Code review – Exercise 1 (better name?)

While reviewing all of your codes in exercise 1, we have encountered many code errors, misuses and design issues so we thought it was best to present some for you all to learn from.
The purpose of this document is to enrich you with practical knowledge regarding C++ best practices, and to help you avoid these kinds of errors in the future.
All the code snippets here are taken directly from your work, so don't be surprised if they look somewhat familiar.
After some deliberation, we have decided **NOT** to deduce points on these errors in this exercise, but you must fix similar errors towards exercise 2's submission. In exercise 2, we will pay more attention to errors presented here in this document, and deduce points accordingly.

Do's and Don'ts

Try putting all global function either in classes or a special utilities class:

For example:

```
char** createBoard(int rows, int cols)
{
	char** board = new char*[rows];
	for (int i = 0; i < 10; i++)
	{
		board[i] = new char[cols];
	}
	return board;
}
```

Also in this case, do not use hard coded parameters – especially when you already have them in the scope.

Do not use `new` with no `delete` within a method scope:

```
	BattleshipGameAlgo* playerA = new BattleshipGameAlgo("BPMD");
	BattleshipGameAlgo* playerB = new BattleshipGameAlgo("bpmd");
```

Must not include .cpp files - Instead, always include only .h files

Bad example:

```
#include "Player.cpp"
```

Using `new` with no `delete` with a class member:

```
	Board(int rows, int cols) {
		numOfRows = rows;
		numOfCols = cols;
		board = new char*[numOfRows];
		for (int i = 0; i < numOfCols; ++i) {
			board[i] = new char[numOfRows];
		}
		//init to clear board
		for (int i = 0; i < numOfRows; ++i) {
			for (int j = 0; j < numOfCols; ++j) {
				board[j][i] = ' ';
			}
		}
	}
	//destructor
	~Board() {
	/*	for (int i = 0; i < numOfRows; ++i) {
			delete[] board[i];
		}
		delete[] board;	*/
	}
```

Will result in memory leaks

No public on inheritance = default is private (no polymorphism)

```
class BasicBattleshipGameAlgo : IBattleshipGameAlgo {}
```

Usage:

```
BasicBattleshipGameAlgo player1, player2;
```

Should be instead:

```
class BasicBattleshipGameAlgo : public IBattleshipGameAlgo {
```

Usage:

```
IBattleshipGameAlgo *player1, *player2;
```

Do not have and excess unused data members

```
class BasicBattleshipGameAlgo : IBattleshipGameAlgo
private:
	char** _board; //unused
```

Do not use C-style string handeling:

```
std::string attackFilename = "some_file_path"
char* tempchar = new char[attackFilename.length() + 1];
strcpy_s(tempchar, attackFilename.length() + 1, attackFilename.c_str());
player.attackList = getAttacksListFromFile(tempchar, player.name);
delete[] tempchar;
```

One can simply use string methods:

```
player.attackList = getAttacksListFromFile(attackFilename, player.name)
```

Don't - Variables that are either on the stack or on the heap:

```
        string str = "";
        string* path=&str;
        string* path1 = &str;
        string* path2 = &str;

                if(endsWith(temp,".sboard") && input_errors[1]!=0)
                {
                        path = (new string(temp));
                }
                if(endsWith(temp,".attack-a") && input_errors[2]!=0)
                {
                         path1 = (new string(temp));
                }
                if(endsWith(temp,".attack-b") && input_errors[3]!=0)
                {
                        path2 = (new string(temp));
                }

        delete path; //can try and delete a non heap allocated variable
        delete path1;
        delete path2;
```

In this case, you might try to delete a stack variable and get an exception.

In general, there is no use of using new/delete with local variables. A simple use of the class string here would have been sufficient:

```
string path;
...
path = temp;
```

Use the trinary operator:

This:

```
scoreB += (result == AttackResult::Sink) ? getPointsOfTool(square) : 0;
```

Instead of this:

```
if (result == AttackResult::Sink && !selfAttack)
{
        scoreB += getPointsOfTool(square);
}
else
{
        scoreB += 0;
}
```

No need for if-else for Boolean expression:

```
bool isPlayerOne(char* board[10], int x, int y)
{
	char c = board[x][y];
	if (c == 'B' || c == 'P' || c == 'M' || c == 'D')
		return true;
	else return false;
}
```

