Práctica 2
Excepciones en C++

Gerardo Aburruzaga, Inmaculada Medina, Francisco Palomo

Marzo de 2007

Índice

1. Introducción **2**

2. Lanzamiento de excepciones **2**
2.1. Excepciones 3
2.2. La instrucción `throw` 4
2.3. La lista `throw` 6

3. Gestión de excepciones **7**
3.1. La instrucción `try` 7
3.2. La instrucción `catch` 8
3.2.1. Captura de cualquier excepción 9
3.2.2. Relanzamiento de una excepción 10
3.3. La función *terminate()* 10
3.4. La función *unexpected()* 12

4. Jerarquía de excepciones **14**
4.1. Excepciones estándares 16

5. Programación con excepciones **18**
5.1. Evitar excepciones 18
5.1.1. No para eventos asíncronos 18
5.1.2. No para errores ordinarios 18
5.1.3. No para control de flujo 18
5.1.4. No si se pueden evitar 19
5.1.5. No para código antiguo 19
5.2. Empleo de excepciones 19

Resumen

Donde se explica el mecanismo de excepciones, empleado por C++ para tratar principalmente las entradas de datos inválidas.

1. Introducción

Supongamos un pequeño (relativamente) programa no interactivo, como uno de manejo de ficheros de un sistema operativo como DOS (o derivados) o Unix; pongamos *mkdir*, que sirve para crear un directorio. Sigamos suponiendo que damos la orden pero sin el parámetro correspondiente, o no tenemos permiso para crear un directorio. ¿Qué debería hacer el programa? Claramente, indicárnoslo mediante el oportuno mensaje de error y acabar inmediatamente, pues no es un programa interactivo.

Supongamos ahora un programa interactivo más complejo, como un editor o procesador de textos de un paquete ofimático. Pulsamos el botón de abrir un fichero y en el cuadro de diálogo escribimos un nombre de fichero incorrecto, o que por algún motivo no podemos abrir. ¿Sería lógico que el programa acabara con un mensaje de error? Más bien debería, sí, informarnos de él, pero dándonos la oportunidad de seguir intentándolo o cancelar la operación.

Bajando de nivel, supongamos que escribimos un programa que hace uso de una biblioteca hecha por otras personas; no disponemos del código fuente, sólo del archivo con los módulos objeto y los ficheros de cabecera. Si llamamos a una función de esa biblioteca pasándole un parámetro con un valor incorrecto, ¿sería lógico que, en tiempo de ejecución, el programa se acabara dando un diagnóstico en la salida de errores? Nuestro programa puede ser gráfico, y esa salida de errores estar conectada a la consola, que no veríamos, o a un fichero de registro, que tampoco veríamos inmediatamente. Quizá nuestro programa sea demasiado «importante» como para acabar en ese punto sin posibilidad de volver atrás a corregir el error.

Todo esto indica que hay diversas formas de tratar los errores, unas más adecuadas que otras según el caso.

El autor de una biblioteca, que en C++ seguramente no se limita a un conjunto de funciones, sino de clases, cada una con sus correspondientes métodos, probablemente se topará con diversos errores que pueda cometer el usuario, sin saber qué hacer con ellos. Las *excepciones*, que vemos en esta práctica, son un mecanismo de control que permite manejar este tipo de errores o problemas de forma que cuando en una función (externa, o interna a una clase), normalmente de una biblioteca, se prevea el encuentro con uno de ellos, se detecte pero se deje el manejo del error en manos del usuario, del *cliente* de la biblioteca o función.

Cuando se detecta un error, el autor de la función *lanza* una excepción. El cliente, si quiere, puede *capturarla* y obrar como le plazca.

2. Lanzamiento de excepciones

Antes de entrar en materia, veamos las alternativas que tiene el autor de una biblioteca en ausencia del mecanismo de excepciones.

Cuando este detecta un problema que no puede tratar directamente o no sabe cómo, la función o método puede optar por:

1. terminar el programa.

   Esto es lo que ocurre incluso en el caso de excepciones cuando el cliente no las captura. A veces, esto es lo natural, como en el primer ejemplo mencionado en §1, pero en otros casos hay cosas mejores que se pueden

hacer. El autor de una biblioteca normalmente no sabe nada acerca del propósito y estrategia general del programa que la usa, y puede no ser admisible emplear *exit()* y aún menos *abort()* para terminar el programa.

Sin embargo, esta es la alternativa que hemos empleado hasta ahora en nuestros ejercicios mediante el empleo de la macro *assert()*, que prueba una condición o expresión y, si es falsa (0), presenta un diagnóstico en la salida de errores y llama a *abort()* para terminar el programa abruptamente. Esta técnica puede ser admisible en la fase de depuración de un programa, y, afortunadamente, todas las aserciones pueden anularse sin tener que borrarlas una a una recompilando el programa con la macro *NDEBUG* definida, se supone que una vez comprobado el funcionamiento correcto. Pero a veces este está determinado por datos que el programa adquiere en el momento de la ejecución, con lo que no vale simplemente anular las aserciones, ni dejarlas tampoco.

2. devolver un valor que represente «error».

   Esta es la alternativa empleada por muchas funciones de la biblioteca estándar de C, donde no existe el mecanismo de excepciones, y por otras muchas bibliotecas. Esta técnica no vale cuando cualquier valor devuelto por la función pueda ser un valor correcto, evidentemente. Y aunque sí se pueda devolver un valor con el significado claro de «error» sin que interfiera con otro válido, el cliente tendría que estar, en cada llamada a la función, comprobando el valor devuelto, lo que hace el código más pesado de leer y aumenta su tamaño, además de que puede que el cliente se olvide de comprobar el valor devuelto.

