Práctica 3: La rebelión de los oprimidos

Programación 2

Curso 2021-2022

Esta práctica consiste en implementar una versión inicial de un juego de recogida de recursos de un mapa, siguiendo el paradigma de programación orientada a objetos. Los conceptos necesarios para desarrollar esta práctica se trabajan en todos los temas de teoría, aunque especialmente en el *Tema 5*.

Condiciones de entrega

- La fecha límite de entrega para esta práctica es el **viernes 27 de mayo**, hasta las **23:59**
- La práctica consta de varios ficheros: `Coordinate.cc`, `Coordinate.h`, `Junk.cc`, `Junk.h`, `Betonski.cc`, `Betonski.h`, `Jankali.cc`, `Jankali.h`, `Map.cc`, `Map.h`, `Util.cc` y `Util.h`. Todos ellos se deberán comprimir en un único fichero llamado `prog2p3.tgz` que se entregará a través del servidor de prácticas de la forma habitual. Para crear el fichero comprimido debes hacerlo de la siguiente manera:

Terminal

```
$ tar cvfz prog2p3.tgz Coordinate.cc Coordinate.h Junk.cc Junk.h Betonski.cc
Betonski.h Jankali.cc Jankali.h Map.cc Map.h Util.cc Util.h
```

Código de honor

Si se detecta copia (total o parcial) en tu práctica, tendrás un **0** en la entrega y se informará a la dirección de la Escuela Politécnica Superior para que adopte medidas disciplinarias

Está bien discutir con tus compañeros posibles soluciones a las prácticas
Está bien apuntarte a una academia si sirve para obligarte a estudiar y hacer las prácticas

Está mal copiar código de otros compañeros para resolver tus problemas
Está mal apuntarte a una academia para que te hagan las prácticas

Si necesitas ayuda acude a tu profesor/a
No copies

Normas generales

- Debes entregar la práctica exclusivamente a través del servidor de prácticas del Departamento de Lenguajes y Sistemas Informáticos (DLSI). Se puede acceder a él de dos maneras:
  - Página principal del DLSI (`https://www.dlsi.ua.es`), opción “ENTREGA DE PRÁCTICAS”

    - Directamente en la dirección https://pracdlsi.dlsi.ua.es
- Cuestiones que debes tener en cuenta al hacer la entrega:
    - El usuario y la contraseña para entregar prácticas son los mismos que utilizas en UACloud
    - Puedes entregar la práctica varias veces, pero sólo se corregirá la última entrega
    - No se admitirán entregas por otros medios, como el correo electrónico o UACloud
    - No se admitirán entregas fuera de plazo
- Tu práctica debe poder ser compilada sin errores con el compilador de C++ existente en la distribución de Linux de los laboratorios de prácticas
- Si tu práctica no se puede compilar su calificación será 0
- La corrección de la práctica se hará de forma automática (**no habrá corrección manual del profesor**), por lo que es imprescindible que respetes estrictamente los textos y los formatos de salida que se indican en este enunciado
- Al comienzo de todos los ficheros fuente entregados debes incluir un comentario con tu NIF (o equivalente) y tu nombre. Por ejemplo:

Junk.h

```
// DNI 12345678X GARCIA GARCIA, JUAN MANUEL
...
```

- El cálculo de la nota de la práctica y su relevancia en la nota final de la asignatura se detallan en las transparencias de presentación de la asignatura (*Tema 0*)

1. Descripción de la práctica

En esta práctica se implementará una versión inicial de un juego en el que unos seres llamados *betonski* recorrerán un mapa obteniendo recursos (*junk*), y otros seres llamados *jankali* pondrán trampas para capturar *betonski* y expoliar los recursos que han recolectado, todo ello utilizando el paradigma de programación orientado a objetos.

2. Detalles de implementación

En el *Moodle* de la asignatura se publicarán varios ficheros que necesitarás para la correcta realización de la práctica:

- `Util.h` y `Util.cc`. El fichero `Util.h` contiene algunos tipos necesarios para la práctica, y la definición de la clase `Util` con métodos auxiliares
- `prac3.cc`. Fichero que contiene el `main` de la práctica. Se encarga de crear los objetos de las clases implicadas en el problema, leyendo instrucciones de un fichero de texto. En la Sección 4 se describen las instrucciones y el funcionamiento del programa. Este fichero no debe modificarse ni incluirse en la entrega final
- `prac3`. Fichero ejecutable de la práctica (compilado para máquinas Linux de 64 bits) desarrollado por el profesorado de la asignatura, para que puedas probarlo con las entradas que quieras y ver la salida correcta esperada
- `makefile`. Fichero que permite compilar de manera óptima todos los ficheros fuente de la práctica y generar un único ejecutable

