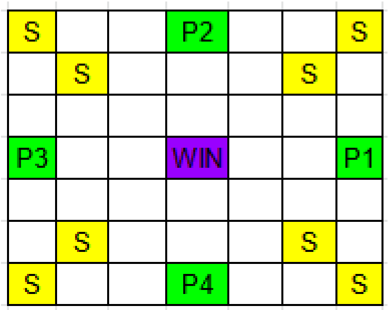
MiniLudu

Sandford Harris & Hasibullah Ghulamhaidar

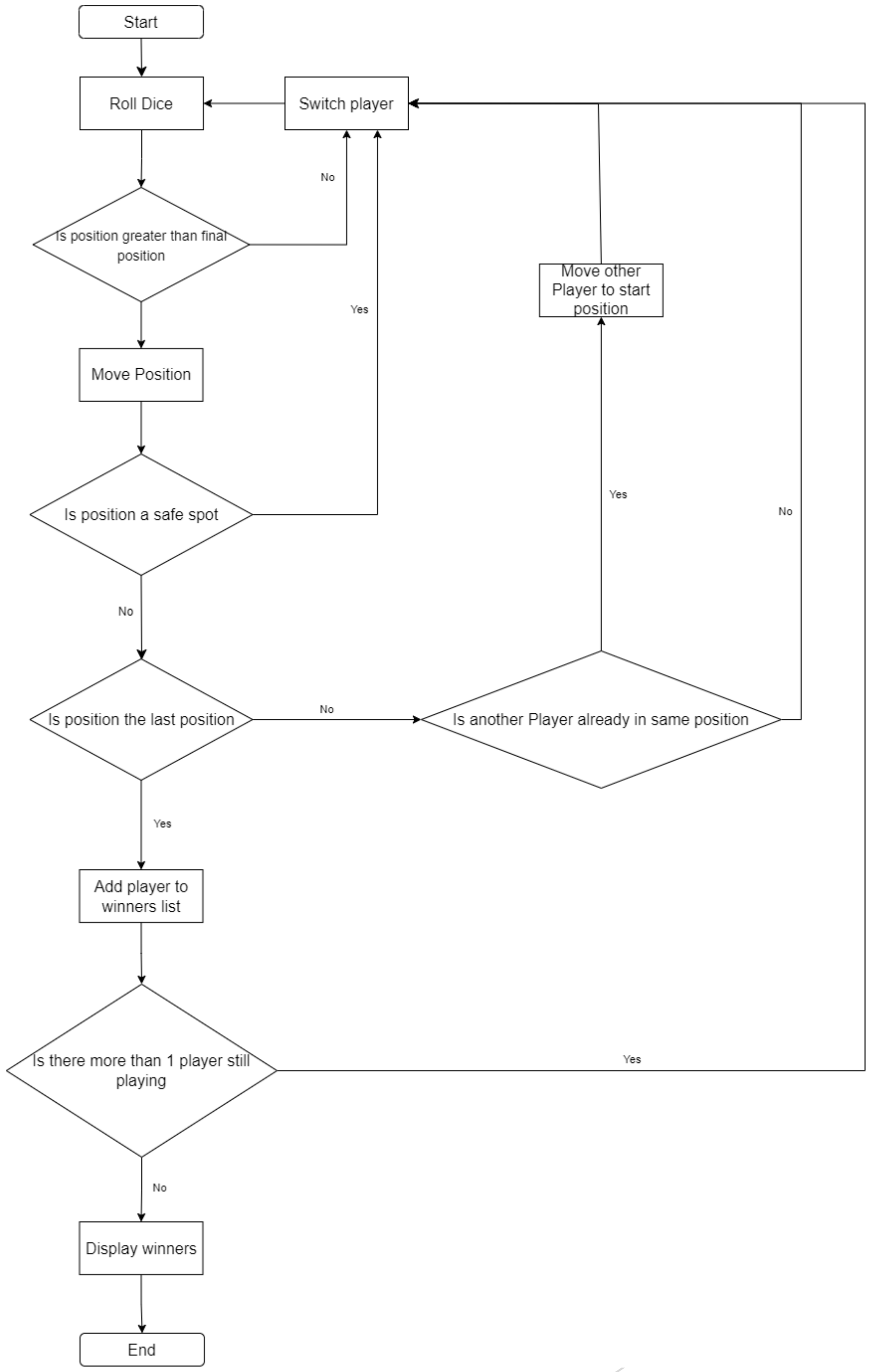
Game overview & requirements



S			P2			S
	S				S	
P3			WIN			P1
	S				S	
S			P4			S

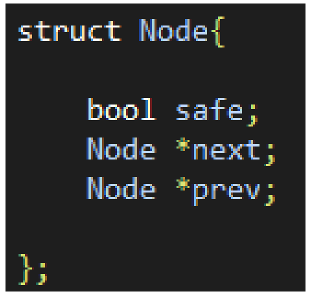
- Played with 4 players. Each player starts on their own starting square and take turns rolling a 4 sided dice.
- Players landing on the same empty space (white sqaures) as another player will kill them, sending them back to their initial starting position (green squares)
- Players on safe spaces (yellow and green squares) cannot be killed.
- Each player moves anti-clockwise around the edge board from their starting location, and clockwise around the inside of the board.
- Players landing on the centre square (purple) win the game. Play continues until 3 players have won.
- Players must roll the exact number to reach the end, otherwise play passes to the next player.
- The game must be implemented using a double linked list

