Práctica 3: Hundir la flota

Programación 2

Curso 2022-2023

Esta práctica consiste en implementar una versión inicial del conocido juego de "Hundir la flota", 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 26 de mayo**, hasta las **23:59**
- La práctica consta de varios ficheros: `Coordinate.cc`, `Coordinate.h`, `Ship.cc`, `Ship.h`, `Player.cc`, `Player.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 Ship.cc Ship.h
          Player.cc Player.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:

Ship.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 del juego de "Hundir la flota" en la que cada jugador tiene un tablero y coloca varios barcos sobre él. Cuando ambos jugadores han colocado sus barcos, pueden comenzar a atacar al jugador contrario por turnos, indicando en cada movimiento una coordenada del tablero contrario; si en esa coordenada hay un barco contrario, el jugador que ha acertado en su ataque repite, y si falla el turno pasa al otro jugador.

En esta práctica se implementarán algunas clases para gestionar el juego, pero no el juego en sí (aunque no sería muy difícil de implementar).

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; el fichero `Util.cc` contiene la implementación de dichos métodos
- `prac3.cc`. Fichero que contiene el `main` de la práctica. Se encarga de crear los objetos de las clases implicadas en el problema, y simular algunos movimientos de una partida. Este fichero no debe incluirse en la entrega final, es solamente una ayuda para probar la práctica
- `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 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, `Ship.h` y `Ship.cc` para los barcos, `Player.h` y `Player.cc` para los jugadores, 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 Ship.cc Player.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 el tipo enumerado `Exception` que hay en `Util.h`:

Terminal

```
enum Exception {
  EXCEPTION_WRONG_ORIENTATION,
  EXCEPTION_WRONG_SHIP_TYPE,
  EXCEPTION_WRONG_COORDINATES,
  EXCEPTION_MAX_SHIP_TYPE,
  EXCEPTION_GAME_STARTED,
  EXCEPTION_OUTSIDE,
  EXCEPTION_NONFREE_POSITION,
  EXCEPTION_ALREADY_SUNK,
  EXCEPTION_ALREADY_HIT,
  EXCEPTION_GAME_OVER
};
```

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). Por simplificar, en esta práctica, casi todas las excepciones se capturan en el fichero

prac3.cc, solamente debes usar un try/catch en la clase Player. En aquellos métodos que lanzan más de una excepción, las excepciones deben lanzarse en el orden en el que se mencionan en la descripción del método.

A continuación se muestra un ejemplo (que no es código real de la práctica) de cómo se lanzaría una excepción desde un método y cómo se capturaría en otro (o en una función):

```
// Método donde se produce la excepción
ShipType Ship::typeFromChar(char type)
{
  ...
  // Si 'type' no es la inicial de ninguno de los tipos de
  // barcos conocidos, lanzamos la excepción con "throw"
  if(...){
    throw EXCEPTION_WRONG_SHIP_TYPE;
  }
  ...
}
...
// Función o método donde se captura la excepción
  ...
  try{
    ...
    ShipType = Ship::typeFromChar(type); // podría lanzar EXCEPTION_WRONG_SHIP_TYPE

    // si es correcto, el código sigue
    Coordinate initial(coord);
    ...
  }
  // Si se produce la excepción, mostramos el error correspondiente
  catch(Exception e){
    if (e == EXCEPTION_WRONG_SHIP_TYPE)
        Util::error(ERR_SHIPS);
    else
        ...
  }
}
```

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 `Coordinate` se declararía de esta manera:

```
class Coordinate{
  protected: // Ponemos "protected" en lugar de "private"
    int row;
    int column;
    CellState state;
    ...
  public:
    Coordinate();
    ...
};
```

