Manejo de Errores: *Excepciones*

Luis Garreta
luis.garreta@javerianacali.edu.co

Ingeniería de Sistemas y Computación
Pontificia Universidad Javeriana – Cali

17 de octubre de 2017

Objetivos

- Saber utilizar try, throw y catch para observar, indicar y manejar excepciones, respectivamente.
- Comprender las ventajas del manejo de errores mediante excepciones frente a la gestión de errores tradicional de la programación imperativa.
- Comprender la jerarquía de excepciones estándar.
- Ser capaz de crear excepciones personalizadas
- Ser capaz de procesar las excepciones no atrapadas y las inesperadas.

Gestion de Errores Tradicional

```
1  int main (void) {
2  int res;
3  if (puedo_fallar () == -1) {
4    cout << "¡Algo falló!" << endl;
5    return 1;
6  }
7  else
8    cout << "Todo va bien..." << endl;
9
10 if (dividir (10, 0, res) == -1){
11   cout << "¡División por cero!" << endl;
12   return 2;
13 }
14 else
15   cout << "Resultado: " << res << endl;
16
17 return 0;
18 }
```

Consecuencias Gestion Tradicional de Errores

- Nos obliga a definir un esquema de programación similar a:

```
1 Llevar a cabo tarea 1
2 Si se produce error
3   Llevar a cabo procesamiento de errores
4
5 Llevar a cabo tarea 2
6 Si se produce error
7   Llevar a cabo procesamiento de errores
```

- A esto se le llama código espaguetti
- Problemas de esta estrategia:
  - Entremezcla la lógica del programación la del tratamiento de errores (disminuye legibilidad)
  - El código cliente (llamador) no está obligado a tratar el error
  - Los 'códigos de error' no son consistentes.

Qúe alternativas al código *espaguetti?*

- Abortar el programa
- ¿Y si el programa es crítico?
  - Usar indicadores de error globales
  - El código cliente (llamador) no está obligado a consultar dichos indicadores.
- USAR EXCEPCIONES

Introducción a Excepciones

- Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.
- Se suelen modelar como objetos (instancias de clases) que contienen información sobre el error.
- Las excepciones se tratan mediante sentencias de control del flujo de error que separan el código para manejar errores del resto mediante :
  - throw, try y catch
- Por defecto, una excepción no se puede ignorar: hará que el programa aborte:
  - Una excepción pasará sucesivamente de un método a su llamador hasta encontrar un bloque de código que la trate.

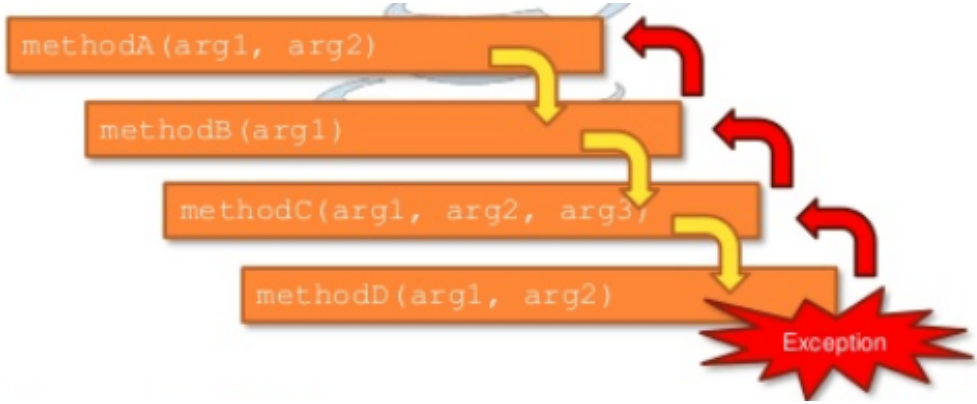
Comportamiento de las Excepciones

- Las excepciones son lanzadas (throw) por un método cuando éste detecta una condición excepcional o de error.
- Esto interrumpe el control normal de flujo y provoca (si el propio método no la trata) la finalización prematura de la ejecución del método y su retorno al llamador.
- Las excepciones pueden ser capturadas (**try/catch**), normalmente por el código cliente (llamador).
- Si el llamador no captura la excepción, su ejecución terminará y la excepción 'saldrá' de nuevo hacia un ámbito más externo y así sucesivamente hasta encontrar un lugar donde es capturada.

