**Battleship Game**

Daniil Kruklinskii

*10453135*

The University of Manchester

OOP in C++ Project

May 2022

# Abstract

This program is an alternative version of the famous Battleship board game. The code is written in C++ programming language using the Object-Oriented Programming paradigm. No errors or unexpected behaviour was observed in the latest version of the program. The game is entertaining, and one may enjoy playing it with another person.

# Introduction and Rules of the Game

The aim of the program is to replicate the well-known Battleship game using a C++ programming language with an Object-Oriented Programming (OOP) approach. The "old-school" version of the game does not provide an opportunity for a wide variety of OOP features to be utilised since it contains trivial objects such as ships, which only differ by size and only one weapon with an impact area of one square. Hence, this program provides an alternative version of the game, which contains different weapons and new game mechanics. The game is played on one computer by two players (people).

The rules of the game in the program are very similar to the original. Each player has a board with two grids (10 by 10): one is used to store the player's ships and receive the opponent's "shots", and the other is used to attack the opponent's ships. Firstly, before the game starts, players must position their five ships (2 two-square, 2 three-square and 1 five-square ship) on their grids. The bulk of the game happens after the ships are placed. The players alternate turns and use their weapons to destroy the opponent's ships by guessing their locations. The game ends when any player loses all his ships, and then the other player is declared a winner.

Apart from the standard weapon, called "canon" in the code, there are two new types: bomb and torpedo. Canon has a blast area of one square and can be used again in the same turn if the previous canon shot is successful. The bomb and the torpedo have blast areas of four and ten (entire row or column), respectively. The new "energy" mechanic is implemented to use bombs and torpedoes. After each turn and after the opponent hits a ship, the player (whose ship is hit) gains two energy points, which can be spent on a bomb (5 energy points required) or/and torpedo (9 energy points required). These weapons can be used multiple times per turn as long as there is enough energy. This makes the game more interesting and provides a good "comeback" opportunity.

# Code Design and Implementation

When a code is becoming too large, it is considered good practice to split it into different files to improve readability. This program contains about 1000 lines of code, so it is split into three files: *battleship.cpp*, *battleship.h* and *battleship\_game.cpp*. The *battleship.cpp* contains all function definitions, *battleship.h* contains class and other custom data types definitions and function declarations, and *battleship\_game.cpp* contains the main() function, which is used to run and play the game. #pragmaonceis used in the header file to prevent name clashes.

enum flags {hit = 'x', empty = ' ', missed = 'o', h\_ship = '-', v\_ship = '|'};

...

enum orientation {Horizontal, Vertical};

Code 1: Battleship.h line 20, 26. flags and orientation definitions.

For more convenience, custom enum data types indicate the limited possibilities for board flags and ship orientations (Code 1). For such custom data types, operator >> and operator << are overloaded so that the user can input orientation and output flags correctly.

This program contains multiple classes. The board class encapsulates the two grids mentioned in the introduction and their dimensions. It also contains different methods such as constructor and destructor. The print\_board() method is implemented using the friend function, which allows it to be defined outside the main body of the class but to still use private data. In addition, it will enable the board to be passed as a parameter, which is more intuitive.

for (auto shape\_it = shape.begin(); shape\_it < shape.end(); ++shape\_it) {

...

}

Code 2: Battleship.cpp line 284. Iterator example.

Ship class is an abstract base class that is never instantiated. All its functions are virtual, which allows them to be overridden in derived classes. Moreover, it allows for polymorphism to occur. Since this class is inherited by three other classes (small\_ship, medium\_ship and large\_ship), its variables have protected access type, meaning that they are also inherited. This class contains a variable shape, which utilises std::pair<> data type. This data type is perfect in this case since each shape vector element must contain only two objects of different types. Many functions in this program use iterators, and change\_flag() is one of them. Code 2 shows how each element of the shape vector is accessed using an iterator. It is a safe and robust way to iterate through the array-like objects.

virtual void use\_ability(...)=0;

Code 3: Battleship.h line 114. Pure virtual function.

