Programación Orientada a Objetos

Tratamiento de excepciones en C++

José Fidel Argudo Argudo
Francisco Palomo Lozano
Inmaculada Medina Bulo
Gerardo Aburruzaga García

UCA
Universidad de Cádiz

Versión 1.0

Tipos de erorres

Errores sintácticos

Impiden la compilación del código por no respetarse las reglas gramaticales del lenguaje de programación.

Errores lógicos

- El código puede compilarse y ejecutarse, pero se obtienen resultados imprevistos debido a un mal diseño del programa al incumplir la especificación del mismo.
- Un programa correcto no tiene errores lógicos.

Tipos de erorres

Errores de ejecución

- Se producen cuando se presentan casos o situaciones anormales no previstas en la especificación (p.e. formato incorrecto o ausencia de un fichero de datos, entrada inválida de datos del usuario, memoria insuficiente, etc.), ante las cuales el programa intenta ejecutar una operación imposible de realizar (p.e. división por cero, acceso a una dirección de memoria prohibida, llamada a una función incumpliendo sus precondiciones).
- El programa ofrecerá resultados inesperados, se bloqueará o terminará bruscamente.

Tratamiento de errores

Errores sintácticos

- Mensajes de errror del compilador.

Errores lógicos

- Depuradores
- Técnicas de prueba de programas
- Técnicas de verificación de programas

Errores de ejecución

- Depuradores para detectar las condiciones en que se producen.
- La prevención y el tratamiento adecuado de los casos anormales o excepcionales que los provocan incrementa la robustez de los programas.

Tratamiento de errores de ejecución

Desde el punto de vista del usuario del programa

- Programa no interactivo: Informar del error y terminar el programa limpiamente.
- Programa interactivo: Informar del error y devolver el programa a un estado desde el que pueda continuar.

Tratamiento de errores de ejecución

Desde el punto de vista del programador

- Un efecto de la aplicación de los principios de modularidad y abstracción (p.e., al usar bibliotecas) es que el punto donde se puede detectar un error de ejecución está separado del punto donde se puede tratar.
- Cuando el autor de una función detecta un problema que no puede tratar directamente, o no sabe cómo hacerlo, puede adoptar diversas estrategias:
  1. Terminar la ejecución del programa con `exit()`, `abort()`, `assert()`.
  2. Devolver un código de error desde la función en que se detecta el error para que sea tratado a partir del punto de llamada.
  3. Terminar la función dejando un código de error en una variable global, que podrá ser comprobada al regresar de la función.

Tratamiento de errores de ejecución

Desde el punto de vista del programador (cont.)

4. Llamar a una función que suministrará el cliente para el tratamiento de errores, aunque ésta tendrá que optar en última instancia por una de las alternativas anteriores.
5. Lanzar una excepción que incluya toda la información que pueda ser útil para el posterior tratamiento del error.

Excepciones en C++

Tratamiento de excepciones

Mecanismo del lenguaje de programación para reaccionar ante circustancias excepcionales, como errores de ejecución, transfiriendo el control a funciones especiales llamadas *manejadores de excepciones*.

Excepción

Objeto de cualquier tipo que se lanza (throw) cuando se detecta un error o problema y que se captura (catch) en otro punto del código donde se puede tratar (try).

Excepciones en C++

```
1  // Excepción lanzada y capturada
2  // en la misma función.
3  int fun()
4  {
5    // ...
6    try { // Código que puede generar
7          // excepciones.
8      if (condición_de_error)
9        throw 10;
10     // ...
11   }
12   catch(int e) {
13     cout << "Exc. capturada. "
14      "Cód. error " << e << endl;
15   }
16   return 0;
17 }
```

```
1  // Excepción lanzada en una función
2  // y capturada en otra.
3  void fun1() {
4    // ...
5    if (condición_de_error)
6      throw 10;
7    // ...
8  }
9  int fun2() {
10   // ...
11   try { // Código que puede generar
12         // excepciones.
13     fun1();
14     // ...
15   }
16   catch(int e) {
17     cout << "Exc. capturada. "
18      "Cód. error "<< e << endl;
19   }
20   return 0;
21 }
```

Excepciones en C++

Lanzamiento de excepciones

- La instrucción

```
throw expresión;
```

  lanza una excepción, es decir un objeto definido por la expresión, que puede ser de cualquier tipo (fundamental o definido por el programador).
- La función actual terminará en este punto y devolverá por valor el objeto lanzado, que lo recibirá el manejador apropiado.
- Los objetos locales se destruyen automáticamente, incluido el propio objeto lanzado del cual se devolverá una copia.
- El control no se transfiere al punto de llamada, sino al manejador del tipo de excepción lanzada que eventualmente exista en un nivel superior del programa.

