const char * what() const throw() {  
        return "Error al abrir el fichero "+nom_fichero; }  
};
```

Gestión de Errores

Excepciones de Usuario en C++



```
int main()
{
    fstream fic1, fic2;
    try {
        fic1.open ("lucas.txt", ios::in);
        fic2.open ("pepe.txt", ios::in);
        if (!fic1)
            throw ExcepcionNoFichero("lucas.txt");
        if (!fic2)
            throw ExcepcionNoFichero("pepe.txt");
        /* Código para tratamiento normal del fichero*/
    }
    catch (ExcepcionNoFichero &ex){
        cout << ex.what();
    }
    fic1.close();
    fic2.close();
}
```

Gestión de Errores

Excepciones de Usuario en C++



```
Error al abrir el fichero lucas.txt  
Press any key to continue
```

Gestión de Errores

Excepciones de Usuario en C++



- EJERCICIO
 - **Definid una excepción de usuario llamada**
`ExcepcionDividirPorCero` que sea lanzada cuando, en el programa de la división por cero, se introduce como valor del divisor 0.
 - Utilizad dicha excepción para modificar el programa de la división por cero para que ahora controle correctamente esta circunstancia

Gestión de Errores

Excepciones de Usuario en C++



- SOLUCIÓN:

```
class ExcepcionDividirPorCero : public exception {  
    public :  
        ExcepcionDividirPorCero() : exception() {}  
        const char * what() const throw() {  
            return "Intentas dividir por cero"; }  
};
```

Gestión de Errores

Excepciones de Usuario en C++



```
int main()
{
    float dividendo, divisor, resultado;
    cout << "PROGRAMA DIVISOR" << endl;
    try {
        cout << "Introduce Dividendo : " ;
        cin >> dividendo;
        cout << "Introduce Divisor : " ;
        cin >> divisor;
        if (divisor == 0)
            throw ExcepcionDividirPorCero();
        resultado = dividendo / divisor;
        cout << dividendo << "/" << divisor
            << "=" << resultado;
        return (0);
    }
    catch (ExcepcionDividirPorCero &exce)
    {
        cerr << "Ocurrio un error:" << exce.what();
    }
}
```

¿Sería correcto
colocar este fragmento
de código detrás del
bloque catch?



PROGRAMA DIVISOR

Introduce Dividendo : 15

Introduce Divisor : 0

Ocurrio un Error Intentas dividir por cero

Press any key to continue

Gestión de Errores

Excepciones: elección de bloques catch



- Cuando se captura una excepción y esta pertenece a una jerarquía de clases, hay que comenzar por la clase más derivada, pues de lo contrario se pierde capacidad de discriminación del tipo de excepción ocurrido.

- **EJEMPLO**

```
#include <stdio.h>
class festival{};
class Verano : public festival{};
class Primavera: public festival{};

void fiesta(int i) {
    if (i==1) throw(Verano());
    else if (i==2) throw (Primavera());
    else throw(festival() );
}
```

Gestión de Errores

Excepciones: elección de bloques catch



```
int main(){
    try { fiesta(0); } // estas sentencias están en el orden adecuado
    catch(const Verano& ) { puts("Festival de Verano"); }
    catch(const Primavera&){ puts("Festival de Primavera" ); }
    catch(const festival& ){ puts("Festival" ); }
    try { fiesta(1); }
        catch(const Verano& ) { puts("Festival de Verano"); }
        catch(const Primavera&){ puts("Festival de Primavera" ); }
        catch(const festival& ){ puts("Festival" ); }
    try { fiesta(2); }
        catch(const Verano& ) { puts("Festival de Verano"); }
        catch(const Primavera&){ puts("Festival de Primavera" ); }
        catch(const festival& ){ puts("Festival" ); }

    /* Si se captura la clase base primero se pierde la posibilidad de comprobar
    la sub-clase de la excepción que ha sido lanzada realmente */
    try { fiesta(1); }
        catch(const festival& ){ puts("Festival (de que tipo??!!)"); }
        catch(const Verano& ) { puts("Festival de Verano" ); }
        catch(const Primavera&){ puts("Festival de Primavera" ); }
    try { fiesta(2); }
        catch(const festival& ){ puts("Festival (de que tipo?!!!)"); }
        catch(const Verano& ) { puts("Festival de Verano" ); }
        catch(const Primavera&){ puts("Festival de Primavera" ); }

    return 0;
}
```

Curso 2007-2008

Gestión de Errores

Excepciones: Eficiencia



- El mecanismo de manejo de excepciones exige almacenar información adicional sobre el objeto de excepción y cada sentencia catch, con el fin de realizar la concordancia en tiempo de ejecución (ya que puede existir polimorfismo) entre la excepción y su manejador. Además, también es necesario guardar información sobre la estructura de cada función, para poder determinar si una excepción fue lanzada desde un bloque try.
- Esto supone una sobrecarga adicional en términos de velocidad y tamaño de programa, incluso cuando no se lanza nunca una excepción.
- Ojo! Incluso si no se usan excepciones directamente, probablemente se estén usando implícitamente. Por ejemplo, los contenedores STL suelen lanzar sus propias excepciones (p.ej. la función `at()` lanza un `out_of_range` si se intenta acceder fuera de los límites del contenedor). También otras funciones de la librería estándar (p.ej. funciones de la clase `string`) lanzan excepciones.


Gestión de Errores

Excepciones: Malos usos de excepciones



- Algunos programadores usan (mal) el manejo de excepciones como una alternativa a bucles for, bloques do-while o simples 'if'.
- Ejemplo:

