Programación en C++ (11) : excepciones

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Octubre 2009

Outline

Outline

Manejar errores

La manera más simple: definir el comportamiento erróneo en términos de un booleano

```
inline void require(bool requirement,
  const std::string& msg = "Requirement failed"){
  using namespace std;
  if (!requirement) {
    cerr << msg << endl;
    exit(1);
  }
}
...
require((a<100),"Value out of bounds");
```

Poco robusto...

Manejar errores

Existe una función estándar (de C) para hacer este tipo de verificaciones, la función assert

```
#include <cassert>
int main() {
   bool d=false;
   assert(d);
}
```

Produce una descripción mínima de lo que pasó y sale...

```
test6.cpp:44: failed assertion ‘d’
Abort
```

Manejar errores

Ese comportamiento frente errores puede estar OK en fase de desarrollo del programa, pero no en fase de release, tenemos que prever cuando es posible un procesamiento en linea de las errores.
En práctica, separar el caso de fase de debug de fase "normal" :

```
#include <cassert>
...
bool d; ...
#ifdef DEBUG
assert(d);
#else
// Prever el procesamiento de error!
#endif
```

Manejar errores

Si, dentro de la función/del método...en que se produce el error tienes todo lo necesario para manejar el error y salir "bien" no hay problemas. Si no es el caso y que tienes que salir del contexto para manejar el error, hay varias maneras de hacerlo en C :

1. Afectar un código de error tuyo al valor de regreso a la función y propagar este código a niveles en que el procesamiento es posible :

```
#define ERR_MALLOC 1
...
if (!(f = malloc(1000*sizeof(double))))
        return ERR_MALLOC;
```

Manejar errores

2. Usar un sistema de códigos global, que provee por ejemplo C a través de errno y perror

```
vloopbacks=
fopen("/proc/video/vloopback/vloopbacks", "r");
if (!vloopbacks) {
  perror("Failed to open '/proc/video/vloopback/vlo
  return -1;
}
```

Las funciones estándar, en caso de error, ponen la variable global errno, y perror imprime tu texto + la descripción del errno

```
13 EACCES Permission denied.  An attempt was made to access a file in a
way forbidden by its file access permissions.
14 EFAULT Bad address.  The system detected an invalid address in
 attempting to use an argument of a call.
15 ENOTBLK Not a block device.  A block device operation was attempted on
  a non-block device or file.
```

Manejar errores

3. Generar una señal desde el programa frente a una error y escribir los mecanismos para manejarlos :

```
#include <signal.h>
int raise(int sig);
```

manejado con

```
void sig_catcher(int signum) {
...
}
int main() {
// catch signal
signal(sig, sig_catcher);
...
raise (SIGERRVAL);
}
```

Manejar errores

4. Mecanismos de saltos (goto) preservando el estado del programa

```
#include <csetjmp>
jmp_buf state;
void testError() {
        ...
        longjmp(state,1); // En caso de error
}
int main() {
  if(setjmp(state) == 0) {
        testError();
  } else {
        // Procesar el error
        ...
  }
```

Manejar errores

Problemas :

- Conexión fuerte entre la "emisión" de los errores y los procesamientos
- Hay que prohibir las herramientas de tipo "goto" en C++ : no manejan los destructores como requerido !

Excepciones

En C++, el mecanismo estándar es el de excepciones : en caso de errores al momento de la ejecución, y que no se puede corregir localmente, la idea es, al lugar de salir de la función en que estamos, lanzar afuera (throw) un "mensaje de destreza en una botella" y esperar que algo en otra parte del código lo pueda manejar ...

Excepciones

Para eso se define objetos que “lanzar” (excepciones : se puede tomar de cualquier tipo) y la palabra llave throw nos permite emitirlos :

```
class HelpMe {
string message;
public:
HelpMe(const string &s) : message(s) {};
const string &getMessage() {return message;}
};
void fonc(bool b) {
        if (!b) {
                throw(HelpMe("I am in trouble !!"));
        }
}
```

Excepciones : throw

El throw :

- actúa como un return, en el sentido de que regresa un valor, el objeto acompañandolo,
- efectúa el cleanup de las variables locales de la función en que se encuentra,
- no regresa al nivel superior en la pila (a la llamada de la función) sino en un lugar especial de la memoria para manejar excepciones,
- enviará preferentemente objetos específicos al error encontrado.

