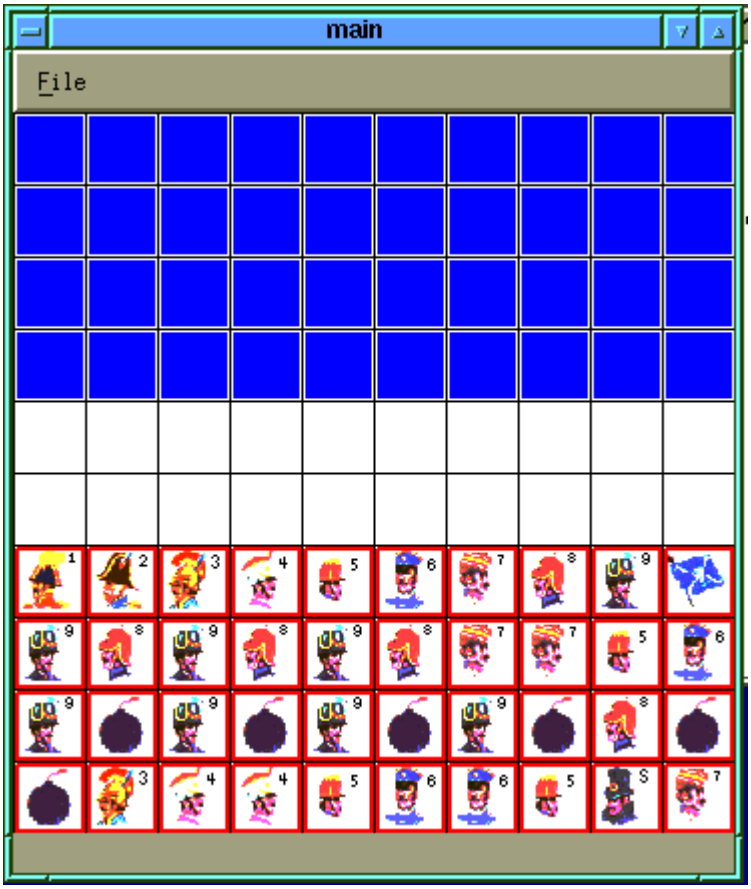
# Assignment 4: Stratego

Originally written by Andy Maag, and recently updated by our own Erik Neuenschwander.

No more random sentences or blackjack for you—this time you're ready for writing video games! Your next CS193D job will hopefully keep you entertained while simultaneously giving you experience with designing and implementing a class structure that uses inheritance and virtual functions to achieve run-time polymorphism. When finished, your program will look something like the following[1]:

**Due Thursday, November 16 at 11:59 p.m.**




## The Game

If you've never played the game Stratego, you've missed out on a truly valuable part of life. It's essentially a twist on chess where different pieces have different abilities and your opponent's pieces are hidden from your view. You only get to see an opponent's piece when you attack that piece with one of your pieces, or that piece attacks one of

[1] Assuming you run the X-Windows version. The text one is similarly stunning…

your pieces. Stratego is a two-player game, and our version will have one human player and one computer player. The true Stratego game is played on a 10 x 10 grid with a river and bridges in the center, but we'll simply play on a 10 x 10 grid where every square is considered valid (meaning a piece can move onto it).

On each turn, a player picks one of his / her pieces to move, and a location to which the piece should move. The rules governing each piece's legal moves is listed in the Pieces section. A piece can never attack another piece which is owned by the same player, and pieces must always stay on the board until removed (via attacking, see Pieces section). A player cannot move his / her enemy's pieces and cannot pass on its turn. A player must move exactly one piece on a turn, and it cannot be moved to the location it was at when the turn began (it must actually *move*). The goal of each player is to capture his / her opponent's flag before the opponent captures his / her flag. The game ends as soon as one player loses his / her flag or has no more moveable pieces.

**Pieces**

Each player starts out with 40 pieces which come in 12 flavors, which you can see to the right. "Initial number" is the number of those pieces with which each player begins the game.

