

[Skip to content](#)

3 SecondFeature.zip

0
Zip Archive

It's time to implement the next feature of this project - I am going to implement

MiniMaxAIPlayer

thought for 2m 18s

Awesome — you already have the stub in place, so this is a clean next step.

Below is a **drop-in MiniMax implementation** that fits your current design (no undo

needed, just Board copies). Then I'll show the **two small TicTacToe changes** to actually let

you select it in the menu.

1) Replace `MiniMaxAIPlayer.cpp` with this

This version:

- uses your existing `Board` API
- avoids undo by copying the board each branch
- scores:
 - +10 if the Minimax AI wins
 - -10 if opponent wins
 - 0 tie
- adds a tiny "prefer faster wins / delay losses" tweak using `filledCells` (you already declared `friend class MiniMaxAIPlayer;`).

//

// `MiniMaxAIPlayer.cpp`

```
//
```

```
#include "MiniMaxAIPlayer.h"
```

```
#include "Board.h"
```

```
#include <limits>
```

```
#include <iostream>
```

```
MiniMaxAIPlayer::MiniMaxAIPlayer(char symbol, const std::string& name)
```

```
    : AIPlayer(symbol, name) {}
```

```
void MiniMaxAIPlayer::makeMove(Board& board) {
```

```
    int bestScore = std::numeric_limits<int>::min();
```

```
    int bestRow = -1;
```

```
    int bestCol = -1;
```

```
// Try every possible move
```

```
    for (int r = 0; r < 3; ++r) {
```

```
        for (int c = 0; c < 3; ++c) {
```

```
            if (!board.isValidMove(r, c)) {
```

```
                continue;
```

```
}
```

```
    Board temp = board;
```

```
    temp.applyMove(r, c, symbol);
```

```
// After we move, opponent tries to minimize our score
```

```
    int score = miniMaxInternal(temp, false);
```

```
    if (score > bestScore) {
```

```
        bestScore = score;

        bestRow = r;

        bestCol = c;

    }

}

}

// Failsafe (shouldn't happen)

if (bestRow == -1 || bestCol == -1) {

    playRandomMove(board);

    return;

}

board.applyMove(bestRow, bestCol, symbol);

}

int MiniMaxAIPlayer::miniMaxInternal(Board& board, bool isMaximizing) {

    int eval = evaluateBoard(board);

    // Terminal states

    if (eval == 10) {

        // Prefer earlier wins

        return 10 - board.filledCells;

    }

    if (eval == -10) {

        // Prefer later losses

        return -10 + board.filledCells;

    }

}
```

```
if (board.tieCheck()) {  
    return 0;  
}  
  
char opp = getOpponentSymbol();  
  
if (isMaximizing) {  
    int best = std::numeric_limits<int>::min();  
  
    for (int r = 0; r < 3; ++r) {  
        for (int c = 0; c < 3; ++c) {  
            if (!board.isValidMove(r, c)) {  
                continue;  
            }  
  
            Board temp = board;  
            temp.applyMove(r, c, symbol);  
  
            int score = miniMaxInternal(temp, false);  
            if (score > best) {  
                best = score;  
            }  
        }  
    }  
    return best;  
} else {  
    int best = std::numeric_limits<int>::max();
```

```
for (int r = 0; r < 3; ++r) {
    for (int c = 0; c < 3; ++c) {
        if (!board.isValidMove(r, c)) {
            continue;
        }
    }

    Board temp = board;
    temp.applyMove(r, c, opp);

    int score = miniMaxInternal(temp, true);
    if (score < best) {
        best = score;
    }
}

return best;
}

int MiniMaxAIPlayer::evaluateBoard(const Board& board) const {
    char me = symbol;
    char opp = getOpponentSymbol();

    if (board.winCheck(me)) {
        return 10;
    }

    if (board.winCheck(opp)) {
        return -10;
    }
}
```

```
}
```

```
    return 0;
```

```
}
```

2) Update TicTacToe.cpp to enable option 4

A) Let users pick it

Change the validation text so **1-4** is allowed and remove the "not implemented" blocking.

Find both Player 1 and Player 2 selection loops and replace the menu text + checks roughly

like this:

```
std::cout << "\nSelect type for Player 1 (X):\n"
    << " 1. Human\n"
    << " 2. Regular AI\n"
    << " 3. Unbeatable AI\n"
    << " 4. Minimax AI\n"
    << "Choice: ";
```

...

```
if (type1 >= 1 && type1 <= 4) {
    break;
} else {
    std::cout << "Invalid type. Please choose 1-4.\n";
}
```

Do the same for Player 2.