Excepciones

Ahora nos falta ver el mecanismo que nos permite asociar una excepción a un procesamiento. El código :

```
fonc(false);
cout << "Life goes on" << endl;
```

genera dicha excepción pero sale diciendo :

```
terminate called after throwing an instance of 'HelpMe'
Abort
```

Excepciones : bloque try

En cualquier lugar del código arriba (en la pila) de la llamada de la función, se puede definir un bloque try que se prepara a recibir excepciones

```
try {
        ...
        fonc(false);
}
```

No se sale de la función en que está este bloque : el procesamiento de los errores puede estar definido en este nivel.

Excepciones : bloque try

El bloque try esta seguido por uno o varios bloques catch que están específicos a la excepción generada dentro del bloque try :

```
try {
        fonc(false);
} catch (int i) {

}
cout << "Life goes on" << endl;
```

La palabra llave catch, seguida de un tipo t y de un identificador id, explícita un bloque donde se va a manejar una excepción id de tipo t. Aquí no tengo manera de manejar HelpMe (solo int) entonces sale igual que sin el bloque try.

Excepciones : bloque try

Si manejo el HelpMe :

```
try {
        fonc(false);
} catch (HelpMe h) {
        cout << "I am here, sweetie !" << endl;
        cout << "I heard your message \""
             << h.getMessage() << "\"" << endl;
}
cout << "Life goes on" << endl;
```

```
I am here, sweetie !
I heard your message "I am in trouble !!"
Life goes on
```

Sigue “normalmente” a la salida del bloque catch.

Excepciones : bloque try

Mas generalmente, tenemos algo como :

```
try {
        ...
} catch (ExceptionType1 ex1) {
...
} catch (ExceptionType2 ex2) {
...
} catch (ExceptionType3 ex3) {
...
} ...
} catch (ExceptionTypeN exN) {
...
}
```

Excepciones

Y si la excepción efectivamente generada no esta prevista, saldrá el programa mencionando el tipo de la excepción

```
terminate called after throwing an instance of 'double'
```

Excepciones

Ese mecanismo es interesante porque :

- separa bien el código normal del código encargado de manejar los errores,
- permite dejar cualquier código de nivel superior en la pila manejar esas errores,
- limpia correctamente al nivel de la función en que se envía la excepción.

Excepciones

Notar que el mecanismo propuesto para excepciones no permite re-empezar el funcionamiento del programa al punto exacto donde falló : la función que llamó tiene sus variables locales limpiadas. Sólo se puede :

- Seguir después del bloque catch.
- Eventualmente re-llamar la función que causó problemas a través de try dentro de ciclos while.

Excepciones : elección del buen handler

- No se necesita correspondencia perfecta entre el tipo de la excepción lanzada y el tipo de recepción por catch : clases derivadas están puestas en correspondencia (por referencia o por valor),

```
class PleaseHelpMe : public HelpMe {
public:
PleaseHelpMe(const string &s) : HelpMe(s) {};
};
```

  El throw con un PleaseHelpMe está manejado

- Preferir pasar las excepciones por referencia para usar eventualmente upcast

Excepciones : elección del buen handler

No se puede contar sobre mecanismo de conversión automática

```
class CanYouPleaseHelpMe {
  string msg;
public:
  CanYouPleaseHelpMe(HelpMe& p) {msg = p.getMessage()
  const string &getMessage() {return msg;}
};
void fonc(bool b) {
  if (!b) throw(HelpMe("I am in trouble !!"));
}
...
try { fonc(false);
} catch (CanYouPleaseHelpMe &p) {
...
```

No lo maneja ...

Excepciones : elección del buen handler

```
try {
    fonc(false);
} catch (HelpMe &h) {
    cout << "I am here, sweetie !" << endl;
    cout << "I got your message \""
         << h.getMessage() << "\"" <<  endl;
} catch (PleaseHelpMe &h) {
    cout << "I am here, darling !" << endl;
    cout << "I heard your message \""
         << h.getMessage() << "\"" <<  endl;
}
```

Al lanzar un PleaseHelpMe, el compilador me avisa :

```
warning: exception of type 'PleaseHelpMe' will be caught
warning:    by earlier handler for 'HelpMe'
```

Manejar todo tipo de excepción

Es posible hacer un handler que agarre toda excepción generada en el try :

```
try {
    fonc(false);
} catch (...) {
    cout << "I am here, sweetheart !" << endl;
}
```

Eso puede ser útil cuando hay que separar el procesamiento de los errores en dos partes : una parte genérica que puede incluir liberación de memoria, por ejemplo; una parte especifica que esta procesada en un segundo tiempo.