The Flag and Bomb pieces can never move. The Scout can move any number of squares in one direction (North, South, East, West) as long as it does not jump over any other piece or leave the board. All other pieces can move exactly one square in one direction (North, South, East, West). No piece can move diagonally.

| Tile Code | Name | Initial Number |
|---|---|---|
| F | Flag | 1 |
| B | Bomb | 6 |
| S | Spy | 1 |
| 1 | Marshall | 1 |
| 2 | General | 1 |
| 3 | Colonel | 2 |
| 4 | Major | 3 |
| 5 | Captain | 4 |
| 6 | Lieutenant | 4 |
| 7 | Sergeant | 4 |
| 8 | Miner | 5 |
| 9 | Scout | 8 |

**There's only room for one of us in this square...**

When a player moves a piece into a square which is occupied by one of his/her opponent's pieces, at most one of those pieces gets to stay. It's an attack, and the losing piece(s) get removed from the board and stay out of play for the remainder of the game. The following rules are used to determine who wins and gets to stay in the square. We'll refer to the piece trying to move (called striking in the Stratego literature) as the **attacker** and the piece being attacked as the **defendant**.

- If any movable piece attacks the Flag, the Flag is captured and the game is over.
- If a Miner attacks a Bomb, then the Miner wins. If any other moveable piece attacks a Bomb, then the bomb prevails. Remember bombs never move, though.
- If the Spy attacks a Marshall, then the Spy wins. If the Marshall attacks a Spy, then the Marshall wins. All Spy/Marshall duels are won by the attacker.
- All other movable pieces prevail over a Spy, regardless of who attacks whom.
- If a numbered Piece attacks another numbered Piece, the Piece with the lower number wins. If both Pieces share the same number, then both die. That's right, mutual suicide, mutually assured destruction, group death. You get it.

Let's look at a few examples:

- A Spy attacks a Flag — Rule 1 says that the Spy wins.
- A Spy attacks a Bomb — Rule 2 says that the Bomb wins.
- A Miner attacks a Bomb — Rule 2 says that the Miner wins.
- A General attacks a Spy — Rule 4 says that the General wins.
- A Marshall attacks a Spy — Rule 3 says that the Marshall wins.
- A Spy attacks a Spy — Rule 5 says both die.
- A Spy attacks a General — Rule 5 says that the General wins.
- A Spy attacks a Marshall — Rule 3 says that the Spy wins.
- A Marshall attacks a Marshall — Rule 5 says both die.
- A General attacks a Marshall — Rule 5 says that the Marshall wins.
- A Flag attacks a Flag – can't happen; flags don't move and therefore can't strike / attack.

We'll give you the base `Piece` class definition, but it's up to you to implement it, as well as provide derived classes which inherit from the `Piece` class and implement the movement and interaction logic. You'll need to extend the `Piece` class definition a little bit to do this. This hierarchy is a wee bit more complicated than that of the `BlackJack` assignment. Inheritance is a key feature of all object-oriented languages, and designing hierarchies and understanding how to structure them is tough to get right. I'm hoping that a second assignment in inheritance will just give you that much more practice with inheritance. You'll also have the opportunity to fiddle with virtual constructors and

double dispatch—those two idioms weren't present in Assignment 3. Basically, this is the point of the assignment. We hope it will be the focus of your time. In exchange, we will make it the focus of your grade. :)

**What We Provide**

We've provided you with the bulk of the code for the assignment. The major sections we've provided are described here:

Main Loop: We've written the main loop which instantiates the board, player, and user-interface objects, sets up the board, alternates taking turns between the two players, and draws the board after every move.

Board: We've also provided the class to implement the board abstraction, including the bounds-checking, piece movement, and file I/O functionality (you don't have to write any file I/O on this whole assignment!).

Move: You're provided with a class to encapsulate moves. It is a simple class which encapsulates the start and destination location of the move, as well as determining if the move is legal or not. This class is defined in `move.h`.

Player: We've provided you with the entirety of the player implementations: a `Player` class, and two specializations in the form of the `HumanPlayer` and `ComputerPlayer` classes. The computer player is only provided in .o format, and actually is pulled from a separate directory. Depending on how the week goes, you might suddenly notice a smarter and tougher computer player half way through… None of this is of your worry.

UI: We've provided you with all of the user-interface code, including a `TextInterface` class and an `XInterface` class. The `TextInterface` class will run on any platform. The `XInterface` will work on the Sun Workstations in Sweet Hall (you must be sitting at the workstation for it to work, or have the capability to run X/Windows programs remotely). Through the beauty of inheritance and virtual functions, none of your code needs to rely on a particular user-interface type — just call the functions in the base `UserInterface` class which we provide in `userinterface.h` and it will work for each user-interface type. You shouldn't have to open or read the `UserInterface` subclasses.

