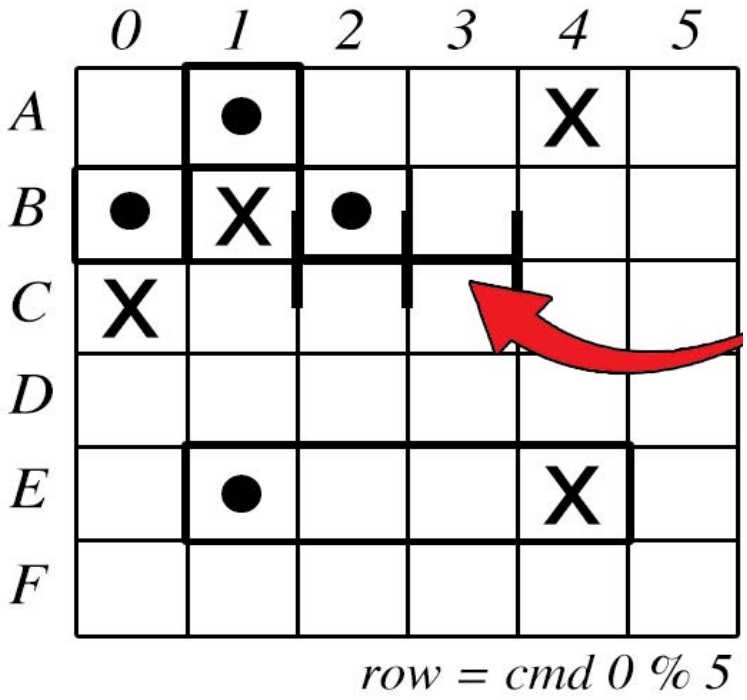


AN INTRO TO GAME LOGIC IN C++

By Chad Jordan - November 25th, 2006

In this guide you will learn:

1. The fundamentals of Propositional Dynamic Logic in game theory
2. The fundamentals of the Object-Oriented Paradigm *(OOP)*
3. Basic data structures and algorithms for game development
4. The use of constructors and destructors with objects and classes
5. Abstract Data Types *(ADT)* with public and private functions, and binary search trees
6. A complete implementation of Artificial Intelligence for a Battleships game in C++

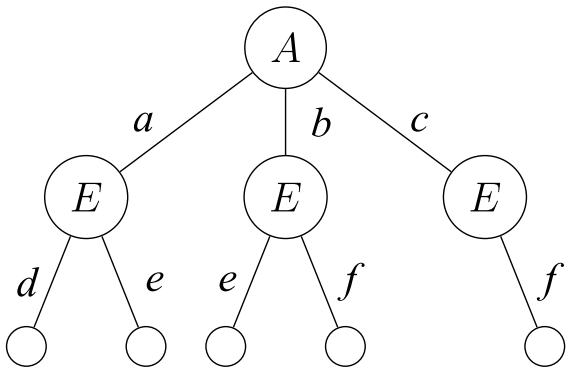
Introduction

While this guide provides an introductory level of information, game logic is considered one of the most difficult areas of programming. Therefore, it's best to treat this guide as a gateway for programmers that are already at a beginner to intermediate level. For my code examples, I'll be using a simple code editor in Linux called Vim. With that in mind, let's begin.

In any aspect of programming, there is always a requirement for systematic problem-solving and logical reason. The principles of how logic and object-oriented methodologies are applied to programming environments are very similar across the board. However, there are also many underlying complexities rooted in cognitive science that the required approach is not always so obvious. This is especially the case when considering game logic. Holistically, game logic on a cognitive scale is some of the most complicated puzzles that a developer can work with. The operational semantics of game theory can be used to reason with determining a game between two players. There are multiple approaches to how we can describe the computational meaning behind game theory, but Propositional Dynamic Logic (PDL) represents the states and events of dynamic systems through modal logic.

PDL is a thoroughly vast system of semantics including proof theory, axiomatizations, and their computational complexity. Put bluntly, it's designed for representing and determining the properties of functions in programs. Applied to game logic, a visual representation looks like this:

Game Structure



Event Model

Moves for Player A

$a - \overset{E}{\underset{}{-}} \; b \qquad c$

Moves for Player E

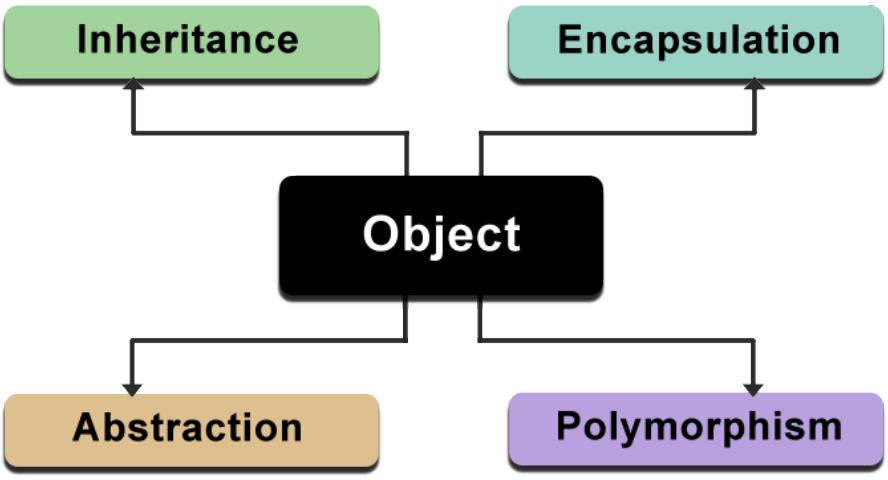
$d - \overset{A}{\underset{}{-}} \; e \qquad f$

Imagine this tree represents a game between two players, in this instance, A & E. At their turn, player '*E*' faces an array of five hypothetical available actions, but is contingent on the decision

that player '*A*' makes. It goes back to the age-old discussions that so many of us have faced when we were younger when we played against an opponent. *"Oh, I should have moved there, that would have changed the outcome in my favor."* Sound familiar? That mentality is referred to as counterfactual conditionals. This analysis also gives way to multiple deliberations, procedures, observations, and interfaces known as epistemic game theory. This area of research encompasses an array of various mathematical frameworks when analyzing games. We have to consider the stages of decision making when we have incomplete information, recall when how to learn from past mistakes, and mixed strategies based on our opponent/s. There are hundreds of PDL formulas that can affect the outcome of the deliberation process. This is especially the case for abstract (raw strategy) games that are primarily designed for analysis and deliberation such as Chess, Reversi, and Go.

Fundamentals in OOP

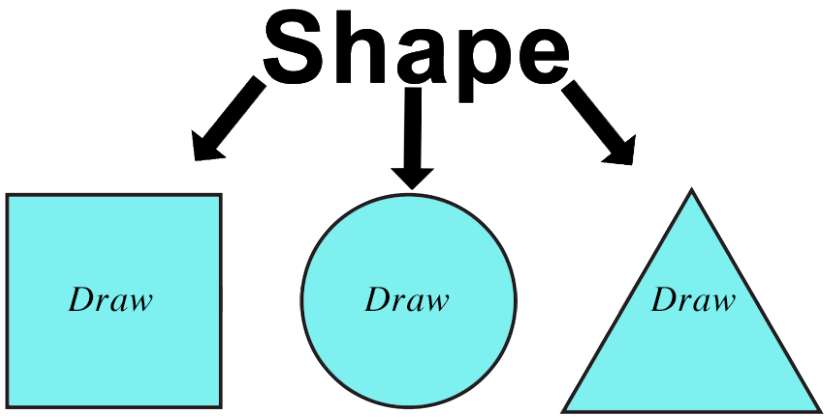
There are plenty of other forms of strategy and puzzles that give way to propositional dynamic, modal, and systematic logic. For the purposes of this article, I'll be looking at Object-Oriented Programming. When moving from procedural programming to object-oriented programming, the advantage is that the code and the operations that manipulate the code are both encapsulated within the object. In other words, when an object is transported across a network, the entire object (including the data and its behaviors) goes with it. The next question you may have is, *What is an object?* Objects are the building blocks of an OO program. Object data and object behavior is exactly what you would expect. Object data is stored within the object and represents the state of the object. Object behavior is what the object can do. Within these objects are subsets of other data attributes, methodologies, interfaces, and classes. A class is the blueprint of an object. When you instantiate an object, you apply a class as the basis for how the object is built. The Object-oriented thought process consists of four primary concepts: Inheritance, Encapsulation, Abstraction, and Polymorphism.



With regard to these four primary concepts in OOP, inheritance allows a class to inherit the attributes and methods of another class. In this instance, we can create brand new classes by abstracting out common attributes and behaviors. When it comes to encapsulation, one of the

advantages of using objects is that the object doesn't need to reveal all of its attributes and behaviors. Good OO design only allows objects to reveal the interfaces needed to interact with it. Therefore, details that are irrelevant to the use of an object should only be hidden from other objects. Abstraction defines a model to create the component of an application. It is the process of hiding the internal details of an application. This is accomplished using abstract classes and interfaces. Polymorphism literally means many shapes. While it's closely related to inheritance, its often cited separately as one of the most important and powerful advantages of object-oriented technologies.

