Universidad de Guadalajara

Centro Universitarios de Ciencias Exactas e Ingenierías

Computación Tolerante a Fallas

(Par. 1) Otras herramientas para el manejar errores

Alumno: Daniel Garcia Figueroa

Código: 217528017

Carrera: Ingeniería Informática

Calendario: 2025A

Objetivo

Investiga otras herramientas para el manejar errores

Introducción

En la programación, los errores son condiciones que interrumpen el funcionamiento normal de un programa y pueden clasificarse en sintácticos, semánticos y de ejecución. Los errores **sintácticos** ocurren cuando el código no cumple con las reglas del lenguaje y son detectados por compiladores o intérpretes. Los errores **semánticos**, en cambio, están relacionados con fallos en los algoritmos que impiden obtener el resultado esperado. Ambos pueden corregirse durante el desarrollo del programa.

Por otro lado, los errores de **ejecución** surgen mientras el programa se está ejecutando y pueden deberse a factores como datos inválidos, mal uso del software, fallos en la memoria o interacción inesperada con otros sistemas. Dado que estos errores no pueden preverse completamente, es necesario implementar mecanismos para detectarlos y manejarlos de manera adecuada.

Existen diversas herramientas diseñadas para facilitar la identificación, monitoreo y gestión de errores en la programación. Este reporte presenta un análisis de algunas de las principales herramientas utilizadas para el manejo de errores, destacando sus características y aplicaciones en diferentes entornos de desarrollo.

Desarrollo

El manejo adecuado de errores es esencial para el desarrollo de programas robustos y confiables. En C++, existen varias herramientas y enfoques para manejar errores de manera eficiente, siendo los dos más comunes el uso de excepciones y la gestión manual de errores.

A continuación, se describen las principales herramientas que se pueden emplear en C++ para gestionar situaciones de error.

1. Manejo de errores con excepciones

Las excepciones en C++ son mecanismos que permiten gestionar los errores de manera controlada sin interrumpir abruptamente el flujo del programa. Las excepciones son útiles

cuando se produce una situación anómala que debe manejarse de forma centralizada. Cuando se lanza una excepción, el flujo de ejecución se detiene y el control se transfiere a un bloque `catch` que maneja el error.

Las excepciones se prefieren en C++ moderno por los siguientes motivos:

- Una excepción obliga al código que llama a reconocer una condición de error y controlarla. Las excepciones no controladas detienen la ejecución del programa.
- Una excepción salta al punto de la pila de llamadas que puede controlar el error. Las funciones intermedias pueden dejar que la excepción se propague. No tienen que coordinarse con otras capas.
- El mecanismo de desenredo de la pila de excepciones destruye todos los objetos del ámbito después de iniciar una excepción, según reglas bien definidas.
- Una excepción permite una separación limpia entre el código que detecta el error y el código que lo controla.

Herramientas

➔ `try:` Se debe usar un bloque try para incluir una o más instrucciones que pueden iniciar una excepción.

➔ `catch:`Para controlar las excepciones que se puedan producir, implemente uno o varios bloques catch inmediatamente después de un bloque try. Cada bloque catch especifica el tipo de excepción que puede controlar.

➔ `throw:` Una expresión throw indica que se ha producido una condición excepcional, a menudo un error, en un bloque try. Se puede usar un objeto de cualquier tipo como operando de una expresión throw. Normalmente, este objeto se emplea para comunicar información sobre el error.

➔ Tipos de Excepciones: C++ ofrece varias excepciones estándar como `invalid_argument`, `out_of_range`, y `runtime_error`, además de permitir la creación de excepciones personalizadas.

```
#include <iostream>
#include <stdexcept> // Necesario para excepciones estándar

// Función para dividir dos números
double dividir(double a, double b) {
   if (b == 0) {
       throw std::invalid_argument("No se puede dividir entre cero");
```

```
    }
    return a / b;
}

int main() {
    try {
        double resultado = dividir(10, 0); // Intentamos dividir por cero
        std::cout << "Resultado: " << resultado << std::endl;
    } catch (const std::exception& e) {
        // Captura cualquier excepción de tipo std::exception
        std::cout << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Ventajas del Uso de Excepciones

- **Claridad del Código:** Las excepciones permiten separar el manejo de errores de la lógica principal del programa, mejorando la legibilidad y organización del código.
- **Manejo Centralizado de Errores:** Permite centralizar el manejo de errores en un solo lugar, evitando la dispersión de verificaciones de errores en múltiples puntos del código.
- **Propagación Automática de Errores:** Las excepciones se propagan automáticamente, permitiendo que se manejen en un nivel superior sin necesidad de hacerlo explícitamente en cada función.

2. Manejo de errores sin excepciones

En C++, el manejo de errores sin excepciones se refiere a técnicas para manejar errores sin recurrir al sistema de excepciones que el lenguaje proporciona (es decir, sin usar `try`, `catch`, o `throw`).

Herramientas

➔ `optional:` Es una clase que se introdujo en C++17, se utiliza para representar valores que pueden estar presentes o ausentes. En lugar de devolver un valor nulo o lanzar una excepción cuando un valor no está disponible, std::optional permite representar esta ausencia de una manera explícita y controlada.

```cpp
#include <iostream>
#include <optional>

// Función que retorna un número positivo o nada
std::optional<int> obtenerNumero(bool valido) {
    if (valido) {
        return 42; // Retorna un número válido
    } else {
        return std::nullopt; // No hay valor
    }
}

int main() {
    auto numero = obtenerNumero(false); // Aquí no tenemos un valor válido

    if (numero) {
        std::cout << "Número: " << *numero << std::endl;
    } else {
        std::cout << "No se encontró el número." << std::endl;
    }

    return 0;
}
```

➔ `expected:` Proporciona una forma de representar cualquiera de los dos valores: un *esperado* valor del tipo T, o un *inesperado* valor del tipo E. std::expected nunca no tiene valor.

