Práctica 3: Batalla de portaaviones

Programación 2

Curso 2024-2025

Esta práctica consiste en simular una batalla entre cazas de portaaviones en una zona de conflicto, 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 9 de mayo**, hasta las **23:59**
- La práctica consta de varios ficheros: `Coordinate.cc`, `Coordinate.h`, `Fighter.cc`, `Fighter.h`, `AircraftCarrier.cc`, `AircraftCarrier.h` `Board.cc` y `Board.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 (en una única línea):

Terminal

```
$ tar cvfz prog2p3.tgz Coordinate.cc Coordinate.h Fighter.cc Fighter.h
          AircraftCarrier.cc AircraftCarrier.h Board.cc Board.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 o pedirle a ChatGPT que te haga la práctica
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
- Al comienzo de todos los ficheros fuente entregados debes incluir un comentario con tu NIF (o equivalente) y tu nombre. Por ejemplo:

Coordinate.h

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

- Superar **todas** las pruebas del autocorrector te da derecho a presentarte al examen de prácticas. Si falla alguna prueba no podrás presentarte al examen y tu calificación en esta práctica será 0
- 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 implementarán varias clases que permitirán simular una batalla entre dos portaaviones, sobre un tablero que representa la zona del conflicto. Los portaaviones enviarán a sus cazas a luchar por cada posición del tablero.

2. Detalles de implementación

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

- `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 una batalla. Este fichero no debe incluirse en la entrega final, es solamente una ayuda para probar la práctica, y se puede modificar o usar como base para otras pruebas
- `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, tras la entrega, se realizará con estas mismas pruebas y será necesario superarlas todas para poder presentarse al examen de esta práctica.

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 en el tablero, `Fighter.h` y `Fighter.cc` para gestionar los cazas, `AircraftCarrier.h` y `AircraftCarrier.cc` para los portaaviones, y `Board.h` y `Board.cc` para el tablero. 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 Fighter.cc AircraftCarrier.cc Board.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 la transparencia 61 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, que son un mecanismo especialmente útil en los constructores porque evitan construir objetos con valores incorrectos.

Para lanzar una excepción debes 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 siempre la excepción estándar `invalid_argument` (que nunca se debe capturar):

Terminal

```
if (...)
 throw invalid_argument(mensaje);
```

donde `mensaje` es una cadena (`string` o vector de `char`); si el argumento inválido es un número, se puede convertir a cadena con la función `to_string`:

Terminal

```
if (...)
 throw invalid_argument(to_string(numero));
```

Para usar la `invalid_argument` debes añadir en tu código `#include <stdexcept>`

2.2. Generación de números aleatorios

Para la simulación de la lucha entre dos cazas es necesario generar un número aleatorio, usando esta función (que debes incluir al principio en el fichero `Fighter.cc`):

Terminal

```
#include <cstdlib> // Para rand()