3. devolver un valor «legal» y dejar el programa en un estado «ilegal».

   Esta alternativa también es empleada donde no se puede la anterior en algunas funciones de la biblioteca estándar de C, sobre todo funciones matemáticas, y, a veces en conjunción con la anterior, en muchas funciones de la biblioteca POSIX.1. Dichas funciones dejan un valor en una variable global llamada *errno*, definida en la biblioteca estándar de C, que indica el error producido. Pero el cliente puede olvidarse de comprobar el estado de esta variable; y el empleo de variables globales es delicado en presencia de concurrencia.

4. llamar a una función que pueda suministrar el cliente en caso de «error».

   Esta alternativa es quizás la mejor de todas, pero en ausencia del mecanismo de excepciones, la función se encuentra en uno de los casos anteriores, si bien al menos es el cliente el que la define.

2.1. Excepciones

¿Qué es exactamente una «excepción»? Bajo el punto de vista del compilador es un objeto; es decir, un valor, de un tipo fundamental incorporado en el lenguaje o de un tipo definido por el usuario (normalmente una clase); un objeto que se *lanza* (en inglés, *throw*) en una función donde se detecta un error o problema, y que se puede recoger o capturar (en inglés, *catch*) en otra donde se pueda tratar o procesar (en inglés, *try*) el suceso. Si no se capturara, está previsto un comportamiento bien definido que veremos más adelante.

2.2. La instrucción `throw`

Una excepción se lanza mediante una instrucción `throw` con la sintaxis siguiente:

`throw` [*expresión*];

La forma donde se omite la *expresión* se emplea para relanzar una excepción a un nivel superior. Estudiaremos este caso más adelante, en §3.2.2.

Primero, con la *expresión* se crea un objeto del tipo correspondiente (y por tanto se llama a su constructor si se trata de un tipo definido por el usuario); ese objeto no existe durante una ejecución «normal» del programa: sólo se creará si se lanza la excepción.

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 */
  ...
  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
                    */
  ...
}
```

Normalmente los objetos que se lanzan son de una clase especial que creará el programador para representar los errores, por lo que la primera forma no se debe emplear: no tiene sentido crear un objeto de tipo *Error* si no se va a producir ninguno.

Entonces la función acaba y devuelve por valor dicho objeto como si de un `return` se tratara, aunque el tipo de devolución de esta sea otro distinto, o *void*, o incluso ni siquiera este, como pasa en los constructores.

Sin embargo, el control no pasa en este caso al punto de llamada a la función, como ocurre con un `return` normal, sino a un manejador de excepción apropiado, si lo hay, que puede estar bastante *lejos* de ese punto de llamada.

Además, los objetos automáticos que se hubieran creado con éxito hasta el lanzamiento de la excepción (y solo ellos) se destruyen, llamando a sus destructores, puesto que la función termina. El objeto lanzado se destruye sólo cuando se acaba el manejador correspondiente (los manejadores se explican en §3.2).

¿Y qué objeto lanzar? Podría lanzarse un simple número que significara un código de error, o una cadena de caracteres con un mensaje. No obstante, es mucho más recomendable crear una clase o tipo para cada error diferente. Algunos errores pueden estar relacionados entre ellos, y así deberían estar también las clases que diseñáramos para representarlos. La idea es encapsular la información que debe llevar la excepción en una clase, incluyendo su nombre, para que el usuario que vaya a tratar con ellas sepa su significado. Si no hace falta incluir ninguna información que haya que pasar al manejador, basta con una clase vacía[1] con un nombre apropiado.

Ejemplo:

La siguiente clase representa un número de identificación fiscal, o NIF, que comprende el número del documento nacional de identidad o DNI más una letra de control calculada en función de dicho número.

