

# 2018년 1학기

# C++ BattleShip Project

## 최종 보고서



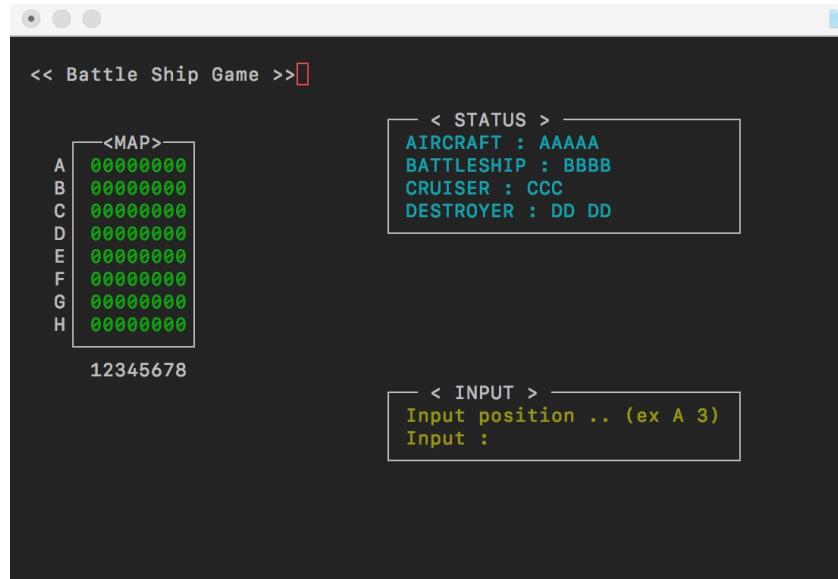
경영학부 경영학전공  
20132651 이성재

### 보고서 개요

1. 1차 과제까지 진행 상황 보고
2. Battleship Game 의 기본 구조
3. 사용자 입력형 모드의 제작
4. Random 공격형 AI 모드의 제작
5. 지능 공격형 AI 모드의 제작
6. 프로젝트 개선점 및 느낀점

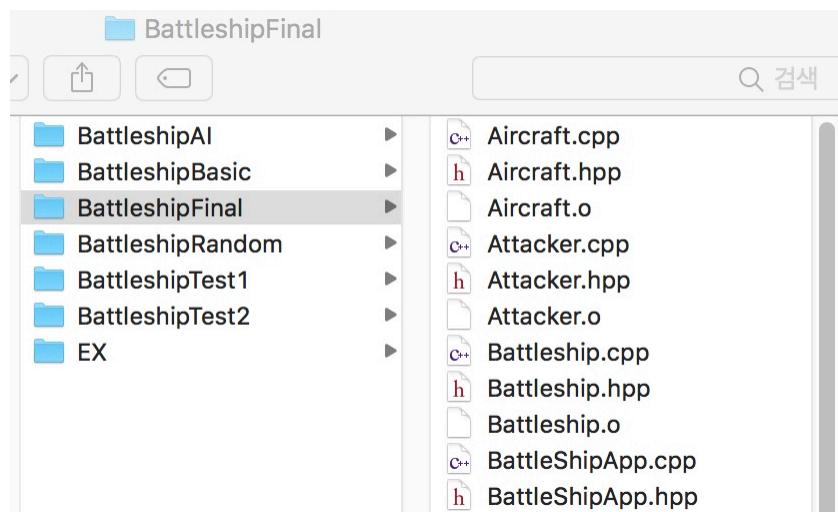
# 1. 1차 과제까지 진행 상황 보고

1차 과제인 사용자 입력형 모드의 제작을 제한시간 내에 제출하지 못하여 이번 최종 과제로 제출하게 되었습니다. 1차 과제에서는 ncurses 의 설치 및 makefile의 작성에 대부분의 시간을 사용하였습니다. cpp 파일 및 헤더파일은 생성하였지만, 실제 내용은 아무것도 작성되지 않은 상태였습니다. 단순하게 참고자료로 주신 UML 구조를 이해하고, 이를 바탕으로 기본 화면만 출력되도록 하는 것이 1차과제까지의 진행상황이었습니다



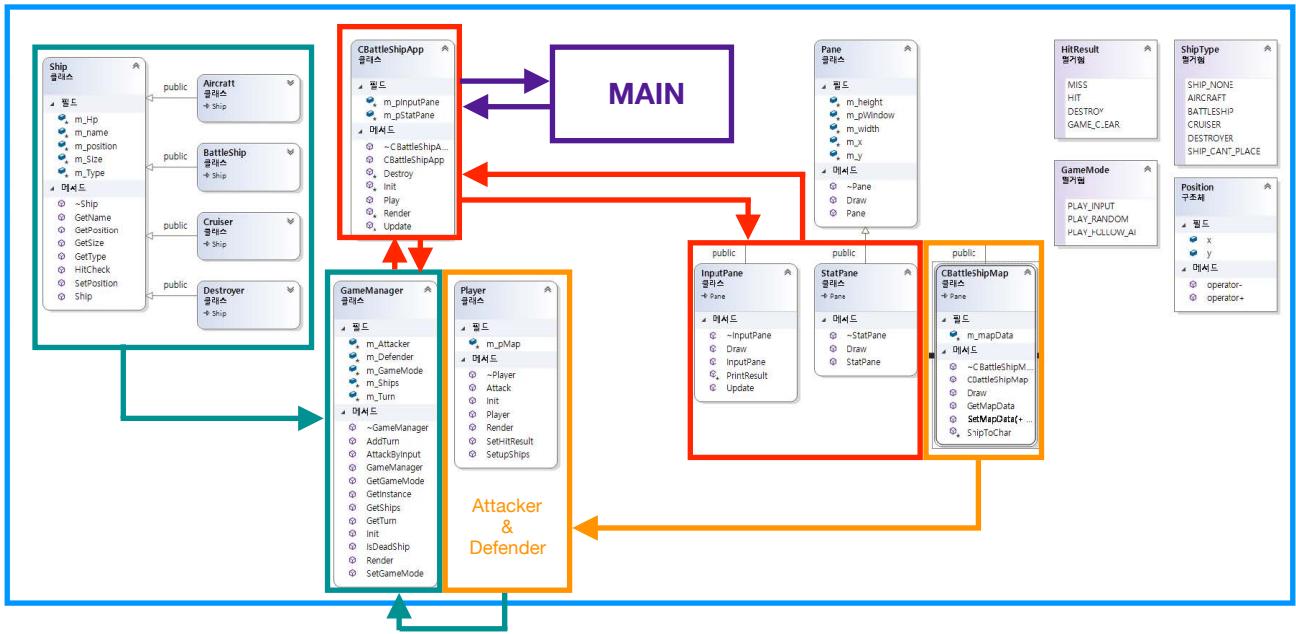
< 기본 화면만 제작된 1차 과제의 진행상황 >

이에 최종 과제 제출까지 진행할 상황을 과제별로 다음과 같이 나누었습니다. 1. 사용자 입력형 모드의 제작. 2. 랜덤 공격형 모드의 제작. 3. 지능 공격형 모드의 제작. 4. 지능 공격형 모드의 반복 및 최소 / 최대 / 평균 turn 수의 출력. 각각의 과제를 순차적으로 완성하여 최종 과제를 제출하는 것이 목표였으며, 포기하지 않고 도전한 결과 시간 내에 완성할 수 있었습니다.