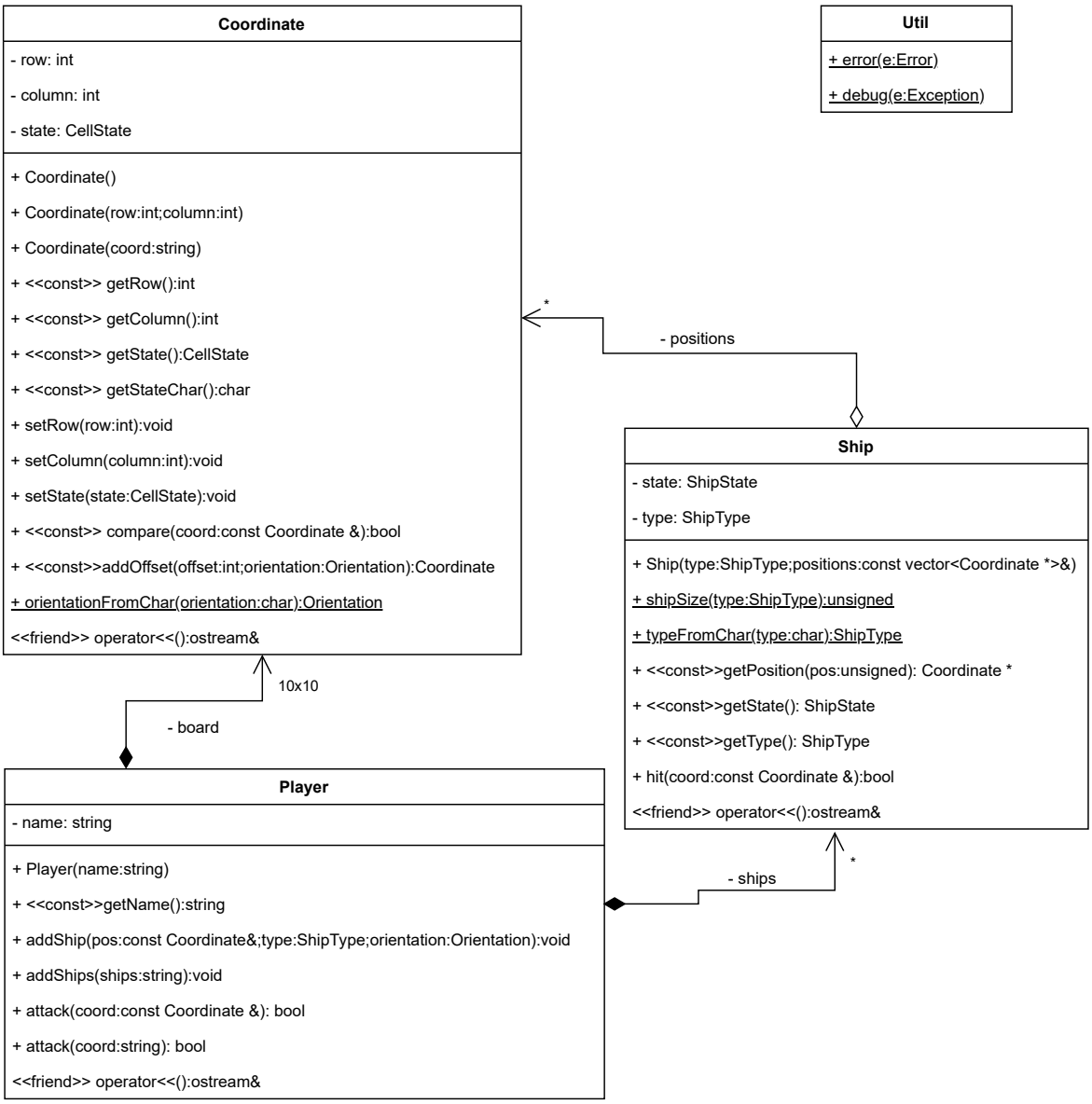
A lo largo de este enunciado, siempre que hablemos de métodos o atributos "privados" nos estaremos refiriendo a aquellos que irán en la parte `protected` de las clases que vas a definir.

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.

En esta práctica no está permitido añadir ningún atributo o método público a las clases definidas en el diagrama, ni añadir o cambiar argumentos de los métodos. 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.

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 el programa, además del método `error` para emitir los correspondientes errores por pantalla. En esta práctica solamente se emiten errores en el fichero `prac3.cc`
- el tipo enumerado `Exception` con todas las posibles excepciones que pueden producirse.
- un método `debug` como ayuda en la depuración de excepciones

- Ten en cuenta que `error` es un método de clase (`static`) y por eso debe invocarse utilizando esta sintaxis: nombre de la clase, seguido de `::` y finalmente el nombre del método. Consulta las transparencias 45 y 46 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`), que normalmente tendrán valores entre 0 y el tamaño del tablero menos uno (p.ej. si el tablero es de tamaño 10, serían entre 0 y 9), pero pueden almacenar cualquier valor **y por tanto no se debe hacer ninguna comprobación en esta clase**. El último atributo es un estado, que será uno de los posibles valores de este tipo enumerado (que debe aparecer en el fichero `Coordinate.h`):

Terminal

```
enum CellState {
  NONE,
  SHIP,
  HIT,
  WATER
};
```

Estos valores representan una posición sin valor (`NONE`), una posición que es parte de un barco (`SHIP`), una posición alcanzada de un barco (`HIT`) y una posición atacada pero que no pertenece a ningún barco (`WATER`).