Program flowchart

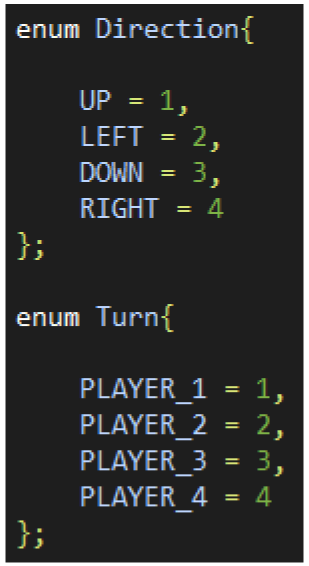

Start
Roll Dice
Switch player
Is position greater than final position
No
Yes
Move Position
Is position a safe spot
No
Is position the last position
No
Is another Player already in same position
Yes
Move other Player to start position
No
Yes
Add player to winners list
Is there more than 1 player still playing
Yes
No
Display winners
End

Initialisation

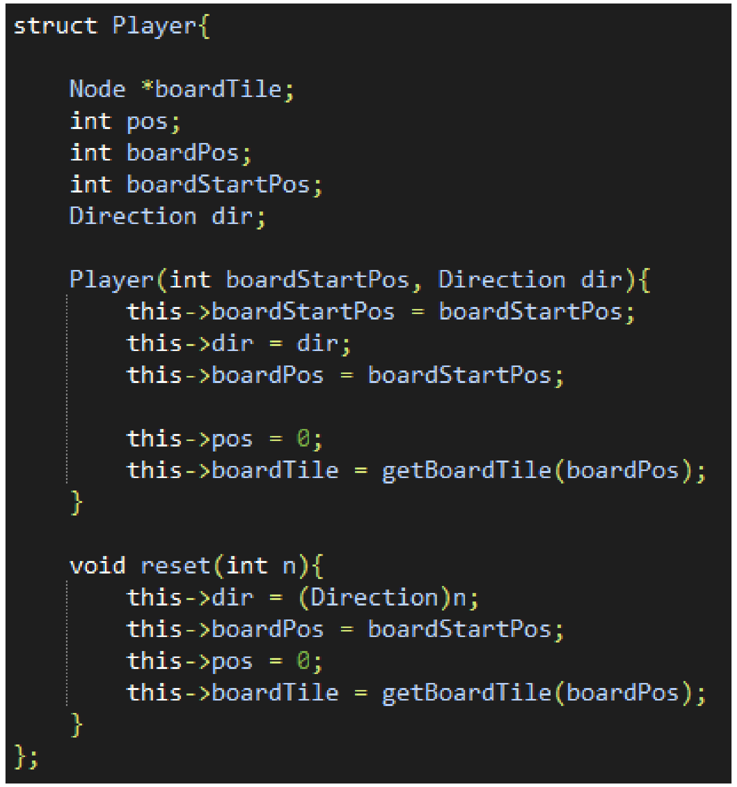
- Node struct
  - Pointers to next and previous Nodes
  - Bool safe – if the node is a safe spot or not
- Player struct
  - Node *boardTile – reference to player's location in the list
  - Int pos – how close player is to its goal
  - Int boardStartPos – node player starts at on the board
  - Int boardPos – player's current node on the board
  - Direction dir – enum tracking which way the player moves across the board
  - Reset() - sets player attributes to initial values
- enum Direction – can be UP, DOWN, LEFT or RIGHT
- enum Turn – keeps track of player turn. Can be PLAYER_1, PLAYER_2, PLAYER_3 or PLAYER_4



```
struct Node{

    bool safe;
    Node *next;
    Node *prev;

};
```



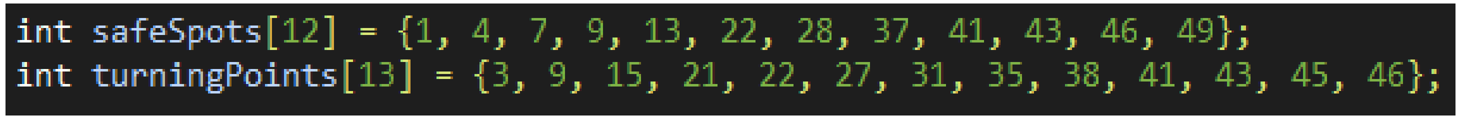
```
enum Direction{

    UP = 1,
    LEFT = 2,
    DOWN = 3,
    RIGHT = 4
};

enum Turn{

    PLAYER_1 = 1,
    PLAYER_2 = 2,
    PLAYER_3 = 3,
    PLAYER_4 = 4
};
```



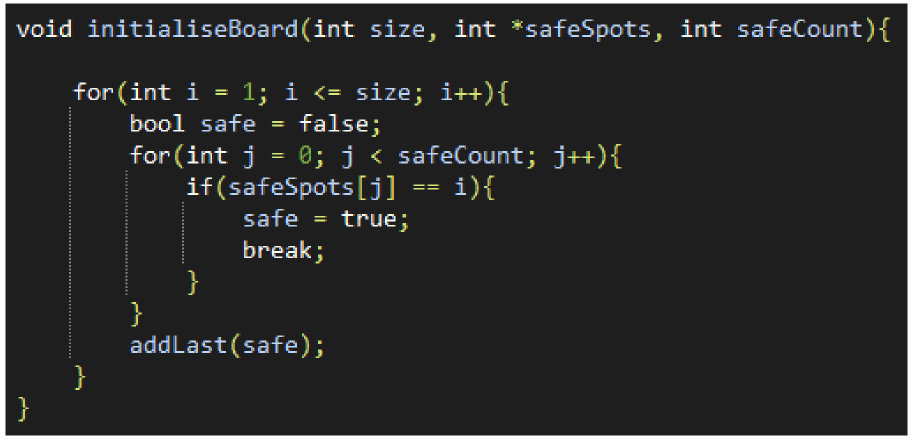
```
struct Player{

    Node *boardTile;
    int pos;
    int boardPos;
    int boardStartPos;
    Direction dir;

    Player(int boardStartPos, Direction dir){
        this->boardStartPos = boardStartPos;
        this->dir = dir;
        this->boardPos = boardStartPos;

        this->pos = 0;
        this->boardTile = getBoardTile(boardPos);
    }

    void reset(int n){
        this->dir = (Direction)n;
        this->boardPos = boardStartPos;
        this->pos = 0;
        this->boardTile = getBoardTile(boardPos);
    }
};
```