Una excepción no capturada provocará que el programa aborte.

Flujo de una Excepción

Un método que lanza excepciones se limita a señalar que se ha producido algún tipo de error, pero, por regla general, no debe tratarlo. Delegará dicho tratamiento en quienes invoquen al método que produce la excepción.



Sintaxis C++: Lanzamiento y Captura

- La instrucción *trow* dispara una excepción hacia el llamador.
- 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

Implementación (Lanzamiento)

```
1  void Func() {
2    if (detecto_error1)
3      throw Tipo1(); // Constructor
4      ...
5    if (detecto_error2)
6      throw Tipo2(); // Constructor
7      ...
8  }
```

Llamado (Captura)

```
1  try {
2      // Codigo de ejecución normal
3      Func(); // puede lanzar
           excepciones
4      ...
5  } catch (Tipo1 &ex) {
6      // Gestión de excep tipo 1
7  } catch (Tipo2 &ex) {
8      // Gestión de excep tipo 2
9  } catch (...) {
10     /* Gestión de cualquier excepción
           no
11     capturada mediante los catch
12     anteriores*/
13 }
```

Excepciones de Usuario: Definición

- Es habitual tipificar el error creando clases de objetos que representan diferentes circunstancias de error.
- La ventaja de hacerlo así es que :
  - Podemos incluir información extra al lanzar la excepción.
  - Podemos agrupar las excepciones en jerarquías de clases.

```
1  class miExcepcion {
2      int x;
3      string msg;
4  public:
5      miExcepcion(int a, string m) : x(a), msg(m) {}
6      string queHaPasado() const {
7          return msg;
8      }
9      int getElCulpable() const {
10         return x;
11     }
12 };
```