- `autocorrector-prac3.tgz`. Contiene los ficheros del autocorrector para probar la práctica con algunas pruebas de entrada. Además contiene varias pruebas unitarias para probar los métodos por separado. La corrección automática de la práctica se realizará con un corrector similar, con estas pruebas y otras más definidas por el profesorado de la asignatura

En esta práctica cada una de las clases se implementará en un módulo diferente, de manera que tendremos dos ficheros para cada una de ellas: `Coordinate.h` y `Coordinate.cc` para las coordenadas, `Junk.h` y `Junk.cc` para los recursos, `Betonski.h` y `Betonski.cc` para los *betonski*, `Jankali.h` y `Jankali.cc` para los *jankali*, `Map.h` y `Map.cc` para el mapa, y `Util.h` y `Util.cc` para los métodos auxiliares. Estos ficheros, junto con `prac3.cc`, se deben compilar conjuntamente para generar un único ejecutable. Una forma de hacer esto es de la siguiente manera:

Terminal

```
$ g++ Coordinate.cc Junk.cc Betonski.cc Jankali.cc Map.cc Util.cc prac3.cc -o prac3
```

Esta solución no es óptima, ya que compila de nuevo todo el código fuente cuando puede que solo alguno de los ficheros haya sido modificado. Una forma más eficiente de realizar la compilación de código distribuido en múltiples ficheros fuente es mediante la herramienta `make`. Debes copiar el fichero `makefile` proporcionado en Moodle dentro del directorio donde estén los ficheros fuente e introducir la siguiente orden:

Terminal

```
$ make
```

- Puedes consultar las transparencias 60 en adelante del *Tema 5* si necesitas más información sobre el funcionamiento de `make`

2.1. Excepciones

Algunos de los métodos que vas a crear en esta práctica deben lanzar excepciones. Para ello deberás utilizar `throw` seguido del tipo de excepción, que en C++ puede ser cualquier valor (un entero, una cadena, etc), pero en esta práctica debe ser uno de los valores definidos en `Util.h`. Algunos de esos valores se han definido de forma que coincidan con alguno de los valores posibles del tipo enumerado `Error`) para que sea más sencillo emitir mensajes de error. Las excepciones se pueden capturar mediante `try/catch` donde quiera que se invoque a un método que pueda lanzar una excepción, y también se puede propagar (en C++ basta con no poner un `try/catch` para que se propague). A continuación se muestra un ejemplo de cómo se lanzaría una excepción desde un método y cómo se capturaría en otro:

```
// Método donde se produce la excepción
Junk Map::getJunk(const Coordinate &coord) const
{
  ...
  // Si la coordenada está fuera del mapa lanzamos la excepción con "throw"
  if(...){
    throw EXCEPTION_OUTSIDE;
  }
  ...
}
...
// Método donde se captura la excepción
  ...
  try{
```

```
    ...
    Coordinate c(-5,8);
    Junk contenido = map.getJunk(c);  // podría lanzar EXCEPTION_OUTSIDE
    ...
  }
  // Si se produce la excepción, mostramos el error correspondiente
  catch(Exception e){
    Util::error((Error) e);
    // como el valor de la excepción coincide con el valor del error, es
    // suficiente con hacer una conversión (cast) para llamar al método
    // error con el valor adecuado
  }
}
```

En esta práctica, la mayoría de las excepciones que se lanzan se deben capturar en el fichero `prac3.cc`, excepto dos: la `EXCEPTION_REBELION`, que debe capturarse en `Jankali.cc`, y la `BETONSKI_NOT_CAPTURED`, que es un error interno de programación y no debe capturarse en ningún sitio

2.2. Cambio de `private` a `protected` para la corrección automática

Para facilitar la posterior corrección por parte del profesorado de la práctica mediante pruebas unitarias (pruebas que evalúan el funcionamiento de cada una de las clases de manera aislada), **deberás declarar los atributos y métodos privados de las clases como `protected` en lugar de como `private`**. Un atributo o método `protected` es similar a uno privado. La diferencia es que los atributos o miembros `protected` son inaccesibles fuera de la clase (como los privados), pero pueden ser accedidos por una subclase (clase derivada) que herede de ella. Por ejemplo, la clase `Junk` se declararía de esta manera:

```
class Junk{
  protected: // Ponemos "protected" en lugar de "private"
    JunkType type;
    int quantity;
    ...
  public:
    Junk();
    ...
};
```

A lo largo de este enunciado, siempre que hablemos de métodos o atributos "privados" nos estaremos refiriendo a aquellos que irán en la parte `protected` de las clases que vas a definir.

