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 2.0

Índice

1. Errores de programación
   - Tipos de errores
   - Tratamiento de errores
2. Excepciones en C++
   - Introducción
   - Lanzamiento de excepciones
   - Bloque try
   - Captura de excepciones
   - Relanzamiento de excepciones
   - Especificación de excepciones
   - Terminación anormal
   - Excepciones en constructores
   - Excepciones estándares

Tipos de errores

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 errores

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, etc.).
- El programa ofrecerá resultados inesperados, se bloqueará o terminará bruscamente.

Tratamiento de errores

Errores sintácticos

- Mensajes de error 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 suministrada por el cliente para el tratamiento de errores, aunque esta 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.

Introducción

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).
- Una excepción propaga información sobre un error de programación desde el punto donde es detectado hasta el punto donde es tratado.

Introducción

```
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     cerr << "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     cerr << "Exc. capturada. "
18      "Cód. error "<< e << endl;
19   }
20   return 0;
21 }
```

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.

Lanzamiento de excepciones

```
1  class Error {/* ... */};
2  Error e;                    // ctor. predeterminado Error()
3
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.
```

Lanzamiento de excepciones

¿Qué objeto lanzar?

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

```
1  class Nif {
2  public:
3    class LetraInvalida {}; // Clase de excepción vacía
4    Nif(unsigned n, char ltr) : dni(n), letra(ltr) // Constructor
5    {
6      if (not letra_valida()) throw LetraInvalida();
7    }
8    // ...
9  private:
10   unsigned int dni;
11   char letra;
12   bool letra_valida();
13 };
```

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, 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` desde los que se llaman a otras funciones con bloques `try`..., y así sucesivamente.

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 ...
```

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.

Captura de excepciones

```
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(Nif::LetraInvalida) {
12       cerr << "Letra␣inválida.␣Por␣favor,␣repita.\a" << endl;
13     }
14   } // Fin del bucle.
15 }
```

Captura de excepciones

Orden de manejadores

- El orden de los manejadores de un bloque `try` es importante:
  - Si existe una relación jerárquica entre los tipos de excepciones tratadas 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 desde 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.
```

Captura de excepciones

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.

Captura de excepciones

```
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 }
```

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.

Relanzamiento de excepciones

```
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:

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

Especificación de excepciones

- El especificador noexcept se usa para declarar si una función puede o no lanzar excepciones. A continuación de la lista de parámetros se añade `noexcept(expresión)`, donde el argumento es una expresión lógica constante que se puede omitir.
- Si el valor de la expresión es `true`, significa que la función no lanza excepciones; si es `false`, se considera que la función es potencialmente lanzadora de excepciones.
- `noexcept` equivale a `noexcept(true)`.
- Salvo en destructores, omitir el especificador `noexcept` es equivalente a una declaración de función `noexcept(false)`, aunque en ningún caso lance excepciones.
- Los destructores son declarados como `noexcept` por omisión, ya que es muy mala idea que lancen excepciones.

Terminación anormal

- Cuando una excepción es lanzada pueden surgir problemas que le impidan alcanzar el manejador correspondiente:
  1. Una excepción lanzada en un punto cualquiera del programa alcanza el nivel de la función `main()` sin ser capturada.
  2. Se lanza una excepción y, antes de que sea capturada, se lanza una segunda.
  3. Una función incumple su especificación `noexcept` lanzando una excepción indebida.
- En estos casos, el programa no puede continuar y se llama automáticamente a la función estándar `terminate()`.

Error de programación

La terminación anormal del tratamiento de excepciones debe considerarse un error lógico.

Terminación anormal

Función `terminate()`

- Función llamada automáticamente cuando una excepción no puede llegar a su manejador.
- `terminate()`, declarada en la cabecera `<exception>`, llama a la función `abort()` y, en consecuencia, el programa termina de repente sin cerrar ficheros abiertos, ni volcar *buffers* y sin llamar a los destructores de los objetos que queden en memoria.
- Aparte de la utilidad descrita, el programador puede usar `terminate()` como una alternativa a `abort()`, siempre que necesite terminar un programa anormalmente.

Terminación anormal

```
1  // Ejemplo de terminación por doble lanzamiento
2  #include <iostream>

4  class Chapuza {
5  public:
6    class Fruta {};
7    void f() { // noexcept(false) por omisión
8      std::cout << "En␣Chapuza::f()\n";
9      throw Fruta();
10   }
11   ~Chapuza() noexcept(false) // por omisión, noexcept ==> terminate()
12   { throw 'c'; }
13 };

15 int main()
16 try {
17   Chapuza ch;
18   ch.f();
19 } catch(...) { std::cout << "En␣catch(...)\n"; }
```

Terminación anormal

```
1  #include <iostream>
2
3  class Chapuza {
4  public:
5    class Fruta {};
6    void f() { // noexcept(false) por omisión
7      std::cout << "En␣Chapuza::f()\n";
8      throw Fruta();
9    }
10   ~Chapuza() { // noexcept por omisión, bien
11     try { throw 'c'; }
12     catch(char c) {
13       std::cout << c << "␣lanzado␣y␣capturado␣en␣~Chapuza()\n";
14     }
15   }
16 };
17
18 int main()
19 try {
20   Chapuza ch;
21   ch.f();
22 } catch(...) { std::cout << "En␣catch(...)\n"; }
```

Excepciones en constructores

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;

5  struct Pieza {
6    Pieza(bool b)
7    { if (!b)
8        throw "Error en Pieza"; }
9  };

11 struct Aparato {
12   Aparato(bool b, const char* ns) :
13     // Posible fuga de memoria
14     nserie{new char[strlen(ns) + 1]},
15     p{b}
16   { strcpy(nserie, ns); }
17   ~Aparato() { delete[] nserie; }
18 private:
19   char* nserie;
20   Pieza p;
21 };
```

```
23 int main() try
24 {
25   Aparato ap1(true, "0001"),
26           ap2(false, "0002");
27   return 0;
28 } catch (const char* e) {
29     cerr << "Exc. capturada: "
30          << e << '\n';
31     return 1;
32 }
```