Initialisation

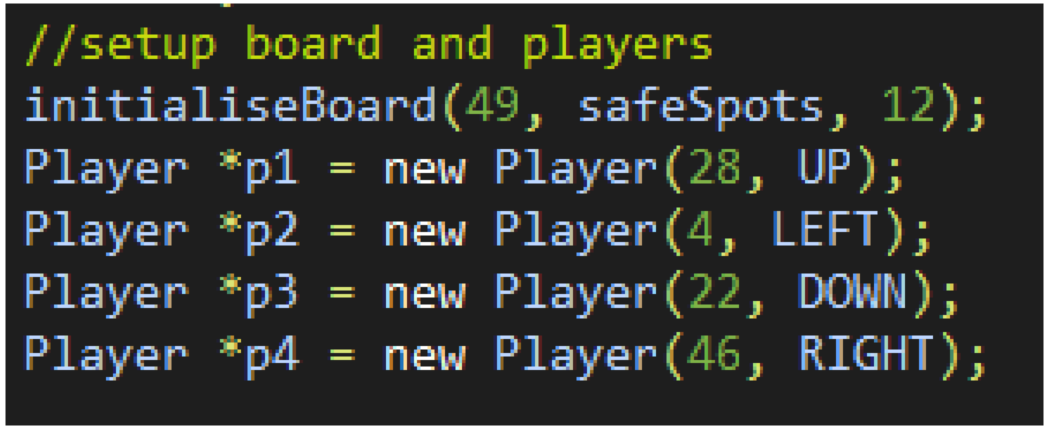
- Int safeSpots[] - which tiles on the board should be considered safe
- Int turningPoints[] - when the player should changed direction of traversal
- InitialiseBoard() - creates the linked list with a given number of nodes, assigns safe spots according to safeSpots[]
- Dice rolls done using rand()
  - Seeded using time with srand(time())



```
int safeSpots[12] = {1, 4, 7, 9, 13, 22, 28, 37, 41, 43, 46, 49};
int turningPoints[13] = {3, 9, 15, 21, 22, 27, 31, 35, 38, 41, 43, 45, 46};
```



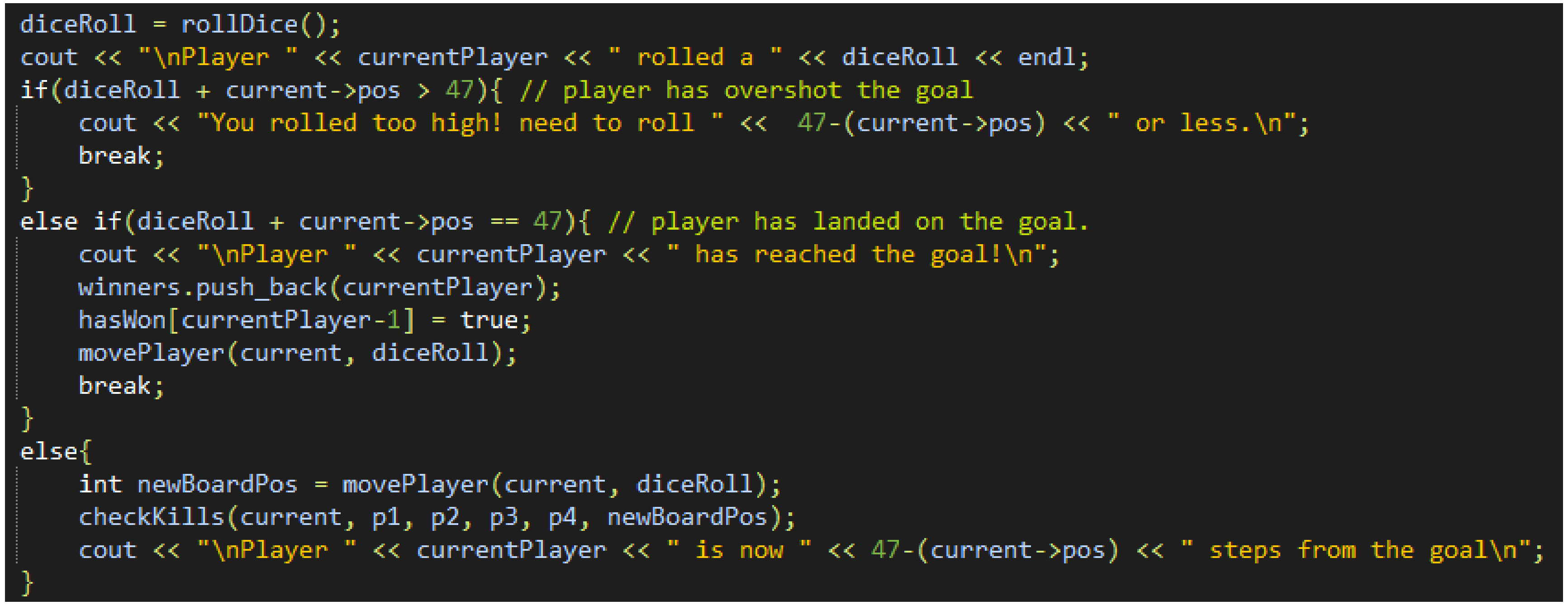
```
void initialiseBoard(int size, int *safeSpots, int safeCount){

    for(int i = 1; i <= size; i++){
        bool safe = false;
        for(int j = 0; j < safeCount; j++){
            if(safeSpots[j] == i){
                safe = true;
                break;
            }
        }
        addLast(safe);
    }
}
```



```
//setup board and players
initialiseBoard(49, safeSpots, 12);
Player *p1 = new Player(28, UP);
Player *p2 = new Player(4, LEFT);
Player *p3 = new Player(22, DOWN);
Player *p4 = new Player(46, RIGHT);
```

