Manejo de errores

M. en I. Fco. Javier Rodríguez García

Programación Avanzada y Métodos Numéricos

Práctica 11: Excepciones

Excepciones

1 Introducción

Hasta el momento hemos estado lidiando con posibles errores que se deriven de nuestros programas cuando se están ejecutando de dos formas: o ignoramos tales errores, o los manejamos con mucha simplicidad. En este documento aprenderemos qué son, cuándo y para qué se usan las *excepciones*, e inclusive cuándo no hay que usarlas.

1.1 Manejo de errores

El manejo de errores o situaciones excepcionales en nuestros programas no debe ser tomado a la ligera ni mucho menos "dejarlo al último". La forma en que vamos a manejar los errores debe ser considerado como parte integral en el desarrollo del software y tomar todas las medidas preventivas desde un principio. Tanto el lenguaje por sí mismo, como el compilador, proveen herramientas que podemos usar e integrar en nuestro software.

Por ejemplo, nuestra primera línea de defensa es el compilador, porque puede atrapar errores lógicos que por una causa u otra hubiéramos dejado pasar, como usar variables u objetos no inicializados. Las banderas **`-Wall`**, **`-Wextra`** y **`-Wpedantic`** se podrán encargar de hacernos saber si "se nos pasó algo". La segunda línea de defensa pueden ser ciertos elementos del lenguaje como marcar a argumentos y métodos como `const`, para que no cambien por error cosas que se supone no debieran cambiar. A esta técnica se le conoce como los *invariantes*. Compañeros inseparables de los invariantes son las *aserciones*. Una aserción, nuestra tercera línea de defensa, se podría encargar de echarnos en cara cuando una función o método no cumple con las pre o post condiciones que se esperan: como precondición deberíamos detectar que un objeto llegue a un método en un estado seguro; mientras que una postcondición se asegurará que el objeto que salga del método lo haga en el mismo estado seguro en el que entró.

1.2 Excepciones

Por último, tenemos a nuestra cuarta línea de defensa, las *excepciones*. Las excepciones son una poderosa herramienta para hacer a nuestros programas más robustos y tolerantes a fallos. La idea detrás de las excepciones es ayudar a llevar información desde el punto donde es detectada una situación inusual dentro del flujo normal del programa hasta el punto donde esta situación pueda ser manejada. Con este mecanismo, además de separar el código normal de nuestros programas del código para manejar errores, se hace visible la estructura del manejo de errores y su posible complejidad.

Es importante mencionar que el flujo normal del programa se detiene cuando la situación anómala es detectada, porque en el punto donde se detectó tal situación el programa termina la función y sale a buscar el bloque de código que se hará cargo, llevando a cabo lo que se conoce como *stack-unwinding*, o desenrrollar la pila (es decir, va “terminando” las funciones previas hasta encontrar el código manejador de la excepción). Si tal manejador no se encuentra, entonces el programa termina.

2 Excepciones en C++

Las excepciones son anomalías en tiempo de ejecución ---cuando el programa ya se está ejecutando--- que se dan fuera del funcionamiento normal de un programa. Así, las excepciones de C++ son un mecanismo donde un bloque de código le notifica a otro bloque de código sobre tales situaciones "excepcionales", o también sobre alguna condición de error. La función o método que encuentra la situación excepcional lanza (*throw*) una excepción, y el código que va a manejar tal situación la atrapa (*catch*). El código que tiene la capacidad de lanzar la excepción deberá encontrarse dentro de un bloque `try`, mientras que el código que la atrapará deberá encontrarse en uno o más bloques `catch`. El código que va a lanzar la excepción debe usar la palabra reservada `throw`.

```
void foo ()
{
   throw val;
   // si algo malo sucediera, lanza una excepción y las siguientes
   // líneas ya no se ejecutarían

   // el código que iría aquí no se ejecutaría si se lanza la excepción
}

// ...
```

```
try {
   foo ();
   // como foo() puede lanzar una excepción, debemos colocarla en
   // un bloque try

   // si todo fue bien en la función foo() el código continuaría aquí
}
catch (val) {
   // código que va a manejar la situación anómala
}
```