// devuelve un número aleatorio entre 0 y maxrnd-1
int getRandomNumber(unsigned int maxrnd){
  return rand() % maxrnd;
}
```

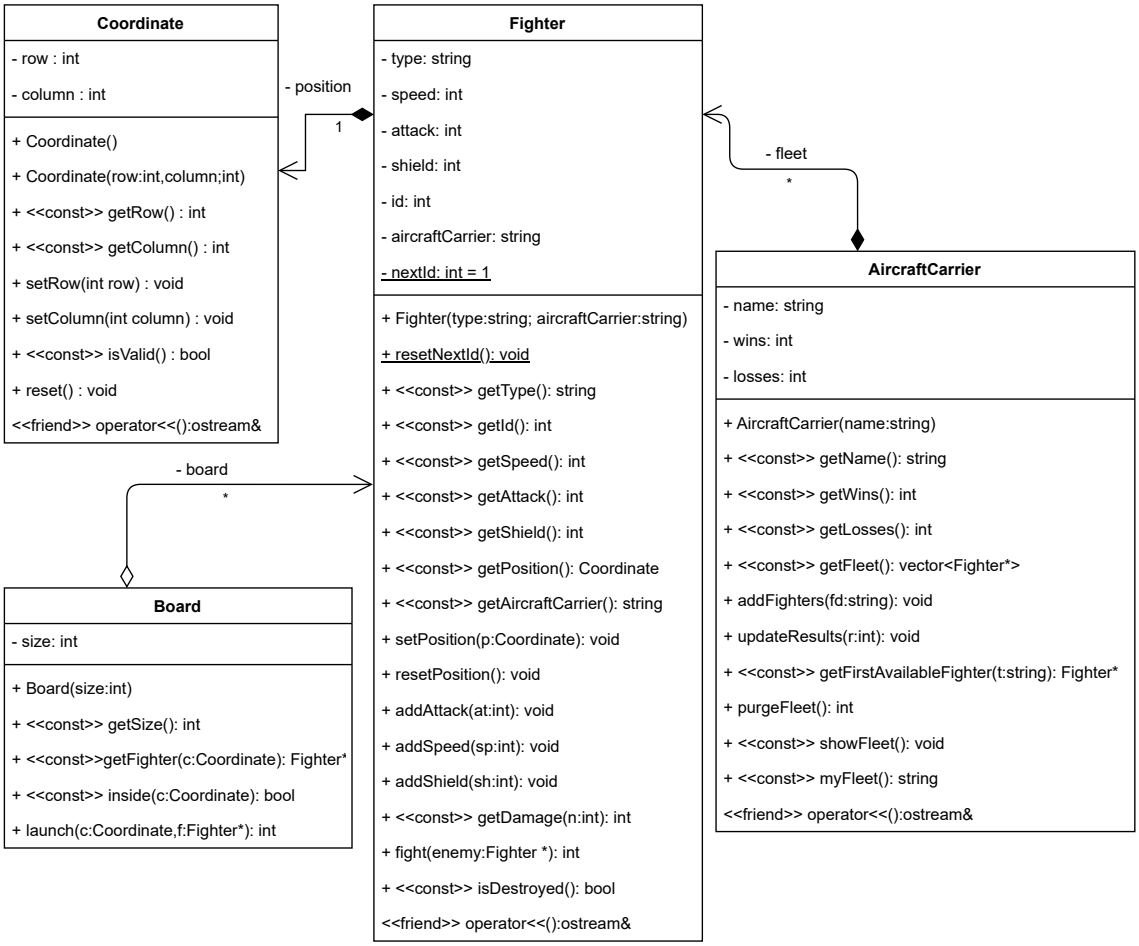
- En el fichero `prac3.cc` se inicializa el generador de números aleatorios llamando a la función `srand`. No debes modificar esa llamada.

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 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. Coordinate

Esta clase gestiona las coordenadas en el tablero. Como se puede ver en el diagrama, tiene dos variables de instancia privadas, "`row`" y "`column`", que representan la fila y la columna en el tablero, respectivamente.

Los métodos de esta clase son los siguientes:

- `Coordinate()`. Crea una coordenada asignando -1 a la fila y la columna. Esta coordenada sería una coordenada no válida.
- `Coordinate(int row,int column)`. Crea una coordenada asignando el valor de `row` a la fila y `column` a la columna.
- `int getRow() const`. *Getter* que devuelve la fila
- `int getColumn() const`. *Getter* que devuelve la columna
- `void setRow(int row)`. *Setter* que modifica la fila
- `void setColumn(int column)`. *Setter* que modifica la columna
- `bool isValid() const`. Devuelve `true` si la fila y la columna son mayores o iguales que 0, y `false` en otro caso.
- `void reset()`. Pone ambos atributos a -1, como si fuera una coordenada recién creada con el constructor sin argumentos.
- `ostream& operator<<(ostream &os,const Coordinate &c)`. Operador salida que muestra la coordenada. Por ejemplo, si la fila es 2 y la columna 7 mostrará `[2,7]`. Si la coordenada no es válida mostrará `[-,-]`. No se debe añadir salto de línea después de imprimir esta información por pantalla.

3.2. Fighter

Esta clase refleja el comportamiento de los cazas, y tiene varios atributos (todos ellos privados):

- `type`: una cadena (`string`) que indica el tipo de caza: `F-35B`, `EuroFighter`, ...
- `speed`: velocidad del caza (no debe ser negativa)
- `attack`: capacidad de ataque del caza (no debe ser negativa)
- `shield`: escudo del caza, si llega a ser 0 o negativo se considerará que el caza está destruido
- `id`: identificador, único para cada caza. Cuando se crea un nuevo caza, se le asigna como identificador el valor de `nextId` y se incrementa `nextId`; como se puede ver en el diagrama, inicialmente `nextId` vale 1
- `position`: coordenada que representa la posición del caza en el tablero; inicialmente será una coordenada no válida (la fila y columna valdrán -1), y se le asignará valor cuando se coloque el caza en el tablero; cuando el caza resulte destruido y se borre del tablero, debe volver a ser una coordenada no válida
- `aircraftCarrier`: nombre del portaaviones al que pertenece el caza

En el diagrama UML aparecen los métodos de esta clase; muchos de ellos son *getters* simples, que devuelven los campos a los que hacen mención. Por ejemplo, `getType` devuelve el atributo `type` del objeto. Los métodos `addAttack`, `addSpeed` y `addShield` actúan de manera similar a los *setters*, pero sumando el valor recibido como argumento al atributo (como el nombre del método indica). Además, en caso de que la velocidad (`speed`) o la capacidad de ataque (`attack`) resulten negativas después de la suma (el argumento podría ser negativo), deben ponerse a `0`.

- `Fighter(string type, string aircraftCarrier)`. Constructor que inicializa un caza, asignando un valor de 100 a la velocidad, 80 al ataque y 80 al escudo. Además, asigna los parámetros a los atributos `type` y `aircraftCarrier`, y asigna valor al identificador (y se incrementa `nextId`). La coordenada se inicializará automáticamente al valor por defecto (`-1,-1`), no es necesario que hagas nada más. Si `type` es una cadena vacía se debe lanzar la excepción `invalid_argument` con la cadena `"Wrong type"`
- `void resetNextId()`. Método estático que inicializa el valor del atributo estático `nextId` a 1. No se debe utilizar en la práctica, pero para realizar pruebas unitarias es necesario
- `void resetPosition()`. Llama al método `reset` para la posición del caza
- `int getDamage(int n) const`. Devuelve el daño infligido por el caza al caza enemigo, que será siempre `(n*attack)/300`. Como todos los números son enteros, la división debe ser entera y el resultado también.
- `int fight(Fighter *enemy)`. Este método simula la lucha entre dos cazas, el que recibe la llamada (el *caza*) y un caza enemigo (`enemy`, el caza *enemigo*). Antes de empezar, si alguno de los dos cazas ha sido destruido, devuelve 0; en caso contrario, inicia un bucle que terminará cuando uno de los dos cazas resulte destruido (es una lucha a muerte). En cada paso del bucle uno de los cazas atacará al otro; para ello se obtendrá un número aleatorio `n` entre 0 y 99 (llamando a la función `getRandomNumber`), y se comparará dicho valor con un umbral u, que se calcula con la siguiente fórmula:

  $$u = \frac{100 v_1}{v_1 + v_2}$$

  donde v_1 es la velocidad del *caza*, y v_2 la del *enemigo*. Como todos los valores son enteros, la división será entera (no uses números reales).

  Si el umbral u es mayor o igual que el número aleatorio `n`,[1] el atacante será el *caza* y se restará al escudo del *enemigo* el daño causado por el caza (que será el valor devuelto por el método `getDamage` pasando `n` como parámetro); en caso contrario, el atacante será el *enemigo* y se restará al escudo del *caza* el daño causado por el *enemigo* (llamando también a `getDamage` del enemigo, pero en este caso pasando `100-n` como parámetro).

  Si después de un ataque un caza resulta destruido, la lucha terminará y el método devolverá 1 si ha ganado el *caza* o -1 si ha ganado el *enemigo*; si ninguno de los cazas ha sido destruido, se seguirá dentro del bucle y se volverá a obtener otro número aleatorio.

  Por ejemplo, si $u = 50$ (ambos cazas tienen la misma velocidad) y el primer número aleatorio n es 80, se le restará 5 al escudo del caza (`this`); si el siguiente número es 88, se le restará 3 nuevamente al caza, y si el siguiente es 13 se le restará 3 al escudo del caza enemigo (`enemy`). El bucle seguirá hasta que uno de los cazas resulte destruido.

  IMPORTANTE: asegúrate de que no llamas a `getRandomNumber` más de una vez dentro del bucle, si haces más llamadas el resultado de esta lucha (y las siguientes) será probablemente diferente y tu práctica funcionará mal.
- `bool isDestroyed() const`. Devuelve `true` si el atributo `shield` es 0 o menor, y `false` en otro caso
- `ostream& operator<<(ostream &os,const Fighter &f)`. Operador de salida que muestra los datos del caza. Por ejemplo, si un `F-35B` colocado en la posición (2,3) tiene como identificador 27, su velocidad es 100, su ataque es 80 y le queda un valor de 36 en su escudo, lo que se debe mostrar sería:

  Terminal
  ```
  (F-35B 27 [2,3] {100,80,36})
  ```

  Si no tuviera posición asignada, simplemente saldría `[-,-]` en vez de `[2,3]` (el operador de salida de `Coordinate` se encargará de eso). No se debe añadir salto de línea al final.

[1] Con esta fórmula, el caza más rápido tendrá un valor mayor de u y por tanto tendrá mayor probabilidad de atacar.

3.3. AircraftCarrier

Esta clase permite gestionar portaaviones, que tendrán varios atributos (privados):

- `name`: nombre del portaaviones
- `wins`: victorias obtenidas por los cazas del portaaviones
- `losses`: derrotas de los cazas del portaaviones
- `fleet`: flota de cazas del portaaviones (debes usar un `vector<Fighter *>` para almacenar los cazas)

Aparte de los getters triviales, los métodos de esta clase son:

- `AircraftCarrier(string name)`. Constructor que inicializa los datos del portaaviones (obviamente los atributos *wins* y *losses* deben inicializarse a 0). Si el nombre es una cadena vacía se debe lanzar la excepción `invalid_argument` con la cadena `"Wrong name"`
- `vector<Fighter *> getFleet() const`. Devuelve la flota de cazas del portaaviones, se usará solamente en las pruebas unitarias
- `void addFighters(string fd)`. A partir de una cadena como `"5/F-35B:12/F-18:3/A6:2/F-35B"`, debe construir los cazas indicados y añadirlos a la flota de cazas del portaaviones. El número de cazas aparecerá al principio, separado del tipo por "/", y si hay más de un tipo de caza se separarán con ":". En la cadena de ejemplo, se crearían 5 `F-35B`, 12 `F-18`, 3 `A6` y otros 2 `F-35B`. Se puede asumir que la cadena no tendrá errores, y por tanto no es necesario comprobarlo. Si la cadena es vacía, el método no hará nada.
- `void updateResults(int r)`. Debe actualizar los valores de *wins* o *losses* en función del valor del argumento *r*; si vale 1, se incrementará *wins*, si vale -1 se incrementará *losses*. Si *r* tiene cualquier otro valor, el método no hará nada.
- `Fighter *getFirstAvailableFighter(string type) const`. Devuelve el primer caza (no destruido y no posicionado en el tablero) de la flota del tipo indicado por el argumento *type*; si *type* es una cadena vacía, devuelve el primer caza (no destruido y no posicionado) de cualquier tipo. Si no hay cazas no destruidos o directamente no hay cazas, devuelve `NULL` (o `nullptr`).
- `int purgeFleet()`. Borra de la flota los cazas destruidos, y devuelve el número de cazas borrados.
- `void showFleet() const`. Muestra la flota completa por la salida estándar, mostrando en cada línea un caza de la flota; si el caza ha sido destruido, al final de la línea se añadirá la cadena `"(X)"` para indicar que se ha destruido, como en este ejemplo:

  Terminal