B) Actually create the Minimax player

Add a case in `createPlayer`:

`case 4:`

```
    return new MiniMaxAIPlayer(symbol);
```

So the full switch becomes:

```
switch (type) {  
    case 1:  
        return new HumanPlayer(symbol, basename + " (Human)");  
    case 2:  
        return new RegularAIPlayer(symbol);  
    case 3:  
        return new UnbeatableAIPlayer(symbol);  
    case 4:  
        return new MiniMaxAIPlayer(symbol);  
    default:  
        std::cout << "Invalid or unsupported player type. Defaulting to  
Human.\n";  
        return new HumanPlayer(symbol, basename + " (Human)");  
}
```

3) Small filename-case gotcha (quick fix)

In `TicTacToe.h`, you currently have:

```
#include "MinimaxAIPlayer.h"
```

But your file is named:

● `MiniMaxAIPlayer.h`

This is fine on Windows, but could break on a case-sensitive setup. Safest fix:

```
#include "MinimaxAIPlayer.h"
```

4) Optional: quick tests you can add

You could extend `test_ai.cpp` later with:

- Minimax vs Random (should never lose)
- Minimax vs Regular (should never lose)
- Minimax vs Unbeatable (should always tie)

Your existing `playAIGame` helper makes this super easy.

Why this approach fits your project

- **No new data structures**
- **No undo API required**
- Uses your existing clean separation:
`Player` → `AIPlayer` → `MinimaxAIPlayer`
- Keeps logic readable for your course level

If you want, I can also draft a small `test_minimax.cpp` in the same style as

your `test_ai.cpp` so you can plug it into CMake with minimal friction.

Why would we use ± 10 instead of ± 1 ?

0

Short version: **it mostly doesn't matter**. Minimax only cares about ordering of outcomes,

so $+1/-1/0$ and $+10/-10/0$ will produce the same moves **as long as you don't add extra**

heuristics.

Longer, practical reasons people use ± 10 :

1. Room for tie-break tweaks

If you later add "win sooner / lose later," you often do something like:

- $10 - \text{depth}$
- $-10 + \text{depth}$

That's cleaner than:

- $1 - \text{depth}$ (which can go negative fast and blur meanings).

2. Consistency with classic examples

Lots of textbook/online minimax demos use ± 10 , so it's familiar.

3. Future expansion

If you ever add non-terminal heuristics (like evaluating two-in-a-row threats), you might

want terminal wins to dominate:

- terminal win: $+10$
- strong advantage: $+3$
- small advantage: $+1$

With ± 1 , you have less expressive space.

Bottom line

For pure TicTacToe with only terminal evaluation:

- ± 1 is totally fine.
- ± 10 is just a convention and makes small scoring adjustments easier.

If you want, you can switch my code to ± 1 and just change the depth tweak to something

like:

- $1 - 0.1 * \text{depth}$ (or drop the tweak entirely).

Please explain the logic of all members of MiniMaxAIPlayer Also, do I need to make any changes to its header file?

I thought for 42s

Here's the intent/logic of **every member** of **MiniMaxAIPlayer** as it exists in your project right now, and how it should behave once you implement it.

I'm referencing your current files from the zip you uploaded (your header already matches the structure you want).

The class itself

```
class MiniMaxAIPlayer : public AIPlayer
```

Meaning:

- It's a specialized AI that inherits:
- the base **Player** interface (**makeMove**)
- shared AI utilities from **AIPlayer** (especially **getOpponentSymbol()**).
- So minimax doesn't need to reinvent basic helper logic like "who is my opponent?"

Forward declaration

```
class Board;
```

Why it exists:

- Prevents circular include headaches and keeps compile times cleaner.
- You only need the full **Board** definition in the **.cpp**, not in the header.

Constructor

```
explicit MiniMaxAIPlayer(char symbol, const std::string& name = "Minimax  
AI");
```

Logic:

- Stores your symbol (**X** or **O**) and name via the **AIPlayer** constructor.
- The default name is just for UI/display and testing readability.