Excepciones : re-lanzar una excepción

Para implementar ese mecanismo de procesamiento en varios tiempos, se necesita propagar la excepción en diferentes niveles : por eso, se usa otra vez la palabra llave throw, que re-lanza la excepción hacia niveles superiores,

```
try {
    fonc(false);
} catch (...) {
    cout << "I am here, sweetheart !" << endl;
    cout << "and I call 066" << endl;
    // Do some cleaning here...
    ...
    throw;
}
```

Excepciones no manejadas

Las excepciones están procesadas nivel por nivel en los bloques `try`; pueden estar manejadas en cualquier de esos niveles. Pero eventualmente no hay ningún bloque `try`, en que se pueda manejar.

Excepciones no manejadas

El comportamiento por default que vimos para excepciones no manejadas es de salir del programa explicando cual excepción no pudo ser manejada :

```
terminate called after throwing an
instance of 'PleaseHelpMe'
Abort
```

Como lo describe el mensaje, se llaman :

- terminate,
- abort, llamada desde terminate, que termina brutalmente el programa (sin llamar destructores de objetos globales o estáticos).

Excepciones no manejadas

El terminate es llamado también cuando :

- cuando un destructor de un objeto local envía una excepción mientras se deshace la pila al salir de una función,
- cuando un destructor o un constructor de un objeto estático o global envía una excepción.

Excepciones no manejadas

Se puede cambiar el comportamiento por default especificando tu propia función terminate gracias a set_terminate.

La función debe ser de regreso void y sin argumentos :

```
void customTerminate() {
   cout << "Sorry, i could not help" << endl;
   exit(0);
}
void (*defaultTerminate)() =
        set_terminate(customTerminate);
```

Excepciones : limpiar memoria

Al salir por lanzar una excepción, se invoca automáticamente el destructor para los objetos construidos al momento de la excepción:

```
class Test {
        static int counter=0;
        int id;
public:
        Test() {
                id=counter++;
                cout << "Constructor " << id << endl;
                if (counter==4) throw 1;
        };
        ~Test() {
                cout << "Destructor " << id << endl;
        }
};
```

Excepciones : limpiar memoria

```
void foncBis() {
        Test t[10];
}
int main() {
        try {
                foncBis();
        } catch (...) {
        }
}
```

Excepciones : limpiar memoria

```
Constructor 0
Constructor 1
Constructor 2
Constructor 3
Destructor 2
Destructor 1
Destructor 0
```

Los objetos ya creados son destruidos. El en que se interrumpió el constructor no está destruido, con razón : no todo ha sido alocado/inicializado al throw, pero supone ocuparse correctamente de lo que tiene que ser parcialmente limpiado.

Excepciones : limpiar memoria

Problema cuando, antes del throw, se ha alocado algo :

```
class Test {
        static int counter=0;
        int id;
        double *data;
public:
        Test() {
                id=counter++;
                data = new double[100];
                cout << "Constructor " << id << endl;
                if (counter==4) throw 1;
        };
        ~Test() {
                delete[] data;
                cout << "Destructor " << id << end'
        }
```

Excepciones : limpiar memoria

En este caso, una primera técnica posible es interceptar excepciones desde el constructor mismo y liberar la memoria desde el catch :

```
Test() {
        try {
                id=counter++;
                data = new double[100];
                cout << "Constructor " << id << endl;
                if (counter==4) throw 1;
        } catch (...) {
                if (data)
                        delete [] data;
                throw;
        }
}
```

Excepciones : limpiar memoria

Otra manera de hacer: encapsular la maquinería de alocación/liberación de recursos como memoria en estructuras *wrapper*, que son típicamente templates :

```
class Test {
Wrapper<double,100> data;
...
};
```

Después de haber generado una excepción en el constructor, por ejemplo, llamará al destructor de Wrapper<double,100>: *Resource Acquisition Is Initialization* (RAII).