Excepciones en C++

```
1  class Error {/* ... */};
2  Error e;                    // ctor. predeterminado Error()

4  throw "Error";              // Lanza un puntero const char*
5  throw 20;                   // Lanza un int
6  throw string("Error");      // Lanza un string.
7                              // Ctor. conversión string(const char*),
8                              // ctor. copia string(const string&) y
9                              // destructor ~string() del primer string.
10 throw e;                    // Lanza un objeto Error.
11                             // Ctor. copia Error(const Error&),
12                             // destructor e.~Error()
13 throw Error();              // Lanza un objeto Error.
14                             // Ctor. predeterminado Error(),
15                             // ctor. copia Error(const Error&),
16                             // destructor ~Error() del primer Error
17 throw new Error;            // Lanza un puntero Error*
18                             // Se crea dinámicamente un objeto con el
19                             // ctor predeterminado Error().
20                             // El manejador deberá liberar la memoria.
```

Excepciones en C++

¿Qué objeto lanzar?

Lo recomendable es definir para cada tipo de excepción una clase, que encapsule la información relevante para que el manejador que vaya a tratar con ella pueda hacerlo. Si no hay que incluir información para el manejador, basta con una clase vacía con un nombre apropiado.

```
1  class Nif {
2    unsigned int dni;
3    char letra;
4    bool letra_valida();
5  public:
6    class LetraInvalida {}; // Clase de excepción vacía
7    // Constructor
8    Nif(unsigned n, char ltr) : dni(n), letra(ltr)
9    { if (not letra_valida()) throw LetraInvalida(); }
10   // ...
11 };
```

Excepciones en C++

El bloque try

- Donde se pueda tratar un tipo de excepción se debe rodear el código que potencialmente genere (directa o indirectamente) esas excepciones con un bloque especial llamado bloque try.
- Un bloque try engloba código que contiene instrucciones throw o, con más frecuencia, que contiene llamadas a funciones que detectan problemas que no pueden resolver y lanzan excepciones con throw.
- En un programa se pueden establecer diferentes niveles de tratamiento de excepciones, puesto que:
  - Los bloques try se pueden anidar y
  - las funciones llamadas dentro de ellos, a su vez, pueden incluir otros bloques try en los que se llaman a otras funciones con bloques try..., y así sucesivamente.

Excepciones en C++

Captura de excepciones

- Inmediatamente a continuación de un bloque try debe escribirse al menos un manejador de excepción, habrá uno para cada tipo de excepción que se capture en esta parte del código.
- Un manejador de excepción se define con la palabra reservada catch seguida de la declaración de un parámetro del tipo de la excepción que captura.

```
try {
  // código que puede lanzar excepciones
} catch(TipoExcep1 idExcep1) {
  // Manejador de excepciones TipoExcep1
} catch(TipoExcep2 idExcep2) {
  // Manejador de excepciones TipoExcep2
}
// etc ...
```

Excepciones en C++

Captura de excepciones

- El nombre del parámetro se puede omitir si no se usa en el código del manejador.
- El parámetro se puede pasar por referencia y así se evita la copia del objeto recibido de la instrucción throw, que a su vez es una copia del objeto lanzado. (Una copia en vez de dos)
- Cuando dentro de un bloque try se genera una excepción, el control se pasa al primer catch cuyo parámetro es del mismo tipo que dicha excepción.
- Una vez ejecutado el manejador, el tratamiento de la excepción termina y se destruyen el parámetro y el objeto lanzado. El resto de manejadores se ignora y el control pasa a la instrucción posterior a la serie de catch del bloque try.
- Si en el bloque try no se lanza ninguna excepción, se termina de ejecutar el mismo y se ignoran todos los catch asociados.