Main loop

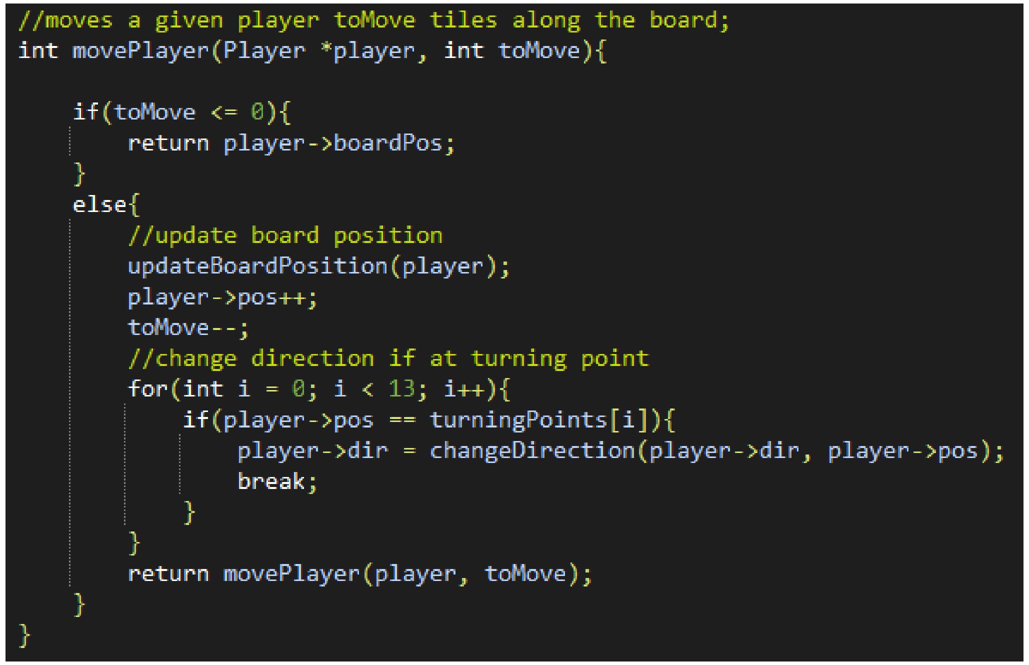
- Player can roll or exit
- Dice is rolled
  - if player overshoots end
    - Pass turn
  - if player lands on end
    - Add to winners list
- move the player amount rolled
- Check if the player kills



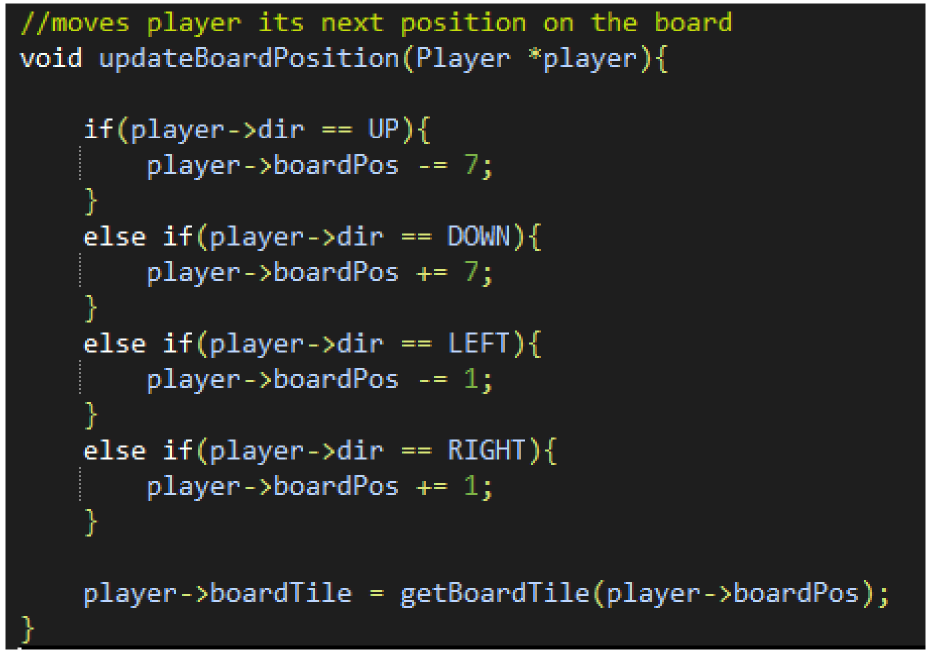
```
diceRoll = rollDice();
cout << "\nPlayer " << currentPlayer << " rolled a " << diceRoll << endl;
if(diceRoll + current->pos > 47){ // player has overshot the goal
    cout << "You rolled too high! need to roll " <<  47-(current->pos) << " or less.\n";
    break;
}
else if(diceRoll + current->pos == 47){ // player has landed on the goal.
    cout << "\nPlayer " << currentPlayer << " has reached the goal!\n";
    winners.push_back(currentPlayer);
    hasWon[currentPlayer-1] = true;
    movePlayer(current, diceRoll);
    break;
}
else{
    int newBoardPos = movePlayer(current, diceRoll);
    checkKills(current, p1, p2, p3, p4, newBoardPos);
    cout << "\nPlayer " << currentPlayer << " is now " << 47-(current->pos) << " steps from the goal\n";
}
```