In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. For example, consider the *Shape* class and the behavior called *Draw*. When you tell someone to draw a shape, their first question should be, "What shape?" You cannot draw a shape, since it's an abstract concept, there must first be a specification of a concrete shape such as *Circle*. Even though *Shape* has a *Draw()* method, *Circle* overrides this method and provides its own *Draw()* method. Overriding basically means replacing an implementation of a parent with one from a child. For example, suppose you have an array of three shapes – Square, Circle, and a Triangle:

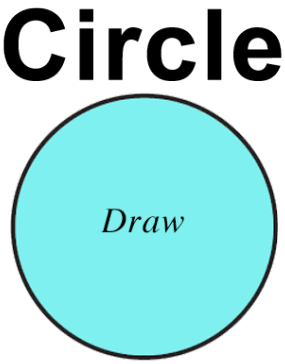



```
public abstract class Shape {
    private double area;
    public abstract double getArea();
}
```

Even though we treat them as shape objects, and send a draw message to each shape object, the end result is still different for each because objects in the array provide the actual implementations. In short, each class is able to respond differently to the same *Draw* method and draw itself. This is what is meant by polymorphism. The *Shape* class in the above example has an attribute called *area* that holds the value for the area of the shape. The method *getArea()* includes an identifier called *abstract*. When a method is defined as *abstract*, a subclass must provide the implementation for this method; in this case, *Shape* is requiring subclasses to provide a *getArea()* implementation. This next example is extra important. We can create a class called *Circle* that inherits from *Shape* (the keyword, **'extends'** signifies that *Circle* inherits from Shape):

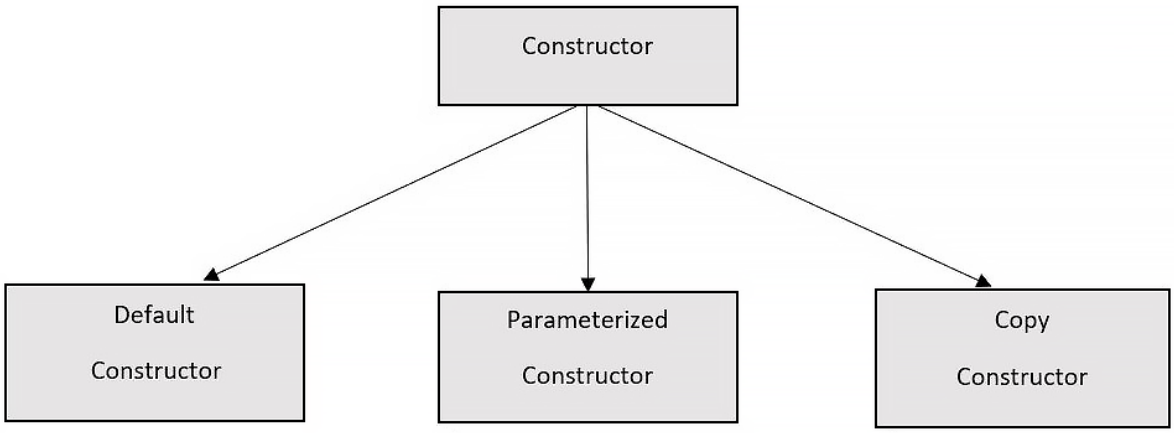
```
public class Circle extends Shape {
    double radius;
    public Circle(double r) {
        radius = r;
    }

public double getArea() {
        area = 3.14*(radius*radius);
        return (area);
    };
}
```



Constructors

We introduce a new concept here called a **constructor**. The *circle class* has a method with the same name, *Circle*. When the names are the same and no return type is provided, the attribute is a special method called a constructor. Constructors are considered advanced object-oriented concepts for structured programming and are also good for performing initializations. There are three main types of constructors: Default, Parameterized, and Copy.

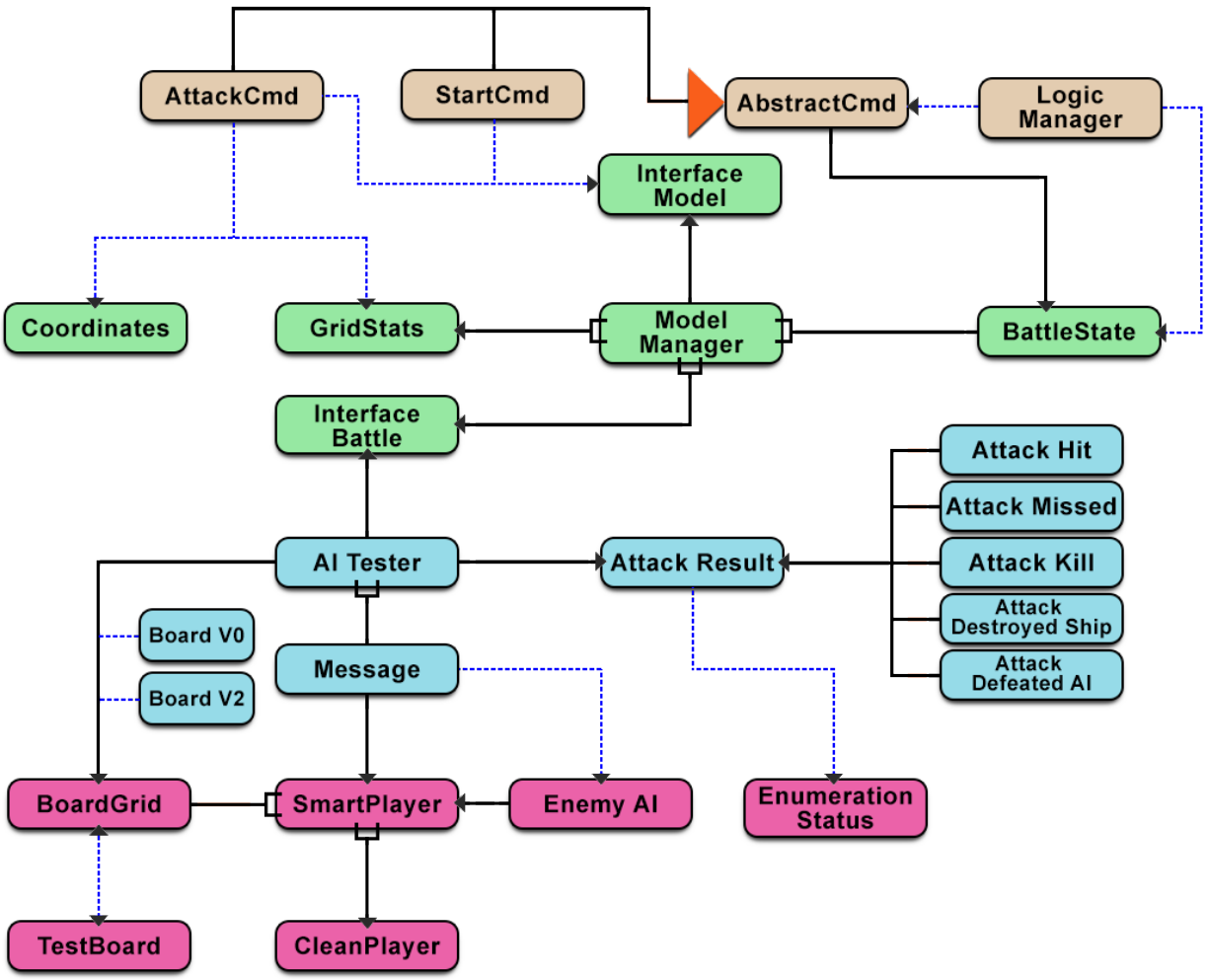


The default constructors do not take any argument and have no parameters. Parameterized constructor arguments help to initialize an object when it is created. Copy constructors are member functions that initialize objects using another object of the same class. In some of my upcoming code examples with my Battleships game, we will see first-hand how constructors can be implemented and assist in the game development process.

Data structures at their core, are the building blocks for software engineering. They define how data is arranged in memory and can be operated on by using various algorithms. An algorithm is code that manipulates data in data structures. In short, an algorithm is a list of rules and instructions that a computer needs in order to carry out various tasks. On their own, we know data structures are arrangements of data in memory, but in-game development when combined with special-purpose algorithms can be processed and used in very unique ways for gaming. Data structures drive many gaming experiences with artificial intelligence. AI systems can be simple, or very complex. Even in simple AI systems data structures and algorithms are

used to control the behavior of dynamic game elements. However, the more complex AI becomes, speed becomes more of an issue. The reason for this lag is because the more complicated the algorithms, the more processing time is required to complete operations. When you consider that numerous objects have to execute individual AI algorithms at once, you can find yourself having to balance realism with performance. This is one example of why I personally prefer C++ since it's one of the faster languages for processing. It's also another reason that AI is typically not as complex in games as it is with real-world applications like robotics, healthcare technologies, self-driving cars, and other software applications.

Good algorithms can take time to create, especially for game development. The more preparation you put into your algorithms, the better your implementation will be. The cornerstone of any computer application whether strictly practical or for gaming requires proper planning of everything you will need in the program in order for it to run as it should. With this in mind, it's vital to have a well-thought-out diagram to follow when implementing your algorithm into your game. The following flow chart is a logic model that I made for my AI Battleships game:



Using Public & Private Functions

At the start of my development, I know that I need to start creating my variables, integers, booleans, and other constant data types that I'll be referencing later in the program. Therefore, I've created a header file to store them. In a game of Battleships, there needs to be a board that the player sees, and a board that the game controller sees. There are also variable declarations for the characters that display on screen when certain functions are triggered and checking if there's a win, loss, or quitting.

```
1
2  #ifndef DEFINES_H
3  #define DEFINES_H
4
5  using namespace std;
6
7  class BattleShipsV0 {
8
9      public:
10         BattleShipsV0( int size );
11         void play();
12
13     private:
14         void initialize(char **board);
15         void placeShip(int size, int startRow, int startCol, int direction);
16         void showBoard(char **board, string boardID);
17         void clearScreen();
18         void processShot(int row, int col);
19         bool checkIfWon();
20
21         char **playerBoard;
22         char **controllerBoard;
23         int boardSize;
24
25         const char SHOT = '^';
26         const char DUPLICATE_SHOT = '!';
27         const char INVALID_SHOT = (char) 0;
28         const char HIT = 'H';
29         const char KILL = 'K';
30         const char MISS = '*';
31         const char SHIP = 'S';
32         const char WATER = '~';
33         const char MARK = 'a';
34         const static int HORIZONTAL = 0;
35         const static int VERTICAL = 1;
36
37         bool won;
38         bool lost;
39         bool tie;
40         bool quitting;
41 };
42
43 #endif
```

Next, I prep my other header file for more source declarations for BoardV2. I define constructors, destructors, and copy constructors for board length == width and height. The assignment operator *(operator=)* is used to copy values from one object to another already existing object. I overload the assignment operator to be used to create an object just like the copy constructor.