Excepciones en C++

```
1  Nif lee_nif() {
2    for (;;) {
3      cout << "Por␣favor,␣introduzca␣su␣número␣"
4              "de␣DNI␣y␣su␣letra␣del␣NIF:␣";
5      unsigned n;
6      char c;
7      cin >> n >> c;
8      try {
9        Nif nif(n, c); // posible excepción
10       return nif;    // salida de la función
11     } catch(LetraInvalida) {
12       cerr << "Letra␣inválida.␣Por␣favor,␣repita.\a" << endl;
13     } // se repite el bucle
14   }
15 }
```

Excepciones en C++

Captura de excepciones

- El orden de los manejadores de un bloque try es importante:
  - Si existe una relación jerárquica entre los tipos de excepciones manejadas en ese bloque, entonces el manejador de un tipo base capturará las excepciones del tipo derivado.
  - También hay que tener en cuenta que un puntero genérico, void*, capturará cualquier puntero.
- El orden adecuado de los manejadores es de los de tipos más específicos a los de tipos más generales.

```
1   catch(void*) { }            // Captura cualquier puntero.
2   catch(char*) { }            // Mal, nunca se llamará.

4   catch(ErrorBase&) { }       // Captura ErrorBase y ErrorDerivado.
5   catch(ErrorDerivado&) { } // Mal, nunca se llamará.
```

Excepciones en C++

Manejador universal

- Una excepción generada en un bloque `try` y no capturada por ninguno de sus manejadores asociados terminará la ejecución del bloque y será transferida al nivel externo superior de tratamiento de excepciones para buscar un manejador adecuado.
- El manejador universal, `catch(...)`, captura cualquier tipo de excepción, por lo que se puede añadir al final de la lista de manejadores de un bloque `try` para capturar cualquier excepción no capturada por los anteriores.
- La excepción no se pierde, pero no podemos hacer gran cosa con ella, sólo darle un tratamiento básico, puesto que no conocemos el objeto lanzado ni su tipo.

Excepciones en C++

```
1  try {
2    // código que puede lanzar excepciones
3  } catch(TipoExcep1 idExcep1) {
4    // Manejador de excepciones TipoExcep1
5  } catch(TipoExcep2 idExcep2) {
6    // Manejador de excepciones TipoExcep2
7  }
8  // etc ...
9  catch(...) { // Manejador universal
10   cerr << "Se ha producido un error, pero no se sabe cuál.\n";
11   //...
12 }
```

Excepciones en C++

Relanzamiento de excepciones

- Una excepción capturada en un manejador puede ser transferida al nivel superior mediante la instrucción `throw;` (sin expresión).
- El relanzamiento puede ser conveniente por diversos motivos:
  - En la función actual no se puede tratar el error detectado,
  - o sólo se puede tratar parcialmente.
  - Estamos en el manejador universal y no tenemos, por tanto, ninguna información sobre la excepción producida.

Excepciones en C++

```
1  try {
2    try {
3      throw 10.0;
4    } catch(int i) {
5      cerr << "Error␣" << i << "␣capturado.\n";
6    } catch(...) { // Manejador universal
7      cerr << "Se␣ha␣producido␣un␣error,␣pero␣no␣se␣sabe␣cuál.\n";
8      throw;  // Relanzamiento a nivel superior
9    }
10 } catch(double d) {
11   cerr << "El␣error␣desconocido␣era␣" << d << endl;
12 }
```

Salida que se obtendrá:

```
Se ha producido un error, pero no se sabe cuál.
El error desconocido era 10
```

Excepciones en C++

Lista throw (desaconsejada en C++11)

- Forma parte de la declaración de una función o método e informa de qué excepciones exactamente puede lanzar.
- El usuario de la función o método puede escribir los manejadores apropiados para capturarlas.
- La lista vacía significa que no se lanzará ninguna excepción.
- La falta de lista significa que la función puede lanzar cualquier excepción, o ninguna.

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

Excepciones en C++

Lista throw (desaconsejada en C++11)

- En el estándar C++11 se desaconseja el uso de listas de especificación de excepciones por razones de eficiencia del código objeto generado por el compilador.
- C++11 ha introducido la palabra reservada noexcept para declarar funciones que no lanzan excepciones. A continuación de la lista de parámetros se añade `noexcept(expresión)`, donde el argumento es opcional y es una expresión lógica constante, que si es `true` significa que la función no lanza excepciones.
- `noexcept(true)` equivale a `noexcept` y tiene el mismo significado que `throw()`, declara una función que no lanza excepciones.