Excepciones de Usuario: Uso

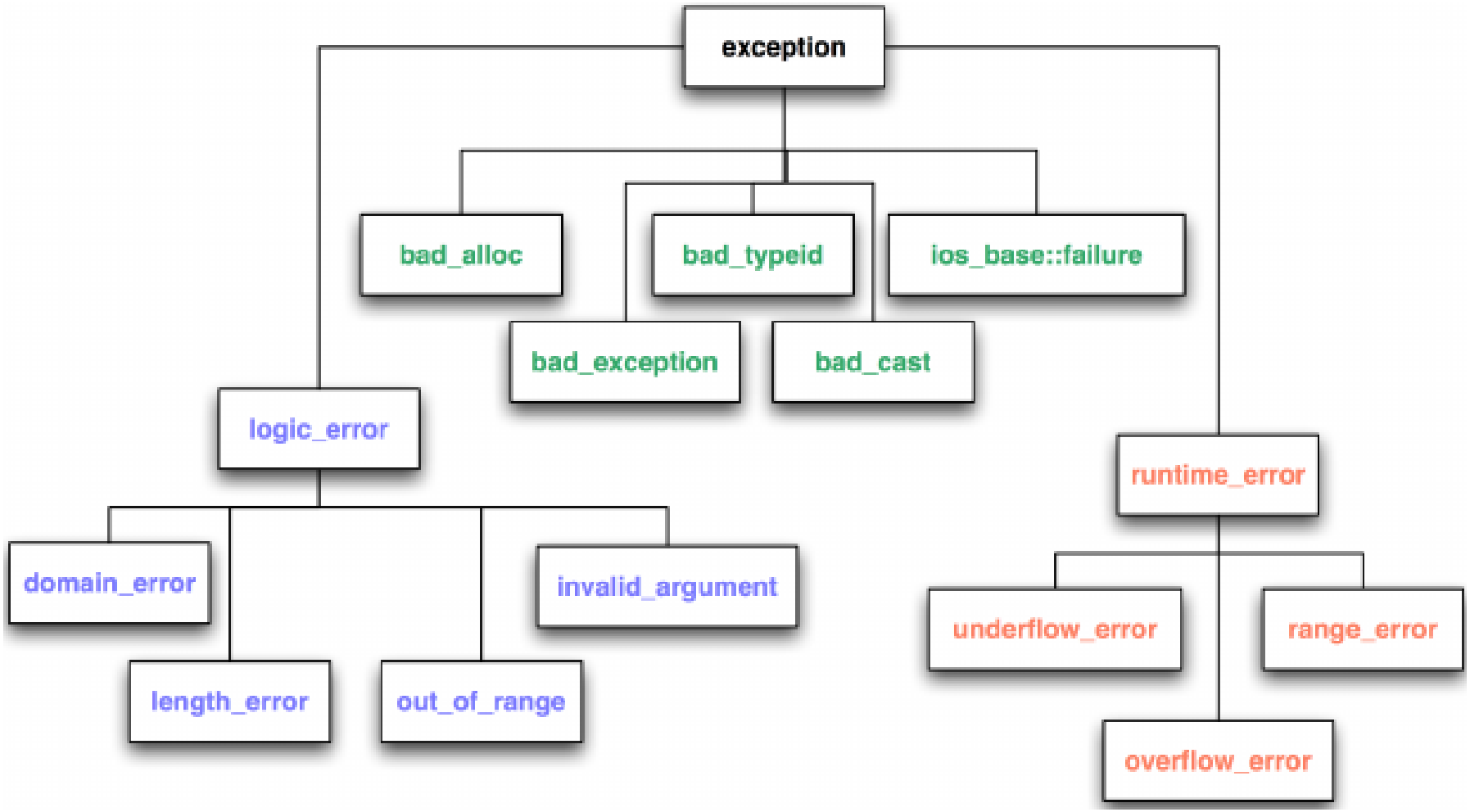
- lanzamiento:

```
1  int LlamameConCuidado(int x) {
2      if (condicion_de_error(x) == true)
3          throw miExcepcion(x,"¡Lo has vuelto a hacer!");
4      //... código a ejecutar si no hay error ...
5  }
```

- Llamado:

```
1  int main() {
2      try {
3          LlamameConCuidado(-0);
4          Argumento
5      }catch (miExcepcion& ex) {
6          por referencia
7          cerr << ex.queHaPasado()
8               << ex.getElCulpable() << endl;
9      }
10 }
```

Excepciones estándares en C++



Ejemplos Tratamiento Excepción Reserva de Memoria

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4  int main() {
5    double *ptr[50];
6    try {
7      for (int i= 0 ; i < 50; i++) {
8        ptr[i] = new double[50000000];
9       cout << "Reservando memoria para elemento " << i <<
             endl;
10     }
11   }catch (bad_alloc &ex) {
12       cout << ex.what() << endl;
13   }
14   cout<<"Termino programa normalmente"<<endl;
15   return (0);
16 }
```

Excepciones de Usuario: Definición

```
1  class ExcepcionDividirPorCero : public exception {
2  public :
3    ExcepcionDividirPorCero() : exception() {}
4    const char * what() const throw() {
5      return "Intentas dividir por cero";
6    }
7  };
```

Excepciones de Usuario: Uso

```
1  int main() {
2      float dividendo, divisor, resultado;
3      cout << "PROGRAMA DIVISOR" << endl;
4      cout << "Introduce Dividendo : " ;
5      cin >> dividendo;
6      cout << "Introduce Divisor : " ;
7      cin >> divisor;
8      try {
9          resultado = div(dividendo, divisor);
10         cout << dividendo << "/" << divisor
11              << "=" << resultado;
12     } catch (ExcepcionDividirPorCero &exce) {
13         cerr << "Error:" << exce.what() << endl;
14
15     }
16     return (0);
17 }
```