```cpp
#include <iostream>
#include <expected> // Solo disponible en C++23

// Función que puede devolver un número o un error
std::expected<int, std::string> obtenerNumero(bool valido) {
    if (valido) {
        return 42; // Número válido
    } else {
        return std::unexpected("Error: valor no válido"); // Error si no es válido
    }
}

int main() {
    auto resultado = obtenerNumero(false); // Simulamos un error

    if (resultado) {
```

```
        std::cout << "Número: " << *resultado << std::endl;
    } else {
        std::cout << "Error: " << resultado.error() << std::endl;
    }

    return 0;
}
```

➔ Valores de Retorno Especiales: Usar valores especiales como `-1` o `nullptr` para indicar que una operación no tuvo éxito. Aunque simple, este enfoque puede ser propenso a errores si no se gestionan adecuadamente.

```
#include <iostream>

// Función que retorna un número o -1 si no es válido
int obtenerNumero(bool valido) {
    if (valido) {
        return 42; // Valor válido
    } else {
        return -1; // Valor especial para indicar error
    }
}

int main() {
    int numero = obtenerNumero(false); // Intentamos obtener el número, pero es inválido

    if (numero != -1) {  // Verificamos si el valor no es el especial (-1)
        std::cout << "Número: " << numero << std::endl;
    } else {
        std::cout << "No se pudo obtener el número." << std::endl; // Error
    }

    return 0;
}
```

➔ Códigos de Error: Algunos programas devuelven códigos de error para indicar que ocurrió una falla. Sin embargo, este enfoque puede ser menos claro y más propenso a errores si no se gestionan adecuadamente todos los códigos posibles.

```
#include <iostream>
```

```cpp
// Definimos códigos de error como constantes
const int ERROR_NO_VALIDO = -1;
const int ERROR_DESCONOCIDO = -2;

// Función que retorna un código de error o un número válido
int obtenerNumero(bool valido) {
    if (valido) {
        return 42; // Número válido
    } else {
         return ERROR_NO_VALIDO; // Código de error indicando que el número no es válido
    }
}

int main() {
    int resultado = obtenerNumero(false); // Intentamos obtener el número, pero es inválido

    if (resultado >= 0) {
        std::cout << "Número: " << resultado << std::endl;
    } else {
        // Comprobamos qué código de error se retornó
        if (resultado == ERROR_NO_VALIDO) {
            std::cout << "Error: El número no es válido." << std::endl;
        } else if (resultado == ERROR_DESCONOCIDO) {
            std::cout << "Error: Desconocido." << std::endl;
        } else {
            std::cout << "Error: Código de error no reconocido." << std::endl;
        }
    }

    return 0;
}
```

➔ Banderas de Error: Se usan variables booleanas o flags para indicar si ocurrió un error, pero esto también puede complicar el flujo del programa si se usan en exceso.

```cpp
#include <iostream>

// Función que retorna un número y usa una bandera de error para indicar si fue exitoso
int obtenerNumero(bool valido, bool& errorFlag) {
    if (valido) {
        errorFlag = false;  // No hubo error
        return 42;  // Número válido
```

```
    } else {
        errorFlag = true;  // Ocurrió un error
        return 0;  // Valor que indica fallo
    }
}

int main() {
    bool errorFlag = false;  // Bandera de error inicializada en false

    int resultado = obtenerNumero(false, errorFlag);  // Intentamos obtener el
número, pero es inválido

    if (errorFlag) {
        std::cout << "Error: El número no es válido." << std::endl;
    } else {
        std::cout << "Número: " << resultado << std::endl;
    }

    return 0;
}
```

Ventajas

- **Manejo Controlado de Ausencia de Valor:** Permite representar la ausencia de un valor de manera clara, sin necesidad de lanzar una excepción.
- **Rendimiento:** Al no requerir el uso de excepciones, el rendimiento puede ser mejor en situaciones donde la ausencia de un valor no es un error crítico.
- **Claridad en el Flujo del Programa:** Los programas que utilizan `optional` dejan claro que un valor puede no estar disponible, lo que mejora la claridad y facilita el manejo de esos casos.

Usar `std::cerr` para el manejo de errores permite que los mensajes de error sean enviados a una salida separada, lo que facilita la depuración y el monitoreo de la aplicación.

Referencias

Fundamentos de programación/Manejo de errores - Wikiversidad. (n.d.). https://es.wikiversity.org/wiki/Fundamentos_de_programaci%C3%B3n/Manejo_de_errores

TylerMSFT. (2023, December 10). *Instrucciones try, throw y catch (C++)*. Microsoft Learn. https://learn.microsoft.com/es-es/cpp/cpp/try-throw-and-catch-statements-cpp?view=msvc-170

Exceptions and Error Handling, C++ FAQ. (n.d.). https://isocpp.org/wiki/faq/exceptions

Guzman, H. C. (n.d.). *Manejo de excepciones y errores | Apuntes lenguaje C++ | Hektor Profe*. https://docs.hektorprofe.net/cpp/13-manejo-excepciones/

Brand, S. (2023, April 18). *Functional exception-less error handling with C++23's optional and expected - C++ Team Blog*. C++ Team Blog. https://devblogs.microsoft.com/cppblog/cpp23s-optional-and-expected/