Starting on **line 17** I also create general public functions for implementing encapsulation via the access specifiers. In OOP, encapsulation represents binding data and functions into one container. Just as I mentioned above regarding encapsulation, this way all functions and classes will be hidden inside the container of the data and the way the functions process the data.

Beginning on **line 29** I put the prototypes for the private data and helper functions here for the control board, the player board, and the label board. Afterward, we use the #endif directive to end multiple inclusion control. If there's an expression written after the *#if* has a nonzero value, the line group immediately following the *#if* directive is kept in the translation unit.

```
 1
 2 #ifndef BOARDV2_H
 3 #define BOARDV2_H
 4
 5 #include <cstdlib>
 6 #include "defines.h"
 7
 8 using namespace std;
 9
10 class BoardV2 {
11     public:
12         BoardV2( int length);
13         ~BoardV2();
14         BoardV2(const BoardV2& other);
15         void operator=(const BoardV2& other);
16
17         char getOpponentView(int row, int col);
18         char getOwnerView(int row, int col);
19         char processShot(int row, int col);
20                 bool isSafe(int row, int col, int length, bool horiz);
21                 bool markShip(int row, int col, int length, bool horiz, char &mark);
22                 bool placeShip(int length, char &mark);
23         bool hasWon();
24                 bool isKilled(int row, int col);
25                 void markAsKilled(int row, int col);
26         const static int MaxBoardSize = 10;
27
28     private:
29         int boardSize;
30         char Board1[MaxBoardSize][MaxBoardSize];
31         char Board2[MaxBoardSize][MaxBoardSize];
32         char Board3[MaxBoardSize][MaxBoardSize];
33 };
34
35 #endif
36
```

Now it's time to begin our programming process by creating a shell for the implementation of the board. We do this by creating a .cpp file and writing the essentials for the Battleships. As mentioned in my previous guide called ***'Understanding Programming'*** we include the general **<iostream>** library for sending data to and from the standard streams input, output, error, and log respectively. I also reference my header file that I created above so that I can reference the characters, private, and public data attributes. Next, I start prepping the board by declaring the integer *boardSize* and converting it as a constructor, creating a player board, followed by the controller's board, and declaring that there has not been a winner or a quitter yet. The board must begin empty with only water, therefore we initialize the board to just water.

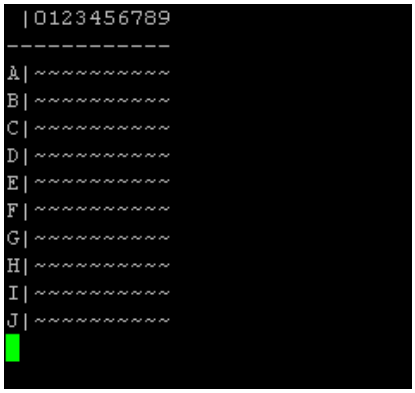
```
 1
 2 #include <iostream>
 3 #include "BattleShipsV0.h"
 4
 5 using namespace std;
 6
 7 BattleShipsV0::BattleShipsV0(int boardSize) {
 8         BattleShipsV0::boardSize = boardSize;
 9
10 playerBoard = new char *[boardSize];
11 for(int i=0; i<boardSize; i++) {
12         playerBoard[i] = new char[boardSize];
13 }
14 initialize(playerBoard);
15
16 controllerBoard = new char *[boardSize];
17 for(int i=0; i<boardSize; i++) {
18         controllerBoard[i] = new char[boardSize];
19 }
20 initialize(controllerBoard);
21
22 won = false;
23 quitting = false;
24 }
25
26 void BattleShipsV0::initialize(char **board) {
27     for(int row=0; row<boardSize; row++)
28         for(int col=0; col<boardSize; col++)
29             board[row][col] = WATER;
30 }
31
```

As we know from a game of Battleship, there is a set number of spaces on the board, or grid if you will, that determines where players can launch torpedo's in an attempt to hit the other player. The for loops iterate through the sections of *boardSize* for a fixed number of times as long as the test condition is true. In for loops, the initialization part is for declaring and initializing any loop control variables. The conditional part of the for loop must be true for the body of the loop to be executed.

Next, we show the program how we want to place ships. The parameters rowVector and colVector are used to determine the direction of the ships from the starting points of row/col. The showBoard function allows the program to display the specified board, then we clear the screen.

```
32 void BattleShipsV0::placeShip(int size, int startRow, int startCol, int direction) {
33     if(direction==HORIZONTAL) {
34                 for(int col=startCol; col<startCol+size; col++) {
35                     controllerBoard[startRow][col] = SHIP;
36                 }
37     }
38     else if(direction==VERTICAL) {
39                 for(int row=startRow; row<startRow+size; row++) {
40                     controllerBoard[row][startCol] = SHIP;
41                 }
42     }
43     else {
44         cerr << "\nInvalid ship direction: " << direction << endl;
45     }
46 }
47
48 void BattleShipsV0::showBoard(char **board, string boardID) {
49         cout << boardID << endl;
50         cout << " |";
51         for(int count=0; count<boardSize; count++) {
52                 cout << count;
53 }
54 cout << endl;
55
56 for(int row=0; row<boardSize+2; row++) {
57         cout << '-';
58 }
59 cout << endl;
60
61 for(int row=0; row<boardSize; row++) {
62         cout << char(('A'+row)) << '|';
63         for(int col=0; col<boardSize; col++) {
64                 cout << board[row][col];
65         }
66 cout << endl;
67 }
68 }
69
70 void BattleShipsV0::clearScreen() {
71     int i;
72     for( i=0; i<25; i++ ) {
73         cout << endl;
74     }
75 }
```

This is our current output as-is with the empty board of water.



This section of code checks all rows and columns on the controllerBoard for any pieces unhit ships, and if any ships are unhit, then the notification is sent to the user. Otherwise, if nothing is unhit equal or greater than (=>) everything being hit equal or greater than (=>) then there is a victory. The switch statement is a multiway branch construct that translates into a jump *(or branch)* table. They're primarily used to reduce repetitive coding and provide more clarity and faster processing through the compiler.

```
 76
 77 void BattleShipsV0::processShot(int row, int col) {
 78         if( row < 0 || row >= boardSize ) {
 79                 cout << "row outside of range!" << endl;
 80                 return;
 81         }
 82         if( col < 0 || col >= boardSize ) {
 83                 cout << "column outside of range!" << endl;
 84                 return;
 85         }
 86     switch( controllerBoard[row][col] ) {
 87                 case SHIP:
 88                         cout << "It's a hit!" << endl;
 89                         controllerBoard[row][col] = HIT;
 90                         playerBoard[row][col] = HIT;
 91                         break;
 92                 case HIT:
 93                 case MISS:
 94                         cout << "You already shot there" << endl;
 95                         break;
 96                 case KILL:
 97                         cout << "Kill not currently implemented" << endl;
 98                         break;
 99                 case '.':
100                         cout << "Splash." << endl;
101                         controllerBoard[row][col] = MISS;
102                         playerBoard[row][col] = MISS;
103                         break;
104                 default:
105                         cerr << "Invalid shot coordinates (row=" << row << ",col="
    << col << ")." << endl;
106                         break;
107     }
108 }
109
110 bool BattleShipsV0::checkIfWon() {
111     for(int row=0; row<boardSize; row++) {
112                 for(int col=0; col<boardSize; col++) {
113
114                         if( controllerBoard [row][col] == SHIP ) {
115                                 return false;
116                         }
117
118                 }
119     }
120
121     return true;
122 }
```

This function starts by allowing the user to create 4 ships on the board, two with 4 cells in length, and two with 3 cells in length. The next section is a mix of do while and another switch statement that provides the variable input and output to the screen for the playerBoard and controllerBoard data attributes.

```
123
124 void BattleShipsV0::play() {
125
126 placeShip(4, 1,1, HORIZONTAL);
127 placeShip(4, 2,0, HORIZONTAL);
128 placeShip(3, 2,3, VERTICAL);
129 placeShip(3, 0,0, VERTICAL);
130 clearScreen();
131
132 char aChar;
133 int col;
134
135 do
136 {
137 showBoard(playerBoard, "Player's view of board");
138 cout << "Enter a shot (ex. A3, 'P' to peek, or 'Q' to quit): " << endl;
139 cin >> aChar;
140 switch(toupper(aChar) )
141 {
142 case 'P':
143         showBoard(controllerBoard, "Opponent's view");
144 break;
145 case 'Q': quitting = true;
146         break;
147 default:
148         cin >> col;
149         processShot(toupper(aChar)-'A',col);
150         won = checkIfWon();
151         break;
152 }
153 cin.ignore(100,'\n');
154 } while (!( won || quitting));
155 if( won )
156 {
157         cout << "Congratulations. You have defeated the enemy." << endl;
158         showBoard(playerBoard, "Player's view of board");
159 }
160 else
161 {
162         cout << "Quitter. Sorry that you can't handle a real game. ;-)" << endl;
163 }
164 cout << endl;
165 }
```

In this case of the switch while loop, the value of the expression is true, so the body of the while loop is executed. Another advantage of using do while loops also make our code more readable and it's better to execute at least one control loop prior to keyboard input. This is also better for the user because the program will run until the user decides to quit the program or the goal within the program is achieved.