Excepciones en C++

Terminación anormal

- Una excepción no capturada en un nivel interno se pasa sucesivamente a los niveles externos superiores hasta que en alguno de ellos un manejador la capture.
- ¿Qué ocurre si la excepción alcanza el nivel más alto, el de la función `main()`, y ningún manejador la captura?
- El programa tiene que terminar inmediatamente, debido a que el proceso de tratamiento de excepciones no puede continuar.

Error de programación

Una excepción no capturada debe considerarse un error lógico.

Solución

Incluir en la función `main()` un bloque `try` cuyo último manejador sea el universal.

Excepciones en C++

Función `terminate()`

- El compilador envuelve todo el código del programa en un bloque try global con un manejador universal que capturará cualquier excepción que le llegue. Este manejador llama a la función `terminate()`.
- `terminate()` es una función declarada en la cabecera `<exception>` que originalmente llama a la función `abort()` y, en consecuencia, el programa termina repentinamente sin cerrar ficheros abiertos, ni volcar buffers y sin llamar a los destructores de los objetos que queden en memoria.
- A parte de la utilidad descrita, el programador puede usar esta función como una alternativa a `abort()`, siempre que necesite terminar un programa anormalmente.

Excepciones en C++

Función `set_terminate()`

- Es muy conveniente cambiar el brusco comportamiento de `terminate()` escribiendo una nueva función que termine el programa de mejor forma que `abort()`.
- Función de terminación:
  - `void terminar();`
  - No debe relanzar ninguna excepción.
  - No debe acabar con `return`.
  - Debe arreglar lo que se pueda y
  - terminar el programa con normalidad con `exit()`.
- `set_terminate()` es una función declarada en la cabecera `<exception>` que instala una nueva función de terminación que será llamada por `terminate()`.

```
terminate_handler set_terminate(terminate_handler f) noexcept;
```

Excepciones en C++

```
1  // Ejemplo de uso de set_terminate()
2  // y de excepciones no capturadas

4  #include <exception>
5  #include <iostream>
6  #include <cstdlib>
7  using namespace std;

9  void terminator()
10 {
11   cerr << "Sayonara, baby!" << endl;
12   exit(EXIT_FAILURE);
13 }

15 void (*terminate_anterior)() = set_terminate(terminator);
```

Excepciones en C++

```
17 class Chapuza {
18 public:
19   class Fruta {};
20   void f() {
21     cout << "Chapuza::f()" << endl;
22     throw Fruta();
23   }
24   ~Chapuza() { throw 'c'; }
25 };

27 int main()
28 try {
29   Chapuza ch;
30   ch.f();
31 } catch(...) {
32   cout << "En␣catch(...)" << endl;
33 }
```

Excepciones en C++

Función `unexpected()`

- Si una función lanza una excepción que no está en su lista throw, se llama automáticamente a la función `unexpected()`.
- `unexpected()` es una función declarada en la cabecera `<exception>` que originalmente llama a `terminate()`.
- La función `set_unexpected()`, declarada en `<exception>`, permite definir un nuevo comportamiento para `unexpected()`.
- Función de excepción inesperada:
  - `void excep_inesperada();`
  - No debe acabar con `return`, sino terminar el programa con `exit()` o `abort()`.
  - O bien, puede relanzar la excepción o lanzar otra distinta, en cuyo caso la búsqueda del manejador comenzará en la llamada a la función que lanzó la excepción inesperada.

Excepciones en C++

Función `unexpected()`

- En C++11 una función declarada `noexcept` que lanza una excepción provoca la llamada a la función `unexpected()`.

Excepciones estándares

Excepciones estándares

- Excepciones definidas en la biblioteca de C++:
  - Algunas pueden ser lanzadas, ante determinadas circustancias, por operadores de C++ y también por diversas funciones y clases de la biblioteca estándar.
  - Otras simplemente están incluidas como otro componente más de la biblioteca estándar a disposición del usuario.
- Están organizadas en una jerarquía cuya raíz es la clase `exception` (declarada en la cabecera `<exception>`), la cual tiene un método virtual `what()` que devuelve una cadena de bajo nivel (`const char*`) con el nombre o la descripción de la naturaleza de la excepción.