< 과제별로 별개의 폴더에 코드를 저장하고 테스트함 >

## 2. Battleship Game 의 기본 구조



< 작성된 UML의 흐름을 표현한 Battleship 구조도 >

간단하게 Battleship Game 구조의 흐름을 설명하자면 다음과 같습니다. 가장 먼저 메인 함수가 호출되면, BattleShipApp의 객체를 생성하고, 내부의 Play, 혹은 PlayAI 함수를 호출합니다. Play 함수는 Init / Render / Update / Destroy의 핵심적인 4가지 기능이 순차적으로 구현되며, GameManager 내부의 GameOver 함수를 체크하여 반환값이 true가 되기 전까지 Update 함수를 반복하여 시행합니다.

Init 함수에서는 random 값의 시드 설정, 기본적인 ncurses 색상 설정, StatPane과 InputPane, GameManager 객체의 생성, turn과 DestroyedShip 변수의 초기화가 이루어집니다. Render 함수에서는 StatPane / InputPane을 그리고, GameManager의 Init 함수를 호출합니다. 이 때 GameManager의 Init 함수는 Attacker와 Defender 객체, Ship 5개의 객체를 생성하며 Defender가 배를 랜덤으로 배치합니다.

Update 함수에서는 Battleship 게임의 핵심적인 기능들이 수행됩니다. 먼저 0으로 설정된 turn을 한 번 호출될 때마다 증가시킵니다. 이렇게 증가된 turn을 StatPane의 DrawTurn 함수로 보내 증가된 값을 화면에 표시합니다. 또한 InputPane의 GetInput, 혹은 GetInputAI 함수로부터 입력값을 받아옵니다. 이는 플레이 모드에 따라 변화합니다. 받아온 값은 App에 있는 input Position 변수에 저장하고, GameManager에게 넘겨 줍니다. GameManager는 받은 input값으로 CheckMap 함수를 호출하며, CheckMap에서는 먼저 Defender에게 현재 위치에 배가 피격되었는지 확인하여 값을 돌려받습니다. 피격되었다면 해당 위치의 배의 체력을 감소시키며, 최종적으로 Attacker에게 해당 위치가 피격 혹은 피격되지 않았음을 알립니다.

한 턴이 진행될 때마다 GameManager는 DestroyedCheck 함수를 이용해 현재의 공격으로 배가 침몰하였는지 확인합니다. 확인된 여부는 'X' 혹은 배의 이름이 반환되며, 이는 DestroyedShip에 저장됩니다. 저장된 값은 Update 함수가 호출되기 직전에 StatPane의 ShipDestroyed 함수로 전달되며, 해당 내용을 StatPane에 표시합니다.

이 과정들이 한 번 진행될 때마다 GameManager의 GameOver 함수가 호출되며, 모든 배의 체력을 확인하여 게임 종료 여부를 체크합니다. 모든 배의 체력이 0이며, GameOver 함수가 true가 되면 StatPane의 GameWin 함수가 호출되며 종료된 Turn 수를 출력합니다.

마지막으로 Destroy 함수로 모든 윈도우를 제거하며 게임을 종료하면 기본적인 알고리즘 진행이 완료됩니다.

### 3. 사용자 입력형 모드의 제작

```
int main(){
    std::vector<int> AIturns;
    int average = 0;
    int playType;

    std::cout << "Play Input : 1 // Play AI : 2" << '\n';
    std::cin >> playType;

    if (playType == 1) {
        BattleShipApp battleShip;
        battleShip.Play();
    }
}
```

```
void BattleShipApp::Play(){
    Init();
    Render();
    while (m_pGameManager->GameOver() == false) {
        m_pStatPane->ShipDestroyed(DestroyedShip);
        Update();
        if (input.x == 100) {
            break;
        }
    }
    m_pStatPane->ShipDestroyed(DestroyedShip);
    m_pStatPane->GameWin(turn);
    Destroy();
}
```

```
void BattleShipApp::Init(){
    srand((unsigned int)time(NULL));
    initscr();
    start_color();
    cbreak();
    refresh();

    init_pair(1, COLOR_GREEN, COLOR_BLACK);
    init_pair(2, COLOR_CYAN, COLOR_BLACK);
    init_pair(3, COLOR_YELLOW, COLOR_BLACK);

    m_pStatPane = new StatPane(30, 3, 30, 8);
    m_pInputPane = new InputPane(30, 15, 30, 4);
    m_pGameManager = new GameManager();
    turn = 0;
    DestroyedShip = "X";
}
```

```
void BattleShipApp::Render(){
    mvprintw(1, 1, "<< Battle Ship Game >>");
    m_pStatPane -> Draw();
    m_pInputPane -> Draw();
    m_pGameManager -> Init();
    refresh();
}
```

#### 1. main 함수

Battleship Game에서 가장 먼저 호출되는 main 함수는 만들어진 버전에 따라 기능이 달라집니다. 원편에 나타난 main함수는 최종 작성된 Battleship Game의 코드로, 사용자가 입력한 PlayType에 따라 입력형 모드, 혹은 AI 공격형 모드로 진행할 수 있습니다. 1번을 입력하면 BattleShipApp 객체가 생성되며 게임을 시작합니다.

#### 2. BattleShipApp 의 Play 함수

실제로 게임이 진행되기 위한 모든것을 총괄하는 곳은 BattleShipApp 클래스입니다. 이곳에서는 Init / Render / GameOver 체크 / DestroyedShip의 표시 / 게임의 진행을 위한 Update / 게임이 종료된 후 GameWin의 표시 / Destroy 가 순차적으로 진행됩니다. 각각의 함수를 상세하게 보면 다음과 같습니다.

##### 2-1. BattleShipApp 의 Init 함수

게임을 진행하기 위한 초기화가 이루어지는 곳입니다. 이곳에서는 랜덤 시드값을 시간으로 초기화 / ncurses의 기본적인 색상 설정 / StatPane과 InputPane의 생성 / GameManager의 생성 / turn과 DestroyedShip의 초기화가 이루어집니다.