Para que una excepción sea útil requiere dar información sobre lo que sucedió. Cualquier tipo de datos (siempre y cuando se pueda copiar) es candidato a llevar dicha información, pero se recomienda utilizar TDUs para acarrear la información. Los siguientes ejemplos didácticos muestran la forma de enviar información desde el código que falló, hasta su manejador utilizando tipos básicos.

```
class Prueba
{
public:
   void UnMetodo();
   // este método lanza una excepción
};

void Prueba::UnMetodo()
{
   throw 43;

   std::cout << “Nunca me voy a ejecutar :(“ << std::endl;
}

int main(void)
{
   Prueba p;

   try
   {
      p.UnMetodo();
   }
   catch (int e) // ‘e’ trae la información dada por el throw
   {
      std::cout << e << std::endl;
   }
```

```
}
```

En lugar de enteros podríamos usar un texto como lo muestra el siguiente ejemplo, pero el lector deberá notar que el argumento del bloque `catch` es una cadena tipo C:

```
class Prueba
{
public:
   void UnMetodo();
};

void Prueba::UnMetodo()
{
   throw "ups, se acaba de lanzar una excepción";
}

int main(void)
{
   Prueba p;

   try
   {
      p.UnMetodo();
   }
   catch (const char * e)
   {
      std::cout << e << std::endl;
   }

}
```

2.1 Clases estándar que capturan excepciones

Antes de que el alumno escriba sus propias clases de excepciones es conveniente que se familiarice con las existentes en la biblioteca estándar. Existen diferentes familias de excepciones que se pueden utilizar en escenarios comunes de programación, como intentar escribir fuera del rango de un arreglo, o una entrada por parte del usuario no esperada, y así por el estilo. Cada una de estas clases de excepciones tiene un sólo metodo, `what()`, que puede ser usado para imprimir la información que viene desde el código que lanzó la excepción. El nombre de la clase de excepción es la única diferencia entre ellas, pero el propósito y contexto lo da el programador.

El siguiente ejemplo usa la excepción estándar `range_error` para indicar que hubo cierto error en el rango esperado por el método:

```
#include <stdexcept>
// para lanzar las excepciones de la biblioteca estándar

class Prueba
{
public:
   void UnMetodo();
};

void Prueba::UnMetodo()
{
   throw std::range_error {"Hubo un error en el rango"};
}

int main(void)
{
   Prueba p;

   try
   {
      p.UnMetodo();
   }
   catch (std::range_error & e)
   {
      std::cout << e.what () << std::endl;
   }

}
```

Como puede observar, el sólo nombre de la familia de la excepción ya nos está dando algo de luz sobre el problema, por lo que también pudimos querer señalar que el error era de lógica:

```
#include <stdexcept>
// para lanzar las excepciones de la biblioteca estándar

class Prueba
{
public:
   void UnMetodo();
};
```

```
void Prueba::UnMetodo()
{
   throw std::logic_error {"Ups, no debió llegar hasta aquí"};
}

int main(void)
{
   Prueba p;

   try
   {
      p.UnMetodo();
   }
   catch (std::logic_error & e)
   {
      std::cout << e.what () << std::endl;
   }

}
```

La siguiente lista muestra las diferentes clases de excepciones estándar:

CLASES DE EXCEPCIONES ESTÁNDAR
exception
runtime_error
range_error
overflow_error
underflow_error
logic_error
domain_error
invalid_argument
length_error
out_of_range

2.2 Software robusto

Una excepción no necesariamente debe terminar el programa de forma abrupta, aún cuando se nos haya informado del fallo. Un código robusto debe hacer al menos dos cosas: tratar de recuperarse del error; y si esto no fuera posible, no terminar sin antes hacer una limpieza, como por ejemplo, devolver los recursos pedidos. A continuación vamos a ver una técnica para tratar de recuperarnos del, y en la siguiente sección trataremos al idiom RAII para explorar la forma en que podemos hacer la limpieza.

