

UNIVERSIDAD DE GUADALAJARA
Centro Universitario de Ciencias Exactas e Ingenierías

Computación Tolerante a Fallas



(Par. 2) Ejemplo: Otras herramientas para el manejar errores

Nombre: Castillo Mares Gilberto

Código: 213330514

Profesor: Dr. Michel Emanuel López Franco

2024-A

Índice

Contenido

Desarrollo	3
Conclusión.....	9

Desarrollo

En los siguientes ejemplos, utilicé el lenguaje de programación C++.

Ejemplo de logging

```
1  #include <iostream>
2  #include <fstream>
3
4  int main() {
5      // Configurar el registro en un archivo
6      std::ofstream logfile("logfile.txt");
7      if (!logfile.is_open()) {
8          std::cerr << "Error al abrir el archivo de
registro." << std::endl;
9          return 1;
10     }
11
12     // Redirigir la salida estándar a la consola y al
archivo de registro
13     std::streambuf* original_cout =
std::cout.rdbuf(logfile.rdbuf());
14
15     try {
16         std::cout << "Inicio del programa" << std::endl;
17
18         // Simular una operación que puede generar un
error
19         int divisor = 0;
20         if (divisor == 0) {
21             std::cerr << "Error: División por cero" <<
std::endl;
22             throw std::runtime_error("Error: División por
cero");
23         } else {
24             float resultado = 10 /
static_cast<float>(divisor);
25             std::cout << "Resultado de la división: " <<
resultado << std::endl;
26         }
27     } catch (const std::exception& e) {
28         std::cerr << "Excepción capturada: " << e.what()
<< std::endl;
29     }
30
31     std::cout << "Fin del programa" << std::endl;
32
33     // Restaurar la salida estándar original
```

```

34     std::cout.rdbuf(original_cout);
35
36     return 0;
37 }

5     // Configurar el registro en un archivo
6     std::ofstream logfile("logfile.txt");
7     if (!logfile.is_open()) {
8         std::cerr << "Error al abrir el archivo de
registro." << std::endl;
9         return 1;

```

Configuración del registro en un archivo: Se crea un objeto `std::ofstream` llamado `logfile` para escribir en un archivo llamado "logfile.txt". Luego, se verifica si el archivo se abrió correctamente. Si hubo un error al abrir el archivo, se imprime un mensaje de error en la salida de error estándar (`std::cerr`) y el programa devuelve un código de error (1).

```

12     // Redirigir la salida estándar a la consola y al
    archivo de registro
13     std::streambuf* original_cout =
std::cout.rdbuf(logfile.rdbuf());

```

Redirección de la salida estándar: Se guarda el puntero al búfer de salida estándar original (`original_cout`). Luego, se redirige la salida estándar (`std::cout`) al búfer del archivo de registro.

```

15     try {
16         std::cout << "Inicio del programa" << std::endl;
17
18         // Simular una operación que puede generar un
error
19         int divisor = 0;
20         if (divisor == 0) {
21             std::cerr << "Error: División por cero" <<
std::endl;
22             throw std::runtime_error("Error: División por
cero");
23         } else {
24             float resultado = 10 /
static_cast<float>(divisor);

```

```

25         std::cout << "Resultado de la división: " <<
resultado << std::endl;
26     }
27     } catch (const std::exception& e) {
28         std::cerr << "Excepción capturada: " << e.what()
<< std::endl;
29     }

```

Bloque try-catch: Se inicia un bloque **try** donde se coloca el código que puede generar una excepción. En este caso, se simula una operación que puede generar un error al intentar dividir por cero. Si ocurre una excepción (**std::runtime_error** en este caso), se captura en el bloque **catch** y se imprime un mensaje de error en la salida de error estándar.