  ```
  (F-35B 24 [-,-] {100,80,-24}) (X)
  (F-35B 25 [1,3] {100,80,80})
  (F-35B 26 [-,-] {100,80,80})
  (F-35B 27 [-,-] {100,80,80})
  (F-35B 28 [-,-] {100,80,80})
  ```

  Se deben mostrar todos los cazas de la flota, incluyendo los destruidos y los posicionados; si la flota no tiene cazas, no hará nada
- `string myFleet() const`. Devuelve una cadena con el formato que admite el método `addFighters` con los cazas (destruidos o no) de la flota. Por ejemplo, si a un portaaviones tiene 7 `F-35B` y 1 `Super Hornet`, devolvería la cadena `"7/F-35B:1/Super Hornet"` (o la cadena `"1/Super Hornet:7/F-35B"`, según si el primer caza es un `F-35B` o un `Super Hornet`). Si la flota no tiene cazas, devuelve una cadena vacía. Debe tenerse en cuenta que deben mostrarse los cazas en el orden que tienen en `fleet` y que deben acumularse todos los cazas del mismo tipo, si por ejemplo un portaaviones tiene en su flota 3 `F-35B`, 2 `F-18`, 4 `F-35B`, 5 `Super Hornet` y 2 `F-35B`, la cadena a devolver debe ser `"9/F-35B:2/F-18:5/Super Hornet"` porque el primer tipo de caza que aparece en la flota es el `F-35B`, luego el `F-18` y finalmente el `Super Hornet`

- `ostream& operator<<(ostream &os,const AircraftCarrier &a)`. Operador de salida que muestra los datos del portaaviones. Por ejemplo, si un portaaviones llamado USS Abraham Lincoln ha conseguido 35 victorias y ha tenido 10 derrotas, y tiene una flota de 12 `F-35B` y 7 `F-18`, se mostraría lo siguiente, sin añadir salto de línea al final:

Terminal