##### 2-2. BattleShipApp 의 Render 함수

위에서 생성된 객체들을 화면에 구성하는 단계의 함수입니다. 이곳에서 StatPane과 InputPane의 기본형태가 Draw 함수를 통해 화면에 그려집니다.

주목할 점은 GameManager의 Init 함수가 호출되며 Attacker / Defender / Ship 객체들이 생성되고, 기본적인 배의 랜덤 배치가 진행된다는 것입니다.

```

void GameManager::Init(){
    m_pAttacker = new Attacker();
    m_pDefender = new Defender();

    m_pAircraft = new Aircraft("Aircraft");
    m_pBattleship = new Battleship("Battleship");
    m_pCruiser = new Cruiser("Cruiser");
    m_pDestroyer1 = new Destroyer("Destroyer1");
    m_pDestroyer2 = new Destroyer("Destroyer2");

    getch();

    m_pDefender->setShip(m_pAircraft);
    m_pDefender->setShip(m_pBattleship);
    m_pDefender->setShip(m_pCruiser);
    m_pDefender->setShip(m_pDestroyer1);
    m_pDefender->setShip(m_pDestroyer2);

}

```

```

void Defender::setShip(Ship* ship){
    int row_col;
    int x, y;
    Position *p = new Position[ship->GetSize()];

    while (true) {

        bool check = true;
        row_col = rand()%2;

        if (row_col == 0) {
            x = rand()%(8 - ship->GetSize() + 1);
            y = rand()%8;

            for (size_t i = 0; i < ship->GetSize(); i++) {
                if (m_pMap->m_mapData[x + i][y] != '-') {
                    check = false;
                }
            }
            if (check == false) {
                continue;
            }
            for (size_t i = 0; i < ship->GetSize(); i++) {
                m_pMap->m_mapData[x + i][y] = ship->GetName()[0];
                m_pMap->DrawShip(x + i, y);
                p[i].x = x + i;
                p[i].y = y;
            }
            ship->SetPosition(p);
        }
        else if(row_col == 1){
            x = rand()%8;
            y = rand()%8 - ship->GetSize() + 1;

            for (size_t i = 0; i < ship->GetSize(); i++) {
                if (m_pMap->m_mapData[x][y + i] != '-') {
                    check = false;
                }
            }
            if (check == false) {
                continue;
            }
            for (size_t i = 0; i < ship->GetSize(); i++) {
                m_pMap->m_mapData[x][y+i] = ship->GetName()[0];
                m_pMap->DrawShip(x, y+i);
                p[i].x = x;
                p[i].y = y + i;
            }
            ship->SetPosition(p);
        }
        break;
    }
}

```

## 2-2-1. GameManager의 Init함수

GameManager는 Attacker 와 Defender 객체, 그리고 4종류의 배 객체를 가지고 있습니다. 이에 따라 각 배에 이름을 붙여 배를 생성하고, 이 배들을 Defender의 setShip 함수로 호출하여 랜덤하게 배치합니다.

### 2-2-1-1. Defender의 setShip 함수 1

가장 중요한 첫 번째 알고리즘인 랜덤 ship 배치 알고리즘일 작성된 곳이 setShip 함수입니다. 먼저 특정한 배를 함수의 매개변수로 전달합니다.

먼저 row\_col / x, y / p 등의 변수를 정의합니다. row\_col 은 배를 가로 혹은 세로의 어느 방향으로 배치할 것 인지를 결정하는 랜덤값이 들어갑니다. x, y에는 배가 배치될 시작점이 들어갑니다. p는 포지션 포인터 배열로 배의 크기만큼 만들어집니다. 배열의 각 값에는 배가 위치할 Position 변수의 값들이 들어갑니다.

while 반복문을 이용해 알고리즘을 진행하며, check 부울 변수를 사용하여 x, y점이 배치 가능한 점인지 여부를 반복 싸이클마다 확인합니다. 가장 먼저 rand() 함수를 이용해 row\_col 이 0 혹은 1의 값을 가지도록 합니다.

만약 row\_col 이 0이라면 세로로 배치하도록 하며, x 와 y값에 0부터 7사이의 랜덤값을 넣습니다. 이 때, x에는 ship의 크기만큼 작은 값이 들어가도록 하여, 만약 ship의 크기가 5라면 0부터 3 사이의 값이 들어가도록 합니다. 이를 통해 배가 배치될 때 칸을 넘어가지 않도록 할 수 있습니다.

다음으로 배가 배치 가능한지의 여부를 반복문을 이용해 확인합니다. Defender에서 생성된 BattleShipMap 의 객체인 m\_mapData 를 확인하여, 만약 아무것도 표시되지 않은 ‘-’ 값이 아니라면 check를 false로 만들어 다른 랜덤 값을 갖도록 합니다.

이러한 과정을 continue 문을 이용해 while문 내에서 반복하다가, 만약 배치 가능한 위치가 나타난다면 배치를 진행합니다. 배의 배치 또한 for 반복문을 이용해 이루어지며, 배의 크기만큼 반복하게 됩니다.

반복문 내에서는 m\_mapData를 수정하게 되며, 해당 배의 이름의 첫 글자를 가져와 입력합니다. m\_mapData에 입력한 다음, BattleShipMap에 정의된 DrawShip 함수를 호출하여 맵의 해당 부분만 변경하여 표시하도록 합니다. 표시가 모두 끝났다면 해당 위치를 앞에서 정의내린 Position 배열에 넣고, 최종적으로 배의 위치들을 SetPosition 함수를 이용해 한꺼번에 배에 저장하도록 합니다.

### 2-2-1-1. Defender의 setShip 함수 2

다음 과정은 row\_col 이 1일 때, 즉 배를 가로로 배치하는 경우이며 앞에서 변경하던 x값이 그대로 작성되고 y값이 변경된다는 차이점만 있습니다. 그 외의 모든 과정이 위의 과정과 동일하게 진행됩니다.

```

bool GameManager::GameOver(){
    if (_pAircraft->m_HP == 0) {
        _pAircraft->Destroyed = true;
    }
    if (_pBattleship->m_HP == 0) {
        _pBattleship->Destroyed = true;
    }
    if (_pCruiser->m_HP == 0) {
        _pCruiser->Destroyed = true;
    }
    if (_pDestroyer1->m_HP == 0) {
        _pDestroyer1->Destroyed = true;
    }
    if (_pDestroyer2->m_HP == 0) {
        _pDestroyer2->Destroyed = true;
    }
    return _pAircraft->Destroyed && _pBattleship->Destroyed && _pCruiser->Destroyed && _pDestroyer1->Destroyed && _pDestroyer2->Destroyed;
}