Instead - just use return:

```
bool Board::isPlayerOne(int x, int y)
{
	char c = _board[x][y]; // _board is data member
	return (c == 'B' || c == 'P' || c == 'M' || c == 'D');
}
```

Or even better:

```
class Board
{
	const char* _board[2];
	const static set<char> _shipChars[2];
public:
	Board() : _board{ "Pp34", "1D34" }
	{}
	char getChar(int x, int y)
	{
		return _board[y][x];
	}
	bool isPlayer(int x, int y, int playerIndex)
	{
		return _shipChars[playerIndex].find(getChar(x, y)) !=
				_shipChars[playerIndex].end();
	}
};

const std::set<char> Board::_shipChars[] = {
	{ 'B', 'P', 'M', 'D' },
	{ 'b', 'p', 'm', 'd' }
};
```

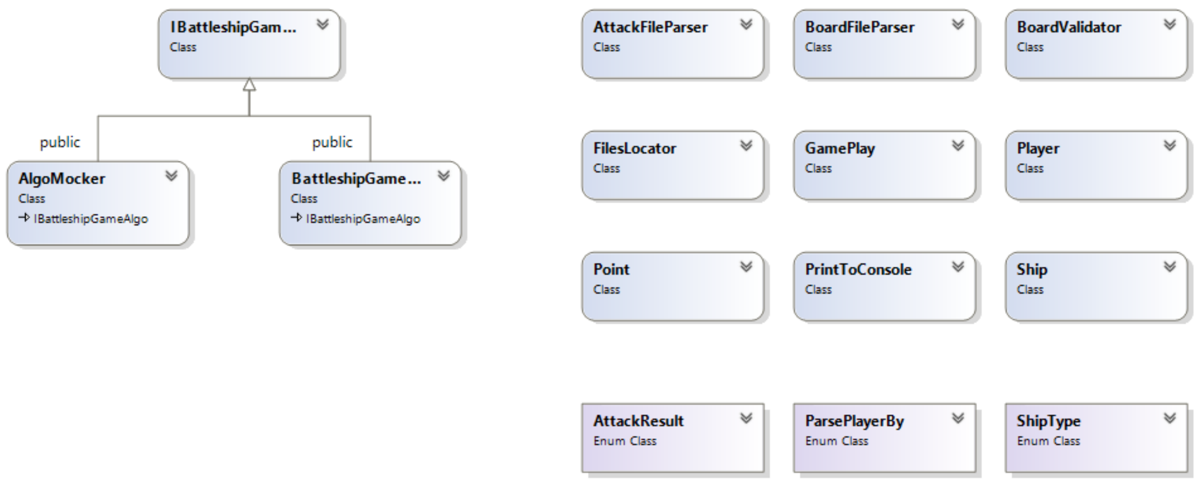
Code Design:

Some students wrote the entire code in just 1 file. Use polymorphism, inheritance to divide your code into classes that talk to each other. Each class should have his own .h and .cpp files.

Bad example – entire game in a single class:

```
class BattleGame
{
        BattleshipGameAlgo playerA, playerB;
        // ...
        // All Methonds
        // ...
        // All Members
};
```

Good Example:



Using uneccesary memory allocations:

```
class Player
{
public:
        queue<pair<int, int>> * attackList;

        explicit Player(char nm) : attackList(nullptr) {}

        ~Player()
        {
                if (attackList != nullptr)
                        delete attackList;
        }
}
```

Instead you could just use the non-pointer version – so you benefit twice:
You do not have to worry about memory leaks, and you don't have to use the (*, ->) syntax on your member.

Not using const on a class method whenever needed:

```
/*return a new raw board*/
/*@post: the returned board is dynamically allocated and must be freed*/
char** GameBoard::getBoard()
{
	char** board = new char*[_rows];

	for (int row = 0; row < _rows; row++)
	{
		board[row] = new char[_cols];
		for (int col = 0; col < _cols; col++)
		{
			board[row][col] = _board[row][col];
		}
	}

	return board;
}
```

This class method does not change anything in the class, hence should be `const`

Or even more classic:

```
bool getIsFailed() {return isFailed;}
```

should be:

```
bool getIsFailed() const {return isFailed;}
```

Not using static whenever needed:

```
int GameBoard::getShipLength(char piece) //should declare static modifier in header
{
	int size = 4;
	char shipTypes[] = { RUBBER, MISSILE, SUB, DESTROYER };
	int shipLengths[] = { RUBBER_LEN, MISSILE_LEN, SUB_LEN, DESTROYER_LEN };

	for (int i = 0; i < size; i++)
	{
		auto shipType = shipTypes[i];
		if (playerShipType(PLAYER_A, shipType) == piece ||
			playerShipType(PLAYER_B, shipType) == piece)
		{
			return shipLengths[i];
		}
	}
	return 0;
}
```

No specific data from a class instance is being used – so we made the method static.