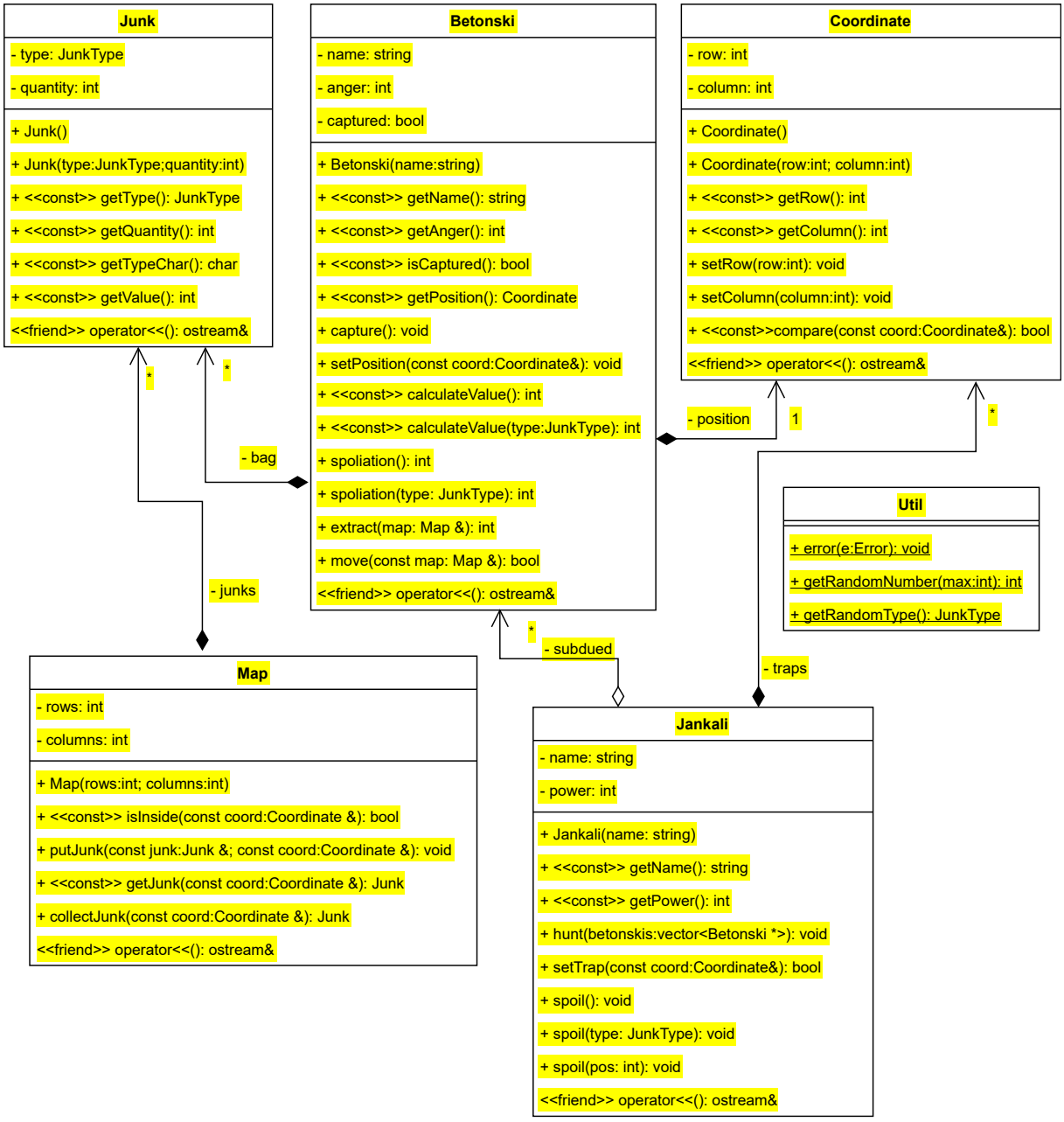
3. Clases y métodos

La figura que aparece en la página siguiente muestra un diagrama UML con las clases que hay que implementar, junto con los atributos, métodos y relaciones que tienen lugar en el escenario de esta práctica.

