VNIVERSITAT DE VALÈNCIA

ESTRUCTURAS DE DATOS Y ALGORITMOS
***Grado en Ingeniería Multimedia (2º).* Curso 2022-23**

Práctica Nº 3: Tratamiento de excepciones

Periodo de realización: Semana del **24/10/2022 a 28/10/2022**

Objetivos

- Aprender el uso de excepciones en C++.
- Implementar precondiciones mediante excepciones.
- Implementar una plantilla de clases.

Introducción

El tipo abstracto de datos Cola se define con la siguiente especificación formal:

TAD Cola

Dominio

E: Tipo de elementos contenidos en una cola

cola: Conjunto de colas que contienen elementos de E

$\mathcal{B}$: Conjunto de valores lógicos = {cierto, falso}

Operaciones

```
c_nueva: → cola
encolar: cola,E → cola
desencolar: cola → cola
primero: cola → E
vacia: cola → B
```

Ecuaciones:* ∀*c:cola;* ∀*x:E

1) desencolar(encolar(c,x)) ≡
   - c_nueva si vacia(c)=cierto
   - encolar(desencolar(c),x) si no

2) primero(encolar(c,x)) ≡
   - x si vacia(c)=cierto
   - primero(c) si no

3) vacia(encolar (c,x)) ≡ falso

4) vacia(c_nueva) ≡ cierto

Ecuaciones de error

5) desencolar(c_nueva) ≡ error

6) primero(c_nueva) ≡ error

Las ecuaciones de error (5 y 6) establecen precondiciones para las operaciones a las que se refieren. Estas precondiciones se pueden implementar en el código mediante asertos (`assert` de *C++*), como ya se ha hecho previamente en la asignatura. Sin embargo, el uso de `assert` en C++ puede ser excesivamente rígido en determinados contextos, ya que cuando se incumple un aserto el programa finaliza inmediatamente y no hay posibilidad de establecer una alternativa. En realidad, lo que ocurre con la macro `assert` es que realiza dos acciones:

(1) Verificar el cumplimiento del aserto y
(2) Tomar una decisión respecto a lo que hacer en caso de incumplimiento (finalizar el programa, sin otra opción).

El problema práctico reside en que `assert` unifica la detección de una situación anómala y su tratamiento. Sin embargo, sería mucho mejor informar al programa cuando se incumpla un aserto (detección) y dejar a criterio del propio programa (desarrollador) la decisión sobre qué hacer en cada caso (tratamiento). Esta decisión puede tener en cuenta la gravedad de la situación en el contexto global de la aplicación. Por ejemplo, el hecho de que la operación *`primero`* del tipo Cola no se pueda realizar por no cumplirse su precondición puede ser más o menos grave dependiendo de la aplicación concreta en la que se use. Es posible que haya aplicaciones donde consultar el "primero" de una cola que esté vacía pueda resultar muy grave y donde esté justificado finalizar inmediatamente el programa porque sea imposible continuar. Sin embargo, en otras aplicaciones el mismo caso puede no resultar especialmente importante y simplemente se requiera esperar hasta que un nuevo dato se almacene en la cola. En cualquier caso, si en la Cola todas las precondiciones se verifican mediante la función `assert`, no será posible tomar una decisión adaptada a cada circunstancia y las dos situaciones descritas previamente tendrán la misma respuesta, finalizar el programa si el aserto se incumple.

Existen alternativas para evitar la rigidez de `assert`. En C++ (y en otros muchos lenguajes de programación) existen las denominadas *excepciones*. Estos elementos del lenguaje permiten separar la detección de una situación anómala de su tratamiento. De manera que, el mismo problema puede tener distintos tratamientos alternativos, incluso dentro del mismo programa.

Las excepciones, como manejo de situaciones inesperadas en el programa, permiten una mayor flexibilidad a la hora de dar una respuesta diferente y personalizada a los "errores" que puedan aparecer según las necesidades de cada aplicación (no exclusivamente para las precondiciones de una función sino para cualquier verificación del estado del programa en cualquier punto de este). Por ese motivo, en esta práctica se debe realizar la implementación de la clase Cola usando excepciones como mecanismo de notificación de situaciones anómalas (indicadas por las ecuaciones de error del TAD).