Utilities: We've provided a set of utility functions which manage a `Location` structure (a column/row pair), and generate random numbers. These are available in `utility.h`.

**Your Part**

The part of the assignment that you must provide is the `Piece` class hierarchy. This requires you to think about a fair amount of object-oriented design in addition to implementation. Be sure to give yourself plenty of time to design the class hierarchy and make any design adjustments as appropriate. Paper sketches are can be helpful here. The amount of coding required in this assignment is not a lot. Even though your classes will be small and simple, **you are expected to build your class hierarchy in such a way that shared code (even two or three lines—remember that represents a larger block of code in industry-scale software projects)** is pressed toward the root of the hierarchy as much as possible.

**Project Task: Design and Implement the `Piece` Hierarchy**

This task is more difficult than the design in blackjack, so give yourself plenty of time to work on it. Remember that a paper design of your class hierarchy to represent Stratego pieces can save you time in the long run. Remember what inheritance and `virtual` (and pure `virtual`) functions are all about—you want to define subclasses that group common behavior between piece types. The two types of behavior you'll be dealing with are the movement rules for the class and the interaction between pieces when they come into contact. Again, I want you to be somewhat academic about your design; by doing so, you'll illustrate your understanding of OOP design, and that's the most important thing you need to get out of this assignment. (Have I stressed this enough yet?)

You should design your class hierarchy carefully because it will affect how much code you must write, and you will be graded on the elegance of your design. The key things to keep in mind: avoid duplicating a lot of code, and avoid `switch` statements on the type of piece you're working with. The only place you should see a `switch` is in the virtual constructor. A `switch` elsewhere means you're coding in a C sense polymorphic behavior that should instead be handled via virtual function dispatch.

To start with, you need to implement the `Piece` class itself. We've provided the basic `Piece` class interface the other classes will call. You'll need to extend this interface with appropriate methods for implementing interaction between pieces with double dispatch. For this assignment, you **must** implement the interaction using double dispatch.

Implementing the Piece class involves implementing several methods:

- ```
  Piece* createPiece(Board& board, Player *owner,
                     const Location& location, char type);
  ```

  This static method is a factory method, or `virtual` constructor, which returns a sub-type of `Piece` depending upon the type provided. The type provided is a character which represents the tile code for the piece. If the type is invalid, `createPiece` should return `NULL`.
  ```

- `Piece(Board& board, Player* owner, const Location& location);`

  The `Piece` constructor should initialize the board, player, and location data members. Because the board data member is a reference, you'll need to initialize this within an initialization list. The piece should add itself to the board in the proper location, and add itself to the owner's list of pieces.

- `virtual ~Piece(void);`

  The Piece destructor should `delete` each piece in the player's piece list. Because the board and player data members are aliases to shared `Player` and `Board` objects, these should not be deleted in the `Piece` destructor. The piece should remove itself from the board and remove itself from its owner's list of pieces.

- `virtual Location getLocation(void) const;`
- `virtual void setLocation(const Location& loc);`

  These are simple accessor and mutator methods for the piece's location.

- `virtual const Player* getOwner(void) const;`

  This is a simple accessor function for the piece's owner.

- `virtual Outcome attack(const Piece* defender) const;`

  This function is a bit of an oddity. It should never be called because each subclass of `Piece` which is able to attack another piece must override it as part of the double-dispatch sequence. However, not every subclass of `Piece` can attack another piece, and you don't want to force those subclasses to override the attack method to replicate code which says "Why am I here?". Thus, you would not want to make it a pure virtual function in the `Piece` class. This function can be implemented by just calling `assert(false)`. It should also return an `Outcome` to avoid generating compiler warnings, since `assert` can be disabled at compile time. `Outcome` is an enumerated type you'll find defined in `utility.h`. Subclasses which override the `attack` method should simply implement it by calling the `defend` method with this as the parameter. This does the double dispatch to reify both types of pieces involved in the interaction. The function should return `Win` if the attacker wins, and `Lose` if the attacker loses. `Draw` is reserved for the mutual annihilation we discussed above.

- `virtual bool isLegalMove(const StrategoMove& move) const;`

  Again, a little bit of an oddity, but this provides a suitable default method which should simply return `false`, so that immobile pieces aren't forced to duplicate this function. Subclasses should override this method whenever appropriate.