Excepciones : auto_ptr

Este tipo de objeto puede ser muy útil, y el C++ provee de hecho una herramienta patrón para crearlos : el patrón auto_ptr, que permite encapsular datos alocados dinámicamente dentro de una clase, y cuyo destructor se encarga de la liberación.

```
std::auto_ptr<Tipo> pt(new Tipo);
```

Se puede usar exactamente como un apuntador (con dereferenciación, incrementación...)

Excepciones : auto_ptr

Dentro de una función susceptible de emitir una excepción :

```
void fonc() {
  Tipo* pt = new Tipo;
  // code... eventually exceptions !
  delete pt;
}
```

se puede re-escribir :

```
void fonc() {
  std::auto_ptr<Tipo> pt(new Tipo);
  // code... eventually exceptions !
  // ahora se liberará la memoria creada
}
```

para transformar el código en código *exception-safe*.

Excepciones : auto_ptr

Una remarca : la creación de un auto_ptr encapsula un apuntador hacia objeto(s), haciendo de la instancia de esa clase el propietario/responsible del apuntador. Hay posibilidad de:

- recuperar el apuntador a través del método get() pero teniendo cuidado a que el objeto se encarga de la liberación ! (entonces, no llamar delete sobre este apuntador),
- abandonar la responsabilidad del apuntador con el método release(), que regresa el apuntador.

Excepciones : auto_ptr

Un ejemplo es, en el constructor, hacer la alocación de los datos dentro de un auto_ptr, temporariamente:

```
Test() {
        id=counter++;
        std::auto_ptr<double> pt(new double[100]);
        cout << "Constructor " << id << endl;
        .... // Maybe exceptions here
        data = pt.release();
}
```

Manejar excepciones al nivel del constructor

Para el caso de constructores (pero más generalmente para función), se puede combinar el bloque `try` con la definición de la función:

```
Test() try : baseClass() {
        id=counter++;
        std::auto_ptr<double> pt(new double[100]);
        cout << "Constructor " << id << endl;
        .... // Maybe exceptions here
        data = pt.release();
} catch (...) {
}
```

La ventaja es que se puede interceptar excepciones de los constructores de la clase de base

Manejar excepciones de la función

Lo mismo existe para funciones, pero sin tanta utilidad:

```
void fonc() try {
        ...
        if (someThing)
                throw 1;
} catch(int) {
        ...
}
```

Excepciones estándar

El lenguaje C++ provee clases para excepciones, que se puede usar gracias a la inclusión del *header* exception. Todas heredan de la clase exception, que contiene un mensaje a que se puede acceder por el método what(). Varias clases heredan de exception:

- logic_error : errores debidas a la lógica interna del programa (que tú hubieras podido prever),
- runtime_error : errores detectables sólo en el momento de la ejecución (por fuentes externas: falta de memoria...),
- ios::failure : errores en la manipulación de iostreams.

Todas esas subclases se construyen con un mensaje string.

Excepciones estándar

Objects of class logic_error or a derived class are thrown when the condition causing the error could have been detected by the client before calling the failing code; that is, the client did not assert preconditions.
Objects of class runtime_error or a derived class are thrown when the error results from a condition that the client could not have tested before calling the failing code.

Excepciones estándar

logic_error :

- domain_error: si una precondición es violada,
- invalid_argument: argumento inválido,
- length_error: en particular en el caso de un contenedor, esta generada si el tamaño excede lo máximo autorizado,
- out_of_range: argumento de valor fuera de rango (a priori),
- bad_cast: error en un dynamic_cast,
- bad_typeid: pasando apuntador nulo a un typeid.

Excepciones estándar

runtime_error :

- range_error: si te das cuenta que el resultado que calculas desde una función cabe fuera de un rango admisible (a posteriori),
- overfow_error: overflow aritmético,
- bad_alloc: error en la alocación (por ejemplo no hay mas memoria).

Especificaciones de excepciones

Al escribir código que estará usado por otra gente, puede ser muy útil especificar cuales son las excepciones que hay que esperar de una función tuya :

```
void fonc() throw(HelpMe,CanYouPleaseHelpMe) {
...
}
```

significa que hay que esperar de esa función excepciones de tipo HelpMe y CanYouPleaseHelpMe.