```
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();
  }
  // ...
};
```

Se ha creado una clase llamada *LetraInvalida* para representar el error consistente en que la letra suministrada al constructor no sea la correcta. Se considera que la información que lleva el propio nombre de la clase es suficiente, por lo que se define vacía. Se ha creado dentro de la propia clase *Nif*, pues no tiene sentido fuera de ella: pertenece a esta clase (hay que acordarse, cuando se haga referencia a ella fuera de la clase, de utilizar el operador de resolución de ámbito).

No hace falta definir antes un objeto del tipo de la excepción y lanzarlo: tras `throw` hacemos una llamada al constructor (en el ejemplo, al no haberse definido ninguno, al predeterminado proporcionado automáticamente por el compilador, que no hace nada) para crear un objeto temporal anónimo que se copia en otro que es el que se lanza, y el primer temporal se destruye inmediatamente. Esta técnica es muy común: sólo debería crearse el objeto excepción cuando se produjera una. Las excepciones se supone que son (como parece evidente del nombre) excepcionales, y por tanto un programa con excepciones debería ejecutarse igual de rápido que otro sin ellas en el caso de que no se produzca ninguna.

[1] En C++, a diferencia de C, puede haber `structs` y clases sólo con funciones miembro, o incluso vacías.

Cuando se lanza una excepción en un constructor, el objeto no ha terminado de construirse, y no se llamará a su destructor. En el ejemplo anterior, esto no tiene ninguna importancia al ser los atributos simples números, pero sí empieza a tenerla cuando en el constructor se adquieren recursos que luego deban liberarse con el destructor.

2.3. La lista `throw`

Uno no tiene por qué informar a la persona que emplee nuestras clases o funciones de qué excepciones puedan lanzar, pero esto sería muy poco considerado porque significaría que dicha persona no podría estar segura de cómo capturar todas las posibles excepciones. Por supuesto, si tiene el código fuente, puede mirarlo buscando todas las instrucciones `throw`, pero muy a menudo una biblioteca se entrega en un archivo, ya compilada, sin fuentes. Por ello, C++ proporciona un mecanismo por el cual se puede informar educadamente al cliente de qué excepciones exactamente podrá lanzar una función o método, de forma que así este podrá escribir los manejadores apropiados cuando las capture. Este mecanismo se conoce como *especificación de excepciones* o *lista throw*, y forma parte de la declaración de la función, apareciendo tras la lista de parámetros.

Se reutiliza la palabra reservada `throw`, seguida de una lista de excepciones separadas por comas, entre paréntesis. La lista vacía significa que la función no lanzará ninguna excepción.

Ejemplo:

Siguiendo con la clase *Nif*, el constructor se escribiría mejor así:

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

Si la lista `throw` vacía significa que la función no va a lanzar ninguna excepción, la falta de la especificación de excepciones significa que la función puede lanzar cualquier excepción, o ninguna.

Ejemplo:

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

Las funciones anteriores no reciben ningún parámetro ni devuelven nada. La primera, *f()*, puede lanzar excepciones de los tres tipos especificados; *g()* podría lanzar cualquier excepción, o no lanzar ninguna, y *h()* no podrá lanzar ninguna excepción. Con esta información, el cliente sabe que cuando llame a *f()* debería capturar esos tres tipos de excepciones, y que cuando llame a *h()* no hará falta que se preocupe de capturar nada. En cambio, al llamar a *g()* no sabrá qué hacer a este respecto.

Es recomendable siempre, cuando se escribe un programa o biblioteca con excepciones, escribir la lista `throw` para cada función. Por seguir un buen estilo, para documentación, y para facilitarle la vida al que emplee nuestras funciones.

> Se podría pensar que las especificaciones de excepciones no están muy bien diseñadas, de forma que el ejemplo anterior debería ser mejor:
>
> ```
> void f() throw(MuyGrande, MuyChico, DivCero);
> void g() throw(...); // cualquier excepción
> void h();            // ninguna excepción
> ```
>
> Esto no es lo mejor por varios motivos. Uno, que no siempre se sabe con certeza qué excepción puede lanzar una función, porque aunque en ella no aparezca un `throw`, puede lanzar una excepción porque llame a otra función que sí la lance y no se recoja en esta, o porque un operador de C++ lance una excepción estándar. Y, lo principal, para mantener inalterado el código escrito antes de que se inventara el mecanismo de excepciones, pues funciones antiguas pueden lanzar de pronto excepciones inadvertidamente si llaman a otras funciones que se hayan actualizado y ahora empleen este mecanismo.

3. Gestión de excepciones

Si una función lanza una excepción, se supone que se capturará y se tratará el error en otro sitio. Una de las ventajas del manejo de excepciones de C++ es que permite a uno concentrarse en el problema que realmente se está intentando resolver, mientras que los errores que ese código pueda producir se tratarán en otro sitio.

3.1. La instrucción `try`

Si en una función se lanza una excepción, o se llama a una función que lanza una excepción (y así recursivamente), esa función acabará en el proceso del lanzamiento. Si no se desea que `throw` provoque el abandono de una función, se puede establecer un bloque especial dentro de la función donde se está intentando resolver el problema real de programación y que potencialmente puede generar excepciones. Este bloque se llama el «bloque *try*», y es un bloque ordinario precedido de la palabra reservada `try`. Si comprende a la función entera, puede sustituir completamente al bloque del cuerpo de la función.

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

En ausencia del manejo de excepciones, un programador cuidadoso tendría que «rodear» cada llamada a función de código que comprobara si su valor devuelto fuera correcto y actuara en consecuencia, incluso si se llamara muchas veces a la misma función. Con el manejo de excepciones, todo se mete en un bloque *try* sin comprobar explícitamente los errores. Esto implica un código más claro y fácil de leer, porque su objetivo o tarea principal no está mezclado con la comprobación de errores.

3.2. La instrucción `catch`

Por supuesto, una excepción lanzada debe acabar en algún sitio, y este es el *manejador de excepción*; habrá uno para cada tipo de excepción que se quiera capturar. Los manejadores de excepción se escriben inmediatamente tras el bloque `try`, y se nombran con la palabra reservada `catch`. Cada cláusula `catch` es como una función que no devuelve nada (ni siquiera *void*) y recibe un parámetro del tipo de la excepción que quiere capturar.

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

Se pueden emplear los identificadores como parámetros formales de una función, y pueden ser objetos, punteros o referencias, según el caso; si no hacen falta porque no se van a emplear en el manejador, se pueden omitir; esto es frecuente por ejemplo cuando la excepción a capturar es una clase vacía cuyo nombre ya posee toda la información necesaria.

Ejemplo:

Sigamos con la clase *Nif*. La siguiente función pide al usuario un NIF y construye un objeto *Nif*; se emplea la técnica de la *rectificación* o *reanudación* mediante bucle, frente al otro modelo de manejo de excepciones: la *terminación*.