Si necesitas incorporar más métodos y atributos a las clases descritas en el diagrama, puedes hacerlo, pero siempre incluyéndolos en la parte privada (`protected`) de las clases. Recuerda también que las relaciones de *agregación* y *composición* dan lugar a nuevos atributos cuando se traducen del diagrama UML a código. Consulta las transparencias 58 y 59 del *Tema 5* si tienes dudas sobre cómo traducir las relaciones de *agregación* y *composición* a código.

- **¡Ojo!** En esta práctica no está permitido añadir ningún atributo o método público (`public`) a las clases definidas en el diagrama, ni añadir o cambiar argumentos de los métodos. Sin embargo, como se ha indicado, sí que puedes añadir atributos y métodos privados (`protected`)

A continuación se describen los métodos de cada clase. Es posible que algunos de estos métodos no sea necesario utilizarlos en la práctica, pero se utilizarán en las pruebas unitarias durante la corrección. Se recomienda implementar las clases en el orden en que aparecen en este enunciado.



3.1. Util

En el Moodle de la asignatura se proporcionará esta clase, que incluirá:

- el tipo enumerado `Error` con todos los posibles errores que se pueden dar en la práctica, además del método `error` para emitir los correspondientes errores por pantalla
- el tipo enumerado `Exception` con todas las posibles excepciones que pueden producirse. Algunos valores coinciden con valores del tipo `Error` para facilitar emitir el mensaje de error
- un método `int getRandomNumber(int max)` que devolverá un número aleatorio entre `0` y `max-1`
- el tipo enumerado `JunkType` (que se describe más adelante), y un método `JunkType getRandomType()` que devolverá un valor aleatorio de dicho tipo enumerado

- Ten en cuenta que `error`, `getRandomNumber` y `getRandomType` son métodos de clase (`static`) y por eso deben invocarse utilizando esta sintaxis: nombre de la clase, seguido de `::` y finalmente el nombre del método. Consulta la transparencia 45 del *Tema 5* si tienes dudas a este respecto. Por ejemplo, para mostrar el mensaje de error `ERR_NAME`, deberás invocar al método `error` pasándole el correspondiente parámetro de la siguiente manera:

  ```
  Util::error(ERR_NAME);
  ```

3.2. Coordinate

Esta clase representa coordenadas en el mapa con dos valores, fila (`row`) y columna (`column`). Los métodos de esta clase son:

- `Coordinate()`. Constructor por defecto, inicializa los dos atributos a -1
- `Coordinate(int row,int column)`. Constructor que inicializa los atributos con los valores pasados como parámetros
- `int getRow() const`, `int getColumn() const`. *Getters* que devuelven los valores de los atributos
- `void setRow(int row)`, `void setColumn(int column)`. *Setters* que modifican los valores de los atributos
- `bool compare(const Coordinate &coord) const`. Método que compara la coordenada actual (`this`) y `coord`), devolviendo `true` si son iguales y `false` en caso contrario
- `ostream& operator<<(ostream &os,const Coordinate &coord)`. Operador de salida que muestra la coordenada. Por ejemplo, si `row` vale 7 y `column` vale 4, se mostraría esto (sin `\n` al final):

Terminal

```
[7,4]
```

3.3. Junk

Esta clase representa los diferentes tipos de recursos que pueden ser recolectados, y la cantidad de recurso disponible. Los tipos se definen con el siguiente tipo enumerado:

Terminal

```
enum JunkType {
  WASTELAND,
  GOLD,
  METAL,
  FOOD,
  STONE
};
```

El tipo `WASTELAND` representa la ausencia de recursos, y por tanto no puede ser recolectado.
Los métodos de esta clase son:

- `Junk()`. Constructor por defecto, inicializa el tipo a `WASTELAND` y la cantidad a 0
- `Junk(JunkType type,int quantity)`. Constructor que inicializa los atributos con los valores pasados como parámetros. Si el valor pasado para `quantity` es menor que 0, debe lanzarse una excepción con valor `EXCEPTION_QUANTITY`
- `JunkType getType() const`, `int getQuantity() const`. *Getters* que devuelven los valores de los atributos
- `char getTypeChar() const`. Devuelve la primera letra del tipo, es decir, "`W`", "`G`", "`M`", "`F`" o "`S`"
- `int getValue() const`. Devuelve el valor del recurso, que será el valor del tipo multiplicado por la cantidad. El valor del tipo se indica en esta tabla:

`WASTELAND`	0
`GOLD`	500
`METAL`	100
`FOOD`	50
`STONE`	20

  Por ejemplo, si el tipo es `GOLD` y la cantidad 12, el método `getValue` devolvería 6000 (500, que es el valor del oro, multiplicado por 12)
- `ostream& operator<<(ostream &os,const Junk &junk)`. Operador de salida que muestra el recurso. Por ejemplo, si `type` es `GOLD` y `quantity` vale 12, se mostraría esto (sin `\n` al final):

Terminal

```
[GOLD:12]
```

- El tipo enumerado `JunkType` está ya en el fichero `Util.h`. Debes hacer un `include` de `Util.h` en tu `Junk.h` para poder usarlo en esta clase
- La tabla con el valor de cada tipo no debe ser visible en otras clases, por lo que debes declararla en el fichero `Junk.cc`

3.4. Map

Esta clase representa el mapa (rectangular) con los recursos del juego. Sus atributos serán el número de filas (`rows`), el número de columnas (`columns`) y una matriz bidimensional de objetos de la clase `Junk` (`junks`). Aunque se podría implementar de otras formas, en esta práctica debes implementar la matriz `junks` como un vector de vectores de `Junk`: `vector<vector<Junk>> junks;`

Para construir la matriz (en el constructor descrito más abajo) tendrás que realizar los siguientes pasos **para cada fila** (es decir, estos cuatro pasos tendrán que repetirse tantas veces como filas tenga la matriz):

1. Declarar una variable de tipo `Junk` creada con el constructor por defecto `Junk()` que la inicializa a `WASTELAND`
2. Declarar una variable que sea un vector de `Junk` (`vector<Junk>`)
3. Añadir a este vector la variable anterior de tipo `Junk` con `push_back` tantas veces como columnas tenga la matriz (de esta manera construimos una fila de la matriz)
4. Finalmente, añadir el `vector<Junk>` a la matriz `junks` usando `push_back`

Los métodos de esta clase son:

- `Map(int rows,int columns)`. Constructor que inicializa los atributos `rows` y `columns`, y crea la matriz de objetos `Junk` como se explica más arriba. Los *junk* serán creados con el constructor por defecto de `Junk`, que los inicializa a `WASTELAND`, aunque con el método `putJunk` podrán reemplazarse por otros objetos `Junk` de otros tipos. Si `rows` o `columns` son menores que 5 se debe lanzar una excepción con valor `EXCEPTION_SIZE` y no crear el mapa (es decir, el tamaño mínimo del mapa es *5x5*)
- `bool isInside(const Coordinate &coord) const`. Devuelve `true` si la coordenada está dentro del mapa y `false` en caso contrario
- `void putJunk(const Junk &junk,const Coordinate &coord)`. Si la coordenada (`coord`) está dentro del mapa, reemplaza el contenido de la matriz `junks` en esa coordenada por el recurso (`junk`) pasado como parámetro. Si la coordenada está fuera del mapa debe lanzarse una excepción con valor `EXCEPTION_OUTSIDE`
- `Junk getJunk(const Coordinate &coord) const`. Si la coordenada (`coord`) está dentro del mapa, devuelve el contenido de esa coordenada (sin modificar el mapa). Como en el método anterior, debe lanzarse una excepción `EXCEPTION_OUTSIDE` si la coordenada está fuera del mapa
- `Junk collectJunk(const Coordinate &coord)`. Si la coordenada (`coord`) está dentro del mapa, devuelve el contenido de la matriz en esa coordenada y lo reemplaza en la matriz por un `Junk` de tipo `WASTELAND`. Como en los métodos anteriores, debe lanzarse una excepción `EXCEPTION_OUTSIDE` si la coordenada está fuera del mapa
- `ostream& operator<<(ostream &os,const Map &map)`. Operador de salida que muestra el mapa con el formato que aparece en el siguiente ejemplo, donde se ve un mapa con 12 filas y 8 columnas:

Terminal

```
   00 01 02 03 04 05 06 07
00    G  F  S     M  M  M
01 G  G  F  F  S  M  M
02
03 M  F  F     S  S  S  S
04 M  M  F     S  S  G  S
05 M                   G
06
07
08 S  S  G  G        F  F
09    S  G              F
10       S  G           M
11          S        M  M
```