```

```

void StatPane::ShipDestroyed(string ShipName){
    char shipname[ShipName.size() + 1];
    int i;
    for (i = 0; i < ShipName.size(); i++) {
        shipname[i] = ShipName[i];
    }
    shipname[i] = NULL;

    if (ShipName[0] == 'X') {
        wattron(m_pWindow, COLOR_PAIR(2));
        mvwprintw(m_pWindow, 6, 1, "          ");
        wattroff(m_pWindow, COLOR_PAIR(2));
        wrefresh(m_pWindow);
    } else {
        wattron(m_pWindow, COLOR_PAIR(2));
        mvwprintw(m_pWindow, 6, 1, " You Destroyed ");
        mvwprintw(m_pWindow, 6, 16, "%s", shipname);
        wattroff(m_pWindow, COLOR_PAIR(2));
        wrefresh(m_pWindow);
    }
}

```

```

void BattleShipApp::Update(){
    turn++;
    m_pStatPane->DrawTurn(turn);
    input = m_pInputPane->GetInput();
    m_pGameManager->input = input;
    m_pGameManager->CheckMap(input);
    DestroyedShip = m_pGameManager->DestroyedCheck(input);
}

```

```

void StatPane::DrawTurn(int turn){
    wattron(m_pWindow, COLOR_PAIR(2));
    mvwprintw(m_pWindow, 5, 2, "Turn : ");
    mvwprintw(m_pWindow, 5, 10, "%d", turn);
    wattroff(m_pWindow, COLOR_PAIR(2));

    wrefresh(m_pWindow);
}

```

```

Position InputPane::GetInput(){
    char input[3];
    Position inputToPosition;
    wattron(m_pWindow, COLOR_PAIR(3));
    mvwscanw(m_pWindow, 2, 10, "%s", input);
    mvwprintw(m_pWindow, 2, 10, "      ");
    wattroff(m_pWindow, COLOR_PAIR(3));
    wrefresh(m_pWindow);
    inputToPosition.x = input[0] - 'A';
    inputToPosition.y = input[1] - '0' - 1;
    if (input[0] == 'x') {
        inputToPosition.x = 100;
    }
    return inputToPosition;
}

```

## 2-3. GameManager의 GameOver 함수

말 그대로 게임이 끝났는지를 확인하는 함수입니다. true 혹은 false를 반환하며, while문의 조건으로 사용되며 Update 를 반복시킵니다. 게임의 종료 여부는 각 배의 체력인 m\_HP를 확인하며 HP가 0인 배를 destroyed 상태로 표시합니다. return 에서 모든 배의 destroyed 여부를 한번에 확인하여, 모든 배의 destroyed 여부가 true일 때 true를 반환합니다.

## 2-4. StatPane 의 ShipDestroyed 함수

파괴된 배의 이름, 혹은 파괴되지 않은 상태를 string 으로 받아 만약 파괴되지 않았다면 빈칸을, 파괴되었다면 해당 배의 이름을 StatPane에 출력해주는 함수입니다.

배가 파괴되었는지의 여부는 아래의 Update 함수에서 확인하며, 파괴되지 않았다면 'X' 가 string 형태로 들어오게 됩니다.

## 2-5. BattleShipApp의 Update 함수

게임의 진행을 한눈에 볼 수 있는 Update 함수입니다. 먼저 앞에서 0으로 초기화 된 턴을 호출될 때마다 1씩 증가시키고, StatPane의 DrawTurn 함수에 보내 StatPane에 출력하도록 합니다.

그 다음으로는 InputPane으로부터 GetInput 함수를 호출하여 입력을 받아 input이라는 Position 변수에 저장합니다. 저장된 input은 GameManager의 input에 저장하고, CheckMap에 보내 Defender와 Attacker가 체크하도록 합니다.

마지막으로 위의 ShipDestroyed 함수에 사용될 DestroyedShip string 변수에 input을 보내 해당 위치에 배가 파괴되었는지, 파괴되었다면 어떤 배가 파괴되었는지 확인한 결과값을 가져옵니다.

### 2-5-1. StatPane 의 DrawTurn 함수

turn 을 받아와 StatPane에 출력합니다.

### 2-5-2. InputPane의 GetInput 함수

먼저 ncurses는 string 사용이 어려워 char 배열 형태의 input을 만듭니다. mvwscanw 함수로 input을 받아와 이를 inputToPosition 이라는 Position 변수의 x, y값으로 각각 저장합니다. 만약 x가 입력되면 100을 저장하여 사용자가 종료할 수 있도록 하며, inputToPosition을 반환합니다.

```
void GameManager::CheckMap(Position input){
    HitOrMiss = m_pDefender->CheckHit(input);
    if (HitOrMiss == 'H') {
        MinusHP(input);
    }
    m_pAttacker->CheckHit(input, HitOrMiss);
}
```

```
char Defender::CheckHit(Position input){
    char HitOrMiss;
    if (m_pMap->m_mapData[input.x][input.y] == '-') {
        HitOrMiss = 'M';
        return HitOrMiss;
    } else {
        HitOrMiss = 'H';
        return HitOrMiss;
    }
}
```

```
void GameManager::MinusHP(Position p){
    for (size_t i = 0; i < m_pAircraft->GetSize(); i++) {
        if (m_pAircraft->m_Position[i].x == p.x && m_pAircraft->m_Position[i].y == p.y) {
            m_pAircraft->m_HP--;
            return;
        }
    }
    for (size_t i = 0; i < m_pBattleship->GetSize(); i++) {
        if (m_pBattleship->m_Position[i].x == p.x && m_pBattleship->m_Position[i].y == p.y) {
            m_pBattleship->m_HP--;
            return;
        }
    }
    for (size_t i = 0; i < m_pCruiser->GetSize(); i++) {
        if (m_pCruiser->m_Position[i].x == p.x && m_pCruiser->m_Position[i].y == p.y) {
            m_pCruiser->m_HP--;
            return;
        }
    }
    for (size_t i = 0; i < m_pDestroyer1->GetSize(); i++) {
        if (m_pDestroyer1->m_Position[i].x == p.x && m_pDestroyer1->m_Position[i].y == p.y) {
            m_pDestroyer1->m_HP--;
            return;
        }
    }
    for (size_t i = 0; i < m_pDestroyer2->GetSize(); i++) {
        if (m_pDestroyer2->m_Position[i].x == p.x && m_pDestroyer2->m_Position[i].y == p.y) {
            m_pDestroyer2->m_HP--;
            return;
        }
    }
}
```

```
void Attacker::CheckHit(Position input, char HitOrMiss){
    m_pMap->m_mapData[input.x][input.y] = HitOrMiss;
    m_pMap->DrawShip(input.x, input.y);
}
```

```
string GameManager::DestroyedCheck(Position p){
    if (HitOrMiss == 'M') {
        return "X";
    } else {
        for (size_t i = 0; i < m_pAircraft->GetSize(); i++) {
            if (m_pAircraft->GetPosition()[i].x == p.x && m_pAircraft->GetPosition()[i].y == p.y) {
                if (m_pAircraft->m_HP == 0) {
                    return m_pAircraft->GetName();
                }
            }
        }
        for (size_t i = 0; i < m_pBattleship->GetSize(); i++) {
            if (m_pBattleship->GetPosition()[i].x == p.x && m_pBattleship->GetPosition()[i].y == p.y) {
                if (m_pBattleship->m_HP == 0) {
                    return m_pBattleship->GetName();
                }
            }
        }
        for (size_t i = 0; i < m_pCruiser->GetSize(); i++) {
            if (m_pCruiser->GetPosition()[i].x == p.x && m_pCruiser->GetPosition()[i].y == p.y) {
                if (m_pCruiser->m_HP == 0) {
                    return m_pCruiser->GetName();
                }
            }
        }
        for (size_t i = 0; i < m_pDestroyer1->GetSize(); i++) {
            if (m_pDestroyer1->GetPosition()[i].x == p.x && m_pDestroyer1->GetPosition()[i].y == p.y) {
                if (m_pDestroyer1->m_HP == 0) {
                    return m_pDestroyer1->GetName();
                }
            }
        }
        for (size_t i = 0; i < m_pDestroyer2->GetSize(); i++) {
            if (m_pDestroyer2->GetPosition()[i].x == p.x && m_pDestroyer2->GetPosition()[i].y == p.y) {
                if (m_pDestroyer2->m_HP == 0) {
                    return m_pDestroyer2->GetName();
                }
            }
        }
    }
    return "X";
}
```