Especificaciones de excepciones

Mientras:

```
void f();
```

significa, por default, que la función puede emitir cualquiera excepción, y

```
void f() throw();
```

significa que la función no va a emitir ninguna excepción.

Especificaciones de excepciones

La especificación puede estar violada:

```
void fonc() throw(HelpMe,CanYouPleaseHelpMe) {
  throw 1;
}
int main() {
  try {
    fonc();
  } catch (...) {
  }
  cout << "Life goes on" << endl;
}
```

```
terminate called after throwing an instance of 'int'
Abort
```

Especificaciones de excepciones

En este caso, si una excepción que no esta entre las especificadas esta lanzada, se lanza una función equivalente a terminate(), que se llama unexpected(). Igualmente a terminate(), se puede cambiar el comportamiento por default gracias a set_unexpected, esperando un argumento de tipo apuntador a función void sin argumento:

```
void custom_unexpected() {
  cout << "unexpected exception thrown" << endl;
  exit(0);
}
int main() {
  set_unexpected(custom_unexpected);
  try {
    fonc();
  } catch (...) {
  }
  cout << "Life goes on" << endl;
```

Excepciones unexpected

Dentro de la función unexpected, se puede aun enviar excepciones, y en este caso:

- se puede enviar una excepción de las que se puede interceptar en el bloque try que no ha interceptado la previa y provocado el unexpected, y el programa sigue como si finalmente la función incriminada había enviado una excepción interceptable,
- se puede re-enviar la misma excepción (por throw) y haber especificado la excepción bad_exception, en este caso se generará la excepción bad_exception,
- sino, se llama el terminate() (o tu propia versión de ese).

Especificaciones de excepciones

Si usas funciones/estructuras/clases que no conoces en detalle, mejor no usar especificaciones de excepciones: es muy posible que no preveas todas, lo que llevará a excepciones $\mathrm{unexpected}$.

Especificaciones de excepciones: herencia

La idea en términos de herencia es que métodos especificando excepciones en una clase base no pueden generar otras excepciones en clases derivadas. Cuando sí se puede especificar toda otra cosa es cuando hay realmente sobrecargamientos (y no polimorfismo).

```
class A {
  virtual void f() throw (HelpMe) {
    throw HelpMe("");
  }
};
class B : public A {
  void f() throw(CanYouPleaseHelpMe) {
    throw CanYouPleaseHelpMe();
 }
};
```

Especificaciones de excepciones: herencia

El mensaje de error:

```
looser throw specifier for
'virtual void B::f() throw (CanYouPleaseHelpMe)'
error:    overriding
'virtual void A::f() throw (HelpMe)'
```

desaparecerá sólo si mi lista de excepciones en B es un subconjunto de la versión de A (las excepciones de B pueden ser derivadas de las de A.

Exception safety

Un aspecto fundamental en la programación en C++ consiste en lo de analizar el código para determinar si esta capaz o no de soportar excepciones. Un comportamiento no previsto en cuanto en excepciones puede tener graves consecuencias sobre el funcionamiento del programa:

```
template<class T> T stack<T>::pop() {
  if(count == 0)
    throw logic_error("stack underflow");
  else
    return data[--count];
}
```

Exception safety

En este caso, preferir escribir su código separando las funcionalidad, de tal manera de que no interfieran malamente a través de excepciones ! Por eso el tipo de $\mathrm{pop}()$ en la implementación real de std es void.
De manera general, si se necesita hacer código muy robusto, cuidar

- constructores,
- alocación memoria y apuntadores "desnudos",
- invocación de otros métodos desde las clases usadas que puedan lanzar excepciones.

Cuando se puede evitar usar excepciones. . .

- eventos asíncronos,
- errorcitas que se podría tratar localmente,
- estructuras de control a través de la pila: no están hechas para esto !
- programas con imperativos fuertes en términos de tamaño del código objeto,
- prohibir absolutamente en destructores.

Cuando se puede usar excepciones. . .

- Constructores,
- para reintentar llamadas a funciones que fallan,
- para intentar otra cosa al lugar de la función que falla,
- para compartir el procesamiento de las errores entre varias capas de tratamiento.