Now that the shell implementation *(BoardV0)* is created, it's time to create a declaration source file for BoardV2. I include my standard libraries and header files for reference, I also include the **<ctime>** library. This header file declares a set of functions, macros and types to work with date and time. Starting on **line 12** I'm creating constructors for length == height and width. Setting length to boardSize tells how much of the board will be used and then the for loop initializes each position in each board to water. On **line 22** I set a destructor since nothing is being dynamically allocated. The copy constructor on **line 24** is copying boardSize from *'other'* to newBoard then the for loop is moving through each location on the board, and then moving the data from one board to the other.

```
 1 #ifndef BOARDV2_CPP
 2 #define BOARDV2_CPP
 3
 4 #include <cstdlib>
 5 #include <iostream>
 6 #include "defines.h"
 7 #include "BoardV2.h"
 8 #include <ctime>
 9
10 using namespace std;
11
12 BoardV2::BoardV2( int length ) {
13         boardSize = length;
14         for( int i = 0; i < boardSize; i++ ) {
15                 for( int j = 0; j < boardSize; j++ ) {
16                         Board1[i][j] = WATER;
17                         Board2[i][j] = WATER;
18                 }
19         }
20 }
21
22 BoardV2::~BoardV2(){}
23
24 BoardV2::BoardV2(const BoardV2& other) {
25         boardSize = other.boardSize;
26         for( int i = 0; i < boardSize; i++ ) {
27                 for( int j = 0; j < boardSize; j++ ) {
28                         Board1[i][j] = other.Board1[i][j];
29                         Board2[i][j] = other.Board2[i][j];
30                 }
31         }
32 }
33
34 void BoardV2::operator=(const BoardV2& other) {
35         boardSize = other.boardSize;
36         for( int i = 0; i < boardSize; i++ ) {
37                 for( int j = 0; j < boardSize; j++ ) {
38                         Board1[i][j] = other.Board1[i][j];
39                         Board2[i][j] = other.Board2[i][j];
40                 }
41         }
42 }
```

The overloaded assignment operator on **line 34** is for creating objects similar to the copy constructor so that we can copy values from Board1 to Board2.

Here we are setting up general-purpose functions with count-controlled loops. The Boolean parameters on **line 49** are randomly giving a zero or a one value which is typecast to true or false. We iterate through to board searching for portions of the ship in row & col.

```
45 bool BoardV2::placeShip(int length, char &mark){
46         int row;
47         int col;
48         srand(time(NULL));
49         bool horiz = bool(rand()%2);
50         for( int i = 0; i < 20; i++ ) {
51                 if( length > boardSize )
52                 return false;
53                 if( horiz ) {
54                         col = rand()% boardSize;
55                         row = rand()% boardSize;
56                 }
57                 else {
58                         col = rand()% boardSize;
59                         row = rand()% boardSize;
60                 }
61                 if( isSafe( row, col, length, horiz )) {
62                         markShip( row, col, length, horiz, mark );
63                         return true;
64                 }
65         }
66                         return false;
67 }
```

Here we print out the character column divider and iterate through boardSize for BoardV2 until we return a Boolean value for row and column.

```
69 char BoardV2::getOpponentView(int row, int col) {
70         cout << "OpponentsView" << endl;
71         cout << "|";
72
73 for(int count=0; count<boardSize; count++) {
74     cout << count;
75 }
76 cout << endl;
77
78 for(int i=0; i<boardSize+2; i++) {
79     cout << '-';
80 }
81 cout << endl;
82
83 for(int row2=0; row2<boardSize; row2++) {
84     cout << char(('A'+row2)) << '|';
85     for(int col2=0; col2<boardSize; col2++) {
86         cout << Board2[row2][col2];
87     }
88         cout << endl;
89 }
90 return Board2[row][col];
91 }
```

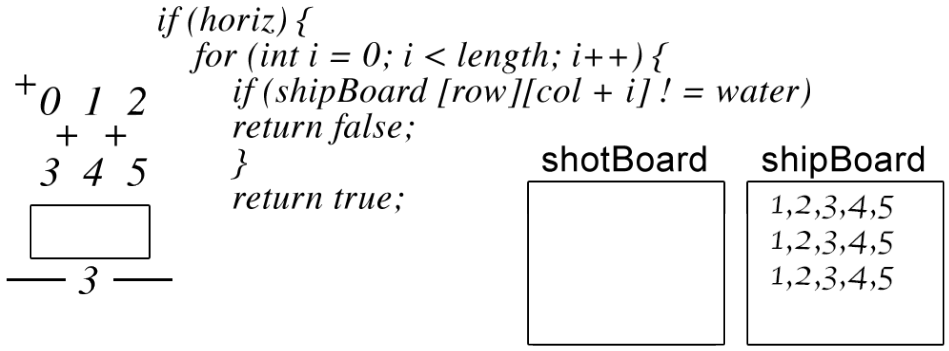
Here the same process continues. I'm still continuing to print out more dividers and characters for opponent and owner views, iterating through boardSize with count-controlled loops.

```
 93 char BoardV2::getOwnerView(int row, int col) {
 94         cout << "OwnerView" << endl;
 95     cout << "|";
 96
 97 for(int count=0; count<boardSize; count++) {
 98     cout << count;
 99 }
100 cout << endl;
101
102 for(int i=0; i<boardSize+2; i++) {
103     cout << '-';
104 }
105 cout << endl;
106
107 for(int row2=0; row2<boardSize; row2++) {
108     cout << char(('A'+row2)) << '|';
109     for(int col2=0; col2<boardSize; col2++) {
110         cout << Board1[row2][col2];
111     }
112         cout << endl;
113 }
114 return Board1[row][col];
115 }
```

This next section tests what is at position row and col against that position in Board1 which holds the information regarding ships. We check for a miss, or a kill.

```
117 char BoardV2::processShot(int row, int col) {
118         switch( Board1[row][col] ) {
119                 case WATER:
120                         Board1[row][col] = MISS;
121                         Board2[row][col] = MISS;
122                         return MISS;
123                 case HIT:
124                 case KILL:
125                 case MISS:
126                         return DUPLICATE_SHOT;
127                 case SHIP:
128                         if( isKilled(row, col)) {
129                                 markAsKilled(row, col);
130                                 return KILL;
131                         }
132                         else {
133                                 Board1[row][col] = HIT;
134                                 Board2[row][col] = HIT;
135                                 return HIT;
136                         }
137         }
138 }
```

The following function provides a visual representation for the length of shipBoard in a two-dimensional array on BoardV2 for row & col.




```
139 bool BoardV2::hasWon() {
140         for( int i = 0; i < boardSize; i++ ) {
141                 for( int j = 0; j < boardSize; j++ ) {
142                         if( Board3[i][j] >= 'a' && Board3[i][j] <= 'z' ) {
143                                 if( Board2[i][j] != KILL ) {
144                                         return false;
145                                 }
146                         }
147                 }
148         }
149 return true;
150 }
```

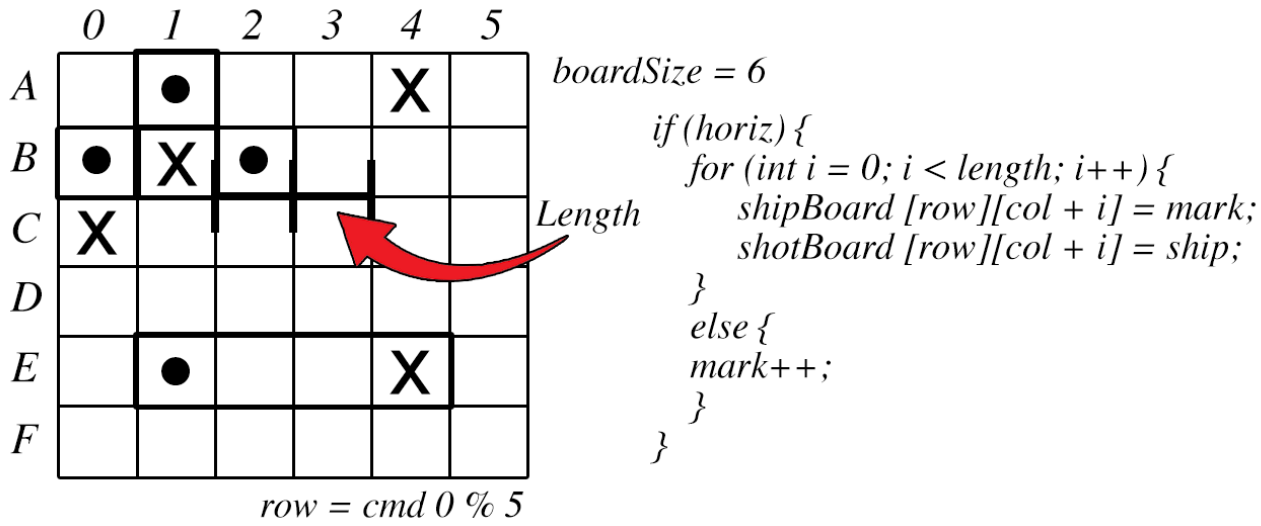
This next function is a counter for rows and columns and returns a Boolean value. Beginning on **line 152** we're putting in prototypes for private data and helper functions. For the third Boolean function beginning on **line 162** we're retrieving labels for the player to test against, and checks each position marked by a label and if it's a hit, it returns true.

```
152 bool BoardV2::isSafe( int row, int col, int length, bool horiz ) {
153         if( horiz ) {
154                 for( int i = 0; i < length; i++ ) {
155                         if( Board1[row][col + i] != WATER )
156                         return false;
157                 }
158                         return true;
159         }
160 }
161
162 bool BoardV2::isKilled( int row, int col ) {
163         char label = Board3[row][col];
164         int shipCount = 0;
165         for( int i = 0; i < boardSize; i++ ) {
166                 for( int j = 0; j < boardSize; j++ ) {
167                         if( Board3[i][j] == label && Board1[i][j] == SHIP ) {
168                                 shipCount++;
169                         }
170                 }
171         }
172         if( shipCount == 0 ) {
173                 return true;
174         }
175         else return false;
176 }
```