## 2-5-3. GameManager의 CheckMap 함수

앞에서 받아오게 된 Position 형태의 input값을 CheckMap 함수에 넘겨주면, GameManager가 이 값을 Defender에게 보내 피격 여부를 확인하고, 다시 피격 여부와 위치를 Attacker에게 보내 맵에 표시하도록 합니다.

그 중간과정에 피격 여부에 따라 해당 위치에 있는 배의 체력을 감소시키는 MinusHP 함수가 호출될 수 있습니다.

### 2-5-3-1. Defender 의 CheckHit 함수

Defender는 위치를 넘겨받아 해당 위치의 mapData를 확인하고, 만약 ‘-’ 이 표시되어 있다면 Miss를 의미하는 ‘M’을, 그 외에는 무언가 배가 표시되어 있었을 것으로 Hit를 의미하는 ‘H’를 반환합니다.

이 때 반환은 HitOrMiss라는 char 변수를 이용해 글자 그 자체를 반환하도록 합니다.

### 2-5-3-2. GameManager의 MinusHP 함수

위치를 넘겨받아 해당 위치를 소유하고 있는 배가 있는지 확인하고, 있다면 그 배의 체력을 깎는 함수입니다. 배의 체력은 각 배의 크기와 동일한 값으로 초기화 되어 있으며, 이는 생성자가 호출될 때 구성됩니다.

각 배의 m\_Position 포지션 배열 변수를 이용하여 x, y값을 비교하게 되며, 모두 동일한 값을 가질 경우에만 해당 배의 HP를 감소시키고 함수를 종료합니다. 이는 하나의 위치에 하나의 배만 존재하기에 종료해도 문제가 생기지 않기 때문입니다.

### 2-5-3-3. Attacker 의 CheckHit 함수

앞의 두 함수를 통해 위치와 피격 여부를 받아와서 Attacker에게 표시하도록 하는 함수입니다. 위치인 input과 피격 여부인 HitOrMiss가 동일한 이름으로 매개변수로 사용됩니다. 해당 위치의 mapData를 HitOrMiss로 변경해주며, BattleShipMap의 DrawShip 함수를 이용해 해당 위치의 값을 업데이트해 줍니다.

## 2-5-4. GameManager의 DestroyedCheck 함수

앞에서 받아온 HitOrMiss 변수를 이용하여 M이라면 “X”를 반환하고, H라면 해당 위치의 배를 찾아내어 그 배의 HP가 0이 되었는지 확인합니다. 이번 피격으로 HP가 0이 되었다면 Ship에 있는 GetName 함수를 이용해 해당 배의 이름을 가져와 반환합니다.

반환된 string 변수는 DestroyedShip에 저장되며 StatPane의 ShipDestroyed 함수에 매개변수로 들어가 StatPane에 출력되게 됩니다.

```

void StatPane::GameWin(int turn){
    wattron(m_pWindow, COLOR_PAIR(2));
    mvwprintw(m_pWindow, 5, 1, " CONGRATURATION !! TURN : ");
    mvwprintw(m_pWindow, 5, 26, "%d", turn);
    wattroff(m_pWindow, COLOR_PAIR(2));
    wrefresh(m_pWindow);
}

```

## 2-6. StatPane의 GameWin 함수

Update를 진행하는 while 문이 종료되고, 게임이 끝나면 GameWin 함수가 호출되어 게임의 종료를 알립니다. 이 때 끝난 turn을 받아와 함께 출력하여 몇 번째 턴에 게임이 종료되었는지 StatPane을 통해 알려줍니다.

### 3 ) 사용자 입력형 모드의 실행 결과

```

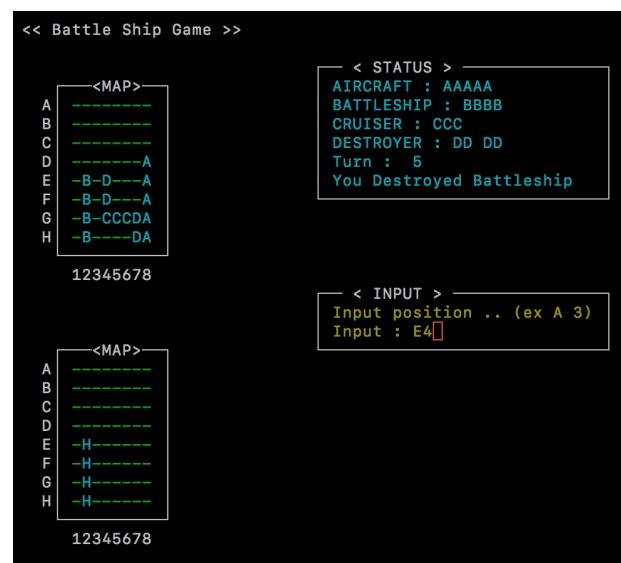
[Sungjaeui-MacBook-Pro:battleshipfinal sungjae$ ./test
Play Input : 1 // Play AI : 2
1

```

사용자가 입력형 모드를 진행할지, AI 공격형 모드를 진행할지 선택할 수 있습니다.  
1을 선택하여 입력형 모드를 실행하였습니다.