Resumen práctico sobre el manejo de excepciones

Para comprender mejor qué son y cómo se usan las excepciones en C++, se recomienda consultar el capítulo 16 del libro:

> *J., Deitel, Paul, M., Deitel, Harvey, CÓMO PROGRAMAR EN C++ 6ED. 6. Madrid, España, Pearson, 2008. INGEBOOK.*

El libro está disponible en formato electrónico en la biblioteca de la Universitat de València en la siguiente url (**el capítulo comienza en la página 724 del documento**)[1]:

http://www.ingebook.com/ib/NPcd/IB_Escritorio_Visualizar?cod_primaria=1000193&libro=1269

También puede ser de utilidad consultar el siguiente (mini)tutorial:

http://www.cplusplus.com/doc/tutorial/exceptions/

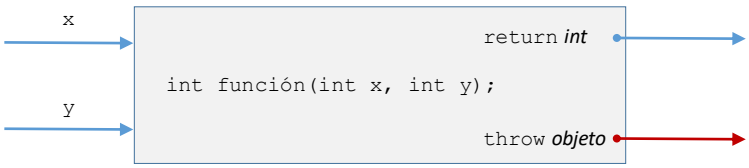
Lanzamiento de una excepción

Cuando una función detecta que no se cumplen las condiciones necesarias para realizar correctamente la tarea prevista (por ejemplo, se incumple una precondición) se puede lanzar una excepción para que la parte del programa que ha invocado a la función reciba esa información y pueda actuar en consecuencia. Por tanto, lanzar una excepción implica enviar un aviso o "alarma", pero solo eso.

Sentencia para lanzar una excepción:

```
throw objeto
```

En una función, la sentencia *throw* es alternativa a la sentencia *return*. Las dos dan por finalizada la ejecución de la función, pero mientras que *return* devuelve el resultado "normal" de la función, *throw* devuelve información sobre la finalización "anormal" de la misma. Por lo tanto, una función es un proceso que recibe unas entradas y genera un resultado a partir de ellas (*return*), pero si se produce alguna circunstancia que impida el correcto procesamiento de la información, la función puede no devolver un resultado sino información sobre el motivo que ha impedido su cálculo (*throw*). La siguiente figura muestra gráficamente este planteamiento:



El lanzamiento de una excepción por parte de una función se debe interpretar como un aviso de que ha ocurrido una situación anómala durante la ejecución de una función y no ha podido finalizar normalmente. Después, será misión del programa o de la función que ha generado la llamada que devuelve una excepción realizar algún tipo de acción como respuesta al aviso recibido. Es lo que se denomina tratar la excepción.

[1] El acceso se debe realizar desde la red de la UV o mediante VPN.

Tratamiento de excepciones

Si una parte del programa invoca a funciones que pueden devolver excepciones y, además, quiere tratarlas de alguna manera deberá realizar dos tareas: (i) capturar las excepciones y (ii) especificar qué hacer cuando ocurran. Para ello, se debe definir un bloque *try*…*catch*:

```
try
{
	//A
	//Todas aquellas sentencias que pueden generar excepciones
}
catch (Tipo1 e)
{
	//Qué hacer cuando ocurre una excepción de Tipo1
}
catch (Tipo2 e)
{
	//Qué hacer cuando ocurre una excepción de Tipo2
}
```

Cuando se produce una excepción en alguna de las sentencias incluidas en el bloque *try* (bloque A) entonces finaliza la ejecución de ese bloque[2] y se busca el bloque *catch* correspondiente al tipo de excepción generada, puesto que una función puede recibir diferentes tipos de excepcios/avisos. Si se encuentra un *catch* del tipo correcto, entonces se ejecutan las sentencias indicadas allí y la ejecución del programa continua después del último *catch* asociado al bloque *try* que ha generado la excepción. Si la excepción generada no se ajusta a ninguno de los tipos indicados en los bloques *catch* entonces la función donde se ha definido el bloque *try*…*catch* finaliza lanzando esa misma excepción. Si el control se devuelve al programa principal y tampoco sabe tratar la excepción, entonces el programa finaliza anormalmente.