Recuperación del código luego de un error

La técnica para recuperarnos de un error, más allá de limpiar y terminar, consiste en insistir hasta que la condición no se dé nuevamente. Esta técnica no funciona en el caso que nos alineemos con la escuela de pensamiento que establece que las excepciones deben ser usadas únicamente en situaciones excepcionales, porque repetir una situación excepcional no es trivial. Pero si mostramos un poco de flexibilidad podemos recuperarnos de errores que no sean exactamente excepcionales, como lo muestra el siguiente código:

```
#include <stdexcept>

class Prueba
{
public:
   void UnMetodo(int v);
};

void Prueba::UnMetodo(int v)
{
   if (v < 5) { return; }

   throw std::overflow_error {"Ups, valor por encima del máximo"};
}

int main(void)
{
   Prueba p;

   auto salir {false};
   while (not salir) {
      try {
         auto v {0};
         std::cout << "Un valor por debajo de 5: " << std::endl;
         std::cin >> v;

         p.UnMetodo (v);
         std::cout << "Ok" << std::endl;

         salir = true;
         // si llegó a está línea quiere decir que no hubo problemas
         // en la llamada al método UnMetodo()
      }
      catch (std::overflow_error & e)
      {
         std::cout << e.what () << std::endl;
```

```
        }
    }

}
```

El método `UnMetodo()` recibe un argumento, y si este es menor que 5, entonces todo va bien. Sin embargo, si el argumento es mayor o igual a 5 se lanza una excepción avisando del error y es capturada en el bloque `catch`. Hemos introducido una variable de control, `salir`, que establece que si es `true` debemos salir del ciclo `while`, mientras que en cualquier otro caso seguiremos hasta que el usuario introduzca un valor dentro de los límites que se piden.

Mientras esta técnica no es exactamente limpia en el sentido de que hemos tenido que introducir un ciclo y una variable de control, nos permite ver de manera muy clara el manejo del error. En estos ejemplos el método `UnMetodo()` no devuelve nada, pero esa no es la norma, por lo que usar excepciones en lugar de códigos de control hace a nuestros programas más claros y menos propensos a errores, o dicho de otra manera, hay que usar excepciones en lugar de códigos de control. Y si hacemos esto, entonces tendremos que ser consistentes a lo largo de todo el programa.

Actividad 1: Agregando excepciones a la clase Reloj

1. En los métodos setters de la clase `Reloj` agregue excepciones del tipo `out_of_range` de tal modo que el programa no termine y le de al usuario la oportunidad de recuperarse, como se ha visto en esta sección.

2. Agregue una excepción `range_error` en el constructor con argumentos y lance una excepción si cualquiera de los atributos horas, minutos o segundos está fuera de rango. El programa deberá terminar.

3. Escriba un constructor copia que se asegure que el objeto del cual se va a copiar esté en un estado seguro antes de hacer la copia. Lance excepciones de no ser así, y termine el programa.

Entregables: El archivo reloj_act1.cpp

2.3 Limpiando y devolviendo los recursos

Si no fuera posible recuperarse de un error o situación excepcional, entonces deberíamos dejar al sistema y sus objetos en un estado estable, devolver los recursos que se hubieran pedido, y no necesariamente terminar, aunque esto último depende de la estructura de nuestro programa.

La forma más conveniente para usar recursos en un programa tolerante a fallos es a través del idiom RAII.

RAII (Resource Acquisition Is Initialization)

RAII es un idiom que establece que si un objeto va a requerir algún tipo de recurso (archivos, memoria, impresoras, puertos, etc.) este se deberá pedir e inicializar desde el constructor y devolverse desde el destructor. De esta forma, cuando el objeto esté completamente construído tendremos la seguridad de que éste se encuentra en un estado seguro, conocido y que los recursos que hubiera pedido estén listos para usarse. Una vez que el objeto llegue al final de su vida (porque su ámbito terminó, o se hizo una llamada explícita a `delete`) debe devolver tales recursos desde el destructor. Para que todo esto sea posible los recursos deberán ser objetos, y deberían utilizarse apuntadores inteligentes, como `unique_ptr` o `shared_ptr`. Focalizando la construcción y destrucción de los objetos tendremos un problema menos en el cual preocuparnos cuando escribamos nuestros programas.