```
Nif lee_nif()
{
  for (;;) {
    cout << "Por favor, introduzca su número "
            "de DNI y su letra del NIF: ";
    unsigned int n;
    char c;
    cin >> n >> c;
    try {
      Nif nif(n, c); // posible excepción
      return nif;    // salida de la función
    } catch(LetraInvalida) {
      cerr << "Letra inválida. Por favor, repita.\a" << endl;
    } // se vuelve al bucle «infinito»
  }
}
```

Cuando dentro de un bloque `try` se lanza una excepción, el control se pasa inmediatamente al primer `catch` cuya *signatura* coincida con el tipo de la excepción. Se buscan de arriba abajo hasta el primero coincidente y, una vez ejecutado su código, se destruye el objeto lanzado y la excepción se considera manejada. La búsqueda acaba aquí; no como en las sentencias `switch` donde había que acabar cada `case` con un `break` si no se quería pasar al siguiente. Una

vez ejecutado el manejador apropiado, el control pasa a la instrucción posterior a todos los manejadores `catch`.

Esto implica que hay que ser cuidadoso con el orden en que se escriben los manejadores. Los más generales hay que ponerlos en los últimos lugares.

EJEMPLO:

```
catch(void*) { }  // capturaría cualquier ptr., incluido char*
catch(char*) { }  // Mal, nunca se llamaría
```

Sin embargo, no se aplican conversiones automáticas definidas por el usuario, como puede verse en el siguiente programa:

```
1  /* Prueba de que no se aplican conversiones
2     definidas por el usuario en la búsqueda
3     de manejadores de excepciones.
4  */
5  #include <iostream>
6  using namespace std;
7
8  class Excepcion1 {};
9  class Excepcion2 {
10 public:
11   Excepcion2(Excepcion1&) { }   // ctor. de conversión
12 };
13
14 int main()
15   try
16 {
17   throw Excepcion1();  // Se lanza un obj. Excepcion1
18 }                      // pero
19 catch(Excepcion2) {    // no se convierte aquí a Excepcion2
20   cout << "En catch(Excepcion2)" << endl;
21 }
22 catch(Excepcion1) {   // por lo tanto se llega aquí
23   cout << "En catch(Excepcion1)" << endl;
24 }
```

Aunque esté definida la conversión desde *Excepcion1* a *Excepcion2*, esta conversión no se efectúa durante el manejo de excepciones, y se acaba en el manejador de *Excepcion1*.

3.2.1. Captura de cualquier excepción

Como se ha explicado en §2.3, si una función no tiene lista `throw`, puede lanzar cualquier tipo de excepción. Una solución para no dejar una excepción sin capturar es definir un manejador que capture cualquier excepción. Esto se consigue escribiendo elipsis en la lista de parámetros (al estilo de C):

Ejemplo:

```
catch(...) {
  cerr << "Se ha lanzado una excepción y la he capturado." << endl;
  // ...
}
```

Evidentemente, este manejador debe ser el último de la lista. Obsérvese que no tenemos ninguna información sobre el tipo de error producido ni podemos definir ningún parámetro formal donde copiar la excepción lanzada.

3.2.2. Relanzamiento de una excepción

A veces conviene relanzar la excepción capturada, por varios motivos:

- puede que en la función donde estemos sepamos o podamos tratar otros tipos de excepciones pero no este.
- puede que solo sepamos o podamos tratar parte del error, pero no todo.
- estamos en el manejador que captura cualquier excepción, el de los puntos suspensivos, y no tenemos por tanto información ninguna sobre la excepción.

En cualquiera de esos casos podemos relanzar la excepción producida con la instrucción `throw` sola:

Ejemplo:

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

La excepción relanzada será capturada con suerte en un nivel superior, hasta llegar a *main()*.

3.3. La función *terminate()*

A estas alturas puede que se esté preguntando: ¿y qué pasa si no se captura una excepción en ningún sitio? Si ninguno de los manejadores `catch` que siguen a un bloque `try` concuerda con una excepción lanzada, esta se pasa al siguiente contexto de mayor nivel; esto es, la función o bloque `try` donde se llamara al bloque que no ha capturado la excepción. Este proceso continúa hasta que en algún nivel un manejador capture la excepción: en este punto, ésta se considera «manejada» y no se sigue buscando.

Si ningún manejador captura la excepción en ningún nivel, siendo el superior la función *main()*, la excepción es «no capturada» o «no manejada». Este caso también se da cuando se lanza una nueva excepción antes de que la existente alcance a su manejador; el caso más común de esto ocurre cuando un constructor de un objeto excepción lanza a su vez otra.

Si una excepción no es capturada, se llama automáticamente a la función *terminate()*. Esta función está implementada como un puntero a función que no recibe ni devuelve nada. Si no se dice otra cosa, llama a la función *abort()*, de la biblioteca estándar de C, que acaba abruptamente la ejecución del programa; esto es, no se cierran ficheros abiertos, ni se vuelcan los *búferes*, ni se llama a los destructores de objetos globales o estáticos.

Se puede (y debe) cambiar este brusco comportamiento instalando 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 la cabecera estándar `<exception>`. La función *set_terminate()* devuelve un puntero a la función *terminate()* que se está reemplazando, por si se quiere restaurar luego.

De todas formas, aunque se haya instalado una función de terminación, no se llama a los destructores de los objetos globales ni estáticos, así que una alternativa mejor a dejar la responsabilidad a *terminate()* sería meter un bloque `try` en *main()* cuyo último capturador fuera el de la elipsis. Por lo general, una excepción no capturada debe considerarse un error de programación.

Nuestra función de terminación no debe recibir ningún parámetro ni devolver nada; además no debe regresar ni relanzar ninguna excepción, sino arreglar lo que se pueda y llamar a alguna función que termine el programa, como *exit()*, de la biblioteca estándar de C, declarada en `<cstdlib>`. Si se llama a *terminate()* se supone que es porque el problema no tiene recuperación posible.

Ejemplo:

En este ejemplo se muestra el empleo de *set_terminate()* y una excepción sin capturar.

terminator.cpp

```
/* Ejemplo de uso de set_terminate()
 * y de excepciones no capturadas
 */
#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();
  }
```