Moving the player

- MovePlayer()
  - Recursive
  - Moves player step by step
  - Changes player direction if they land on a turning point
  - Updates player boardPos, pos, and tile reference
- UpdateBoardPosition()
  - Sets player's board position based on its current direction



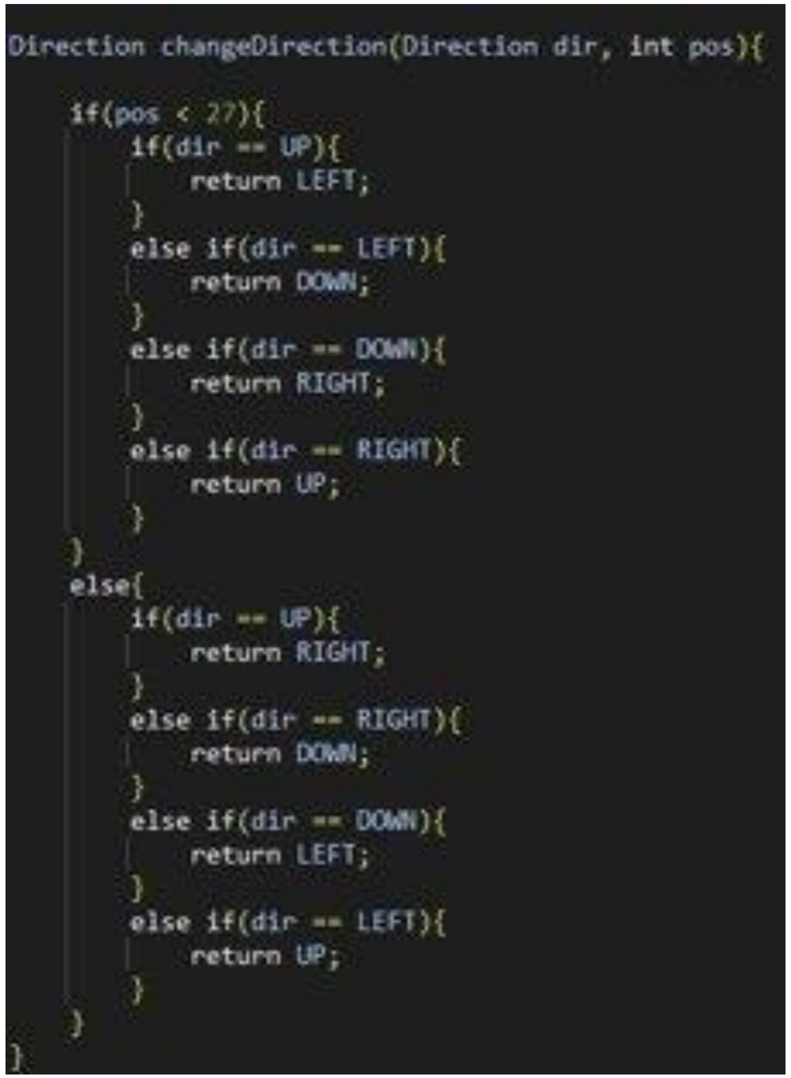
```
//moves a given player toMove tiles along the board;
int movePlayer(Player *player, int toMove){

    if(toMove <= 0){
        return player->boardPos;
    }
    else{
        //update board position
        updateBoardPosition(player);
        player->pos++;
        toMove--;
        //change direction if at turning point
        for(int i = 0; i < 13; i++){
            if(player->pos == turningPoints[i]){
                player->dir = changeDirection(player->dir, player->pos);
                break;
            }
        }
        return movePlayer(player, toMove);
    }
}
```



```
//moves player its next position on the board
void updateBoardPosition(Player *player){

    if(player->dir == UP){
        player->boardPos -= 7;
    }
    else if(player->dir == DOWN){
        player->boardPos += 7;
    }
    else if(player->dir == LEFT){
        player->boardPos -= 1;
    }
    else if(player->dir == RIGHT){
        player->boardPos += 1;
    }

    player->boardTile = getBoardTile(player->boardPos);
}
```

ChangeDirection()