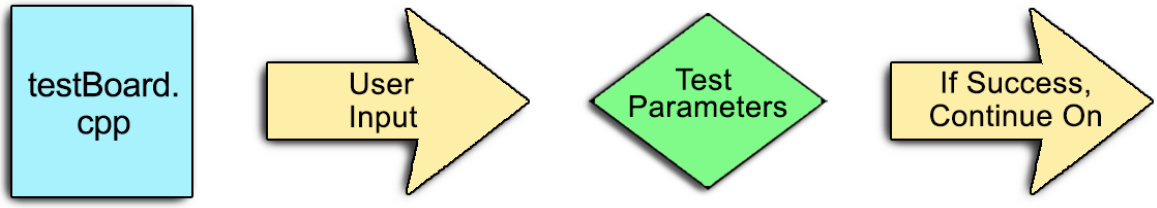
The last function for this file performs the same iterations as the previous ones except this time we're checking for if a ship is marked as killed. We can use this function for marking ships on the player board and labeling them in a parallel array. #endif is the end of multiple inclusion control.

```
177 void BoardV2::markAsKilled(int row, int col ) {
178         char label = Board3[row][col];
179         for( int i = 0; i < boardSize; i++ ) {
180                 for( int j = 0; j < boardSize; j++ ) {
181                         if( Board3[i][j] == label ) {
182                                 Board1[i][j] = KILL;
183                                 Board2[i][j] = KILL;
184                         }
185                 }
186         }
187 }
188 bool BoardV2::markShip( int row, int col, int length, bool horiz, char &mark ) {
189         if( horiz ) {
190                 if( row + length > boardSize ) return false;
191                         for( int i = 0; i < length; i++ ) {
192                                 Board1[row][col + i] = SHIP;
193                                 Board3[row][col + i] = mark;
194                         }
195         }
196         else {
197                 if( row + length > boardSize ) return false;
198                         for( int i = 0; i < length; i++ ) {
199                                 Board1[row + i][col] = SHIP;
200                                 Board3[row + i][col] = mark;
201                         }
202         }
203 }
204
205 #endif
```

I made this visual representation as a concept for the ships marked on the board. It provides the length of the ship in comparison to boardSize and if the shot marks a hit on the ship or not.



Now, with this portion of our board logic completed, we need to write a testBoard file to check the overall functionality of what we've implemented so far in our various .CPP files. This way we can execute the input and output based on what the user wishes to test.



In this next file we just need to test what we've created for the board thus far. To speed this up we can copy **lines 135 to 165** from the *BattleshipsV0.cpp* file into a new file called *testBoard.cpp* and this will help us save a lot of time.

```
#include <iostream>
#include "BoardV2.cpp"

using namespace std;

int main()
{
int col;
bool quitting = false;
bool won = false;
char mark = MARK;
char aChar;

BoardV2 Board(10);
do{
cout << "Enter a shot (ex. A3, 'P' to place ship, or 'Q' to quit): " << endl;
cin >> aChar;
switch(toupper(aChar) )
{
case 'P': cout << "enter ship length (2-4)"; cin >> col; while( col < 2 || col > 4 ) {
     cout << "only from 2-4"; cin >> col; } Board.placeShip(col, mark);
         mark++;
         Board.getOwnerView(toupper( aChar )-'A',col);
         break;
case 'Q': quitting = true;
    break;
default:
    cin >> col;
    cout << "Shot was" << Board.processShot(toupper(aChar)-'A',col);
        cout << "Opponent is" << Board.getOpponentView(toupper(aChar)-'A',col);
        cout << "Owner is" << Board.getOwnerView(toupper(aChar)-'A',col);
    won = Board.hasWon();
    break;
}
cin.ignore(100,'\n');
} while (!( won || quitting));
if( won )
{
    cout << "Congratulations. You have defeated the enemy." << endl;
}
else
{
    cout << "Quitter. Sorry that you can't handle a real game." << endl;
}
cout << endl;

}
```

As you can see from the above code, there's no need to re-write the wheel when we've already built it and we know it compiles with no errors. There were only a few small tweaks with Booleans and char data types that we returned to run the test board. With this in place, we can test for the input and output of our ship placement with respect to rows and columns, and finally, the response from our program for each type of input. Since this test program passes all input and output parameters correctly, it's time to start implementing the structure for the AI of our game. If you haven't created a separate directory for AI yet, now is a good time to create it and copy your defines.h file into it. Now we create another header file that defines a primitive message class and I've called mine Message.h. In this file we define constructors, general set and get functions for message type functions, row, column, and string functions.

```
 1
 2 #include <string>
 3 #include "defines.h"
 4
 5 using namespace std;
 6
 7 #ifndef MESSAGE_H
 8 #define MESSAGE_H
 9
10 class Message {
11     public:
12         Message( char messageType );
13         Message( char messageType, int row, int col, string str );
14
15         void setMessage( char messageType, int row, int col, string str );
16         void setMessageType( char messageType );
17         char getMessageType( );
18         void setRow( int row );
19         int getRow( );
20
21         void setCol( int col );
22         int getCol( );
23
24         void setString( string str );
25         string getString( );
26
27     private:
28         char messageType;
29         int row;
30         int col;
31         string str;
32 };
33
34 #endif
35
```

Next, we know we have to retrieve message data that will eventually get printed out to the screen for our AI. Therefore, we have to write a CPP file to accept the declarations that we created in the previous header file in order to initialize the object message type as well as the message data to the specified values and then return the data for row, col, and string. This portion of my program is displayed on the next page. The arrow operators in this next code example allow access to the pointer variables in the structures and unions.

```cpp
#ifndef MESSAGE_CPP
#define MESSAGE_CPP

#include "Message.h"

using namespace std;

Message::Message( char messageType ) {
    this->messageType = messageType;
    this->row = -1;
    this->col = -1;
    this->str = "";
}

Message::Message( char messageType, int row, int col, string str ) {
    setMessage( messageType, row, col, str );
}

void Message::setMessage( char messageType, int row, int col, string str ) {
    this->messageType = messageType;
    this->row = row;
    this->col = col;
    this->str = str;
}

void Message::setMessageType( char messageType ) {
    this->messageType = messageType;
}

char Message::getMessageType( ) {
    return messageType;
}

void Message::setRow( int row ) {
    this->row = row;
}

int Message::getRow( ) {
    return row;
}

void Message::setCol( int col ) {
    this->col = col;
}

int Message::getCol( ) {
    return col;
}

void Message::setString( string str ) {
    this->str = str;
}

string Message::getString( ) {
    return str;
}

#endif
```

Next, we create another header file to store our class for PlayerV1 to make choices on the board. It gets the players move choice, then is returned to the caller. The parameter then returns the Message, *move*. The *getMove* parameter is a pure virtual function. The Player class declares, but does not define it. That allows a class to force all derived classes to implement the functionality. The *moveResult* parameter informs the player of the result of a previous move. The player updates its internal representation of the opponent's board to reflect the result. The message param 'msg' will have the shot coordinates row, col, and the shot result available via the messageType.

```
 1
 2 #ifndef PLAYERV1_H
 3 #define PLAYERV1_H
 4
 5 #include "Message.h"
 6
 7 class PlayerV1 {
 8     public:
 9         virtual Message getMove() = 0;
10
11         virtual void moveResult( Message msg );
12
13     protected:
14         PlayerV1( int boardSize );
15
16         int boardSize;
17         const static int MaxBoardSize = 10;
18         char board[MaxBoardSize][MaxBoardSize];
19 };
20
21 #endif
```

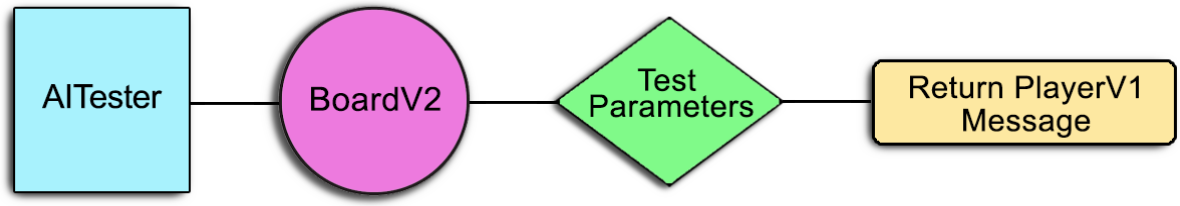
Here we set *boardSize*, and then initialize the board to water.

```
 1
 2 #ifndef PLAYERV1_CPP
 3 #define PLAYERV1_CPP
 4
 5 #include <iostream>
 6
 7 #include "PlayerV1.h"
 8
 9 PlayerV1::PlayerV1( int boardSize ) {
10     this->boardSize = boardSize;
11     for(int row=0; row<boardSize; row++) {
12         for(int col=0; col<boardSize; col++) {
13             board[row][col] = WATER;
14         }
15     }
16 }
17
18 void PlayerV1::moveResult( Message msg ) {
19     board[msg.getRow()][msg.getCol()] = msg.getMessageType();
20 }
21
22 #endif
```