```
24    ~Chapuza() { throw 'c'; }
25  };
26
27  int main()
28    try
29  {
30    Chapuza ch;
31    ch.f();
32  } catch(...) {
33    cout << "En catch(...)" << endl;
34  }
```

La clase *Chapuza* no solo lanza una excepción en su método *f()*, sino en su destructor. Como se ha dicho, esta es una de las situaciones donde se produce una llamada a *terminate()*. Aunque parezca que se debería llegar al `catch` que captura todas las excepciones, se llama en su lugar a *terminate()* porque en el proceso de limpiar los objetos de la pila para manejar una excepción, se llama a otra en el destructor de *Chapuza*, y eso genera una segunda excepción, forzando la llamada a *terminate()*. Se deduce de esto que un destructor nunca debería lanzar una excepción ni provocar indirectamente el lanzamiento de una. Si se lanza una excepción en un destructor, debe tratarse allí mismo, metiendo en él un bloque `try` con los capturadores correspondientes.

> La función de la biblioteca estándar *uncaught_exception()*, declarada en `<exception>`, devuelve `true` si una excepción se ha lanzado pero no se ha capturado aún. Esto permite al programador especificar acciones diferentes en un destructor dependiendo de si un objeto se está destruyendo normalmente o como parte del proceso del mecanismo de excepciones.

3.4. La función *unexpected()*

Si una especificación de excepción, o lista `throw` (§2.3) *miente*, el compilador castigará nuestro pecado llamando automáticamente a la función de la biblioteca estándar *unexpected()*. Esta función está implementada como un puntero a función, de forma que podemos cambiar su comportamiento predefinido, que consiste en llamar a *terminate()* (V. §3.3); para ello haremos una llamada a la función *set_unexpected()*, declarada en `<exception>`, pasándole la dirección de una función que no reciba ni devuelva nada. Asimismo, *set_unexpected()* devuelve la dirección de la función anterior que estamos reemplazando.

Nuestra función *unexpected()* no debe recibir ni devolver nada, y además no debe regresar: debe acabar llamando a una función que haga terminar el programa, como *exit()* o *abort()*, pero también, y en esto se diferencia de *terminate()*, puede lanzar una excepción, incluso relanzar la misma. En este caso, la búsqueda de su manejador empieza en la llamada a función que lanzó la excepción inesperada.

Ejemplo:

excep-unexpec.cpp

```
1  /*
2   * Especificaciones de excepciones y unexpected()
```

```
 */
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;

class Arriba {};
class Abajo {};
void g();

void f(int i) throw(Arriba, Abajo)
{
    switch (i) {
        case 1: throw Arriba();
        case 2: throw Abajo();
    }
    g();
}

#ifndef UNEXPECTED
void g() {} // Versión 1: correcto
#else
void g() { throw 666; } // Versión 2: excepción inesperada en f()
#endif

void el_inesperado()
{
    cerr << "Se ha lanzado una excepción a traición." << endl;
    exit(EXIT_FAILURE);
}

int main()
{
  (void) set_unexpected(el_inesperado);
  for (int i = 1; i <= 3; i++)
      try {
          f(i);
      } catch(Arriba) {
          cout << "Arriba capturado." << endl;
      } catch(Abajo) {
          cout << "Abajo capturado." << endl;
      }
}

```

La función *f()* promete que sólo va a lanzar excepciones de los tipos *Arriba* y *Abajo*, y eso es lo que parece mirando su código. La versión 1 de *g()* no lanza nada, así que es verdad. Sin embargo, algún día alguien cambia *g()* de forma que ahora lanza una excepción de tipo *int*, y enlaza *g()* con *f()* (podrían estar en ficheros separados). Ahora *f()* lanzará también indirectamente una excepción de tipo *int*, violando su promesa, o sea, su lista `throw`, aunque quizá sin el conocimiento de su autor, y provocando una llamada a *unexpected()*.

Esto nos enseña que siempre que empleemos excepciones y listas `throw`,

deberíamos escribir nuestra versión de *unexpected()*, como en este ejemplo, para al menos registrar el suceso y que nos demos cuenta, y además relanzar el mismo error, lanzar otro nuevo, o terminar el programa lo mejor posible, como hacemos aquí.

Otra forma de abordar el problema del lanzamiento de excepciones que no estén en la lista `throw` es añadir a esta la excepción estándar *bad_exception* (§4.1), declarada en `<exception>`. En este caso, *unexpected()* lanzará esta excepción en lugar de llamar a una función.

Ejemplo:

```
class X { };
class Y { };
void f() throw(X, std::bad_exception)
{
  // ...
  throw Y(); // lanza una excepción «mala»
}
```

En este caso, la función *f()* no acaba lanzando la excepción *Y*, ni *unexpected()* llama a la función instalada con *set_unexpected()* o, si no se ha instalado ninguna, a *terminate()*, sino que se lanza la excepción estándar *bad_exception*, que será la que habrá que capturar. Obsérvese que de todas formas se pierde información incluso de qué excepción es la que realmente se ha producido.

4. Jerarquía de excepciones

ATENCIÓN: Esta sección requiere haber visto antes el mecanismo de la herencia. Sáltela si no lo conoce aún, y vuelva a ella una vez estudiado.

Una excepción es un objeto de alguna clase (también puede ser de un tipo fundamental) que representa la ocurrencia de un suceso excepcional, que normalmente se ve como un problema o error. A menudo, las excepciones se pueden agrupar de forma natural en familias. Esto implica que la herencia puede ser un mecanismo útil a veces para estructurarlas. Por ejemplo, las excepciones para una biblioteca matemática podrían estructurarse así:

```
class ErrorMatematico { };
class DesbordamientoSuperior: public ErrorMatematico { };
class DesbordamientoInferior: public ErrorMatematico { };
class DivisionPorCero: public ErrorMatematico { };
// ...
```

Esto nos permitiría manejar cualquier error matemático sin importarnos de qué clase sea. Por ejemplo:

```
void f()
{
  // ...
  try {
    // ...
  } catch(DesbordamientoSuperior) {
     // manejo de DesbordamientoSuperior o algo derivado de él
  } catch(ErrorMatematico) {
     // manejo de cualquier otro error matemático (no el anterior)
  }
}
```

Si estas excepciones se hubieran expresado de forma independiente unas de otras, sin agruparse en esa jerarquía, una función que quisiera poder capturar todas las excepciones de nuestra biblioteca matemática tendría que listar todos los manejadores, uno por excepción. Esto es tedioso y propenso a olvido de alguna excepción. Pero además, requeriría añadir otro manejador a la lista si por algún motivo se añadiera otra excepción a la biblioteca.

El empleo de jerarquías de excepciones lleva a manejadores interesados sólo en un subconjunto de la información expresada por las excepciones. O sea, que una excepción se captura por un manejador para su clase base en lugar de para su tipo exacto. En este caso, si no se emplean punteros ni referencias, parte de la información de la excepción lanzada se pierde en el proceso de la captura. Por ejemplo:

```
class ErrorMatematico {
  // ...
public:
  virtual void ver_traza() const { cerr << "Error matemático"; }
};

class Desbordamiento_int: public ErrorMatematico {
  const char* op;
  int a1, a2;
  // ...
public:
  Desbordamiento_int(const char* p, int a, int b)
  : op(p), a1(a), a2(b) { }
  virtual void ver_traza() const {
    cerr << op << '(' << a1 << ", " << a2 << ')';
  };

void f()
{
  try {
    g();
  } catch(ErrorMatematico m) {
     // ...
  }
}
```

Cuando se alcanza el manejador para *ErrorMatematico*, *m* es un objeto de este tipo, aunque la llamada a *g()* haya provocado un error del tipo derivado *Desbordamiento_int*, con lo cual la información extra de este tipo se pierde.

Para evitar esta pérdida de información, deben emplearse punteros o referencias. Por ejemplo:

```
int sumar(int x, int y)
{
  if ((x > 0 && y > 0 && x > INT_MAX - y)
      ||
      (x < 0 && y < 0 && x < INT_MIN - y))
    throw Desbordamiento_int("+", x, y);
  return x + y;
}