```
Aircraft Carrier [USS Abraham Lincoln 35/10] 12/F-35B:7/F-18
```

3.4. Board

Esta clase representa el tablero cuadrado en el que se desarrollará el juego, y tiene dos atributos:

- `size`: tamaño del tablero (debe ser siempre positivo). Las coordenadas dentro del tablero irán de 0 a *size-1*
- `board`: matriz cuadrada de punteros a cazas para almacenar los cazas en posiciones del tablero. . Debes declararla como:

```
vector<vector<Fighter *>> board;
```

Los métodos de esta clase son:

- `Board(int size)`.Constructor que inicializa los datos del tablero. Si el valor de `size` no es mayor que `0` debe lanzarse la excepción `invalid_argument` con la cadena `"Wrong size"`; si es correcto, se debe crear la matriz cuadrada de tamaño `size` e inicializar todos los punteros a `NULL` o `nullptr` (más tarde se almacenarán cazas).
- `Fighter *getFighter(Coordinate c) const`. *Getter* que devuelve el contenido del tablero en la posición indicada, que será el caza que la ocupa, o `NULL` (`nullptr`) si no hubiera nada.
- `bool inside(Coordinate c) const`. Devuelve `true` si la coordenada está dentro del tablero, y `false` en otro caso. Las coordenadas están dentro del tablero si las componentes de la coordenada están entre 0 y *size-1* (ambos valores incluidos)
- `int launch(Coordinate c,Fighter *f)`. Intenta colocar un caza en una posición del tablero (si no está ya posicionado). Obviamente, si la coordenada no está dentro del tablero no hará nada, como tampoco hará nada si la posición está ocupada por otro caza del mismo portaaviones. Sin embargo, si la posición está ocupada por un caza de otro portaaviones (un caza enemigo), el nuevo caza atacará y peleará con el que ocupaba dicha posición. Al acabar la lucha (en la que siempre gana uno de los cazas y el otro resulta destruido), se quedará en la posición el caza ganador, y el caza perdedor reseteará su posición. Finalmente, si la posición estaba vacía, el caza se colocará en dicha posición (actualizando también la posición del caza con el *setter* correspondiente).

  Por ejemplo, si tenemos en la posición `[2,3]` del tablero un `SF-1` y queremos lanzar un caza atacante `F-35B` a esa posición, el `F-35B` peleará con el `SF-1`, y pueden ocurrir dos cosas:

  - si gana el `F-35B` se le asignará esa posición del tablero (y se actualizará su posición), y al `SF-1` se le reseteará la posición.
  - si gana el `SF-1` ninguno de los cazas cambiará de posición (el `F-35B` no debe tener posición para ser lanzado, se queda igual)

  El método debe devolver el resultado de la lucha con el caza enemigo: un 1 si gana el caza atacante o un -1 si gana el enemigo (el que estaba en el tablero). En cualquier otro caso (ya estaba en el tablero, o no hay caza, o es amigo, o la coordenada no está en el tablero) el método devolverá 0.

  Finalmente, si el método devuelve el resultado de la lucha ese valor debe usarse en otra parte del código (p.ej. en el prac3.cc) para actualizar las estadísticas de victorias/derrotas de los portaaviones de ambos cazas.

4. Programa principal

El programa principal está en el fichero `prac3.cc` publicado en el Moodle de la asignatura. 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.

Este fichero contiene código para crear varios portaaviones y cazas, y para simular una batalla en el tablero:

```
#include <iostream>
#include "AircraftCarrier.h"
#include "Board.h"

using namespace std;

int main(){
     srand(888);

     Board board(5);  // 5x5

     AircraftCarrier one("USS Acme");
     one.addFighters("3/F-35B:2/F-18:3/A6");

     AircraftCarrier two("Spectra One");
     two.addFighters("4/SF-1:3/SB-3:2/SI-6B");

     // position some fighters on the board
     Coordinate c1(2,3);
     Fighter *f1 = one.getFirstAvailableFighter("F-18");
     board.launch(c1,f1);

     Coordinate c2(3,2);
     Fighter *f2 = two.getFirstAvailableFighter("");
     board.launch(c2,f2);

     cout << "After launching one fighter of each aircraft:" << endl;
     cout << one << endl;
     one.showFleet();
     cout << two << endl;
     two.showFleet();

     Fighter *f3 = one.getFirstAvailableFighter("");
     int result = board.launch(c2,f3);
     if (result != 0)
     {
       one.updateResults(result);
       two.updateResults(-result);
     }
     cout << "After a fight between two fighters:" << endl;     cout << one << endl;
     one.showFleet();
     cout << two << endl;
     two.showFleet();


    return 0;
}
```

La salida sería:

```
After launching one fighter of each aircraft:
Aircraft Carrier [USS Acme 0/0] 3/F-35B:2/F-18:3/A6
(F-35B 1 [-,-] {100,80,80})
(F-35B 2 [-,-] {100,80,80})
```

```
(F-35B 3 [-,-] {100,80,80})
(F-18 4 [2,3] {100,80,80})
(F-18 5 [-,-] {100,80,80})
(A6 6 [-,-] {100,80,80})
(A6 7 [-,-] {100,80,80})
(A6 8 [-,-] {100,80,80})

Aircraft Carrier [Spectra One 0/0] 4/SF-1:3/SB-3:2/SI-6B
(SF-1 9 [3,2] {100,80,80})
(SF-1 10 [-,-] {100,80,80})
(SF-1 11 [-,-] {100,80,80})
(SF-1 12 [-,-] {100,80,80})
(SB-3 13 [-,-] {100,80,80})
(SB-3 14 [-,-] {100,80,80})
(SB-3 15 [-,-] {100,80,80})
(SI-6B 16 [-,-] {100,80,80})
(SI-6B 17 [-,-] {100,80,80})

After a fight between two fighters:
Aircraft Carrier [USS Acme 1/0] 3/F-35B:2/F-18:3/A6
(F-35B 1 [3,2] {100,80,4})
(F-35B 2 [-,-] {100,80,80})
(F-35B 3 [-,-] {100,80,80})
(F-18 4 [2,3] {100,80,80})
(F-18 5 [-,-] {100,80,80})
(A6 6 [-,-] {100,80,80})
(A6 7 [-,-] {100,80,80})
(A6 8 [-,-] {100,80,80})

Aircraft Carrier [Spectra One 0/1] 4/SF-1:3/SB-3:2/SI-6B
(SF-1 9 [-,-] {100,80,-3}) (X)
(SF-1 10 [-,-] {100,80,80})
(SF-1 11 [-,-] {100,80,80})
(SF-1 12 [-,-] {100,80,80})
(SB-3 13 [-,-] {100,80,80})
(SB-3 14 [-,-] {100,80,80})
(SB-3 15 [-,-] {100,80,80})
(SI-6B 16 [-,-] {100,80,80})
(SI-6B 17 [-,-] {100,80,80})
```