Now is a good time to create the main driver of our Battleships implementations, the AI Tester. Just like my other header files, we make sure to account for all variables that we will need to reference during the AI gameplay.

```
1
2  #ifndef AITESTER_H
3  #define AITESTER_H
4
5  #include "Message.h"
6  #include "BoardV2.h"
7  #include "PlayerV1.h"
8
9  using namespace std;
10
11 class AITester {
12   public:
13     AITester(PlayerV1* player, BoardV2* playerBoard, string playerName,
14     int boardSize, bool silent);
15     void play(int delay, int& totalMoves);
16
17   private:
18     void placeShips(BoardV2* board);
19     void showBoard(BoardV2* board, bool ownerView, string playerName,
20     bool hLMostRecentShot, int hLRow, int hLCol );
21     void clearScreen();
22     void updateAI(PlayerV1 *player, BoardV2 *board);
23     void snooze(int seconds);
24     bool processShot(string playerName, PlayerV1 *player, BoardV2 *board,
25     int row, int col);
26
27     PlayerV1 *player;
28     BoardV2 *playerBoard;
29     string playerName;
30     int boardSize;
31     bool silent;
32     bool playerWon;
33 };
34
35 #endif
```

This diagram displays the interaction between the files and commands during the execution of the AI Tester and messages returned for the player board.



With this concept in mind, C++ implementation is required for our AI Tester.

We Include the previous header files to tie in the integers, Booleans, parameters, and constant values that we reference in our logic CPP files. Beginning on **line 15** we use arrow operators again to setup player 1 followed by general access functions from **lines 19 – 21**. Beginning on **line 24** we write a function to place a ship where rowVector and colVector are used to determine direction of the ships when starting from row and column.

```
 1
 2 #include <iostream>
 3 #include <string>
 4 #include <unistd.h>
 5
 6 #include "defines.h"
 7 #include "Message.h"
 8 #include "BoardV2.h"
 9 #include "AITester.h"
10
11 using namespace std;
12
13 AITester::AITester( PlayerV1* player, BoardV2* playerBoard, string playerName,
14     int boardSize, bool silent ) {
15     this->player = player;
16     this->playerBoard = playerBoard;
17     this->playerName = playerName;
18
19     this->boardSize = boardSize;
20     this->silent = silent;
21     this->playerWon = false;
22 }
23
24 void AITester::placeShips( BoardV2* board ) {
25     const int NumShips = 6;
26     string shipNames[NumShips]  = { "Submarine", "Destroyer", "Aircraft Carrier",
27                             "Destroyer 2", "Submarine 2", "Aircraft Carrier 2" };
28     int shipLengths[NumShips] = { 3,3,4,3,3,4 };
29
30     int maxShips = boardSize-2;
31     if( maxShips > NumShips ) {
32         maxShips = NumShips;
33     }
34
35     for( int i=0; i<maxShips; i++ ) {
36         if( board->placeShip(shipLengths[i]) == false ) {
37             cerr << "Couldn't place "<<shipNames[i]<<" (length "<<shipLengths[i]<<")"<<endl;
38         }
39     }
40 }
```

For our stand-alone statement we can have a static number of ships *(constant value)* that is set in stone in our header files so we won't change it later. We can set the ship names and their length and assign it back to an integer. If the ship placement is improperly set outside of its declared size, then the *cerr* data object will print an error message.

In this function, we return a Boolean value for hLMostRecentShot to check for a possible shot for row and column. Aside from the standard iteration from my other CPP files one of the big differences with this file is adding ship highlighting and we retrieve the data variable getOwnerView, and if it's not the owner view, we use arrow operators to get access to the opponent view and update the regular game value for row and col. Then, we clear the screen with a clearScreen function.

```
42 void AITester::showBoard(BoardV2* board, bool ownerView, string playerName,
43     bool hLMostRecentShot, int hLRow, int hLCol ) {
44     if( silent ) return;
45 
46     cout << playerName << endl;
47     cout << " |";
48     for(int count=0; count<boardSize; count++) {
49         cout << count;
50     }
51     cout << endl;
52 
53     for(int row=0; row<boardSize+2; row++) {
54         cout << '-';
55     }
56     cout << endl;
57 
58     for(int row=0; row<boardSize; row++) {
59         cout << (char)(row+'A') << "|";
60         for(int col=0; col<boardSize; col++) {
61             if( ownerView == true ) {
62 #ifdef _POSIX_SOURCE
63                 char value = board->getOwnerView(row,col);
64                 if( value >= 'a' && value <= 'f' ) {
65                     cout << "\033[7m" <<value<< "\033[0m";
66                 } else {
67                     cout << value;
68                 }
69 #else
70                 cout << board->getOwnerView(row,col);
71 #endif
72             } else {
73 #ifdef _POSIX_SOURCE
74                 if( hLMostRecentShot && row==hLRow && col==hLCol )
75                     cout << "\033[7m" <<board->getOpponentView(row,col) << "\033[0m";
76                 else
77 #endif
78                     cout << board->getOpponentView(row,col);
79             }
80         }
81         cout << endl;
82     }
83 }
84 
85 void AITester::clearScreen() {
86     if( silent ) return;
87 #ifdef _POSIX_SOURCE
88     cout << "\033[H\033[2J";
89 #else
90     for(int i=0; i<25; i++) cout << endl;
91 #endif
92 }
93 
```

The usleep variable takes the argument in microseconds, so we have to convert milliseconds to microseconds. Just like previous files, tweaking our switch statement for the current variables of processShot is the best method for user input.

```
 94 void AITester::snooze( int milliSeconds ) {
 95     long sleepTime = 1000 * milliSeconds;
 96     usleep(sleepTime);
 97 }
 98
 99 void AITester::updateAI(PlayerV1 *player, BoardV2 *board) {
100     Message killMsg( KILL, -1, -1, "");
101     for(int row=0; row<boardSize; row++) {
102         for(int col=0; col<boardSize; col++) {
103             if(board->getOwnerView(row,col) == KILL) {
104                 killMsg.setRow(row);
105                 killMsg.setCol(col);
106                 player->moveResult(killMsg);
107             }
108         }
109     }
110 }
111
112 bool AITester::processShot(string playerName, PlayerV1 *player, BoardV2 *board,
113     int row, int col) {
114     bool won = false;
115     if( !silent ) cout << "Processing " << playerName
116                        << "'s shot [" <<row<< "," <<col<< "]." << endl;
117     Message msg = board->processShot( row, col );
118     msg.setRow(row);
119     msg.setCol(col);
120     switch( msg.getMessageType() ) {
121         case MISS:
122             if( !silent ) cout << "Miss" << endl;
123             player->moveResult(msg);
124             break;
125         case HIT:
126             if( !silent ) cout << "Hit." << endl;
127             player->moveResult(msg);
128             break;
129         case KILL:
130             if( !silent ) cout << "It's a KILL! " << msg.getString() << endl;
131             won = board->hasWon();
132             updateAI(player, board);
133             break;
134         case DUPLICATE_SHOT:
135             if( !silent ) cout << "You already shot there." << endl;
136             break;
137         case INVALID_SHOT:
138             cout << playerName << " shot at invalid board coordinates [row="<<row<<
139                     ", col="<<col<<"]." << endl;
140             break;
141         default:
142             if( !silent ) cout << "Invalid return from processShot: "
143                                << msg.getMessageType() << "(" << msg.getString() <<
    ")" << endl;
144             break;
145     }
146
147     return won;
148 }
```

With access functions running the conversion of milli seconds to micro seconds, our delay function will work. Starting on **line 161** this code kicks in if running in batch mode but we want to see what is happening with the occasional game if playerBoard and processShot are in trouble at runtime. However, if there is not an error, the iterations properly run between playerBoard and processShot and then check for the amount of shots and ships that have been placed on the board.

```
149 void AITester::play( int milliSecondsDelay, int& totalMoves ) {
150     int maxShots = boardSize*boardSize*2;
151     int shotCount = 0;
152     totalMoves = 0;
153     clearScreen();
154     placeShips(playerBoard);
155
156     do {
157         clearScreen();
158
159         Message shot = player->getMove();
160         shotCount++;
161         if( silent && shotCount>70 ) {
162             silent = false;
163             milliSecondsDelay = 1000;
164         }
165         playerWon = processShot(playerName, player, playerBoard, shot.getRow(),
166         shot.getCol());
167
168         if( ! silent ) {
169             if( playerWon ) {
170                 showBoard(playerBoard, true, "Final status of " + playerName +
171                 "'s board", true, shot.getRow(), shot.getCol());
172             } else {
173                 showBoard(playerBoard, false, playerName + "'s Board", \
174                 true, shot.getRow(), shot.getCol());
175             }
176             if( milliSecondsDelay > 0 ) {
177                 snooze( milliSecondsDelay );
178             }
179         }
180
181         totalMoves++;
182     } while ( !playerWon && shotCount < maxShots );
183
184     if( ! silent ) {
185         cout << endl;
186         cout << "Moves = " << totalMoves << ", percentage of board shot at = "
187              << (100.0*(float)totalMoves)/(boardSize*boardSize) << "%." << endl;
188         cout << endl;
189         snooze( milliSecondsDelay );
190     }
191 }
```