```
#include <iostream>
using namespace std;
class Exit{}; //los objetos exit son usados como una excepción
int main()
{
    int num;
    cout<< "Introduce un número (99 para salir)" <<endl;
    try
    {
        while (true) //infinitely
        {
            cin>>num;
            if (num == 99)
                throw Exit(); //exit the loop
            cout<< "introdujiste: " << num << ". Introduce otro número " <<endl;
        }
    }
    catch (Exit& )
    {
        cout<< "Fin!" <<endl;
    }
}
```



Muy
ineficiente!!
Mejor usar
break

Gestión de Errores

Excepciones: Resumen



- **Ventajas:**

- **Separar el manejo de errores del código normal:**

- El manejo de excepciones permite al programador quitar el código para manejar errores de la línea principal.
 - Permite escribir programas más claros , robustos y tolerantes a fallos.
 - Agrupar los tipos de errores y la diferenciación entre ellos
 - Detectar el error en un lugar (código 'servidor') y tratarlo en otro diferente (código cliente).

- **Inconvenientes**

- Sobrecarga del sistema



■ Ejercicio

- Define una clase genérica Pila que gestione una pila de objetos arbitrarios. La pila tiene una capacidad máxima de 10 elementos, pero puede crearse con cualquier capacidad inferior. Queremos que la pila controle el posible desbordamiento al intentar apilar un nuevo elemento mediante una excepción de usuario ExcepcionDesbordamiento. Definir en C++ la clase genérica Pila y los métodos Pila(), ~Pila() y apilar(). Implementar un programa main() de prueba que intente apilar 20 elementos y capture la excepción en cuanto se produzca (dejando de intentar apilar el resto de elementos).

Gestión de Errores

Excepciones: Ejercicio propuesto



■ Solución

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

class ExcepcionDesbordamiento : public exception {};

template <class T>
class Pila{
public:
    Pila(int tam=10);
    void apilar (T);
    T desapilar();
    ~Pila();

private:
    int nelementos; //numElementos con que se crea la pila
    T** elementos; //array de punteros a elementos de tipo T
    int cima; //número de elementos actualmente en la pila-1
    static const int tmaximo=10;
};
```

Gestión de Errores

Excepciones: Ejercicio propuesto



■ Solución

```
template <class T>
Pila<T>::Pila(int tam){
    if (tam<=tmaximo){
        elementos=new T*[tam];
        nelementos=tam;
    }
    else {
        elementos=new T*[tmaximo];
        nelementos=tmaximo;
    }
    for (int i=0;i<nelementos;i++)
        elementos[i]=NULL;
    cima=-1;
}
template <class T>
Pila<T>::~~Pila(){

    for (int i=0;i<nelementos;i++){
        delete elementos[i];
        elementos[i]=NULL;
    }
    delete []elementos;
    nelementos=0;
    cima=-1;
}
```

Gestión de Errores

Excepciones: Ejercicio propuesto



■ Solución

```
template <class T>
void Pila<T>::apilar (T elem){
    cout<<"Llamo método apilar"<<endl;
    try{
        cima++;
        if (cima<nelementos){
            elementos[cima]=new T(elem);
            cout<<"Apilo"<<endl;
        }
        else
            throw ExcepcionDesbordamiento();
    }
    catch(exception& e){
        cout<<"Catch método apilar"<<endl;
    }
}

int main(){

    Pila<int> pi(2);
    for (int i=0;i<20;i++)
        pi.apilar(i);
    return (0);
};
```

¿Qué salida
produce este
programa?



■ Solución

```
int main() {  
  
    Pila<int> pi(2);  
    try{  
        for (int i=0;i<20;i++)  
            pi.apilar(i);  
    }  
    catch(exception &e){  
        cerr<<"Catch de main: tipo error exception"<<endl;  
    }  
    catch(...){  
        cerr<<"Catch de main: tipo error desconocido"<<endl;  
    }  
    return (0);  
};
```

Gestión de Errores

Excepciones: Ejercicio propuesto



- **Alternativa: no capturar excepción en Apilar**

- En este caso es obligatorio que el main capture la excepción si no queremos que el programa termine de manera inesperada

```
template <class T>
void Pila<T>::apilar (T elem){
    cout<<"Intento apilar"<<endl;
    cima++;
    if (cima<nelementos)
        elementos[cima]=new T(elem);
    else
        throw ExcepcionDesbordamiento();
};

int main(){
    Pila<int> pi(2);
    try{
        for (int i=0;i<20;i++) pi.apilar(i);
    }
    catch(exception &e){
        cerr<<"Catch de main: tipo error exception"<<endl;
    }
    catch(...){
        cerr<<"Catch de main: tipo error desconocido"<<endl;
    }
}
```

Gestión de Errores

Excepciones: Ejercicio propuesto



■ ¿Cómo lanzarías una excepción capturada en un nivel de anidamiento X hacia los niveles superiores?

```
template <class T>
void Pila<T>::apilar (T elem){
    try{
        cout<<"Intento apilar"<<endl;
        cima++;
        try{
            if (cima<nelementos){
                elementos[cima]=new T(elem);
                cout<<"Elemento apilado"<<endl;
            }
            else
                throw ExcepcionDesbordamiento();
        }
        catch(...){
            cerr<<"Catch interno de apilar"<<endl;
            throw;
        }
    }
    catch(...){
        cerr<<"Catch externo de apilar"<<endl;
    }
}
```

```
int main(){
    Pila<int> pi(2);
    try{
        for (int i=0;i<5;i++)
            pi.apilar(i);
    }
    catch(...){
        cerr<<"Catch del main"<<endl;
    }
    return (0);
};
```

¿Qué salida produce este programa?