Excepciones en constructores

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;

5  struct Pieza {
6    Pieza(bool b)
7    { if (!b)
8        throw "Error en Pieza"; }
9  };

11 struct Aparato {
12   Aparato(bool b, const char* ns) try
13     : nserie{new char[strlen(ns) + 1]}, p{b}
14   { strcpy(nserie, ns); }
15   catch(const char* e) {
16     delete[] nserie;
17     // Relanzamiento implícito
18   }
19   ~Aparato() { delete[] nserie; }
20 private:
21   char* nserie;
22   Pieza p;
23 };
```

```
25 int main() try
26 {
27   Aparato ap1(true, "0001"),
28           ap2(false, "0002");
29   return 0;
30 } catch (const char* e) {
31     cerr << "Exc. capturada: "
32          << e << '\n';
33     return 1;
34 }
```

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 un componente más de la biblioteca estándar a disposición del programador.
- Están organizadas en una jerarquía de clases cuya raíz es la clase `exception` (declarada en la cabecera `<exception>`), la cual tiene el método

```
virtual const char* what() const noexcept;
```

que devuelve una cadena con el nombre o descripción de la excepción.

Excepciones estándares

Excepciones derivadas de la clase `exception`		
Tipo	**Cabecera**	**Descripción**
`bad_alloc`	`<new>`	Fallo al asignar memoria con los operadores `new` o `new[]`.
`bad_cast`	`<typeinfo>`	Expresión de tipo inválido en operador `dynamic_cast`.
`bad_typeid`	`<typeinfo>`	El operador `typeid` recibe un puntero nulo.
`bad_function_call`	`<functional>`	Llamada a un objeto función vacío.
`logic_error`	`<stdexcept>`	*Descripción en págs. ss.*
`runtime_error`	`<stdexcept>`	*Descripción en págs. ss.*

Excepciones estándares

Errores lógicos y de ejecución

- Las excepciones de la clase `logic_error` se deben a errores lógicos, como son la violación de especificaciones de funciones o de invariantes de clases. Podrán ser lanzadas en fase de depuración, pero no cuando el programa sea correcto.
- Las excepciones de la clase `runtime_error` se producen durante la ejecución debido a causas externas, por lo que pueden aparecer aunque el programa sea correcto.
- El parámetro que admiten los constructores de estas clases (al igual que los de sus derivadas) es el texto descriptivo que devolverá el método `what()`.

```
logic_error(const std::string& s);
logic_error(const char* s);
runtime_error(const std::string& s);
runtime_error(const char* s);
```

Excepciones estándares

Excepciones derivadas de la clase `logic_error`		
Tipo	**Cabecera**	**Descripción**
`length_error`	`<stdexcept>`	Excedido el límite de tamaño máximo de un objeto.
`domain_error`	`<stdexcept>`	Incumplimiento de las precondiciones de una función.
`invalid_argument`	`<stdexcept>`	Recibido valor inválido en parámetro de función.
`out_of_range`	`<stdexcept>`	Parámetro de función fuera del rango válido.

Excepciones estándares

Excepciones derivadas de la clase `runtime_error`		
Tipo	**Cabecera**	**Descripción**
`range_error`	`<stdexcept>`	El resultado de una función o los cálculos intermedios están fuera del rango del tipo de destino.
`overflow_error`	`<stdexcept>`	Error aritmético de desbordamiento superior.
`underflow_error`	`<stdexcept>`	Error aritmético de desbordamiento inferior.
`system_error`	`<system_error>`	Errores de bajo nivel y del sistema operativo.

Excepciones estándares

Excepciones derivadas de la clase `system_error`		
Tipo	**Cabecera**	**Descripción**
`ios_base::failure`	`<ios>`	Fallo de entrada/salida.
`filesystem::filesystem_error`	`<filesystem>`	Fallo del sistema de ficheros.

Tratamiento de excepciones estándares

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

4  // ...
5    try {

7    // código que puede lanzar  excepciones

9  } catch(bad_alloc& e) { // Manejador específico de excepción estándar
10       cerr << "Error␣de␣memoria:␣" << e.what() << endl;
11 } catch(overflow_error&) { // Manejador específico de excepción estándar
12       cerr << "Error␣de␣desbordamiento␣superior.\n";
13 } catch(runtime_error& e) { // Manejador genérico de runtime_error
14       cerr << "Error:␣" << e.what() << endl;
15 } catch(exception& e) { // Manejador genérico de excepciones estándares
16       cerr << "Error:␣" << e.what() << endl;
17 } catch(...) { // Manejador de otras excepciones
18       cerr << "Error␣desconocido.\n";
19   //...
20 }
```

Tratamiento de excepciones estándares

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

4  class Hora {
5  public:
6    class Incorrecta : public std::logic_error {
7    public:
8      explicit Incorrecta(const char* s) : std::logic_error(s) {}
9      // hereda el método público what() de logic_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 // ...
```