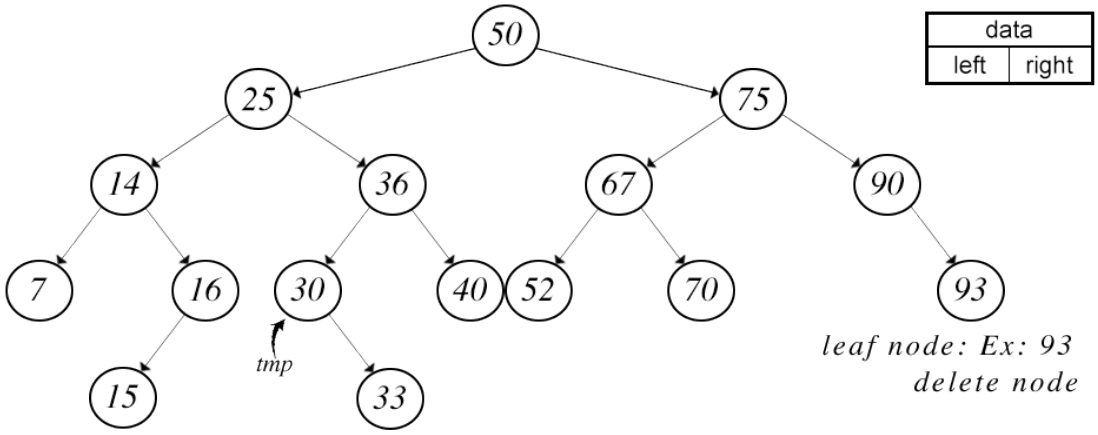
With this file complete, it's time to start prepping our multiple player AI's.

Algorithm Design With Binary Search Trees

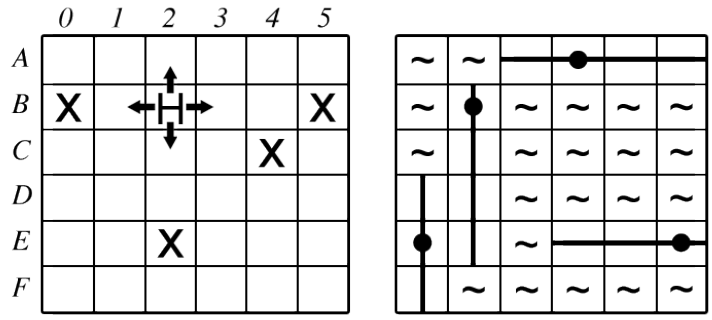
As far as our AI players are concerned, our Battleships game should have multiple levels of difficulty for our AI's. However, due to the extensive length of this guide, I will only be providing my smartest AI. In my AI Battleships game, there are three types of AI that I've created:

1) A gambling player that is sloppier and takes too many risks at random that I call gamblerPlayer.
2) A clean player that plays better than the gambler, and utilizes cleaner moves at a more intermediate level of intelligence which I call cleanPlayer.
3) Finally, I have my most intelligent player of the three that uses a scanning technique to look for ship length and shots already fired across the spaces of the board called smartPlayer.

Creating a smart AI that understands and performs the best methods of search patterns on a board grid first involves a human understanding how the computer already thinks and searches in a matrix. The computer will understand how it needs to search, but just like any other programming methodologies and algorithms, we have to create that communication for the computer so it can perform exactly what we want. Using binary search trees is a great way to create a thought structure and algorithmic approach to our smart player AI. Binary Trees are structure nodes that hold hierarchical relationships at different levels. We search through these trees in order to locate, retrieve, manipulate, and delete data. Here is an example of a binary search tree that I used for my smartPlayer AI:

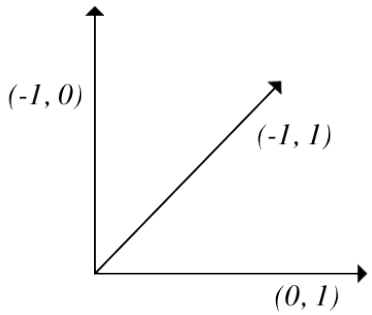


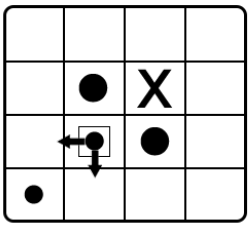
This next page has a more visual in-depth look at how logarithmic principles are applied to my algorithm for searching through the board grid for hits, misses, or water. The following visual representation is some of the most important parts of the thought process for my smartPlayer AI.




```
void hunter (startR, shotC, &shotR, &shotC) {
   if (goNorth (startR, shotC, shotR, shotC)) {
      return;
   }
   else if (goEast (startR, shotC, shotR, shotC)) {
      return;
   }
         continual functions (goSouth)
                             (goWest)
}
```

```
bool goNorth(startR, shotC, & shotR, & shotC) {
    for (i = 1; startR - i < 0; i++) {
        switch ( board[startR - i] [startC] ) {
          case WATER:
                shotR = startR - i;
                shotC = startC;
        return true;
          case MISS:
          case HIT:
          case KILL:
            return false;
        }
      return false;
  }
```





```
bool genericHunter (startR, startC, shotR,
     shotC, deltaX, deltaY) {
   for (int i = 1; isOnBoard (startR + i (i * deltaX)
                              startC + i (i * deltaY)); i++) {
      }
   }
}
```

Searching Performance

Unsorted Data 0 (N)
Linear Search

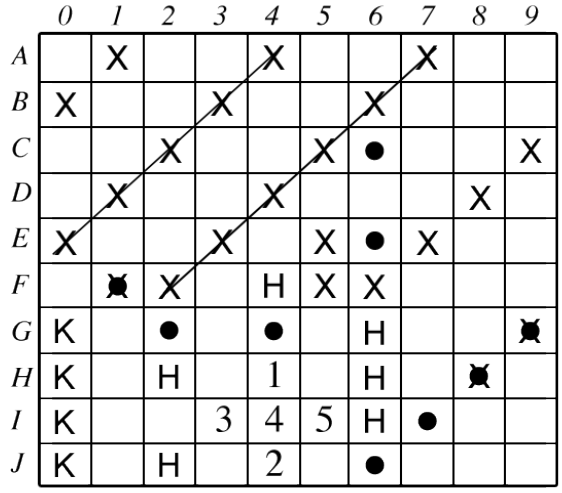
Sorted Data 0 ($Log_2 N$)
Binary Search
Log_2 (1024) = 10 2^{10}= 1024

Search Random Data 0 (K) (constant)
Hashing ← *if $Log_N N = 1$*

Sorting Performance

0 (N^2) sorts bubble
Selection Insertion

0 (N $Log_2 N$) sorts N
quicksort
heapsort $Log_2 N$
mergesort x = 8
$Log_2 8 = 3$
x = 3



Logarithmic equations are good methodologies to apply for this particular algorithm of my AI. The same is said for how we apply the principles of binary search trees with the object-oriented paradigm in our algorithms. So the question becomes what does this all mean to those who don't study or research it? Essentially, I'm creating an advanced scanner that runs multiple iterations and levels of scans through the grid of our board searching for a hit or miss until there is a kill. With these concepts in mind, it's time to implement my smartPlayer AI.

Implementing the AI

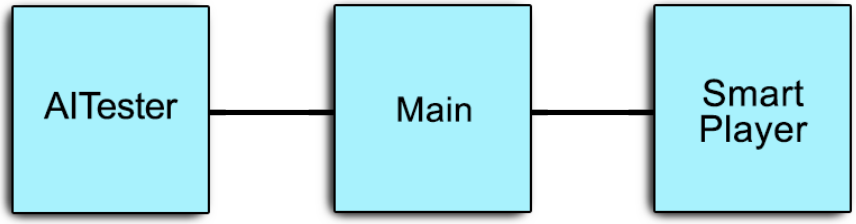
This first portion of the program gets the computer's shot choice, then returns it to the caller. The most important parts of the return message are the row and column values. Position 0 of the int array should hold the row and position 1 of the column.

```
 1
 2 #include <iostream>
 3 #include <cstdlib>
 4 #include <cassert>
 5
 6 using namespace std;
 7
 8 #include "SmartPlayer.h"
 9
10 SmartPlayer::SmartPlayer( int boardSize ):PlayerV1(boardSize) {
11     lastRow = 0;
12     lastCol = -3;
13 }
14
15 Message SmartPlayer::getMove() {
16         int hitRow;
17         int hitCol;
18         if( HitFound( hitRow, hitCol )== true ) {
19                 int shotRow;
20                 int shotCol;
21                 if(HuntShip(lastRow, lastCol, shotRow, shotCol))
22                 {
23                         Message result( SHOT, shotRow, shotCol, "Bang" );
24                 return result;
25                 }
26         }
27         do {
28                 lastCol += 3;
29                 if( lastCol >= boardSize ) {
30                         lastCol = (lastRow + 1) % 3;
31                         lastRow++;
32                 }
33                 if( lastRow >= boardSize ) {
34                         lastCol = 0;
35                         lastRow = 0;
36                 }
37         }
38         while( board[lastRow][lastCol] == MISS || board[lastRow][lastCol] == KILL );
39
40     Message result( SHOT, lastRow, lastCol, "Bang" );
41     return result;
42 }
43
44 bool SmartPlayer::HitFound(int &row, int &col) {
45         for( int i = 0; i < boardSize; i++ ) {
46                 for( int j = 0; j < boardSize; j++ ) {
47                         if( board[i][j]==HIT ) {
48                                 row = i;
49                                 col = j;
50                                 return true;
51                         }
52                 }
53         }return false;
54 }
```

