EXCEPCIONES EN C++

UCA Universidad de Cádiz

Programación Orientada a Objetos

Curso 2010 / 2011

Procedimientos o formas para tratar errores

- Terminar el programa.
- Devolver un valor que represente “error”.
- Devolver un valor “legal” y dejar el programa en un estado “ilegal”.
- Llamar a una función que puede suministrar el cliente en caso de “error”.

¿Qué es una excepción?

- Un objeto que se lanza (*throw*) en una función donde se detecta un error o problema y que se puede capturar (*catch*) en otra donde se puede tratar (*try*).

Instrucción throw (I)

- Para lanzarla una excepción:
  - *throw [expresión];*
- Ejemplo:

```
class Error { /* ... */ };
void f() {
...
Error e; // constructor predeterminado
...
throw e; /* se lanza un objeto temporal, copia de "e",
            creado con el constructor de copia, y se
            destruye ((e)) con el destructor */
...
```

Instrucción throw (II)

```
throw Error(); /* se crea un objeto temporal con el ctor.
                  predeterminado, se copia en otro temporal,
                 que es el que se lanza, con su constructor
                 de copia, y se destruye el primer temporal
                 */
...
throw new Error; /* se lanza un puntero a Error, que apunta
                        a un objeto anónimo creado dinámicamente
                        */
...
}
```

Instrucción throw (III)

- Recomendable: crear una clase o tipo para cada error diferente.
- Si no hace falta incluir ninguna información que haya que pasar al manejador, basta con una clase vacía con nombre apropiado.

```
class Nif {
   unsigned int dni;   char letra;   bool letra_valida();
   public:
      // Clase de excepción, anidada
     class LetraInvalida { /* vacía */ };
     // Constructor
     Nif(unsigned int n, char ltr): dni(n), letra(ltr)
    {
       if (not letra_valida()) throw LetraInvalida();
    }
   // ...
};
```

La lista throw (I)

- Se utiliza para informar al cliente de qué excepciones exactamente podrá lanzar una función o método.
- Forma parte de la declaración de la función, apareciendo tras la lista de parámetros.
- Recomendable: escribir la lista *throw* para cada función.
- **Lista vacía ≠ lista inexistente.**
- Lista vacía: la función no va a lanzar ninguna excepción.
- Lista inexistente: la función puede lanzar cualquier excepción o ninguna.

La lista throw (II)

Ejemplo:

```
Nif(unsigned int n, char ltr) throw(LetraInvalida)
      : dni(n), letra(ltr)
   {
      if (not letra_valida()) throw LetraInvalida();
   }
```

La lista throw (III)

Ejemplo:

- void f() throw(MuyGrande, MuyChico, DivCero);
- void g();
- void h() throw();

Instrucción try

- Bloque especial dentro de la función donde se está intentando resolver el problema real de programación y que potencialmente puede generar excepciones.
- Implica un código más claro y fácil de leer, porque su tarea principal no está mezclado con la comprobación de errores.

```
try {
   // código que puede generar excepciones ...
}
// ...
```

Instrucción catch o manejador de excepción (I)

- Habrá uno por cada tipo de excepción que se quiera capturar.
- Se escriben inmediatamente tras el bloque try.

```
try {
   // código que puede lanzar excepciones
} catch(Tipo1 id1) {
   // trata con excepciones de Tipo1
} catch(Tipo2 id2) {
   // trata con excepciones de Tipo2
}
// etc...
```

Instrucción catch o manejador de excepción (II)

- Cuando se lanza una excepción dentro de un bloque *try*, el control se pasa inmediatamente al primer *catch* cuya *signatura* coincida con el tipo de la excepción.
- Hay que ser cuidadosos con el orden en que se escriben los manejadores.
- Para capturar cualquier excepción (debe ser el último de la lista):

```
catch(…) {
}
```

Instrucción catch o manejador de excepción (III)

- Para relanzar una excepción se utiliza la instrucción *throw* sola.

```
catch(...) {
   cerr << "Se ha lanzado una excepción." << endl;
   throw;
}
```

¿Para qué sirven las excepciones? (I)

- Corregir el problema y llamar de nuevo a la función que lo causó.
- Corregir el problema y seguir, sin volver a llamar a la función que lo causó.
- Calcular algún resultado alternativo en vez del que la función se supone que produciría.
- Hacer lo que se pueda en el contexto en curso y relanzar la misma excepción a uno superior.
- Hacer lo que se pueda en el contexto en curso y relanzar otra excepción diferente a uno superior.
- Terminar el programa ordenadamente.

¿Para qué sirven las excepciones? (II)

- Envolver funciones que empleen otros esquemas de error, de forma que generen excepciones.
- Simplificar. El empleo de excepciones debe ser fácil, claro y cómodo.
- Hacer una biblioteca o programa más seguro y robusto.

Consejos:

- **Emplee siempre especificaciones de excepción.**
- **Capture por referencia, no por valor.**
- **Lance excepciones en constructores** (garantiza que el objeto no está construido).
- **Empiece con las excepciones estándar.**
- **No lance o cause excepciones en destructores.**
- Anide sus propias excepciones.
- Emplee jerarquías de excepciones.
- Herencia múltiple.
- Evite punteros “desnudos”.

¿Cuándo evitar excepciones?