```
virtual const Board& getBoard(void) const;
```

This is a simple accessor for the board data member.

You will also need to add some defend methods to the `Piece` class and its subclasses which implement the second part of the double-dispatch sequence. How many you need to implement depends on how you design your class hierarchy. We'll give you a hint to say that my solution, which I feel is pretty well designed, defined and implements three `defend` methods, and it does not use any type-field switching. Design your class hierarchy carefully so that you don't have to implement a grotesquely large number of defend methods.

There are also several methods in the `Piece` class which are pure `virtual` functions and must be implemented in classes derived from `Piece`.

```
virtual char kind(void) const = 0;
```

Subclasses should override this method to return the tile code for the piece. The tile codes are listed in the table on page 2. (Hint: you don't necessarily need one subclass per tile code.)

```
virtual Outcome defend(const ????& attacker) const;
```

As mentioned above, subclasses will need to provide `defend` methods to implement the second half of the double dispatch. These take references because there's not reason to provide support for `NULL`. Remember that in these methods, `Win` means that the defender loses, and `Lose` means that the defender wins, and `Draw` is a sad state for both parties when two pieces of equal strength meet. Another way to put this is that the outcome is taken from the attacker's perspective. These need to be defined in the `Piece` base class, but can be pure `virtual` functions at the base class level.

**Miscellaneous notes**

- Before you turn in your assignment, **please set it up to use** the text-based user-interface.
- Text-based versus X-Windows based interfaces are chosen by the `USE_X` constant at the top of the Makefile. Uncomment it to use X, comment with # at the beginning of the line to use text.
- We've commented out the call to `Randomize` in `main.cc`. This is useful for debugging, but will make the computer player's moves always the same. When you're done debugging, you should restore the call to `Randomize` before submitting the program.

- All the design guidelines we gave you for the first three assignments still apply (compiles cleanly, is `const`-correct, properly deallocates memory, don't overuse the heap, etc.).
- While we're happy to answer your questions, please do not send us questions like "How do I design my `Piece` class hierarchy?". Since the design of this is an integral part of the assignment, answering that sort of question is like writing the code for you. We will answer design-related questions if they're really specific.
- Again, to use the X/Windows user-interface, you need to uncomment the `USE_X = true` line in the `Makefile`.
- You may find it easier to use the text user-interface for debugging your program because you can "look back" several moves. However, it's much more fun to play with the fancier user-interface, you can select to see all your enemies pieces (choose "Cover" off the menu), which is nice during debugging.

**Extras**

You are more than encouraged to invest some time into a good computer player (or players!) if you'd like to, but only do so after you've completed and tested the rest of the assignment. In fact, if you're looking for a challenge, you might design the `ComputerPlayer` class as a abstract base class, and implement different heuristics in different subclasses, apply all heuristics asking each to come up with a list of good moves, and then come up with some voting scheme to decide which of the proposed moves seems best. This isn't as easy as it sounds, but it's a very interesting problem.

**Getting Started**

In our leland class directory `/usr/class/cs193d/assignments` you'll find the project directory `hw4` which contains the starting code for you and a makefile to build the project. You can also get the starting files via the web page. The update directory includes the .o file for the computer player on Unix, the one that will get automagically updated.

No matter where you do your development work, when you're done, be sure your project will compile and run on the `elaine` workstations. All assignments will be electronically submitted there and that is where we will compile and test your code.

**Playing Stratego with the Text-Based User-Interface**

If you're unfamiliar with how to play Stratego, it may be a good idea to play around with the sample application in the class directory. To play the game with the text interface, you first need to enter the name of the board file to use. The board that we've provided is called `board.txt`. When the board is drawn, your pieces are at the bottom of the screen and the computer's pieces are at the top of the screen. Because all of the computer's pieces are hidden from your view, they are shown as `X`'s.

You can then begin taking turns with the computer player. To move a piece, type the name of the grid location of the piece you'd like to move (A7, for example). Then type in the name of grid location you'd like to move it to (A6, for example). The board will redraw, then the computer will make its move, and the board will redraw again. You can scroll up in your terminal window to see moves which have scrolled too far up. When two pieces collide, the types of the pieces will be output.

You can create a new board file if you wish, but it must be 10 rows by 10 columns, and have 2 empty lines for lines 5 and 6 of the file. Each line must contain exactly 10 characters.