Next is the ability class, an abstract base class that is never instantiated similarly to the ship class. However, the ability class contains a pure virtual function use\_ability() (Code 3). It is useful since every derived class (canon, bomb and torpedo) has a different use\_ability() function.

std::vector<std::shared\_ptr<ship>> ships;

std::vector<std::unique\_ptr<ability>> abilities;

Code 4: Battleship.h line 180, 181. Smart pointers.

Last is the player class, containing a vector of shared pointers to store ships and a vector of unique pointers to store abilities as private members (Code 4). Smart pointers are a modern and robust way of heap memory allocation. Unlike the new keyword, smart pointers do not require a delete() analogue, making them safer and more convenient. The reason that a shared pointer is used for ships is the return\_ships() function. It returns a reference to ships, which creates another pointer preventing the use of a unique pointer.

player(...) constructor demonstrates polymorphism. Even though ships and abilities vectors should contain ship and ability base classes, respectively, it is possible to put derived classes such as small\_ship and canon with their unique constructors and other attributes, which is very powerful.

The player's is\_valid\_ship\_location() checks whether the input location is valid for the newly placed ship. Many errors may occur, such as one ship overlapping with the existing one. To handle all these errors, try-catch construction is used. It provides an easy and convenient way to deal with different errors individually. Also, this function utilises std::count(), which is a convenient and fast way to count identical objects and is certainly better than writing own algorithm.

return !string.empty() && std::find\_if(string.begin(), string.end(),

[](unsigned char character) { return !std::isdigit(character); }) == string.end();

Code 5: Battleship.cpp line 25. Lambda function.

Location is a very important variable in this code since it is constantly input by the user, who can type something that the program may not expect. Therefore, the input must be checked thoroughly. One coordinate must be an integer, which is checked by the is\_number(...) function. Each character in the string must be checked, which is handled by std::find\_if(). It needs a function to be passed as the last parameter, and since this function is quite simple, the lambda function is used (Code 5).

# Results

The user communicates with the program via the command line terminal. There are many potential inputs and outputs that are important to demonstrate the functionality of this program, but in this section, only the most obvious will be demonstrated.

Graphical user interface, text, application

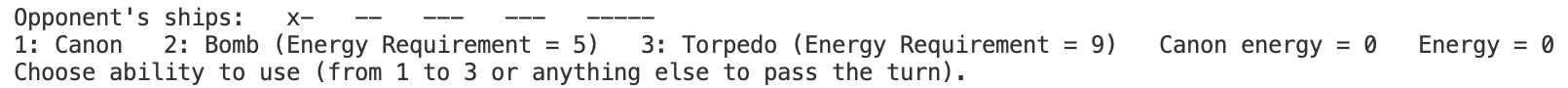
Description automatically generatedText

Description automatically generated

Output 1: This tests whether a user can choose an invalid location or place ships incorrectly.

Output 1 demonstrates what the program does if the user inputs a location in the wrong format or outside the grid (top image). The location is handled correctly. The first ship occupies squares 2 J and 3 J. Therefore, the incorrectly placed ship is also handled correctly (bottom image).

A picture containing clock

Description automatically generated

Output 2: This tests whether a canon shot correctly changes the board.

Output 2 demonstrates that the canon ability is used correctly. Firstly 1 A was chosen for the shot, and then C 1. 1 A square indeed had a ship, and C 1 did not. Therefore, grid output is accurate, and the canon energy is updated correctly.

# Conclusion

The code serves its purpose, and I had an entertaining time playing this game with my friends. Multiple games showed that the output was consistently correct. The user output can be improved to make the ships more visible and possibly make the squares bigger. The extension possibilities are limitless. For instance, different maps and weapons can be added, command line terminal can be abandoned in favour of more visually appealing alternatives.