The HitFound function for SmartPlayer is straightforward. The AI will search through boardSize and if it finds a part of a ship, then we've made a hit, and the AI will continue the sequence of hits for the length of the ships until we've made a kill. Otherwise, the AI will continue to scale the board. HuntShip is a function that is told how to search the board for ships from North, South, East, and West directions for all rows and columns on the board.

```
56 bool SmartPlayer::HuntShip(int startRow, int startCol, int &endRow, int &endCol) {
57         // moving north
58         endRow = startRow;
59         endCol = startCol;
60         while( endRow >= 0 && board [endRow][endCol] == HIT ) {
61                 endRow--;  // moves hunter north
62         }
63         if( !(endRow < 0) && board [endRow][endCol] == WATER ) {
64                 return true;
65         }
66         // moving south
67         endRow = startRow;
68         endCol = startCol;
69         while( endRow < boardSize && board [endRow][endCol] == HIT ) {
70                 endRow++;  // moves hunter south
71         }
72         if( !(endRow >= boardSize) && board [endRow][endCol] == WATER ) {
73                 return true;
74         }
75         // moving east
76         endRow = startRow;
77         endCol = startCol;
78         while( endCol < boardSize && board [endRow][endCol] == HIT ) {
79                 endCol++;  // moves hunter east
80         }
81         if( !(endCol >= boardSize) && board [endRow][endCol] == WATER ) {
82                 return true;
83         }
84         // moving west
85         endRow = startRow;
86         endCol = startCol;
87         while( endCol >= 0 && board [endRow][endCol] == HIT ) {
88                 endCol--;  // moves hunter west
89         }
90         if( !(endCol < 0) && board [endRow][endCol] == WATER ) {
91                 return true;
92         }
93         return false;
94 }
95
```

The final piece to this program, is connecting everything together with our main.cpp file.



These include declarations will access and setup the random number generator. The **<cctype>** header declares functions to classify and transform individual characters such as checking whether a character is uppercase or not. The **<cassert>** header (from the C standard **<assert.h>** header) declares the assert macro and can be included in multiple instances for different error handling instances. The **<ctime>** header file is used for time manipulation like retrieving date and time information.

The Battleships project header files are of course included from all of the earlier variables and data types that we also need now when writing our main function for program initialization. This main file is basically one large function.

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cctype>
 4 #include <cassert>
 5 #include <cstdlib>
 6 #include <ctime>
 7
 8 #include "BoardV2.h"
 9 #include "AITester.h"
10 #include "PlayerV1.h"
11
12 #include "SmartPlayer.h"
13 #include "CleanPlayer.h"
14 #include "GamblerPlayer.h"
15
16 using namespace std;
17
18 int main() {
19     PlayerV1 *player=NULL;
20     BoardV2 *board;
21     string playerName;
22     int boardSize;
23     bool silent = false;
24     float secondsPerMove = 1;
25     int gameMoves = 0;
26     double totalMoves = 0.0;
27     int totalGames = 0;
28     char displayChoice = 'y';
29
30     srand(time(NULL));
```

Line 30 seeds the random generator. This only needs to occur once per execution of the program being run. In order to generate random-like numbers, in our case ship coordinates and lengths of ships in their randomly placed positions on the board.

Starting on **line 32** the program prints out a message to the screen asking the user what size board they want to play on. Any size less than 3x3 or greater than 10x10 will return false and exit the program. Afterward, the user is prompted to choose which AI they would like to test.

```
31
32      cout << "What size board would you like? Lower than 3-10 exits.";
33      cin >> boardSize;
34      if ( !cin || boardSize < 3 || boardSize > 10 ) {
35          cout << "Exiting" << endl;
36          return 1;
37      }
38      cout << endl << "Which AI would you like to test? " << endl
39           << "1\tSmart Player" << endl
40           << "2\tClean Player" << endl
41           << "3\tGambler Player" << endl
42           << endl << "Your choice: ";
43      int choice;
44      cin >> choice;
45      while( !cin.good() || choice < 1 || choice > 3 ) {
46          cout << endl << "Invalid choice." << endl;
47          if(!cin.good()) {
48              cerr << "Error: cin went into failsafe mode." << endl;
49              cin.clear();
50              cin.ignore(100, '\n');
51          }
52          cout << "Which AI would you like to test? " << endl
53               << "1\tSmart Player" << endl
54               << "2\tClean Player" << endl
55               << "3\tGambler Player" << endl
56               << endl << "Your choice: ";
57          cin >> choice;
58      }
```

Next, we can output a question of how many times you would like to test the AI and input the total amount of games, followed by determining display options.

```
59
60      cout << "How many times should I test the game AI? ";
61      cin >> totalGames;
62      cout << "Would you like to see each game displayed? [y/n] ";
63      cin >> displayChoice;
64      if( tolower(displayChoice) == 'n') {
65          silent = true;
66      }
67      if( ! silent ) {
68          cout << "How many seconds per move? 0 = fastest. Decimals allowed";
69          cin >> secondsPerMove;
70      } else {
71          secondsPerMove = 0;
72      }
```

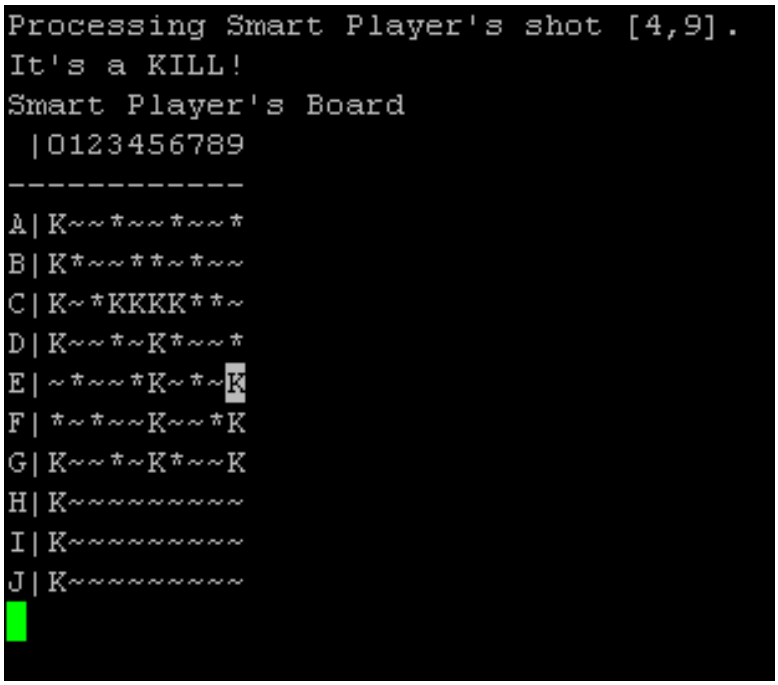
Now the code iterates through the count-control loop based on the number of games we specify, and a new AI and board awaits user input for which AI we want to test with using the switch statement.

```
73
74      for( int count=0; count<totalGames; count++ ) {
75          switch( choice ) {
76              case 1:
77                  player = new SmartPlayer(boardSize);
78                  playerName = "Smart Player";
79                  break;
80              case 2:
81                  player = new CleanPlayer(boardSize);
82                  playerName = "Clean Player";
83                  break;
84              case 3:
85                  player = new GamblerPlayer(boardSize);
86                  playerName = "Gambler Player";
87                  break;
88              default:
89                  cout << "Invalid player choice! Exiting ..." << endl;
90                  exit(1);
91          }
92          assert(player!=NULL);
93          board = new BoardV2(boardSize);
94
95          AITester tester(player, board, playerName, boardSize, silent);
96          tester.play(int(1000*secondsPerMove), gameMoves);
97
98          totalMoves += gameMoves;
99
100         delete player;
101         delete board;
102     }
```

Since the tester does timing in milliseconds, not seconds, we convert to milliseconds, update the total amount of moves, get rid of the used player AI and board, and then print out the stats:

```
103
104     cout << "Done testing \"" << playerName << "\" AI." << endl
105          << "Total games = " << totalGames << endl
106          << "Average shots per game was " << showpoint << setprecision(3)
107          << totalMoves/totalGames << endl
108          << "Percentage of board shot at = " <<
109              float(100*totalMoves/totalGames)/(boardSize*boardSize) << endl;
110
111     return 0;
112 }
```

That's it! Now we compile everything and run the program to see the results of our AI.



Everything is running as it should! This is the result of my SmartPlayer AI in action. Sinking ships and taking names! The user inputs the length of the ships, and the 'H' for 'hit' is auto-replaced by a 'K' for 'kill' once the enemy is killed. The lettering is represented by the length of each ship. Overall, smartPlayer is one smart AI and gets the job done!

Conclusion

I hope this guide has been helpful for anyone who reads it. While this is an introductory guide, I realize there is a lot of information included. Writing a program like this actually requires a lot more work and studying in the field in order to understand the complete algorithmic and programming approach to creating an AI Battleships game in C++. If anything, there's still a lot more theoretical, and computer science concepts that I've left out of this guide. Everything that I've included is pertinent to the purposes of understanding the fundamentals of game logic. This is one of the primary reasons why I stated that this guide is intended for programmers that are already at a beginner to intermediate level. To further expand upon game development, the industry has very advanced game engines in place to help them put their games together faster, but even with game engines, developers are still having to write a good amount of code. Even if you aren't a full-blown programmer, this guide still provides insight to how programmers implement and execute fairly advanced programming techniques without the use of a game engine. This program was also one of my last assignments for my *Intro to Computer Science II* course during my undergraduate degree. All diagrams and code in this guide I created and converted digitally from my original notes. If you have any questions about this guide or any other general inquiries, you can email me at technologicguy@gmail.com

Resources Used:

- Dale, Nell – Weems, Chip. *Programming and Problem Solving With C++ (4th Edition)* – 2004