Los métodos de esta clase son:

- `Coordinate()`. Constructor por defecto, inicializa la fila y la columna a -1 y el estado a `NONE`
- `Coordinate(int row,int column)`. Constructor que inicializa la fila y la columna con los valores pasados como parámetros, y el estado a `NONE`
- `Coordinate(string coord)`. Constructor que inicializa la fila y la columna a partir de la cadena, como en el juego original. El primer carácter de la cadena será una letra mayúscula entre `'A'` y `'Z'` (no es necesario comprobarlo), que representa la fila: la `'A'` sería la primera fila (la fila 0), la `'B'` la fila 1, etc. Los siguientes caracteres serán un número de una o dos cifras que representan la columna, donde `'1'` sería la primera columna (es decir, el atributo `column` valdría 0), `'2'` la segunda columna (`column` valdría 1), etc. El formato será siempre correcto, y no es necesario comprobar que sólo hay dos cifras, es decir, los valores irán desde `'A1'` hasta `'Z99'`.

  Si la cadena es por ejemplo "`"B9"`", esa posición correspondería a la segunda fila y a la penúltima columna (en un tablero de tamaño 10), y por tanto el atributo `row` debe valer 1 y el atributo `column` debe valer 8. Como en los métodos anteriores, el estado se inicializa a `NONE`

- `int getRow() const`, `int getColumn() const`, `CellState getState() const`. *Getters* que devuelven los valores de los atributos correspondientes
- `char getStateChar() const`. Método que devuelve la letra inicial de el estado de la coordenada, es decir, devuelve `'N'` (`NONE`), `'S'` (`SHIP`), `'H'` (`HIT`) o `'W'` (`WATER`)
- `void setRow(int row)`, `void setColumn(int column)`, `void setState(CellState state)`. *Setters* que modifican el valor de los atributos correspondientes (sin realizar ninguna comprobación)
- `bool compare(const Coordinate &coord) const`. Método que compara la coordenada actual (`this`) y `coord`, devolviendo `true` si la fila y la columna son iguales y `false` en caso contrario. El estado de ambas coordenadas no se tiene en cuenta en la comparación, es irrelevante.
- `Coordinate addOffset(int offset,Orientation orientation) const`. Método que devuelve una coordenada que es resultado de sumar a la coordenada (`this`) un número de `offset` posiciones en la orientación (`orientation`) indicada. Esta orientación puede ser uno de los valores del siguiente tipo enumerado (que debe declararse en el fichero `Coordinate.h`):

  Terminal
  ```
  enum Orientation {
    NORTH,
    EAST,
    SOUTH,
    WEST
  };
  ```

  Por ejemplo, si una coordenada tiene la fila 3 y la columna 4 (`"D5"`), si se le suman 2 posiciones en la orientación `NORTH` se obtendría la fila 1 y columna 4 (`"B5"`); si fuera en la orientación `EAST`, se obtendría la fila 3 y la columna 6 (`"D7"`). No es necesario realizar ninguna comprobación sobre la coordenada resultante, y en todos los casos, el estado de la nueva coordenada será `NONE`.
- `Orientation orientationFromChar(char orientation)`. Método estático que devuelve la orientación correspondiente a la indicada como argumento, que se corresponderá con la primera letra de la orientación. Por ejemplo, si el argumento es una 'S' devolverá `SOUTH`. Si el carácter que se pasa como argumento no es la primera letra de una orientación (p.ej. si es un '2', un ':' o una 's' minúscula), se lanzará la excepción `EXCEPTION_WRONG_ORIENTATION`.
- `ostream& operator<<(ostream &os,const Coordinate &coord)`. Operador de salida que muestra la coordenada con el siguiente formato:
  - La fila será una letra entre 'A' y 'Z' (se puede asumir que no va a haber valores mayores y no es necesario comprobarlo)
  - La columna será un valor entre 1 y el tamaño del tablero (tampoco hace falta comprobar nada), y podría tener una o dos cifras
  - El estado se representará con la inicial, excepto si es `NONE`, que no se representa
  - No se debe imprimir un '`\n`' al final (es posible que se escriban varias coordenadas en la misma línea en otro método de la práctica)