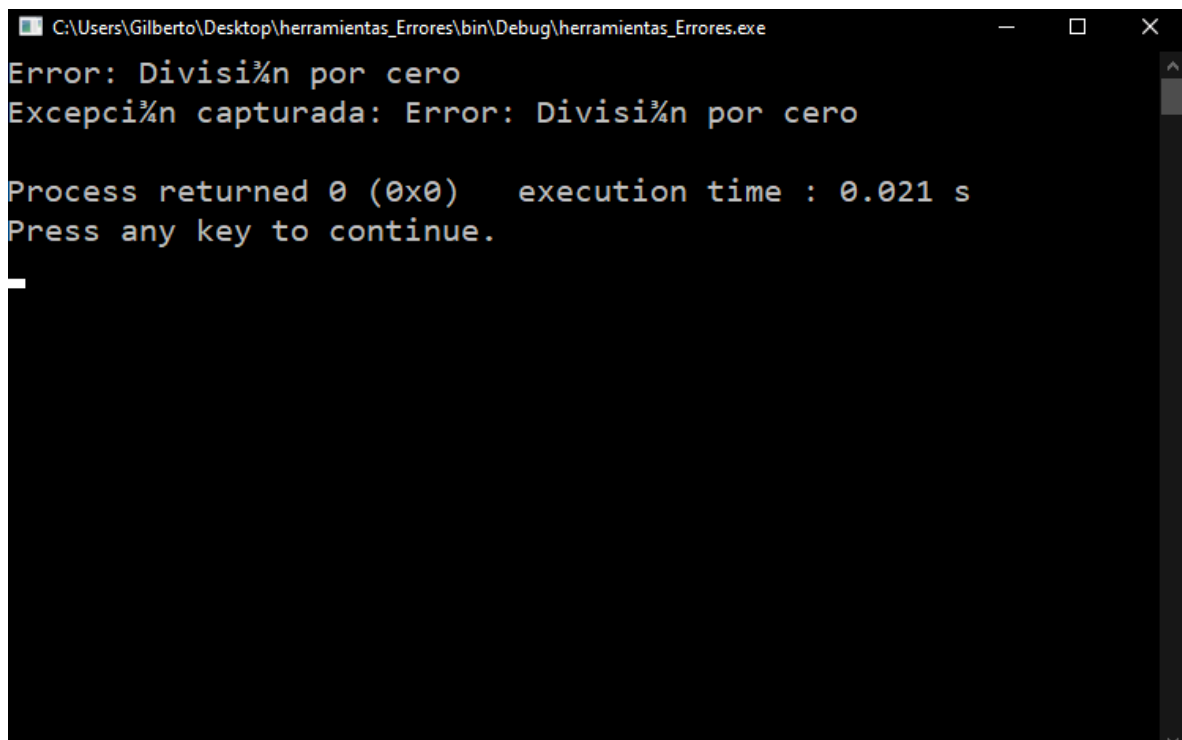
```

33     // Restaurar la salida estándar original
34     std::cout.rdbuf(original_cout);

```

Restauración de la salida estándar: Se restaura la salida estándar original antes de finalizar el programa. Esto asegura que las futuras operaciones de salida estándar se realicen como se esperaba.

Captura de pantalla:



```

C:\Users\Gilberto\Desktop\herramientas_Errores\bin\Debug\herramientas_Errores.exe
Error: Divisi3n por cero
Excepci3n capturada: Error: Divisi3n por cero

Process returned 0 (0x0)   execution time : 0.021 s
Press any key to continue.

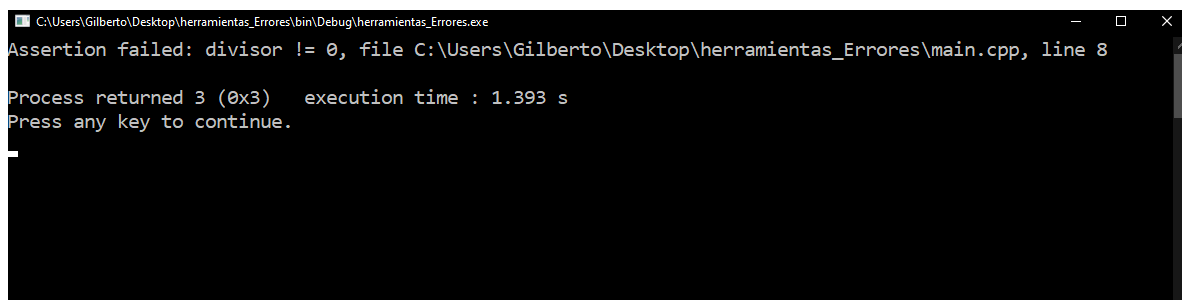
```