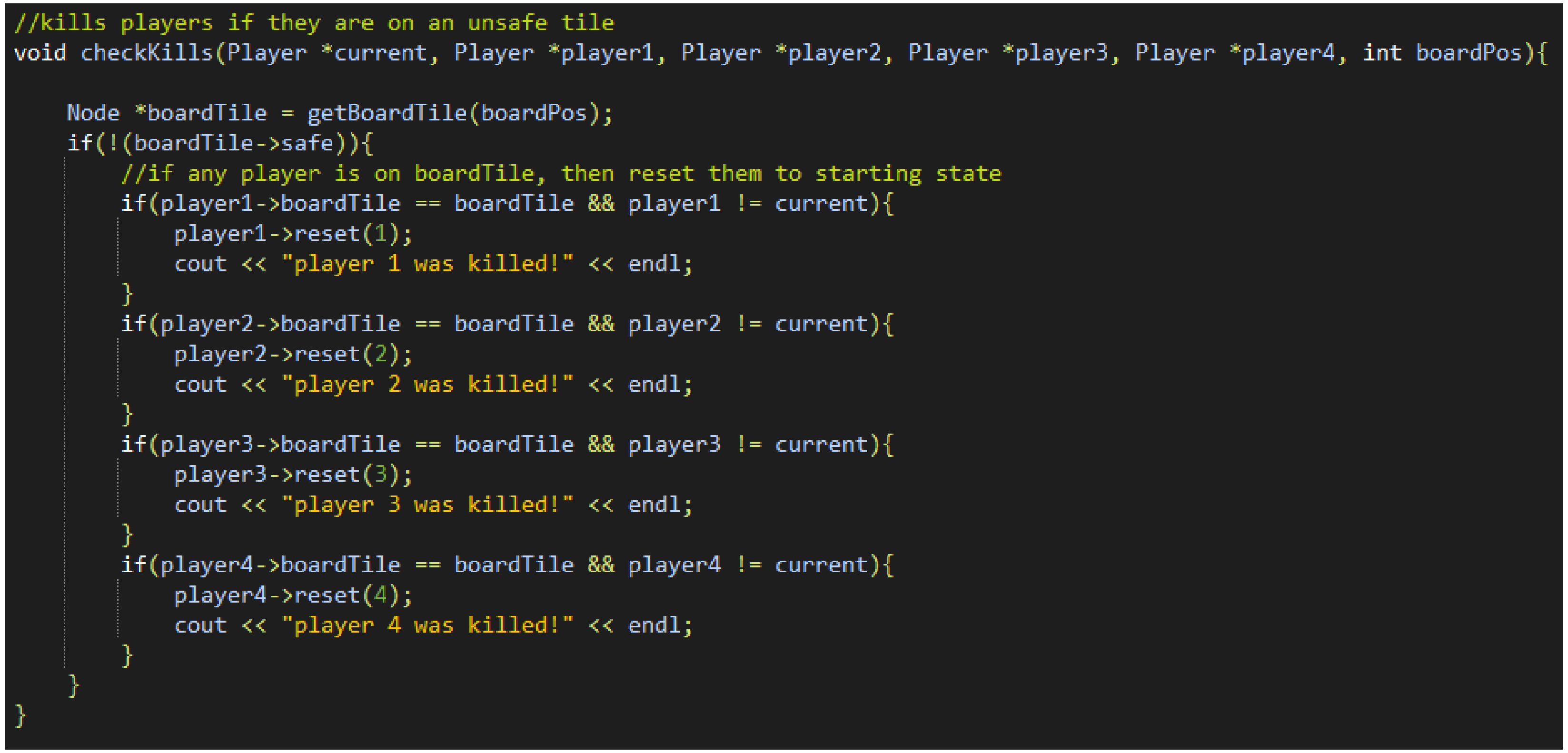
- ChangeDirection()
  - Returns a new direction for the player to move in based on current direction and the player's current position
    - Controls when the player switches from anti-clockwise to clockwise



```
Direction changeDirection(Direction dir, int pos){

    if(pos < 27){
        if(dir == UP){
            return LEFT;
        }
        else if(dir == LEFT){
            return DOWN;
        }
        else if(dir == DOWN){
            return RIGHT;
        }
        else if(dir == RIGHT){
            return UP;
        }
    }
    else{
        if(dir == UP){
            return RIGHT;
        }
        else if(dir == RIGHT){
            return DOWN;
        }
        else if(dir == DOWN){
            return LEFT;
        }
        else if(dir == LEFT){
            return UP;
        }
    }
}
```

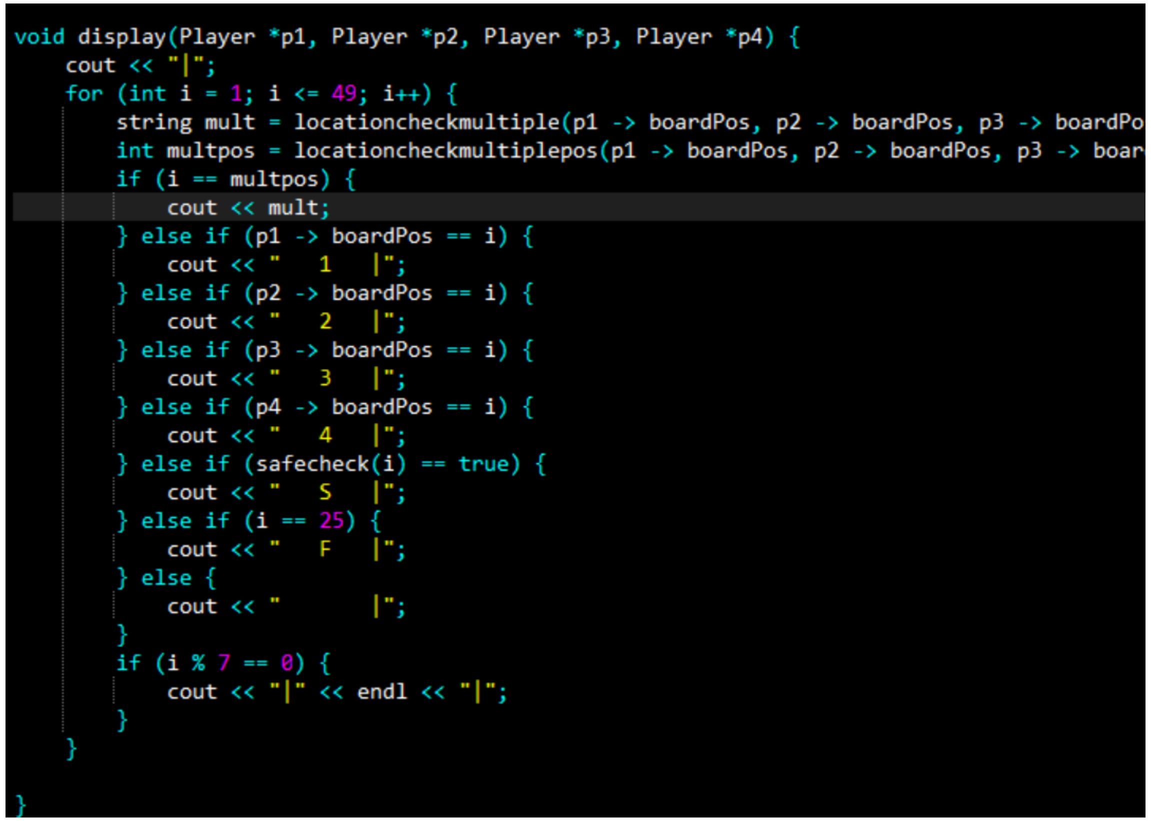
CheckKills()