  Por ejemplo, una coordenada cuya fila es 3, la columna es 11 y el estado es `WATER` se imprimiría como `"D12W"`. Si la fila o la columna tienen un valor negativo, la cadena que se debe imprimir es `"--"`

3.3. Ship

Esta clase representa los diferentes tipos de barcos que se pueden utilizar en el juego. Los tipos se definen con el siguiente tipo enumerado (que debe aparecer en el fichero `Ship.h`):

Terminal

```
enum ShipType {
  BATTLESHIP,
  DESTROYER,
  CRUISE,
  SUBMARINE
};
```

Cada tipo de barco ocupa el número de posiciones en el tablero que indica la siguiente tabla:

Tipo	Posiciones
BATTLESHIP	4
DESTROYER	3
CRUISE	2
SUBMARINE	1

Cada barco guardará en un vector de punteros las posiciones (coordenadas) del tablero que ocupa, de manera que cuando resulte alcanzada una posición del barco, el cambio de estado en la coordenada alcanzada se reflejará en el tablero automáticamente, usando los punteros.

El barco puede estar en uno de los estados definidos por este tipo enumerado (que debe declararse en `Ship.h`):

Terminal

```
enum ShipState {
  OK,
  DAMAGED,
  SUNK
};
```

Estos valores representan que el barco está bien (`OK`, que es el estado inicial del barco), que el barco ha sido alcanzado en una o más de sus posiciones (`DAMAGED`), y que el barco ha resultado hundido (`SUNK`).

Los métodos de esta clase son:

- `Ship(ShipType type,const vector<Coordinate *> &positions)`. Constructor que crea un barco e inicializa sus posiciones con las del vector que se le pasa como argumento; estas posiciones se corresponderán con coordenadas del tablero para que cualquier cambio en el barco se refleje automáticamente en el tablero, de ahí que se utilicen punteros.

  El estado inicialmente es `OK`, y el tipo de barco será el indicado como argumento. Si el tamaño del vector de posiciones no coincide con el tamaño del tipo de barco indicado, se lanzará la excepción `EXCEPTION_WRONG_COORDINATES`; si el tamaño del vector es correcto, este método cambiará el estado de dichas posiciones del tablero al estado `SHIP` para indicar que están ocupadas por un barco.

- `unsigned shipSize(ShipType type)`. Método estático que devuelve el número de posiciones que ocupa un barco del tipo indicado como argumento, según la tabla que aparece más arriba.

- `ShipType typeFromChar(char type)`. Método estático que devuelve el tipo de barco correspondiente al tipo indicado como argumento, que se corresponderá con la primera letra del tipo. Por ejemplo, si el argumento es una `'S'` devolverá `SUBMARINE`. Si el carácter que se pasa como argumento no es la primera letra de un tipo, p.ej. si es un `'2'`, un `':'` o una `'s'` (minúscula), se lanzará la excepción `EXCEPTION_WRONG_SHIP_TYPE`.

- `Coordinate *getPosition(unsigned pos) const`. Devuelve el puntero a la coordenada situada en la posición `pos` del vector de posiciones (los valores de `pos` irán de 0 en adelante), o bien devolverá `NULL` si dicha posición queda fuera del vector `positions`.

- `ShipType getType() const`, `ShipState getState() const`. *Getters* que devuelven los valores de los atributos.
- `bool hit(const Coordinate &coord)`. Este método comparará la coordenada que se le pasa como argumento con las coordenadas del barco. Puede ocurrir que:
  - La coordenada coincide (usando el método `compare` de la clase `Coordenada`) con una de las del barco, en cuyo caso hay que hacer algunas comprobaciones y cambios, según se explica más abajo
  - La coordenada no coincide con ninguna de las del barco, en cuyo caso el método devolverá `false`

  Si la coordenada pasada como argumento coincide con alguna de las posiciones del barco, puede ocurrir que el barco ya esté hundido, en cuyo caso se lanzará la excepción `EXCEPTION_ALREADY_SUNK` (que se capturará en la clase `Player`); si no está hundido y dicha posición no ha sido atacada previamente (es decir, su estado es `SHIP`), se cambiará el estado de la posición del barco a `HIT` y se actualizará el estado del barco de la siguiente forma:
  - Si el barco estaba `OK` pasará a estar `DAMAGED` (excepto si es un submarino, que pasará directamente a `SUNK`)
  - Si estaba `DAMAGED` y queda alguna posición sin ser alcanzada (con estado `SHIP`), el estado del barco no cambiará, seguirá siendo `DAMAGED`; si todas las posiciones han sido alcanzadas, el estado del barco cambiará a `SUNK`

  Si es una posición previamente atacada, lanzará la excepción `EXCEPTION_ALREADY_HIT`, que debe capturarse en la clase `Player`.