void f() try
{
  int i1 = sumar(1, 2);
  int i2 = sumar(INT_MAX, -2);
  int i3 = sumar(INT_MAX, 2);  // excepción
} catch(ErrorMatematico& m) {
    // ...
    m.ver_traza();
}
```

En este caso se llama en el manejador a *Desbordamiento_int::ver_traza()*; si se hubiera capturado por valor, se hubiera llamado en cambio a la función de la clase base: *ErrorMatematico::ver_traza()*.

4.1. Excepciones estándares

ATENCIÓN: Para aprovechar mejor esta sección se requiere haber visto antes el mecanismo de la herencia. Si no lo conoce aún, léala de todas formas pero vuelva a hacerlo una vez estudiado dicho mecanismo.

Algunos operadores del lenguaje C++, y el propio lenguaje, pueden lanzar excepciones. Algunos métodos de clases de la biblioteca estándar también. Estas excepciones y otras más que se definen aunque no las lance ni el lenguaje ni la biblioteca estándar están disponibles además para quien quiera emplearlas. Hay quien opina (Stroustrup no) que estas excepciones proporcionan una forma fácil y rápida de emplear excepciones sin tener que definir unas propias; también se pueden definir excepciones derivándolas de éstas.

Las excepciones estándar forman una jerarquía. A continuación se describen brevemente. El sangrado indica de quién deriva cada clase.

exception Definida en `<exception>`. Es la clase base para todas las excepciones estándares. Posee un método virtual llamado *what()* que redefinen las otras excepciones derivadas de ésta. Este método devuelve una cadena de caracteres de bajo nivel que describe la excepción o da su nombre.

- **logic_error** Definida en `<stdexcept>`. Se supone que los errores lógicos podrían detectarse antes de la ejecución del programa o comprobando los parámetros pasados a funciones y constructores.
  - **length_error** Definida en `<stdexcept>`. Indica un intento de producir un objeto cuya longitud sea mayor o igual que *npos* (el valor más grande representable en un *size_t*).

domain_error Definida en `<stdexcept>`. Para informar de violaciones de precondiciones.

out_of_range Definida en `<stdexcept>`. Lanzada por el método *at()* de varias clases contenedoras como *string* o *vector* y por el operador de índice *operator* `[]` de *bitset* para informar de un parámetro cuyo valor está fuera de un rango válido.

invalid_argument Definida en `<stdexcept>`. Lanzada por el constructor de *bitset*, indica un parámetro inválido para la función desde donde se lanza.

bad_alloc Definida en `<new>`. Lanzada por el operador `new` en caso de fallo en la petición de memoria, si no se ha instalado una función para tratar el caso mediante la función *set_new_handler()* y si no se ha pasado el argumento *nothrow*, que haría que `new` devolviera 0.

bad_exception Definida en `<exception>`. Lanzada por el lenguaje cuando una función lanza una excepción que no está en su lista `throw` pero ésta contiene precisamente *bad_exception* (p. 14).

ios_base::failure Definida en `<ios>`. Lanzada por *ios_base::clear()* para indicar un fallo de entrada/salida.

bad_typeid Definida en `<typeinfo>`. Lanzada por el operador de identificación de tipo en tiempo de ejecución (RTTI, *Run Time Type Identification*) `typeid` cuando se le pasa un puntero nulo.

bad_cast Definida en `<typeinfo>`. Lanzada por el operador de RTTI `dynamic_cast` cuando se le pasa una referencia que no es del tipo esperado.

runtime_error Definida en `<stdexcept>`. Para informar de errores que se producirán sólo cuando el programa esté ejecutándose.

range_error Definida en `<stdexcept>`. Para informar de violaciones de una postcondición.

overflow_error Definida en `<stdexcept>`. Lanzada por el método *bitset::to_ulong()*, para informar de un desbordamiento superior matemático.

underflow_error Definida en `<stdexcept>`. Para informar de un desbordamiento inferior matemático.

Para más claridad se presentan las excepciones estándar lanzadas por el lenguaje en la tabla 1 y las lanzadas por la biblioteca estándar en la tabla 2.

Excepción	Lanzada por	Cabecera
bad_alloc	`new`	`<new>`
bad_cast	`dynamic_cast`	`<typeinfo>`
bad_typeid	`typeid`	`<typeinfo>`
bad_exception	*lista* `throw`	`<exception>`

Cuadro 1: Excepciones estándares lanzadas por el lenguaje C++

La jerarquía de excepciones estándares se verá mejor en la figura 1.

Excepción	Lanzada por	Cabecera
out_of_range	*at()*	`<stdexcept>`
	bitset::operator[]()	
invalid_argument	*constructor de bitset*	`<stdexcept>`
overflow_error	*bitset::to_ulong()*	`<stdexcept>`
ios_base::failure	*ios_base::clear()*	`<ios>`

Cuadro 2: Excepciones estándares lanzadas por la biblioteca estándar

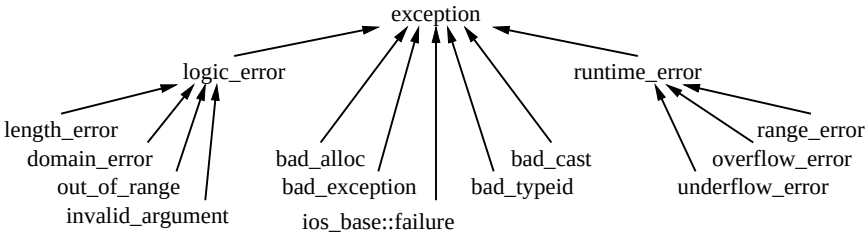

Figura 1: Jerarquía de excepciones estándares

5. Programación con excepciones

Las excepciones no estaban en el C++ original; se añadieron un tiempo después. A continuación se presentan unos consejos y advertencias: cuándo, cuándo no, y cómo emplear excepciones.

5.1. Evitar excepciones

Las excepciones no son la panacea o la solución a todos los problemas. A veces es mejor no emplearlas. He aquí cuándo.

5.1.1. No para eventos asíncronos

El sistema de señales basado en la función de la biblioteca estándar de C *signal()* maneja eventos o sucesos asíncronos; es decir, aquellos que ocurren fuera del ámbito del programa y que por tanto este no puede preveer. Las excepciones no sirven para este tipo de sucesos. Sin embargo pueden *asociarse* a ellas. El manejador de señal puede hacer su trabajo rápidamente, regresar, y luego el programa puede lanzar una excepción basándose en alguna variable puesta por este manejador de señal.

5.1.2. No para errores ordinarios

Si se posee suficiente información como para tratar un error, no es una excepción. Si el error se puede tratar en el contexto en el que se produce, hágase ahí.

Tampoco son excepciones de C++ los sucesos a nivel de máquina como división por cero; estos se manejan mediante otros mecanismos, proporcionados

por el sistema operativo o la circuitería (aunque también pueden asociarse a excepciones).

5.1.3. No para control de flujo

Las excepciones pueden verse como otro mecanismo de control de flujo, una especie de mezcla entre `return` y `switch`. Por lo tanto podrían emplearse sin que se produzcan realmente errores o problemas en el programa: excepciones que no son errores.

No obstante, esto suele ser una mala idea. En parte porque las excepciones son mucho menos eficientes que la ejecución normal del programa: las excepciones son sucesos raros, excepcionales, y el programa, en ausencia de ellos, no debería verse afectado en su rendimiento. Y en parte porque emplear excepciones para lo que no han sido diseñadas es confuso.

5.1.4. No si se pueden evitar

No es obligatorio emplear excepciones. Algunos programas, como se dijo en 1, son bastante (relativamente) simples: se toma una entrada, se procesa de alguna forma, y se muestra una salida. Es posible que falte memoria, que no se pueda abrir algún fichero o que el usuario haya dado algún parámetro incorrecto. Es perfectamente válido y recomendable en este tipo de programas presentar un mensaje de diagnóstico y acabar ordenadamente; incluso emplear *assert* aunque sea en la fase de pruebas, o *abort()*. No valdría la pena trabajar extra para meter excepciones cuando lo más fácil para el usuario es volver a ejecutar el programa.

5.1.5. No para código antiguo

Si se tiene que modificar un programa antiguo que no emplea excepciones y se le añade una biblioteca que sí las utiliza, ¿habrá que modificar todo el programa para adaptarlo a las excepciones? La respuesta es no, o no mucho. Quizás se pueda aislar el código que genere excepciones en un bloque `try` con manejadores que conviertan las excepciones al esquema de manejo de errores que tuviera el programa antiguo. En cualquier caso se podría rodear de un bloque `try` la porción de programa más grande que pudiera generar excepciones (esto puede ser *main()*) seguido de un manejador para cualquier excepción (`catch(...)`) que generara mensajes básicos de error. Esto podría refinarse más adelante según se viera, pero el código a añadir es muy poco.

5.2. Empleo de excepciones

A continuación se expone qué cosas se pueden hacer con excepciones, o para qué sirven.

- 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.
- Envolver funciones (por ejemplo y especialmente de alguna biblioteca de C) que empleen otros esquemas de error, de forma que generen excepciones.
- Simplificar. El empleo de excepciones debe ser más fácil, claro y cómodo que otro que se pueda usar.
- Hacer una biblioteca o programa más seguro y robusto.

Y por último algunos consejos:

Emplee siempre especificaciones de excepción Las especificaciones de excepción o listas `throw` son como prototipos de funciones: les dicen al usuario que le conviene escribir manejadores de excepción, y cuáles, y les dicen al compilador qué excepciones pueden lanzarse desde la función.

Empiece con las excepciones estándar Antes de crear sus excepciones, estaría bien que mirara a ver si alguna de las que ya hay (§4.1) le sirve. Si es así, es posible que sea más fácil para el usuario reconocerla y tratarla. Si no, puede intentar derivar la suya de una de las existentes o directamente de *exception*. Así el usuario siempre podrá esperar llamar en sus manejadores al método *what()* definido en la clase de interfaz *exception*.

Anide sus propias excepciones Si se crea excepciones para una clase particular, anídelas en ella (esto es, defínalas dentro); esto le dice claramente al usuario que esas excepciones solo son para esa clase; además impide la proliferación de nombres en el espacio de nombres global o en el que se defina la clase anfitriona. Este anidamiento puede hacerse aunque las excepciones deriven de las estándares.

ATENCIÓN: Este párrafo y los dos siguiente emplean el concepto de herencia, que aún no se ha visto. Sáltelos y vuelva a leerlos cuando conozca dicho concepto.

Emplee jerarquías de excepciones Pues proporcionan un modo muy conveniente de clasificar los diversos tipos de errores que puedan encontrarse en la clase o biblioteca que se esté construyendo. Dan información valiosa a los usuarios, les asisten en la organización de su código, y les da la opción de capturar solamente el tipo base, despreciando todos los tipos específicos de error; además, al añadir una nueva excepción a la jerarquía, ésta se capturará en el manejador de la clase base, sin necesidad de tocar el código cliente con los manejadores.

Precisamente las excepciones estándar son un buen ejemplo de esto (fig. 1).

Herencia múltiple Uno de los pocos sitios donde la herencia múltiple puede estar justificada es precisamente en las jerarquías de excepciones, porque un manejador de una clase base de excepción de cualquiera de las raíces de la clase de excepción heredada múltiplemente puede manejar dicha excepción.

Ejemplo:

```
class E_Fichero_NFS: public E_Red, public E_SistemaFicheros {
  // ...
};
```

Un error relativo a un fichero remoto (*E_Fichero_NFS*) puede ser capturado en funciones que traten con excepciones o errores de red (*E_Red*):

```
void f()
try {
  // ...
} catch(E_Red& e) {
    // ...
}
```

tanto como por funciones que traten con excepciones o errores del sistema de ficheros (*E_SistemaFicheros*):

```
void g()
try {
  // ...
} catch(E_SistemaFicheros& e) {
    // ...
}
```

Esto es importante en casos donde haya servicios transparentes al usuario; obsérvese que el autor de *g()* puede no saber o no importarle que hay una red por medio; sin embargo, su manejador también capturará los errores de red (*E_Fichero_NFS*).

Capture por referencia, no por valor Si se lanza un objeto de una clase derivada y se captura por valor en el manejador de la clase base, los miembros añadidos en la derivación se pierden[2] y el objeto copiado en el parámetro formal del manejador se comportará como un objeto de la clase base.

En vez de por referencia, se podrían lanzar y capturar punteros, pero esto añadiría mayor complejidad. El lanzador y el capturador tendrían que ponerse de acuerdo en cómo el objeto de excepción obtiene y libera memoria.

De todas formas, aunque no haya una jerarquía de excepciones, siempre suele ser mejor capturar por referencia, debido al ahorro de una copia.

Lance excepciones en constructores Estos son uno de los sitios más importantes donde lanzar excepciones, pues son la mejor alternativa al tratamiento de errores, ya que la construcción (correcta) de objetos es fundamental en C++. El lanzamiento de una excepción en un constructor garantiza que el objeto no

[2] A esto se le llama *rebanar* el objeto.

está construido (lo cual es diferente de que esté mal construido). Sin embargo, hay que tener cuidado cuando el objeto contiene punteros para los que se reserve memoria en el constructor.

No lance o cause excepciones en destructores Puesto que se llama a destructores en el proceso de lanzar otras excepciones, si se lanza una excepción en un destructor, se lanzará *antes* de que se alcance la cláusula `catch` de la excepción originalmente lanzada, lo que provocará una llamada a *terminate()*.

Y si en el destructor se llamara a alguna función que pudiera lanzar excepciones, debería meterse su llamada en un bloque `try` de forma que ninguna excepción escapara del destructor.

Evite punteros «desnudos» Un simple puntero significa vulnerabilidad en el constructor si se reserva memoria u otro recurso para él. Puesto que un puntero no posee destructor, esos recursos no serán liberados en el caso de que se lance una excepción en el constructor, como se ha mencionado antes.

> A este respecto, la biblioteca estándar viene en nuestra ayuda proporcionándonos un tipo pseudo-puntero llamado *auto_ptr*, definido en `<memory>`, que posee la curiosa e interesante propiedad de que el objeto al que apunte será implícita y silenciosamente borrado cuando el objeto *auto_ptr* salga de ámbito.