< 그림 1. Ship의 배치 및 입력 대기 >



< 그림 2. Battleship에 대한 공격 >

그림1. Ship의 배치 및 입력 대기

게임이 시작되면 먼저 비어있는 맵이 그려지고, 한 번의 키 입력을 getch()를 이용해 진행되면 배의 랜덤 배치가 맵에 나타납니다.

그림2. Battleship에 대한 공격

위의 Defender의 맵을 보고 특정 Ship인 Battleship을 공격하였습니다. E1, F1, G1, H1을 차례로 입력한 결과 STATUS창에 Battleship을 파괴했음이 표시되는 것을 확인할 수 있습니다.

그림3. 게임 완료 전 종료 기능의 구현

당연히 모든 게임이 끝나면 종료되지만, 편의성을 위해 추가적인 게임종단 기능을 넣었습니다. Input에 x 값을 넣으면 강제적으로 게임이 종료되도록 합니다.



< 그림 3. x 입력 후 강제 게임종료 화면 >

# 4. Random 공격형 AI 모드의 제작

```
void BattleShipApp::PlayAI(){
    Init();
    Render();
    while (m_pGameManager->GameOver() == false) {
        m_pStatPane->ShipDestroyed(DestroyedShip);
        UpdateAI();
        getch();
    }
    m_pStatPane->ShipDestroyed(DestroyedShip);
    m_pStatPane->GameWin(turn);
    Destroy();
}
```

## 1. BattleShipApp 의 PlayAI 함수

랜덤 공격형 AI 프로그램을 작성하기 위해 PlayAI 함수를 BattleShipApp에 새롭게 작성하였습니다. Random 형 공격의 경우 따로 선택지를 만들지 않아 main에서 직접 Play 함수를 PlayAI 함수로 변경하여 실행해야 합니다.

PlayAI 함수는 위의 Play 함수와 Input을 받아오는 방법 이외에 모든 것이 동일합니다. Input을 받아오는 과정은 기존의 Update 함수에 존재하므로, UpdateAI라는 함수도 하나 더 만들도록 합니다.

```
void BattleShipApp::UpdateAI(){
    turn++;
    m_pStatPane->DrawTurn(turn);
    input = GetInputAI();
    m_pGameManager->input = input;
    m_pGameManager->CheckMap(input);
    DestroyedShip = m_pGameManager->DestroyedCheck(input);
}
```

## 2. BattleShipApp 의 UpdateAI 함수

기존의 Update 함수를 대체하기 위한 UpdateAI 함수입니다. 마찬가지로 모든 기능이 동일하지만 input을 받아오는 GetInput 함수가 사라지고, BattleShipApp 내에 GetInputAI 함수를 새롭게 만들어 input 값을 저장하도록 합니다.

```
Position BattleShipApp::GetInputAI(){
    int x, y;
    Position returnPosition;
    while (true) {
        x = rand()%8;
        y = rand()%8;
        if (AIAttackMap[x][y] == 1) {
            continue;
        }else{
            AIAttackMap[x][y] = 1;
            returnPosition.x = x;
            returnPosition.y = y;
            return returnPosition;
        }
    }
}
```

## 3. BattleShipApp 의 GetInputAI 함수

랜덤 공격을 하기 위해서는 하나의 조건이 필요한데, 이는 기존에 공격을 했던 점인지 아닌지 판별할 수 있어야 한다는 것입니다. 이러한 문제를 해결하기 위해 AIAttackMap이라는 int 형 2차원 배열을 만들어 0으로 초기화하고, 공격할 때마다 해당 위치에 1로 표시합니다.

만약 랜덤하게 뽑아낸 x, y값이 이미 공격한 점이라면, 다시 뽑아내도록 continue 문을 실행시킵니다. 아니라면 해당 위치를 returnPosition이라는 Position 변수에 저장하여 결과값으로 반환합니다.

## 4 ) Random 공격형 AI 모드의 실행 결과

```
<< Battle Ship Game >>

< MAP >
A B-D-----
B B-D---D-
C B-----D-
D B-----D-
E -AAAAAA-
F C-----C-
G C-----C-
H C-----C-

12345678

< STATUS >
AIRCRAFT : AAAAA
BATTLESHIP : BBBB
CRUISER : CCC
DESTROYER : DD DD
Turn : 1

< INPUT >
Input position .. (ex A 3)
Input :
```

< 그림 1. 초기 Ship 배치 및 첫 선택 화면 >

```
<< Battle Ship Game >>

< MAP >
A B-D-----
B B-D---D-
C B-----D-
D B-----D-
E -AAAAAA-
F C-----C-
G C-----C-
H C-----C-

12345678

< STATUS >
AIRCRAFT : AAAAA
BATTLESHIP : BBBB
CRUISER : CCC
DESTROYER : DD DD
Turn : 31

< INPUT >
Input position .. (ex A 3)
Input :
```

< 그림 2. 31회의 무작위 선택 화면 >

```
<< Battle Ship Game >>

< MAP >
A B-D-----
B B-D---D-
C B-----D-
D B-----D-
E -AAAAAA-
F C-----C-
G C-----C-
H C-----C-

12345678

< STATUS >
AIRCRAFT : AAAAA
BATTLESHIP : BBBB
CRUISER : CCC
DESTROYER : DD DD
CONGRATURATION !! TURN : 62
You Destroyed Aircraft

< INPUT >
Input position .. (ex A 3)
Input :
```

< 그림 3. 무작위 선택 게임종료 화면 >

그림1. 초기 Ship 배치 및 첫 선택 화면

게임이 시작되면 **Ship**들을 배치하고, 키 입력에 따라 랜덤 공격이 이루어집니다. 한 번의 키 입력으로 D7 위치에 랜덤한 공격이 이루어진 것을 볼 수 있습니다.

그림2. 31회의 무작위 선택 화면

31회 동안의 키 입력으로 31회의 무작위 공격이 진행된 모습입니다. Hit 여부에 상관없이 무작위로 공격이 이루어져 아무런 연관성이 없는 **Attack Map** 이 보여집니다.

그림3. 무작위 선택 게임종료 화면

마지막으로 **Aircraft** 배를 파괴하며 62턴만에 게임이 종료되는 모습입니다. 총 64개의 점중에서 62개를 공격한 것으로 매우 나쁜 효율을 가진다는 것을 알 수 있습니다.

