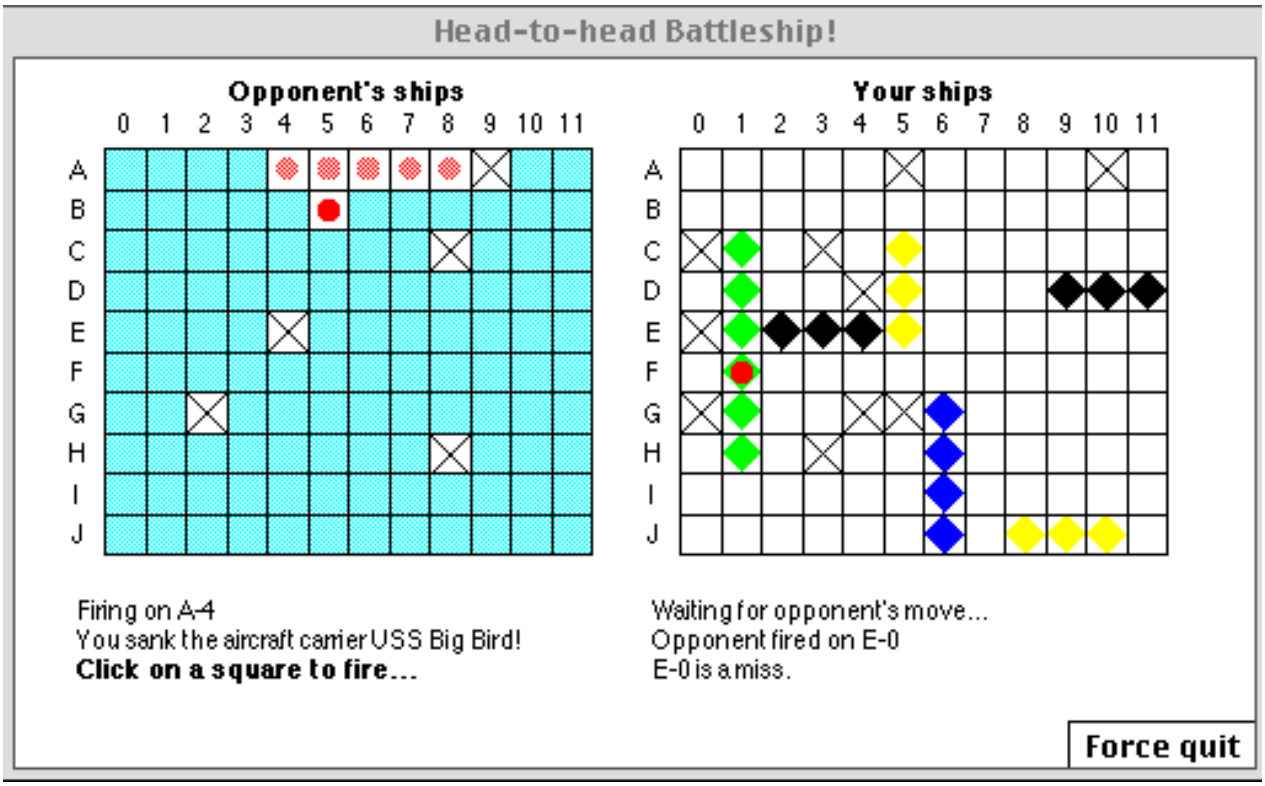
Assignment #4: Network Battleship

Due: Friday, April 28th in class

For your next task, you will implement the game of Battleship™. This assignment is comparable to the last assignment of a CS106A course. It is a comprehensive program covering material from the entire first quarter spectrum, with particular emphasis on pointers, arrays, structures, and file processing. As an added bonus, I recently spruced it up to support networking, making it possible to play head-to-head Battleship over a network, wow!

The program itself can be straightforward if you take care to design a reasonable data structure and follow a sensible decomposition strategy, but you must develop and test in stages rather than trying to attack the entire problem at once. Starting early is essential to ensure you have enough time to work through the design issues, completely debug the functionality, and get help if needed.

Below is a screen shot from a sample run of the program:



The game

In our version of Battleship, the user is pitted against an opponent, who can be either another networked user or a simple computer player. Each player keeps their own two-dimensional board onto which the program randomly positions a set of ships of various sizes. Each square of the board is called a cell. Each ship occupies either a horizontal or vertical line of cells. For variety, each ship is of a particular class (battleship, aircraft carrier, rowboat, etc.) and the user's ships are drawn in different colors according to their class to help the user keep track of which ship is which. Each player knows where their own ships are placed, but starts with no information about their opponent's ships— scouting them out is the object of the game.

Once the ships are laid out, the battle begins. The user clicks on a cell on the opponent's board to fire a shot. The shot is then communicated to the opponent, who responds by telling the user whether the location was a hit or a miss. The opponent then has a chance to fire a shot back at the user's ships and the user responds with the result. The game continues like this, alternating shot and response. A hit in the last remaining cell of a ship sinks that ship. The game is won by the first player to sink all of the opponent's ships.

