Code review – Exercise 1

While reviewing your code in exercise 1, we have encountered many code errors, misuses, bad practices and design issues.
The purpose of this document is to present the code issues that we have found to help you avoid these kind of errors in the following submissions.
All the code snippets here are taken directly from your work, so don't be surprised if they look somewhat familiar. It might be from your exercise.
We have decided **NOT** to deduce points for code issues in exercise 1. Even not for major errors such as memory leak or bad OOP design, as we saw that many of you still have such errors and this may mean that we need first to make sure you are aware of these problems.
BUT – you must fix these errors and all other code issues in your next submissions. From exercise 2 and on we will pay attention to code errors and deduce points for any code issue.

Do's and Don'ts

Try putting all global function in classes (if there is no relevant class, consider 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;
}
```

This function could have been in class Board. Or in class Game.

Hard coded numbers:

Do not use hard coded numbers – especially when you already have them in the scope.

In above case, the number 10 is a mistake – there is a parameter for rows!

In other cases, hard coded numbers can be replaced with enums, static const etc.

Note that putting enums and static const inside a class is the proper way for handling constant values.

Memory Leaks

- **Do not use new with no delete:**

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

// there is no delete, nowhere, oops
```

- **Dtor in comment or not deleting all memory allocated in the class:**

```
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;	*/
}
```

The dtor above probably got into comment because the program crashes when it was active. Reason may be miss of copy ctor (better to block the copy ctor and assignment operator with =delete to make sure Board is not copied).

- **Using uneccesary memory allocations:**

```
class Player {
public:
    queue<pair<int, int>> * attackList;
    explicit Player(char nm) : attackList(nullptr) {}
    // allocating attackList somewhere in class
    ~Player() {
	if (attackList != nullptr) // redundant 'if' – it's ok to delete nullptr
		delete attackList;
    }
    //...
};
```

Instead you could just use `queue<pair<int, int>> attackList` – with no pointer – so you benefit twice: you do not have to worry about memory leaks, and you don't have to worry whether the member is alive (allocated) each time you access it.

- **Asymmetric allocations – allocating in one class and deallocating in another**
- **Deallocating in a method (and not in a dtor) and forgetting to call it**

The correct way is to deallocate in the dtor of the same class that allocates the memory. Other methods that may deallocate should put the pointer back to nullptr or to a valid new allocation.

- **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));
}

// some code here...

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

In the case above, you might try to delete a stack variable.
In general, there is no use of using new/delete on local variables.
A much simpler use of the class string for this case would be:

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

Other unnecessary string plays:

```
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 make the method getAttacksListFromFile get const string& and simply call it: `player.attackList = getAttacksListFromFile(attackFilename, player.name);`

Or if there is no option for making getAttacksListFromFile get const string& one can still simply do: `getAttacksListFromFile(attackFilename.c_str(), player.name);`

For that the method getAttacksListFromFile should get const char* - getting char* in getAttacksListFromFile the source of trouble that requires the ugly and inefficient code above...

Memory management - summary:

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

1) **Do not use `new` unless it is absolutely necessary – if you can do it without allocation that's usually better.**
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. Or you can search for 'new' in your code and make sure that any 'new' is covered with a proper 'delete' in the right places.**
5) **If memory was allocated with `new <type>[]` it MUST be `deleted` with `delete[]`**
6) **In the future – we will learn about smart pointers, and then all your memory allocation troubles will be over!**

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

Bad example:

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

NEVER include cpp files!

Each class should have a cpp and header. Include the header! Not the cpp!

Forgetting 'public' on the inheritance: the default is private (and no polymorphism)

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

Using the class:

```
BasicBattleshipGameAlgo player1, player2;
```

Should be instead:

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

Usage:

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

Note: the game code should not be aware which actual algorithm is used! This is why the abstract base class has virtual methods!

Do not have and excess unused data members

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

If you do not need a member remove it. Don't leave unnecessary and unmanaged members in your code! (In the above case, if you never initialize the member _board or you do not manage it right, because you don't actually need it – remove it from class).

No need for if-else for boolean expression:

Not so nice:

```
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 return the boolean expression:

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

(Note by the way that [y] comes before [x]: `_board[y][x]` as the rows in array are accessed before the cols. Most of you did that right, however very few missed that which led to bugs).

Or even better, something like:

```
bool Board::isPlayer(int x, int y, int playerIndex) {
   return _shipChars[playerIndex].find(getChar(x, y)) !=
		_shipChars[playerIndex].end(); // e.g. using std::set<char>
}
```

(above getChar(x, y) is a method that we suggest to have in class Board, or in class Game: here x comes before y as this is our method, inside the method we will access the array).

Not using `const` on a class method whenever needed:

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

should be:

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

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;
};
```

Explanation:

- `const` **on** `other` **– meaning the method cannot change** `other`
- `GameBoard&` `other` **– value passed by reference, without copying** `GameBoard` `object.`
- `GameBoard&` `(return value)` **– we return a reference to** `this` **instead of a copy.**

(A question that rises – why do you need assignment operator? better block it?)

Not using `static` whenever needed:

```
int GameBoard::getShipLength(char piece) //should declare static modifier in header
{
	int size = 4;
	// below can be declared static in the method or in class
	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.

Data that can be static – better be static.

Use of `exit`:

Do not use `exit`!

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.

Same goes for other warnings – do not leave warnings!

Code Design:

Some students wrote the entire code in just 1 file. That's really bad.

Others missed the use of polymorphism.

Others missed the need of classes and created a big main or a bulk of global functions.

You MUST use proper classes. Each class MUST have its .h file and in most cases a .cpp file (ONLY if this is a template class or if all methods can be inline then there would be only an .h file without a .cpp file).

Note that too many classes (some that do almost nothing and can be a simple primitive type member in another class) is also not good. Use common sense to decide which classes are needed.

Keeping the file open throughout the game:

Not so good:

```
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, it is a better practice – when the amount of data to read is not huge – to read the data into memory and then iterate over it.

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;
	// or other error message as relevant
	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();
}
```

Why?
If some of your members have different initialization in their empty ctor and in the ctor which takes params, it's better to initialize the object once through the relevant ctor and not first time through the empty ctor and then override it with another initialization.

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.

Others:

- **Very long methods**
- **Code duplication**
- **Inefficient code (we are not looking for super-efficiency, but when there is something which is very inefficient compared to another simple implementation one should prefer the more efficient way)**
- **Bad names for classes, methods, data members and variables – we will subtract points for that in the 2nd exercise on! Put some thought in your names!**