# 5. 지능 공격형 AI 모드의 제작

```
    }else{
        for (size_t i = 0; i < 10; i++) {
            BattleShipApp battleShip;
            battleShip.PlayAI();
            AIturns.push_back(battleShip.lastTurn);
        }

        sort(AIturns.begin(), AIturns.end());
        for (size_t i = 0; i < AIturns.size(); i++) {
            average += AIturns[i];
        }

        average = average / AIturns.size();
        std::cout << "Minimum Turn : " << AIturns[0] << '\n';
        std::cout << "Maximum Turn : " << AIturns.back() << '\n';
        std::cout << "Average Turn : " << average << '\n';
    }
    return 0;
}
```

```
Position BattleShipApp::GetInputAI(){
    int x, y;
    Position returnPosition;
    Position pushPosition;

    for (size_t x = 0; x < 8; x++) {
        for (size_t y = 0; y < 8; y++) {
            AIAttackMap[x][y] = m_pGameManager->m_pAttacker->m_pMap->m_mapData[x][y];
            if (AIAttackMap[x][y] == 'H') {

                if (x < 7 && AIAttackMap[x+1][y] == '-') {
                    pushPosition.x = x + 1;
                    pushPosition.y = y;
                    nextAttack.push_back(pushPosition);
                }

                if (y < 7 && AIAttackMap[x][y+1] == '-') {
                    pushPosition.x = x;
                    pushPosition.y = y + 1;
                    nextAttack.push_back(pushPosition);
                }

                if (0 < x && AIAttackMap[x-1][y] == '-') {
                    pushPosition.x = x - 1;
                    pushPosition.y = y;
                    nextAttack.push_back(pushPosition);
                }

                if (0 < y && AIAttackMap[x][y-1] == '-') {
                    pushPosition.x = x;
                    pushPosition.y = y - 1;
                    nextAttack.push_back(pushPosition);
                }
            }
        }
    }
}
```

```
    while (true) {

        x = rand()%8;
        y = rand()%8;

        while (true) {
            if (!nextAttack.empty()) {
                returnPosition = nextAttack.back();
                if (AIAttackMap[returnPosition.x][returnPosition.y] == '-') {
                    nextAttack.pop_back();
                    return returnPosition;
                }else{
                    nextAttack.pop_back();
                    continue;
                }
            }else{
                break;
            }
        }

        if (AIAttackMap[x][y] == '-') {
            returnPosition.x = x;
            returnPosition.y = y;
            return returnPosition;
        }else{
            continue;
        }
    }
}
```

## 1. main 함수

사용자 입력형 모드와 동일하게, 사용자의 입력을 받아 AI 모드를 선택하고, 진행하게 됩니다. 다른 점은 총 10번의 시행이 있으며 해당 진행에 대해 걸린 turn 수에 대한 결과값을 Altturns라는 벡터 변수에 저장한다는 점입니다.

저장된 최종 결과값은 algorithm 라이브러리의 sort 함수를 이용하여 오름차순으로 정렬되며, vector의 첫 번째 변수와 마지막 변수를 꺼내와 Minimum Turn과 Maximum Turn을 출력하도록 합니다.

평균적인 Turn 수를 구하기 위해 Altturns 벡터 크기 만큼의 for 문을 사용하여 average라는 변수에 모든 벡터값을 저장하고, 크기만큼 나누어 줍니다.

## 2. BattleShipApp의 GetInputAI 함수

다른 모든 함수는 위의 Random AI 공격 함수와 동일하며, GetInputAI 함수 하나만 다르게 작성되었습니다.

먼저 returnPosition과 pushPosition 두 개의 포지션 변수를 정의합니다. returnPosition은 최종적으로 선택되어 반환할 변수이며, pushPosition은 나중에 사용될 포지션을 포지션 벡터 변수에 저장해 둘 변수입니다.

가장 먼저 AI가 효과적으로 공격하기 위해서는 지금 까지의 공격 결과를 알아야 하므로, GameManager가 내부의 Attacker가 가진 Map의 m\_mapData를 가져와 AIAttackMap에 복사합니다. 이 때 AIAttackMap은 기존의 것과 다르게 char 변수를 가지고 있습니다.

하나씩 복사하는 과정에서 AIAttackMap에 피격된 위치라는 'H'가 복사될 경우, 그 위치의 상하좌우를 nextAttack 포지션 벡터 변수에 push\_back 합니다. 이 때 고려해야 하는 조건은 다음과 같습니다. 첫째, 상하좌우로 이동하였을 경우 그 위치가 벽으로 막혀 이동 불가능하지는 않은지, 둘째, 해당 위치가 이미 피격되어 다시 피격할 필요 없는 위치는 아닌지를 확인해야 합니다. 이러한 조건을 모두 만족하였다 해당 위치를 pushPosition에 저장하여 nextAttack 배열에 저장합니다.

이러한 저장 과정은 매 턴 이루어지며, 과정이 종료된 다음 본격적인 공격이 시작됩니다. 공격은 먼저 저장된 nextAttack의 값을 하나씩 꺼내와 공격하는 것, 그리고 nextAttack에 더이상 값이 없다면 랜덤하게 공격하는 것으로 나누어집니다.

먼저 while 반복문 속에서 nextAttack이 비어있지 않다면 returnPosition을 nextAttack의 마지막 값으로 지정하고 해당 위치에 공격이 가능한지 판단합니다. 공격이 가능하다면 nextAttack에서 해당 위치를 삭제한 다음, 반환합니다. 공격이 불가능하다면 마찬가지로 해당 위치를 삭제한 다음 다음 배열값을 가져옵니다.

nextAttack이 비었다면 반복문을 종료하고, 다음 조건문으로 이동합니다. 다음 조건문에서는 x, y에 랜덤값을 부여하여 해당 위치가 비어있다면 공격, 아니라면 새로운 랜덤값을 찾습니다.