- No para eventos asíncronos.
- No para errores ordinarios.
- No para control de flujo.
- No si se puede evitar.
- No para código antiguo. Se recomienda uso de catch(…)

Funciones *terminate()* y *set_terminate()* (I)

- Si una excepción no es capturada, se llama automáticamente a la función *terminate()*, que llama a la función *abort()* → acaba abruptamente la ejecución del programa.
- Se puede instalar una función escrita por el usuario que sea la que *terminate()* llame en lugar de *abort()*.
- Esta función se instala pasando su dirección a la función estándar *set_terminate()*, declarada en <*exception*>.
- Pero no se llama a los destructores de los objetos globales ni estáticos.
- Solución: meter un bloque try en *main()* cuyo último capturador sea el de la elipsis.
- Nuestra función de terminación debe arreglar lo que se pueda y llamar, por ejemplo, a *exit()*.

Funciones *terminate() y set_terminate()* (II)

```
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;
void terminator()
{
   cerr << "Sayonara, baby!" << endl;
   exit(EXIT_FAILURE);
}
void (*terminate_anterior)() = set_terminate(terminator);

class Chapuza {
   public:
      class Fruta {};
         void f() {
            cout << "Chapuza::f()" << endl;
            throw Fruta();
         }
      ~Chapuza() { throw 'c'; }
};
```

Funciones *terminate() y set_terminate()* (III)

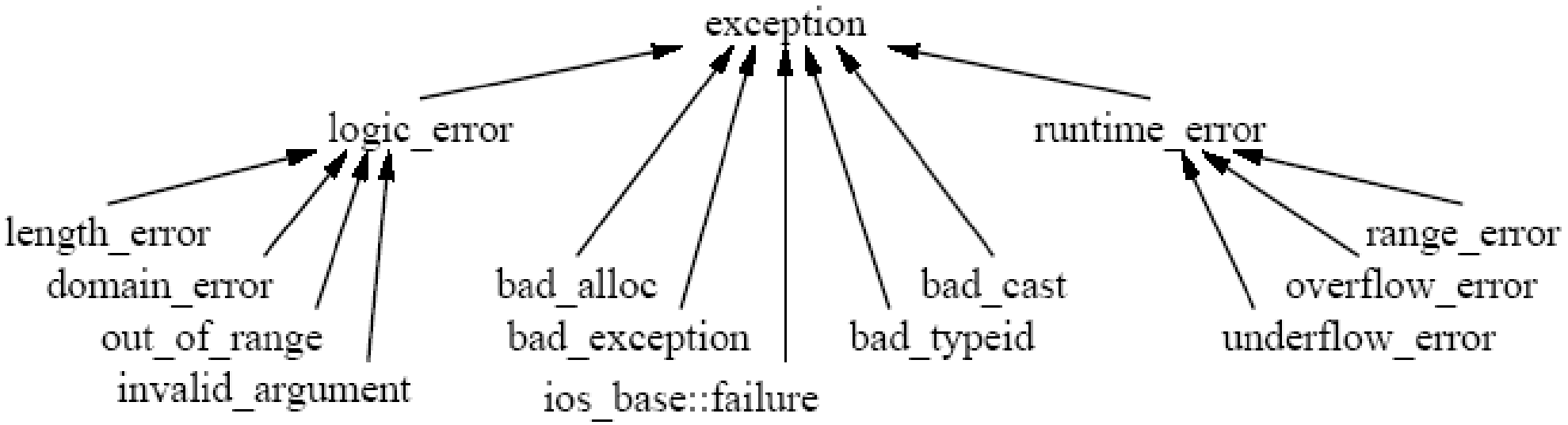
```
int main()
try
{
   Chapuza ch;
   ch.f();
} catch(...) {
   cout << "En catch(...)" << endl;
}
```

Funciones *unexpected() y set_unexpected()*

- Si una especificación, o lista *throw miente*, el compilador llamará automáticamente a la función de la biblioteca estándar *unexpected()*, que llamará a *terminate()*.
- Podemos cambiar este comportamiento predefinido utilizando la función *set_unexpected()*.
- *unexpected()* no debe recibir ni devolver nada, debe llamar a una función como *exit()* o *abort()*.
- **Puede lanzar una excepción, incluso relanzar la misma.**
- Alternativa: añadir a la lista *throw* la excepción estándar *bad_exception*.

  Ej: void f() throw(X, std::bad_exception) {}

Jerarquía de excepciones estándares



Ejercicio:

1. Escriba las líneas que imprimiría el siguiente programa.
2. Diga qué anomalías hay.
3. Corríjalo reescribiendo el método *Objeto::lanzamiento()* para que lance un objeto creado dinámicamente, o su dirección, y el capturador de la excepción en *main()*.

```
#include <iostream>
using namespace std;
class Objeto {
public:
    Objeto(char const *nombre): nombre_(nombre) { cout << "Constructor de Objeto para " <<
                                                          nombre_ << endl; }
    ~Objeto() {  cout << "Destructor de Objeto para " << nombre_ << endl; }
    void lanzamiento() {
      Objeto o("'objeto local de lanzamiento()'");
      cout << "Método lanzamiento() para " << nombre_ << endl;
      throw &o;
    }
    void saludo() { cout << "Hola de parte de " << nombre_ << endl; }
private:
    char const *nombre_;
};
int main() {
    Objeto o("'objeto de main()'");
    try {
      o.lanzamiento();
    } catch(Objeto *o) {
      cout << "Excepción capturada" << endl;
      o->saludo();
    }
}
```