Not using const on a method argument and not using reference:

Bad example:

```
GameBoard GameBoard::operator=(GameBoard other)
{
	setBoard(other._board, other._rows, other._cols);
	return *this;
};
```

Should be instead:

```
GameBoard& GameBoard::operator=(const GameBoard& other)
{
	setBoard(other._board, other._rows, other._cols);
	return *this;
};
```

- `const` – **meaning the method cannot change** `other`
- `GameBoard& other` – **meaning the value is passed by reference, and not copying the entire** `GameBoard object`.
- `GameBoard& (return value)` – **meaning we return a reference to** `this` **instead of returning a copy.**

Compilation warnings:

You must avoid all compilation warning!

Most common are:

1) **'initializing': conversion from 'size_t' to 'int', possible loss of data:**

   ```
   std::string line;
   ...
   for (int col = 0; col < line.length(); ++col)
   ```

   line.length() – size_t

   col – int

2) **'e': unreferenced local variable**

   ```
   catch (std::exception& e) {}
   ```
   Remove unused variables before compilation.

Keeping the file open throughout the game:

Bad example:

```
class BattleshipGameAlgo :IBattleshipGameAlgo
{
public:
ifstream attackFile;
…
}

std::pair<int, int> BattleshipGameAlgo::attack()
{
	pair<int, int> attackMove;
	string line;
	if (!getline(this->attackFile, line))
	{
		//end of file
		//attack moves over
		return std::pair<int, int>(-1, -1);
	}
…
}
```

Instead – read it once, keep a list of moves, and iterate over it locally. There are several reasons for this: For once, if takes significantly more time to read from a file than it does from the stack.

A good example:

```
// get next attack from the player's moves queue
std::pair<int, int> Player::attack()
{
    if(movesQueue.size() > 0)
    {
        std::pair<int,int>& nextAttack(movesQueue.front());
        movesQueue.pop();
        return nextAttack;
    }
    return make_pair(-1,-1);
}
```

Not checking for return errors:

Bad example:

```
ifstream fin("tmp_bmk.txt");
// read it to tmp and later save it in path
getline(fin, tmp);
```

Should be:

```
std::ifstream fin("filesInDirectory.txt");
if (!fin.is_open())
{
	std::cout << "Error while open internal file" << std::endl;
	return;
}
```

Not using initializer list:

```
// Constructor
Player::Player(int playerNum, string attackFilePath) {
       _playerNum = playerNum;
       _playerScore = 0;
       _finishedAttacks = false;
       _shipsMap.clear();
       _attackFilePath = attackFilePath;
       _attackFileOpened = false;
       _numActiveShips = 5;
}
```

A better practice is:

```
// Constructor
Player::Player(int playerNum, string attackFilePath) :
       _playerNum(playerNum),
       _playerScore(0),
       _attackFile(attackFilePath),
       _attackFileOpened(false),
       _finishedAttacks(false),
       _numActiveShips(5)
{
       _shipsMap.clear();
}
```

Creating temporary files and not deleting them:

Many of you used system() in order to handle directories, but some of you just left the created temporary files laying around.

Memory Leaks:

There were MANY memory leaks. Here are a few things to notice:

1) **Do not use new unless it is absolutely necessary – For example, in this particular exercise, you could have avoided using it completely.**
2) **If you do use new, make sure to delete – keep in mind that whoever made the allocation is responsible for the release of the memory.**
3) **Never use new with string! This is not C. Strings behave like any other class.**
4) **Use Visual Studio's diagnostic tools to catch memory leaks. You can use any other tool you like as well.**
5) **In the future – we will learn about smart pointers, and then all your memory allocation troubles will be over!**