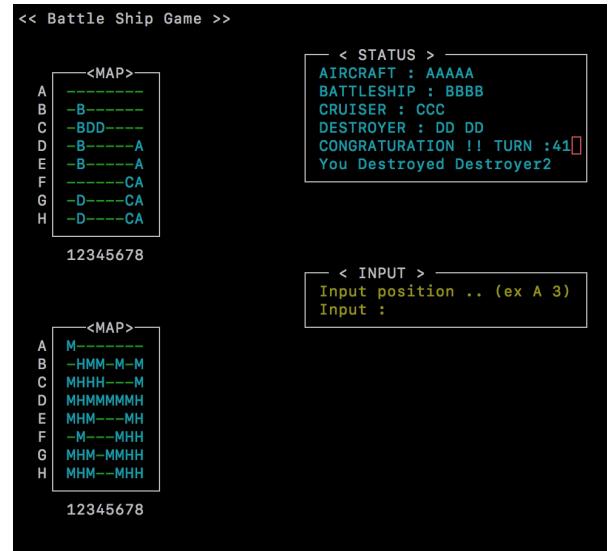
## 5 ) 지능 공격형 AI 모드의 실행 결과

```
[Sungjaeui-MacBook-Pro:battleshipfinal sungjae$ ./test
Play Input : 1 // Play AI : 2
2
```

사용자가 입력형 모드를 진행할지, AI 공격형 모드를 진행할지 선택할 수 있습니다.  
2를 선택하여 AI 모드를 실행하였습니다.



< 그림 1. 35턴이 걸린 한 게임의 종료 화면 >



< 그림 2. 41턴이 걸린 한 게임의 종료 화면 >

```
[Sungjaeui-MacBook-Pro:battleshipfinal sungjae$ ./test
Play Input : 1 // Play AI : 2
2
Minimum Turn : 32
Maximum Turn : 55
Average Turn : 43
Sungjaeui-MacBook-Pro:battleshipfinal sungjae$
```

< 그림 3. 10회의 AI 공격 결과, 최소 32턴, 최대 55턴, 평균 43턴의 기록을 출력하는 화면 >

그림 1. 35턴이 걸린 한 게임의 종료 화면

지능형 AI 모드의 경우 여러 게임을 진행해야 하기 때문에 `getch()` 한 번으로 한 게임이 진행되도록 제작하였습니다. 이에 10번의 엔터키 입력으로 10게임이 진행되는 모습을 한눈에 볼 수 있습니다.

그림 2. 41턴이 걸린 한 게임의 종료 화면

그림 3. 최종 결과 화면

평균적으로 40턴 초반대의 결과를 보였으며, 가장 결과가 잘 나왔을 때는 29턴이었습니다. 하지만 결과가 잘 나오지 않을 때는 50턴 후반대의 결과가 자주 나오며, 변동폭이 큰 알고리즘으로 보입니다.

# 6. 프로젝트 개선점 및 느낀 점

## 프로젝트 개선점

### 1. 사용자의 입력에 대한 제한 필요

사용자의 입력을 제어하지 못해 문제가 발생하는 경우가 많습니다. 사용자가 x 키를 눌러 게임을 중단하게 만든 점은 높이 평가할 수 있지만, 기존에 입력했던 지점을 다시 입력하거나, 맵에 존재하지 않는 위치를 입력하였을 때 적절하게 대처하지 못합니다.

이를 개선하기 위해 사용자 입력 맵을 새롭게 생성하여, 입력 차원인 `BattleShipApp` 차원에서 에러를 발생시키고, 사용자에게 다른 값을 입력하도록 지시하는 과정이 필요할 것으로 보입니다.

### 2. 파괴된 배의 정보를 지속적으로 표시

현재는 배가 파괴되었을 경우 그 상황에만 한 번 사용자에게 표시하는 것이 전부입니다. 하지만 `StatPane`에 지속적으로 어떤 배들이 파괴되었는지 표시해 준다면 더욱 좋은 사용자 경험을 제공할 수 있을 것입니다. 상세한 구현은 파괴된 배들의 정보를 `GameManager` 측에서 `BattleShipApp`으로 보내주고, 이를 다시 `StatPane`에 보내 표시하는 방식으로 구현 가능합니다.

### 3. 더 나은 AI 공격 알고리즘의 개발

현재의 AI 알고리즘은 단순히 피격된 장소의 상하좌우를 다음 공격점으로 두고, 순차적으로 공격해 나가는 방식입니다. 이는 쓸데없는 공격 지점까지 공격하게 되어 턴수를 증가시키는 단점이 존재합니다. 이를 해결하기 위해 다음과 같은 방법이 가능하다고 생각합니다. 첫째로 Hit 를 연속하여 달성하였을 경우, 같은 방향으로 공격하게 하는 것입니다. `bool` 변수를 두개 지정하여,  $n$  번째와  $n + 1$  번째 모두 Hit를 달성하였다면, 동일한 방향으로 공격하거나, 180도 회전된 반대 방향으로 공격하게 하면 더 좋은 성적을 낼 것입니다. 둘째로 현재 어떤 배가 남았는지를 체크하여 Destroyer 하나만 남았다면 2칸 이상 지속되지 않는다는 점을 AI가 알게된다면 더욱 효율적인 공격이 가능할 것입니다. 마지막으로 확률상 배가 가운데 배치될 가능성이 높기 때문에 랜덤하게 선택해야 할 경우 가운데부터 공격하도록 하는 것이 효과적일 것입니다.

C++ 실력의 한계로 완벽한 알고리즘을 구현하지는 못하였지만 시간날 때 마다 위의 사항을 조금씩 구현하여 업데이트 해 보도록 하겠습니다.

## 프로젝트를 진행하며 느낀 점

이번 프로젝트는 처음에 정말 막막하게 다가왔지만, 진행하며 C++ 실력을 상당히 높일 수 있는 기회가 되었습니다. 우선 `Makefile` 과 헤더파일의 사용을 확실하게 익히며 리눅스 기반의 컴파일 환경에 대한 이해를 할 수 있었습니다. 이처럼 많은 파일로 분할되어 개발을 진행해 본 적은 처음이었지만, 이러한 분할에 익숙해져야 한다는 점을 느꼈습니다. 큰 프로젝트가 될 수록 작은 기능으로 나누어 구현해야 하며, 나중에 효과적으로 코드의 이용이 가능하다는 점을 알게 되었습니다.

무엇보다 논리적으로 프로그램을 설계하는 과정의 중요성에 대해 깨닫게 되었습니다. 아무리 쉬운 프로그램이라도 공책에 논리적인 구성 요소들을 적고, 그 관계에 대해 정리를 하지 않으면 구현하는데 어려움이 많고, 반대로 어려운 프로그램이라도 상세하고 정확하게 구성 요소를 작성한다면 구현이 크게 어렵지 않다는 것을 알게 되었습니다. 앞으로의 프로그래밍에 있어 논리적인 구성 과정을 꼭 거쳐서 진행해야 한다는 점을 크게 느꼈습니다.

`Battleship` 프로젝트는 정말 어려웠지만 역시 마지막까지 포기해서는 안된다는 점을 알려준 과제였습니다. 끝까지 버티고 도전하면 해결할 수 있다는 점을 다시 한 번 깨닫게 해 주었다는 점에서 프로젝트 진행 동안 알찬 시간을 보냈다고 생각됩니다.