  El método devolverá `false` si la coordenada pasada como parámetro no coincide con ninguna de las posiciones del barco; en caso contrario se entiende que el ataque ha sido un acierto y el método devolverá `true` (si no se ha lanzado ninguna excepción, claro).
- `ostream& operator<<(ostream &os,const Ship &ship)`. Operador de salida que muestra el estado del barco. Por ejemplo, un crucero situado en la esquina superior izquierda que ha sido alcanzado en la segunda posición se imprimiría así:

  ```
  CRUISE (D): A1S A2H
  ```

  donde la primera palabra coincide con el nombre utilizado en el tipo enumerado, la `D` entre paréntesis indica el estado del barco (`DAMAGED`) y después de los dos puntos aparecen las coordenadas del barco y finalmente un `\n`

3.4. Player

Esta clase modeliza a uno de los jugadores, que tendrá un nombre, un tablero (de coordenadas) de tamaño `10x10` y un vector de barcos (inicialmente vacío). El número máximo de barcos de cada tipo que puede haber en el tablero se indica en esta tabla:

Tipo	Máximo
`BATTLESHIP`	1
`DESTROYER`	2
`CRUISE`	3
`SUBMARINE`	4

Los métodos de esta clase son:

- `Player(string name)`. Constructor que inicializa el nombre con el que se pasa como argumento. Puesto que el tamaño del tablero es constante (`10x10`), se puede definir el tablero (`board`) como una matriz cuadrada de `10x10` de coordenadas:

```
const int BOARDSIZE=10;
...
class Player{
...
    Coordinate board[BOARDSIZE][BOARDSIZE];
...
};
```

Puesto que el constructor por defecto de la clase Coordinate pone a -1 la fila y la columna (y el estado a `NONE`), es necesario asignar a cada coordenada el valor adecuado para su fila y su columna (el estado debe dejarse a `NONE`).

- `string getName() const`. *Getter* que devuelve el nombre del jugador
- `void addShip(const Coordinate &pos,ShipType type,Orientation orientation)`. Método que añade un barco al tablero del jugador. Para ello debe hacer lo siguiente:

  1. Comprobar que el jugador no tenga ya en el tablero el máximo número de barcos de ese tipo. En caso de que se intente añadir un tipo de barco cuyo máximo ya se ha alcanzado (p.ej. si se intenta añadir un destructor y el jugador ya tiene 2 destructores), se lanzará la excepción `EXCEPTION_MAX_SHIP_TYPE`
  2. Comprobar que la partida no se ha iniciado ya, es decir, que no hay posiciones del tablero cuyo estado sea `WATER` ni `HIT`; si la partida ya ha empezado no se pueden añadir más barcos y se lanzará la excepción `EXCEPTION_GAME_STARTED`
  3. A partir de los parámetros, construir el vector de punteros a coordenadas necesario para crear el barco, con ayuda del método `addOffset` de la clase `Coordinate` y teniendo en cuenta que todas las coordenadas deben cumplir las siguientes restricciones:
     - Deben estar dentro del tablero (obviamente, esta comprobación debe hacerse *antes* de obtener el puntero a la coordenada del tablero); si no se cumple esta condición se debe lanzar la excepción `EXCEPTION_OUTSIDE`
     - La coordenada del tablero debe estar vacía (con estado `NONE`), en caso contrario se debe lanzar la excepción `EXCEPTION_NONFREE_POSITION`. Por simplificar la práctica, en esta versión del juego se permite que los barcos se coloquen en posiciones adyacentes a otros barcos, no es necesario comprobar nada al respecto

     El número de coordenadas que debe almacenarse en el vector se determinará en función del tipo de barco, según la tabla de la Sección 3.3 (usando el método `Ship::shipSize`). Por ejemplo, para un `DESTROYER` se almacenarán 3 posiciones.
  4. Una vez se ha construido el vector de punteros a las coordenadas que va a ocupar el barco, se puede construir el barco y almacenarlo en el vector de barcos del jugador.