Existe una notación especial para especificar un bloque *catch* que permita tratar cualquier tipo de excepción[3]:

```
catch (...)
{
	//Qué hacer cuando ocurre cualquier tipo de excepción
}
```

Trabajo a realizar

Git: Debes trabajar en la carpeta "Pr3" del repositorio que ya tienes creado para la asignatura. Descarga los archivos disponibles en Aula Virtual y realiza un primer *commit* ("Pr.3: Versión inicial") y *push* con ese material inicial. Respeta la estructura de carpetas que se ha utilizado en las anteriores prácticas (`src`, `include`, etc.). A partir de aquí, ya sabes que debes registrar periódicamente los cambios realizados. La política de gestión de los repositorios es responsabilidad de los estudiantes y se valorará en el apartado "Organización" de los criterios de evaluación de las prácticas.

[2] Es importante entender esto. En el momento en que alguna sentencia del bloque `try` genera una excepción, el resto de las sentencias de ese bloque ya no se ejecutan. Por tanto, forma parte del diseño del programa decidir qué sentencias se incluyen en el mismo bloque `try`.

[3] El argumento de catch es (literalmente) tres puntos.

Tarea 1: Completar la clase Cola (con excepciones)

Se debe terminar de implementar el tipo Cola como una clase de *C++* de acuerdo con su TAD y los siguientes criterios (ver los comentarios incluidos en el código):

1) Se proporciona una versión de la clase Cola implementada como una plantilla (*template*) que permite definir colas para almacenar cualquier tipo de valores (tipo E de la definición formal).
   `template <class E> class Cola;`
   Notas de implementación:
   - Se ha utilizado el tipo estándar `queue` para facilitar la implementación. En realidad, solo se ha realizado una adaptación de este tipo.
   - Se debe leer la documentación de las operaciones para entender su significado y forma de uso.
   - Al tratarse de una plantilla, interfaz e implementación están en el mismo archivo y deberá incluirse en las aplicaciones que lo usen (`#include "cola.h"`).
2) Se debe completar la implementación proporcionada, de manera que aquellas operaciones que tengan alguna precondición se comporten de la siguiente forma:
   a. Comprueben mediante una condición (no con `assert`) si se cumple la precondición como primer paso en su implementación.
   b. Si no se satisface la precondición, la operación deberá lanzar una excepción del tipo *runtime_error* incorporando un *string* con el nombre de la operación que ha generado el problema. Por ejemplo, la forma de lanzar la excepción en la operación `Cola<E>::Primero` sería:

      ```
      throw ( runtime_error("Cola::Primero()") );
      ```

      Nota de implementación:
      - La clase `runtime_error` es un tipo estándar de excepción.
      - Para poder usar la clase `runtime_error` se ha incluido `<stdexcept>`

Se recomienda validar esta tarea realizando un pequeño programa de test que declare una cola de algún tipo simple (`int`, `char`, etc.), donde se almacenen unos pocos elementos (<10) y después se fuerce el lanzamiento de excepciones eliminando de la cola más elementos de los existentes y consultando el primero cuando la cola está vacía. Este programa no tiene por qué tratar las excepciones, se trata simplemente de visualizar los errores producidos al generarse la excepción.