En cada posición se representa el tipo de recurso que hay en ella, excepto si se trata de `WASTELAND`, que se representa con un espacio en blanco. Como se puede ver en el ejemplo, las coordenadas empiezan en 0 tanto para filas como para columnas (y siempre se deben poner con dos dígitos, poniendo ceros a la izquierda para rellenar si fuera necesario). Se puede asumir que no va a haber mapas de un tamaño superior a *100x100* (no hace falta comprobarlo)

3.5. Betonski

Esta clase modelará el comportamiento de los *betonski*, que son seres pacíficos y felices que van recolectando recursos por el mapa, pero que cuando son capturados y expoliados por un *jankali* se van enfadando y pueden llegar a rebelarse y romper su vínculo con el opresor. Los atributos de esta clase son el nombre (`name`), el nivel de enfado (`anger`), el saco en el que van guardando lo que recolectan (`bag`), su posición en el mapa (`position`) y si han sido capturados o no (`captured`).

Los métodos de esta clase son:

- `Betonski(string name)`. Constructor que inicializa el nombre (`name`) al valor pasado por parámetro. Inicialmente, los *betonski* son libres (`captured` vale `false`) y por tanto no están enfadados (`anger` vale `0`), y no han sido situados aún en el mapa (`position` vale `[-1,-1]`). Si la cadena que se pasa con el nombre está vacía, debe lanzarse una excepción con `EXCEPTION_NAME`
- `string getName() const`, `int getAnger() const`, `bool isCaptured() const` y `Coordinate getPosition() const`. *Getters* que devuelven el valor de los atributos
- `void capture()`. Indica al *betonski* que ha sido capturado
- `void setPosition(const Coordinate &coord)`. Asigna la posición en `coord` al *betonski*. No es necesario comprobar la coordenada. Podemos asumir que siempre será correcta
- `int calculateValue() const`. Devuelve el valor total de los recursos recolectados por el *betonski*
- `int calculateValue(JunkType type) const`. Devuelve el valor total de los recursos del tipo indicado por el parámetro `type` de entre los recolectados por el *betonski*
- `int spoliation()`. Obtiene el valor de los recursos recolectados y lo devuelve, vaciando el saco y sumando al nivel de enfado dicho valor. Este método será llamado por los *jankali* para expoliar a sus *betonski*. Al principio del método, antes de hacer nada más, se deben hacer un par de comprobaciones (en el siguiente orden):
  1. Si el *betonski* no ha sido capturado (tal y como se describe en la Sección 3.6), lógicamente se negará a entregar sus recursos y el método debe lanzar una excepción `BETONSKI_NOT_CAPTURED`
  2. Si el valor de los recursos que le van a expoliar, sumado con el nivel de enfado que ya tenía el *betonski*, supera el valor `5000`, entonces el *betonski* se rebelará y se liberará. Es decir, dejará de estar capturado, volverá a ser feliz (el valor de `anger` será `0`) y obviamente no devolverá el valor de sus recursos ni vaciará su saco. El método lanzará la excepción `EXCEPTION_REBELION` para indicar a su opresor *jankali* que se ha rebelado y liberado de su opresión. El *jankali* debe capturar la excepción y borrar al *betonski* de su conjunto de *betonski* dominados, como se explica en la siguiente sección
- `int spoliation(JunkType type)`. Como el método anterior, pero solamente devuelve el valor de los recursos del tipo indicado como parámetro, eliminando del saco solo dichos recursos. El nivel de enfado aumenta con dicho valor, como en el método anterior, y también deben hacerse las mismas comprobaciones iniciales (lanzando las excepciones si fuese necesario)
- `int extract(Map &map)`. Si el *betonski* está dentro del mapa, recolecta el recurso de `map` situado en su posición. Si no es `WASTELAND` lo añade al final de su saco y devuelve su valor. Si es `WASTELAND` devolverá `0`
- `bool move(const Map &map)`. Si está dentro del mapa, el *betonski* se mueve a una casilla adyacente. Si está fuera, debe lanzar una excepción `EXCEPTION_OUTSIDE`. Para moverse, primero se obtiene un número aleatorio entre `0` y `7` (usando el método `getRandomNumber()` de `Util.h`) que representa la dirección del movimiento, según este diagrama:

7	0	1
6		2
5	4	3

En función de la posición del *betonski* y la dirección del movimiento obtenida aleatoriamente, se calcula la nueva posición: si está dentro del mapa el *betonski* se moverá a esa posición y el método devolverá "`true`"; si está fuera del mapa, el *betonski* no se moverá y el método devolverá "`false`"

- `ostream& operator<<(ostream &os,const Betonski &betonski)`. Operador de salida que muestra los datos del *betonski* siguiendo el formato que se muestra en este ejemplo para un *betonski* llamado "`Sluk`":

```
Terminal
Betonski "Sluk" Captured 1550 [7,4]
[GOLD:5][STONE:30][FOOD:100][GOLD:3][METAL:10][FOOD:30]
```

En la primera línea se indica el nombre, si ha sido capturado o está libre (en cuyo caso se mostraría "`Free`"), el nivel de enfado y su posición. En la segunda línea se imprime el contenido de su saco. Si está vacío se debe imprimir una línea en blanco.