- Para obtener un puntero a una coordenada del tablero habría que hacer algo así:

  ```
  Coordinate *pcoord =  &(board[2][3]);
  ```

  De esa forma se declara un puntero `pcoord` y se le asigna la coordenada `(2,3)` del tablero

- `void addShips(string ships)`. Método que permite añadir uno o más barcos al jugador. La cadena que se pasa como argumento tendrá el formato del siguiente ejemplo:

```
B-B3-E S-J10-N   C-D7-W  D-A1-S
```

- La primera letra de cada subcadena indica el tipo de barco (si es erróneo se lanzará la excepción correspondiente en el método `typeFromChar`, y no se debe capturar)
- Después del primer guión aparece la coordenada
- La orientación aparece después del segundo guión; si fuera incorrecta, se lanzará la excepción correspondiente en el método `orientationFromChar`, que tampoco hay que capturar

- El formato de las subcadenas será siempre correcto, aunque puede contener datos incorrectos (p.ej. `K-Z99-A`); entre cada par de subcadenas habrá uno o más blancos

El método debe ir extrayendo y procesando una a una las subcadenas, añadiendo el barco correspondiente al jugador (usando el método `addShip`, evidentemente).

Si al procesar alguna subcadena se produce una excepción el método terminará por acción de la excepción y dejará sin procesar el resto de la cadena; los barcos que se hayan añadido anteriormente a la subcadena errónea se quedarán en el tablero, no se deben borrar. Si se produce alguna excepción ya se capturará (o no) en el programa principal. Este método no debe capturar excepciones.

- `bool attack(const Coordinate &coord)`. Este método simula el ataque del jugador contrario a una posición del tablero; se debe recorrer el vector de barcos llamando al método `hit` y:
  - Si el método `hit` lanza la excepción `EXCEPTION_ALREADY_SUNK` o la `EXCEPTION_ALREADY_HIT` se debe capturar inmediatamente y devolver `false` (sin terminar de recorrer el vector de barcos y sin modificar el tablero).
  - Si algún barco resulta alcanzado se debe hacer lo siguiente:
    - Si el barco resulta hundido, se debe comprobar si todos los barcos están hundidos, en cuyo caso se debe lanzar la excepción `EXCEPTION_GAME_OVER` (el jugador ha perdido la partida)
    - En caso contrario, se debe devolver `true` inmediatamente, sin terminar de recorrer el vector de barcos (en una posición solo puede haber un barco, luego los barcos restantes no pueden ser alcanzados).
  - Si la posición indicada no coincide con ninguna de las de los barcos, se cambiará el estado de dicha coordenada en el tablero a `WATER` y se devolverá `false`.
- `bool attack(string coord)`. Este método construye una coordenada a partir de la cadena que se pasa como argumento, y llama al método anterior (obviamente, devuelve lo que devuelva el método anterior). Por ejemplo, si la cadena es `"B7"` se crearía la coordenada con el constructor correspondiente (que crearía una coordenada que tendría la fila 1 y la columna 6) y se llamaría al método anterior.
- `ostream& operator<<(ostream &os,const Player &player)`. Operador de salida que muestra el nombre, el tablero y los barcos del jugador, como en el siguiente ejemplo:

Terminal

```
Juan Sin Miedo
   1  2  3  4  5  6  7  8  9  10
A                 W  W
B     S  S  S  H
C                           S  S
D                     W
E  S
F  S
G  S                           H
H
I
J
CRUISE (O): C9S C10S
BATTLESHIP (D): B5H B4S B3S B2S
DESTROYER (O): G1S F1S E1S
SUBMARINE (S): G10H
```

Al mostrar el tablero, cada columna tiene 3 caracteres de los que el primero y el último son blancos, excepto en la primera fila, la que muestra los números de las columnas, en la que para los números con dos cifras (en esta práctica sólo el 10) la segunda cifra ocupa la posición del último blanco. En

cada posición se muestra una letra que representa el estado de la coordenada, excepto si es `NONE`, en cuyo caso se muestra un blanco.

En el ejemplo, el crucero y el destructor tienen una letra O mayúscula (no un cero) porque su estado es `OK`.

4. Programa principal

El programa principal está en el fichero `prac3.cc` publicado en el Moodle de la asignatura. Este fichero contiene código para crear dos jugadores y añadirles barcos, y para simular algunos movimientos de la partida. Es simplemente un ejemplo de cómo se podrían usar las clases que has creado en un programa. Puedes modificarlo como quieras para probar tus clases, no se debe entregar con el resto de ficheros (aunque si lo entregas no pasa nada, se ignorará).