- Takes a reference to all players and current player and the board position
- If the tile at that position isn't safe and a player on that tile is not the current player, they are killed
- Killed players are reset to their starting position



```
//kills players if they are on an unsafe tile
void checkKills(Player *current, Player *player1, Player *player2, Player *player3, Player *player4, int boardPos){

    Node *boardTile = getBoardTile(boardPos);
    if(!(boardTile->safe)){
        //if any player is on boardTile, then reset them to starting state
        if(player1->boardTile == boardTile && player1 != current){
            player1->reset(1);
            cout << "player 1 was killed!" << endl;
        }
        if(player2->boardTile == boardTile && player2 != current){
            player2->reset(2);
            cout << "player 2 was killed!" << endl;
        }
        if(player3->boardTile == boardTile && player3 != current){
            player3->reset(3);
            cout << "player 3 was killed!" << endl;
        }
        if(player4->boardTile == boardTile && player4 != current){
            player4->reset(4);
            cout << "player 4 was killed!" << endl;
        }
    }
}
```

Displaying the board

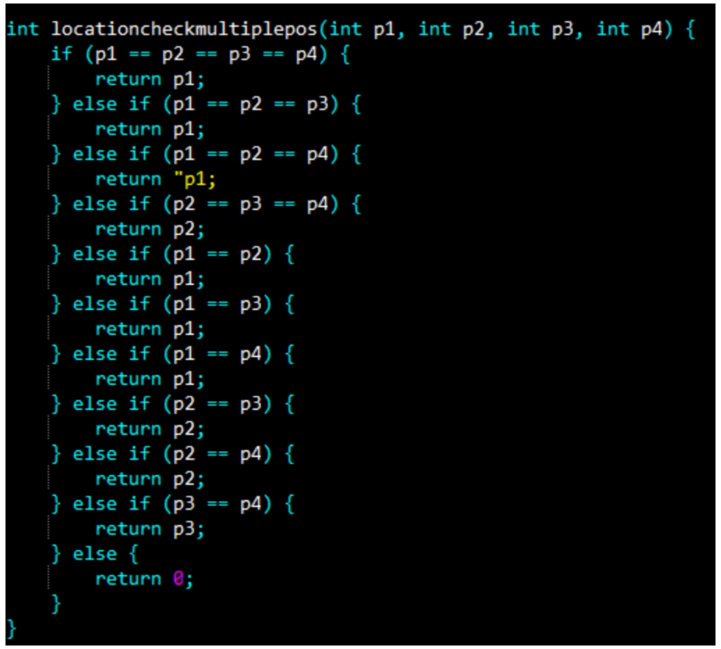


```
void display(Player *p1, Player *p2, Player *p3, Player *p4) {
    cout << "|";
    for (int i = 1; i <= 49; i++) {
        string mult = locationcheckmultiple(p1 -> boardPos, p2 -> boardPos, p3 -> boardPo
        int multpos = locationcheckmultiplepos(p1 -> boardPos, p2 -> boardPos, p3 -> boar
        if (i == multpos) {
            cout << mult;
        } else if (p1 -> boardPos == i) {
            cout << "   1   |";
        } else if (p2 -> boardPos == i) {
            cout << "   2   |";
        } else if (p3 -> boardPos == i) {
            cout << "   3   |";
        } else if (p4 -> boardPos == i) {
            cout << "   4   |";
        } else if (safecheck(i) == true) {
            cout << "   S   |";
        } else if (i == 25) {
            cout << "   F   |";
        } else {
            cout << "       |";
        }
        if (i % 7 == 0) {
            cout << "|" << endl << "|";
        }
    }
}
```

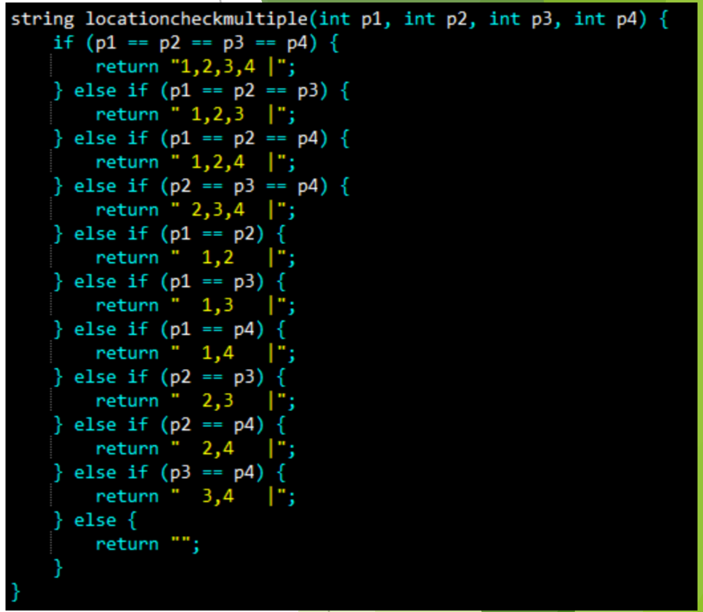
- display() – the function will display the board of the game along with where the players are and where the safe spots are located
- mult – used if there is more than one player in a position
- multpos – gives the position where more than one player is in