Excepciones estándares

Jerarquía de excepciones estándares

- De esta clase base general `exception` se derivan todas las excepciones estándar:
  - `bad_alloc` (`<new>`). Lanzada por los operadores `new` y `new []` ante un fallo de asignación de memoria.
  - `bad_exception` (`<exception>`). Lanzada, bajo determinadas condiciones, cuando una función lanza una excepción que no está en su lista `throw`.
  - `bad_cast` (`<typeinfo>`). Lanzada por el operador `dynamic_cast` cuando recibe una referencia que no es del tipo esperado.
  - `bad_typeid` (`<typeinfo>`). Lanzada por el operador `typeid` cuando se le pasa un puntero nulo.
  - `bad_function_call` (C++11, `<functional>`). Lanzada cuando se invoca a un objeto función vacío.
  - `logic_error` (`<stdexcept>`). *Descripción en págs. ss.*
  - `runtime_error` (`<stdexcept>`). *Descripción en págs. ss.*

Excepciones estándares

Erores lógicos y de ejecución

- Los constructores de todas las excpeciones de tipo `logic_error` y `runtime_error` admiten una cadena de caracteres como único parámetro con el texto descriptivo que devolverá el método `what()`.

Excepciones estándares

Erores lógicos

- `logic_error` (`<stdexcept>`). Denota un error lógico del programa, por lo que podría ser lanzada en fase de depuración, pero no cuando el programa esté terminado.
  - `length_error` (`<stdexcept>`). Indica un intento de exceder el límite definido en la implementación como tamaño máximo de un objeto.
  - `domain_error` (`<stdexcept>`). Informa del incumplimiento de las precondiciones de una función.
  - `invalid_argument` (`<stdexcept>`). Representa que un parámetro de una función tiene un valor no válido. Es lanzada por algunos componentes de la biblioteca estándar.
  - `out_of_range` (`<stdexcept>`). Indica que el parámetro de una función tien un valor fuera del rango válido. Es lanzada por las clases `vector`, `deque`, `string` y `bitset`.

Excepciones estándares

Erores de ejecución

- `runtime_error` (<stdexcept>). Errores provocados en tiempo de ejecución por causas externas al programa.
  - `range_error` (<stdexcept>). Informa de la violación de una postcondición o de un error de rango en los cálculos internos de una función.
  - `overflow_error` (<stdexcept>). Error aritmético de desbordamiento superior.
  - `underflow_error` (<stdexcept>). Error aritmético de desbordamiento inferior.

Tratamiento de excepciones estándares

```
1  #include<exception>
2  #include<stdexcept>

4  // ...

6  try {

8    // código que puede lanzar excepciones

10 } catch(bad_alloc& e) { // Manejador específico de excepción estándar
11   cerr << "Error de memoria: " << e.what() << endl;
12 } catch(overflow_error&) { // Manejador específico de excepción estándar
13   cerr << "Error de desbordamiento superior.\n";
14 } catch(exception& e) { // Manejador genérico de excepciones estándares
15   cerr << "Error: " << e.what() << endl;
16 } catch(...) { // Manejador de otras excepciones
17   cerr << "Error desconocido.\n";
18   //...
19 }
```

Tratamiento de excepciones estándares

```
1  // hora.h
2  #include<stdexcept>

4  class Hora {
5  public:
6    class Incorrecta : public std::runtime_error {
7    public:
8      explicit Incorrecta(const char* s) : std::runtime_error(s) {}
9      // hereda el método público what() de runtime_error
10   };
11   Hora(int h, int m) : h(h), m(m)
12   {
13     if (h < 0 || h > 23) throw Incorrecta("horas fuera de rango");
14     if (m < 0 || m > 59) throw Incorrecta("minutos fuera de rango");
15   }
16   //...
17 private:
18   int h, m;  // horas y minutos
19 };
```

Tratamiento de excepciones estándares

```
1  #include<iostream>
2  #include "hora.h"

4  int horas, minutos;

6  // Se obtienen valores para horas y minutos
7  // ...
8  try {
9    Hora horeja(horas, minutos);
10   // Hora correcta
11   // ...
12 } catch (Hora::Incorrecta& hora_mala) {
13   std::cerr << "Hora␣errónea:␣" << hora_mala.what() << std::endl;
14 }
15 // ...
```