3.6. Jankali

Esta clase modela el comportamiento de los *jankali*, que son seres opresores cuya aspiración es obtener el máximo de poder explotando a cuantos más *betonski* mejor, para lo que colocan trampas para capturarlos y poderlos explotar. Los atributos de esta clase son el nombre (`name`), el vector de *betonski* capturados (`subdued`), el vector de trampas (`traps`) y el poder que tienen (`power`).

Los métodos de esta clase son:

- `Jankali(string name)`. Constructor que inicializa el nombre (`name`) al valor pasado como parámetro y el poder (`power`) a 300. Si la cadena con el nombre está vacía, debe lanzar una excepción `EXCEPTION_NAME`

- `string getName() const` y `int getPower() const`. *Getters* que devuelven el valor de los atributos correspondientes

- `void hunt(vector<Betonski *>betonskis)`. Si alguno de los *betonski* ha caido en alguna de las trampas del *jankali* (es decir, si la posición del *betonski* coincide con alguna trampa) y no ha sido capturado por ningún otro *jankali*, el *jankali* lo captura y lo añade al final de su vector `subdued`. El vector `betonskis` pasado como parámetro no debe modificarse

- `bool setTrap(const Coordinate &coord)`. Si la coordenada pasada como parámetro no está ya entre las trampas del *jankali* y tiene suficiente poder para costear la trampa, se añade una nueva trampa al final del vector `traps` y se descuenta el coste de la trampa del poder del *jankali*. El coste de la trampa depende de la coordenada y se calcula como `(row+2)*(column+2)`. El método devuelve `true` si ha conseguido poner la trampa y `false` en caso contrario

- `void spoil()`. Este método permite expoliar a todos sus *betonski* (contenidos en el vector `subdued`) sumando el valor obtenido en cada expoliación a su poder. Debe tenerse en cuenta que si un *betonski* se ha liberado lanza una excepción `EXCEPTION_REBELION` (como se comentó en la sección anterior) que debe ser capturada en este método para eliminar al *betonski* del vector `subdued`. Si se diera ese caso, el *jankali* debe seguir intentando expoliar a los demás *betonski* de su vector `subdued`

- `void spoil(JunkType type)`. Igual que el método anterior, pero en este caso solamente expolia los recursos del tipo indicado por el parámetro

- `void spoil(int pos)`. Expolia al *betonski* concreto que está en la posición `pos` del vector `subdued`. Si la posición no es correcta (no está dentro del vector `subdued`) no hace nada

- `ostream& operator<<(ostream &os,const Jankali &jankali)`. Operador de salida que muestra los datos del *jankali* siguiendo el formato del siguiente ejemplo, donde se muestra un *jankali* llamado "`Darkwave`" que tiene 3500 de `power`, ha capturado tres *betonski* y ha colocado cinco trampas:

Terminal

```
Jankali "Darkwave" 3500
Betonski "Sluk" Captured 1550 [7,4]
[GOLD:5][STONE:30][FOOD:100][GOLD:3][METAL:10][FOOD:30]
Betonski "Siut Tisk" Captured 5000 [12,6]
[STONE:300][FOOD:200][GOLD:20][METAL:1000][FOOD:350]
Betonski "Tiste Foo" Captured 0 [11,5]
[STONE:300][FOOD:200][GOLD:20][METAL:1000][FOOD:350]
Traps [0,7][2,0][8,5][11,15][5,6]
```

En la primera línea se indica el nombre y su poder, a continuación se muestra la información de los *betonski* que ha capturado y, finalmente, se muestran en la última línea las coordenadas donde están situadas las trampas

4. Programa principal

El programa principal está en el fichero `prac3.cc` publicado en el Moodle de la asignatura. Este fichero contiene código para leer un fichero con instrucciones de la partida que se pasará al programa como argumento por línea de comando, como se describe más abajo. Este programa principal hace uso de las clases que has definido en tu código para construir un juego completo que puedas probar. **Revísalo, pero no es necesario que lo modifiques**. Es simplemente un ejemplo de cómo se podrían usar las clases que has creado en un programa.