Tarea 2: Programar con tratamiento de excepciones

Se debe crear programa, llamado "`main.cpp`", que cumpla los siguientes criterios:

1) El programa debe simular la conexión y desconexión de jugadores a un juego en línea.
2) En una partida del juego solo pueden participar 10 jugadores. Si un nuevo jugador se conecta y hay menos de 10 jugadores activos, entonces pasará directamente a jugar, en caso contrario, pasará a una sala de espera hasta que haya un sitio libre en la partida.
3) Los jugadores que participan en la partida (hasta 10) se almacenan en un `vector` y la sala de espera se implementa con una `cola` para respetar el orden de conexión.
4) La información de conexiones y desconexiones de usuarios reside en un archivo de texto, donde cada línea hace referencia a una operación de este tipo con un formato sencillo: (i) una

conexión está identificada con la letra "C" seguida por el nombre del usuario (que no contiene espacios) y (ii) una desconexión se identifica con la letra "D", sin más información. Ejemplo de entrada:

```
C aragorn
C frodo
C gandalf
D
C arwen
D
…
```

El significado de esta entrada sería: se conecta el usuario "aragorn", después se conectan "frodo" y "gandalf", a continuación, se desconecta un jugador (más adelante se explica cuál), se conecta "arwen", se desconecta un jugador, etc.

5) El programa debe leer línea a línea el archivo de entrada (*datos.txt*) y actuar de la siguiente manera:
   a. Si la línea representa una conexión, entonces:
      i. Si hay plazas disponibles en el vector de jugadores de la partida, se debe añadir el nuevo usuario (string) a este vector. Además, se debe mostrar por pantalla el mensaje "<usuario>: Jugador activo", donde "<usuario> es el nombre de usuario leído del archivo.
      ii. Si no hay plaza en la partida, se debe enviar al usuario a la sala de espera (añadir a la cola). Si esta operación se puede realizar correctamente (no hay excepción), se mostrará por pantalla el mensaje "<usuario>: Espera turno en la sala de espera". Si hubiera algún error el mensaje sería "<usuario>: Imposible conectar, inténtelo más tarde". En ningún caso, el programa finalizará porque se lance una excepción desde Cola.
   b. Si la línea leída indica una desconexión, entonces se debe eliminar de la partida al primer jugador del vector de jugadores activos[4], pero antes mostrará el mensaje "<usuario>: Ha sido desconectado del juego", donde <usuario> es el nombre de usuario eliminado. Además, se debe llevar al primer usuario de la sala de espera al vector de jugadores activos en la partida (eliminar de la cola y añadir al vector). Si es posible realizar este proceso e incorporar un jugador a la partida entonces, se mostrará el mensaje "<usuario>: Pasa a ser Jugador activo". En caso contrario, si al extraer un usuario de la sala de espera se generará un error (excepción) entonces, se debería mostrar el mensaje "Hay espacio disponible en la partida". En ningún caso, el programa finalizará porque se lance una excepción desde Cola.
6) Para validar los resultados es necesario que todos los mensajes mostrados por pantalla se vuelquen también a un archivo de texto, denominado "salida.txt".

[4] myvector.erase (myvector.begin())

Tarea 3: Verificación de resultados

El programa debe generar, a partir del archivo de datos proporcionado, un archivo de salida con 1.200 líneas, cuyas primeras y últimas líneas deben ser:

```
carmenmerc: Jugador activo
fernandolo: Jugador activo
mariaeuseb: Jugador activo
carmenmerc: Ha sido desconectado del juego
Hay espacio disponible en la partida
doloresjul: Jugador activo
zuriamusle: Jugador activo
genowefagu: Jugador activo
dioselinam: Jugador activo
daniulianv: Jugador activo
…
karinegraj: Ha sido desconectado del juego
mariacarme: Pasa a ser Jugador activo
roberthagu: Espera turno en la sala de espera
franciscoa: Espera turno en la sala de espera
olgasusana: Espera turno en la sala de espera
helenacarm: Espera turno en la sala de espera
andresnico: Espera turno en la sala de espera
antoniomar: Espera turno en la sala de espera
vanessaale: Espera turno en la sala de espera
yuanyuanco: Espera turno en la sala de espera
surjeetdum: Espera turno en la sala de espera
nataliamar: Espera turno en la sala de espera
rociocinta: Espera turno en la sala de espera
```

Tarea 4: Generación de documentación

El código del programa deberá estar correctamente documento con Doxygen, de acuerdo con la guía de estilo. Una vez finalizado el programa, se deberá ejecutar Doxygen para verificar la correcta generación de la documentación del programa.