Implementation should be simple:

```
MiniMaxAIPlayer::MiniMaxAIPlayer(char symbol, const std::string& name)  
: AIPlayer(symbol, name) {}
```

makeMove(Board& board)

```
void makeMove(Board& board) override;
```

This is the **entry point** the game calls.

What it should do:

1. Look at every valid empty cell.
2. For each candidate move:
 - copy the board
 - apply the move
 - call **miniMaxInternal(...)** to evaluate the result
3. Choose the move with the best score and apply it to the real board.

Important design detail in your codebase:

- Your **Board** is small (3×3).
- Copying is cheap.

- So this “copy-per-branch” approach avoids needing an `undo()` API.

`miniMaxInternal(Board& board, bool isMaximizing)`

```
int miniMaxInternal(Board& board, bool isMaximizing);
```

This is the **recursive minimax engine**.

Parameters:

- `board`: a working board state (usually a copy).
- `isMaximizing`:
- `true` → it's **this AI's turn** to choose the best outcome.
- `false` → it's the **opponent's turn**, trying to make things worst for us.

Core logic:

1. **Check terminal states** first:

- if we've won → return positive score
- if opponent has won → return negative score
- if tie → return 0

2. Otherwise:

- If maximizing:
 - try all valid moves for `symbol`
 - return the **maximum score**.

- If minimizing:
 - try all valid moves for opponent symbol
 - return the **minimum score**.

This structure guarantees perfect play.

```
evaluateBoard(const Board& board) const
```

```
int evaluateBoard(const Board& board) const;
```

This is your **scoring function**.

The simplest correct version:

- +10 (or +1) if **we** win
- -10 (or -1) if **opponent** wins
- 0 otherwise

This function is intentionally separate because:

- it keeps minimax clean
- it makes it easy to add minor refinements later (like preferring faster wins)

The friend in Board

```
friend class MiniMaxAIPlayer;
```

This lets minimax access private members like **filledCells** if you want.

You **don't have to** use this.

But it's useful if you add a "win sooner / lose later" tie-break:

- return 10 - **filledCells** for a win
- return -10 + **filledCells** for a loss

That's optional polish, not required for correctness.

Do you need to change the header file?

Probably no. Your header is already exactly what you need.

It already has:

- correct inheritance
- correct override
- correct helper declarations
- forward declaration

Only change it if you choose to change your implementation signature.

For example:

- If you decide to add `depth`:

```
int miniMaxInternal(Board& board, bool isMaximizing, int depth);
```

then yes, update the header accordingly.

Optional style tweak (not required)

Your other AI headers usually only take `(char symbol)` constructors.

So you could simplify to match that style:

```
explicit MiniMaxAIPlayer(char symbol);
```

But that's purely consistency. Your current version is fine.

Quick mental model

- `makeMove` = "top-level chooser"
- `miniMaxInternal` = "recursive game tree solver"
- `evaluateBoard` = "who is winning here?"

If you want, I can paste a version of `MiniMaxAIPlayer.cpp` that matches **your exact**

header and uses only your current `Board` API (copy-based, no undo), so it drops in cleanly

with minimal edits elsewhere.

Remind me, what does explicit do?

o

explicit in C++ tells the compiler:

“Don’t use this constructor for automatic (implicit) type conversions.”

It mainly matters for **single-argument constructors**.

Without **explicit**

```
class PlayerType {  
public:  
    PlayerType(int level) {}  
};
```

```
void foo(PlayerType p) {}
```

```
foo(3); //  allowed: int 3 is implicitly converted into PlayerType
```

This can be convenient, but it can also cause confusing bugs.

With **explicit**

```
class PlayerType {  
public:  
    explicit PlayerType(int level) {}  
};
```

```
void foo(PlayerType p) {}
```

```
foo(3); //  error
```

```
foo(PlayerType(3)); //  okay
```

Now you must be intentional.

Why it's good

It prevents accidental conversions like:

- passing the wrong type to a function
- surprising behavior in overloads
- weird bugs when you add more constructors later

In your context

If you had something like:

```
explicit MiniMaxAIPlayer(char symbol);
```

That would stop this:

```
MiniMaxAIPlayer ai = 'X'; // ✗ blocked by explicit
```

and require:

```
MiniMaxAIPlayer ai('X'); // ✓
```

Quick rule of thumb

If a constructor can be called with **one argument**, it's usually a good idea to make

it **explicit** unless you really want implicit conversion.

That's all it is — a safety belt.

ChatGPT can make mistakes. Check important info.

||