Como se puede ver en el código existente, inicialmente se crea un mapa de `10x10`, un vector de punteros a `Betonski` y un vector de punteros a `Jankali`. El fichero pasado como parámetro con instrucciones para el juego contendrá una instrucción por línea. El formato de estas instrucciones siempre será correcto.

A continuación se describen las distintas instrucciones que se pueden dar en el juego, en caso de que quieras hacer tu propio fichero de instrucciones para crear nuevas partidas:

- `newBetonski` *name*. Crea un nuevo `Betonski` en memoria dinámica y lo añade al final del vector correspondiente
- `newJankali` *name*. Como en la instrucción anterior, crea un nuevo `Jankali` en memoria dinámica y lo añade al final del vector correspondiente
- `posBetonski` *name row column*. Intenta posicionar el *betonski* con ese nombre en la coordenada indicada por `row` y `column`. Si no existe en el vector un *betonski* con ese nombre emitirá el error `ERR_NAME` y no hará nada más. Igualmente, si la coordenada no está dentro del mapa emitirá el error `ERR_OUTSIDE` y no hará nada más. Si todos los datos son correctos asignará al *betonski* la posición indicada por la coordenada
- `putJunk` *type quantity row column*. Crea un recurso de tipo *type* y lo pone en el mapa. Si el tipo es incorrecto emite el error `ERR_TYPE`. Si al crear el recurso o al posicionarlo en el mapa lanza alguna excepción emitirá el error indicado por la excepción
- `setTrap` *name row column*. Permite a un *jankali* colocar una trampa. Si no existe un *jankali* con ese nombre en el vector se emite el error `ERR_NAME` y si la coordenada no está dentro del mapa emitirá el error `ERR_OUTSIDE`
- `exit`. Termina la partida
- `play`. Realiza una jugada de la partida llevando a cabo los siguientes pasos:
  1. Mueve todos los *betonski* (en el orden en que aparecen en el vector)
  2. Hace que todos los *betonski* extraigan del mapa los recursos que encuentren en su nueva posición (también en el orden en que aparecen en el vector, de manera que si dos *betonski* tienen la misma posición, el primero que aparezca en el vector se lleva los recursos)

3. Los *jankali* capturan a los *betonski* que hayan caido en sus trampas (en el orden en que aparecen en el vector, de manera que si un *betonski* está en una posición en la que hay varias trampas de varios *jankali*, el primer *jankali* que aparezca en el vector será el que capturará al *betonski*)
4. Por último, los *jankali* expolian a todos sus *betonski*, para lo cual obtienen un tipo de recurso aleatoriamente (llamando al método `getRandomType` de la clase `Util`). Si el tipo de recurso es `WASTELAND`, se expoliarán todos los recursos. En caso contrario, solamente los recursos del tipo obtenido

Un ejemplo de fichero de entrada sería el siguiente (ten en cuenta que por simplificar los nombres sólo tienen una palabra):

Terminal

```
newBetonski Sluk
newBetonski Tisk
newBetonski Foo
newJankali Darkwave
newJankali TrollMaximus
posBetonski Sluk 2 3
posBetonski Tisk 6 7
posBetonski Foo 5 5
putJunk GOLD 12 4 4
putJunk STONE 10 4 5
putJunk FOOD 20 4 6
putJunk FOOD 30 5 4
putJunk FOOD 40 5 6
putJunk METAL 10 6 4
putJunk METAL 15 6 5
putJunk METAL 12 6 6
setTrap Darkwave 4 4
setTrap TrollMaximus 6 6
play
exit
```

Por último, antes de procesar cada línea se muestra el vector de `Betonski`, el vector de `Jankali` y el mapa. Con el fichero anterior, la salida al final del programa, antes de procesar la instrucción `exit`, sería:

Terminal

```
   00 01 02 03 04 05 06 07 08 09
00
01
02
03
04            G  S
05            F     F
06            M  M
07
08
09
Betonski "Sluk" Free 0 [1,2]

Betonski "Tisk" Captured 1200 [6,6]

Betonski "Foo" Free 0 [4,6]
[FOOD:20]
Jankali "Darkwave" 264
Traps [4,4]
Jankali "TrollMaximus" 1436
Betonski "Tisk" Captured 1200 [6,6]

Traps [6,6]
```