Aunque dejar a un objeto en un estado seguro no es ni exacta ni xx para el programa, es lo mejor que podemos hacer para evitar que el objeto en estado corrupto tenga daños colaterales en el resto de la aplicación. Por ejemplo, ¿qué podemos hacer con la hora del reloj de la Actividad 1 si uno de los atributos está fuera del rango? Aunque en la actividad insistimos en que el usuario lo arreglara, esto no siempre será posible. Por lo tanto, lo que podríamos hacer es que la excepción ponga la hora completa en un estado seguro, tal vez 00:00:00, y continuar. De esta forma no tendríamos porqué terminar el programa. Quizás 00:00:00 no sea lo más conveniente para la aplicación, pero el programa seguiría en ejecución.

Actividad 2: Dejando a los objetos en un estado seguro

1. En los métodos setters de la clase `Reloj` agregue excepciones del tipo `out_of_range` de tal modo que el programa ponga al objeto en un estado seguro. Después de las

llamadas a los setters imprima la hora para ver si sus excepciones funcionaron adecuadamente.

2. Escriba un constructor copia que se asegure que el objeto del cual se va a copiar esté en un estado seguro antes de hacer la copia. De no ser así, deberá dejar a ambos objetos en un estado seguro, y lanzar una excepción que le avise al cliente sobre esta situación.

Entregables: El archivo reloj_act2.cpp

3 Cuándo sí y cuándo no usar excepciones

3.1 Algunas notas filosóficas

Los LOO, incluyendo a C++, incorporan un mecanismo avanzado para manejar errores en tiempo de ejecución: las excepciones. Y como el nombre lo indica, se trata de manejar situaciones excepcionales en nuestros programas, situaciones que se supone no deberían (o deben) haber sucedido. Existe un debate en cuanto al uso y abuso de las excepciones, y mientras una escuela de pensamiento establece que hay que usar excepciones siempre, otra escuela propone usarlas cuando realmente sucedan cosas excepcionales, y aquí hay que definir lo que vamos a entender por *excepcionales*. Incluso hay autores que indican que las excepciones son un tema tan amplio y delicado a la vez, que merecen libros completos únicamente para hablar de ellas. Nosotros vamos a estudiar a las excepciones como una (poderosa) herramienta más, y será la experiencia y sentido común quien dicte cuándo y bajo qué circunstancias es necesario o no utilizarlas.

Sin embargo, podemos tomar en cuenta los siguientes tres puntos como punto de partida para el uso de excepciones:

1 **Código existente**. Si vamos a trabajar (por ejemplo, para agregar funcionalidad o arreglar errores) sobre un código que no maneja excepciones, entonces no hay que usarlas. Esto incluye al código antiguo.

2 **Código nuevo**. Si vamos a trabajar en un código nuevo, podemos usar excepciones. Cada uno decide, en base a lo ya mencionado, si lo hace o no, y ser consistentes con su utilización. No se vale usarlas en unas partes y en otras no, o en unas partes usarlas de una forma y en otras de otra. Y en caso de decidir usarlas, ceñirse a situaciones realmente excepciona-

les – por ejemplo, si intentamos abrir un archivo que no existe en la ruta – , ¿es esto una situación excepcional? No. El que un archivo no esté en la ruta es algo que esperamos cuando escribimos nuestra lógica.

3 **Sistemas embebidos o sistemas en tiempo real**. En ambos casos se requiere que el código se complete dentro de unas ventanas de tiempo muy rígidas. Las excepciones, cuando se activan, rompen con este esquema: "brincan" a alguna parte del código (al manejador de la excepción), por lo que la tarea que lanzó la excepción no va a terminar su trabajo, y podría ser que el tiempo que tome llegar al manejador de la misma sea demasiado lento y se pierda la ventana. Además, se podría dejar al sistema en un estado no válido. Estas tres situaciones son inaceptables en este tipo de software.

Antes de que el alumno comience a decidir sobre utilizarlas o no, es importante que entienda lo que son para que en el futuro no tome una escuela u otra de pensamiento a rajatable, sino que se coloque en un punto intermedio y se cree un criterio, y quizás las llegue a usar de manera limpia, correcta y elegante en situaciones donde uno no lo esperaría, haciendo a sus programas más seguros y robustos.