Displaying the board

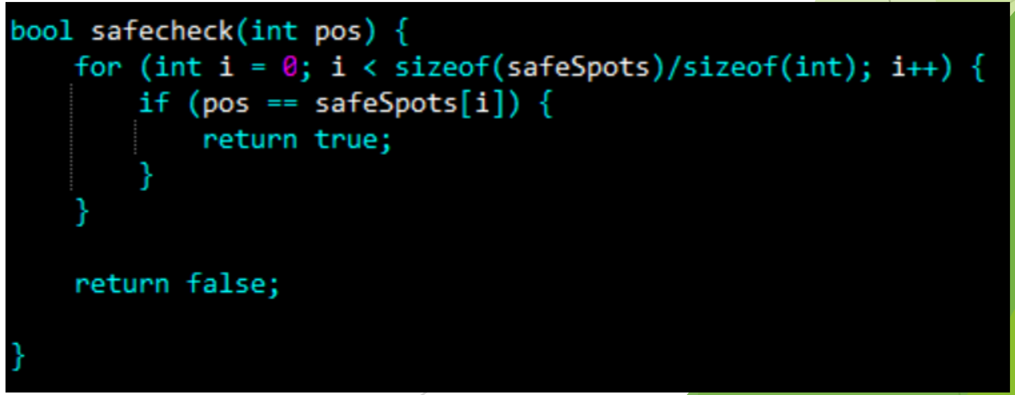
- locationcheckmultiplepos() – returns the position where there is more than one player, given that the position is a safe spot.
- locationcheckmultiple() – gives a string that shows the players at a position if there is more than one, and if the position is a safe sport
- safecheck() – checks to see if a given position is a safe spot



```
int locationcheckmultiplepos(int p1, int p2, int p3, int p4) {
    if (p1 == p2 == p3 == p4) {
        return p1;
    } else if (p1 == p2 == p3) {
        return p1;
    } else if (p1 == p2 == p4) {
        return "p1;
    } else if (p2 == p3 == p4) {
        return p2;
    } else if (p1 == p2) {
        return p1;
    } else if (p1 == p3) {
        return p1;
    } else if (p1 == p4) {
        return p1;
    } else if (p2 == p3) {
        return p2;
    } else if (p2 == p4) {
        return p2;
    } else if (p3 == p4) {
        return p3;
    } else {
        return 0;
    }
}
```



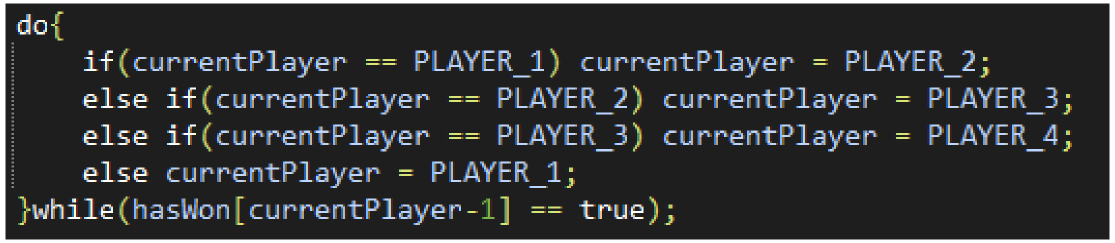
```
string locationcheckmultiple(int p1, int p2, int p3, int p4) {
    if (p1 == p2 == p3 == p4) {
        return "1,2,3,4 |";
    } else if (p1 == p2 == p3) {
        return " 1,2,3  |";
    } else if (p1 == p2 == p4) {
        return " 1,2,4  |";
    } else if (p2 == p3 == p4) {
        return " 2,3,4  |";
    } else if (p1 == p2) {
        return "  1,2   |";
    } else if (p1 == p3) {
        return "  1,3   |";
    } else if (p1 == p4) {
        return "  1,4   |";
    } else if (p2 == p3) {
        return "  2,3   |";
    } else if (p2 == p4) {
        return "  2,4   |";
    } else if (p3 == p4) {
        return "  3,4   |";
    } else {
        return "";
    }
}
```



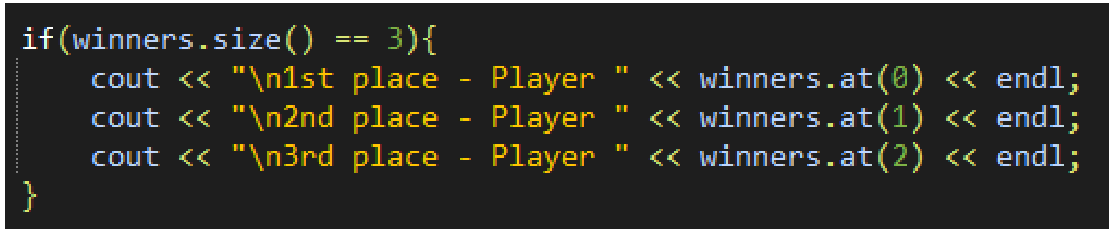
```
bool safecheck(int pos) {
    for (int i = 0; i < sizeof(safeSpots)/sizeof(int); i++) {
        if (pos == safeSpots[i]) {
            return true;
        }
    }

    return false;

}
```

Winning the game

- Next player chosen from those who havent reached the end
- When 3 players have reached the end, the game ends
- Output winners in the order they were added



```
do{
    if(currentPlayer == PLAYER_1) currentPlayer = PLAYER_2;
    else if(currentPlayer == PLAYER_2) currentPlayer = PLAYER_3;
    else if(currentPlayer == PLAYER_3) currentPlayer = PLAYER_4;
    else currentPlayer = PLAYER_1;
}while(hasWon[currentPlayer-1] == true);
```



```
if(winners.size() == 3){
    cout << "\n1st place - Player " << winners.at(0) << endl;
    cout << "\n2nd place - Player " << winners.at(1) << endl;
    cout << "\n3rd place - Player " << winners.at(2) << endl;
}
```