Ejemplo de assert:

```
1  #include <cassert>
2  #include <iostream>
3
4  int main() {
5      int divisor = 0;
6
7      // Utilizando assert para verificar que el divisor no
8      sea cero
9      assert(divisor != 0);
10
11     // El código continuar; aquí- si la aserción es
12     verdadera
13     std::cout << "Divisor no es cero. Continuando con la
14     ejecución." << std::endl;
15     return 0;
16 }
```

- Se incluye la cabecera **<cassert>** que proporciona la macro **assert**.
- Se declara una variable **divisor** y se utiliza **assert(divisor != 0)** para verificar que el divisor no sea cero.
- Si la aserción es verdadera, el programa continuará ejecutándose normalmente; de lo contrario, si la aserción es falsa, se producirá un fallo en tiempo de ejecución y el programa se cerrará, indicando la línea y el archivo donde falló la aserción.

Captura de pantalla:



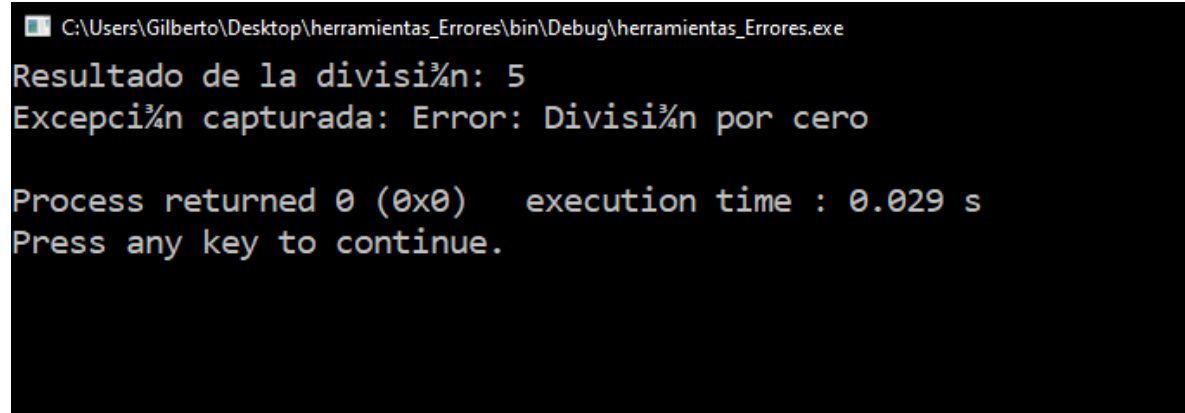
Ejemplo de raise:

Esta excepción como tal no se encuentra en el lenguaje C++, sin embargo, utilicé la excepción **throw** que tiene un comportamiento similar.

```
1  #include <iostream>
2  #include <stdexcept>
3
4  void dividir(int a, int b) {
5      if (b == 0) {
6          // Lanzar una excepción en lugar de usar 'raise'
7          throw std::runtime_error("Error: División por
cero");
8      }
9
10     int resultado = a / b;
11     std::cout << "Resultado de la división: " <<
resultado << std::endl;
12 }
13
14 int main() {
15     try {
16         dividir(10, 2);
17         dividir(8, 0); // Esto provocará una excepción
18     } catch (const std::exception& e) {
19         std::cerr << "Excepción capturada: " << e.what()
<< std::endl;
20     }
21
22     return 0;
23 }
```

- La función **dividir** verifica si el divisor (**b**) es cero. Si es cero, en lugar de usar **raise**, lanza una excepción de tipo **std::runtime_error** con un mensaje descriptivo.
- En la función **main**, se llama a **dividir** dos veces. La primera llamada tiene un divisor válido (2), mientras que la segunda llamada tiene un divisor cero, lo que provoca que se lance una excepción.
- En el bloque **catch**, se captura la excepción y se imprime el mensaje asociado a la excepción.

Captura de pantalla:



```
C:\Users\Gilberto\Desktop\herramientas_Errores\bin\Debug\herramientas_Errores.exe
Resultado de la divisi3n: 5
Excepci3n capturada: Error: Divisi3n por cero

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```


Conclusión

Creo que al poder utilizar ***assert*** resulta muy útil porque durante el desarrollo y pruebas del software se pueden verificar condiciones críticas, mientras que la excepción ***throw*** es esencial para señalar y manejar situaciones en tiempo de ejecución, proporcionando un sistema robusto para el manejo de errores.