Looking at the sample run on the first page, the user has incorrectly guessed A9, C8, and G2 (among others) and had a hit at B5. On their last turn, the user sunk the opponent's ship "USS BigBird" which occupied the horizontal line A4 through A8. The commentary at the bottom tells us that BigBird was of class "aircraft carrier". Looking at our opponent's guesses, they haven't been doing so well, there are misses scattered throughout the board and one lone hit at location F1.

Graphics

The game activity is drawn and updated in the graphics window. This task separates nicely into its own module and we provide you with the entire `gbattle` implementation, in order to allow you to concentrate your efforts on the interesting parts of the program. A brief mention of the functions is given here to get you started. The `gbattle.h` interface has comments describing each of the functions in more detail. You can also poke around in the `gbattle.c` file if you're curious or would like to change anything about the display.

— There is a new struct **`coord`** for a row-col pair and an enum **`board`** to distinguish the left and right boards of the display.

— There are #define-d constants for the number of rows and columns in the board.

— The **`InitBattleGraphics`** function configures the starting display with the two empty boards, one each for the user and opponent. You call this function once at the start of the game.

— The **`GetLocationChosenByUser`** function is for the user's turn. It waits for the user to click in the graphics window and returns the location of the cell chosen.

— The **`MarkShip`** function is used to fill a colored diamond in a cell to represent a ship piece. This is used when drawing the user's ships at the beginning of the game.

— The **`MarkMiss`** function exposes and draws a black 'X' over a cell location. It is used to mark both the user's and opponent's missed shots.

— The **`MarkHit`** function fills a red circle over a cell location. It is used to mark both the user's and opponent's hits.

— The **`MarkLineAsSunk`** function dims out a line of cells. It is used to mark sunken ships, both for the user and opponent.

— The **`DrawPrintfMessage`** function prints a message on the display. It is used for running commentary during game play about shots and responses. The function is just like printf—you can give it a printf-style format string with % directives and matching arguments. It displays the message in the graphics window instead of the text console.

Networking

The other module that we provide is the `netbattle` module containing the routines you need to establish a network connection and send and receive turn information between players. Networked applications share data across the network between computers, called hosts. Data is sent in small blocks called *packets* and are *routed* from one host to another. The Internet Protocol (IP) is the mechanism for routing these packets on the Internet and the Transmission Control Protocol (TCP) is a method of allowing communication between hosts. The library code we have provided allows for TCP/IP connectivity. Every host has a unique numeric IP address of the form *x.x.x.x* (for instance,

my dear old NeXT computer is 171.64.64.150) and often a host name (such as monolith.stanford.edu).

In connecting to another computer, one computer is said to be the *host* that sets up a communication channel and waits for the connection and the other is the *client* that establishes a connection to the host. For a network game, one player will host the game, the other will connect to that host. Once a connection has been established, both send and receive symmetrically.

Our library handles the complexity of networking behind the wall of abstraction and provides a simple interface specifically designed to support the Battleship game. The starter code provided in `battlemain.c` shows how to configure the starting connection. From there, it is largely just exchanging send and receive calls between the user and opponent. A brief mention of the network functions is given here. The `netbattle.h` interface has comments describing each of the functions in more detail.

— There is an enum **`shotResult`** to describe the various possible responses to a fired shot and a struct **`response`** that gathers information returned to the player on a turn.

— The **`HostGame`** and **`WaitForOtherToConnect`** functions are used at the beginning of a game when the user wants to host a new game and wait for another player to join.

— The **`ConnectToHost`** function is used at the beginning of a game when the user wants to join a game hosted by another player.

— The **`ConnectToComputerPlayer`** function is used at the beginning of a game when the user wants to play a game against the computer player.

— The **`CloseConnection`** function shuts down an existing connection. It is used at the end of the game to clean-up.

— The **`SendShotToOpponentAndWaitForResponse`** function sends a shot to the connected opponent and waits for their response. It is used on the user's turn.

— The **`WaitForOpponentShot`** function waits for an incoming shot from the connected opponent. It is used at the beginning of the opponent's turn.

— The **`SendResponseToOpponentShot`** function sends back a response to a shot received by the above function. It is used at the end of the opponent's turn.

The network library also provides a dumb computer player as a possible connected opponent. Communicating with the computer player uses exactly the same calls as the network player. This allows you to test your game in isolation without involving the network until you have a stable base. The computer player has no intelligence at all and is an easy opponent to beat. To facilitate debugging, it always places its ships in the top rows of the board and fires on the user's ships in a totally random fashion.

Designing a data structure

One important initial consideration when structuring this program is how to store the data for this game—in particular, the ships, their classes, and their locations on the user's board. At times you want to be able to go quickly from a board location to the ship that occupies it. At other times, you will need to go quickly from a ship to its placement on the board. Although you could always determine one given the other, it is easier to build a bit of redundancy into your data structures in order to facilitate both types of access.

For the user's board, you will want to maintain a two-dimensional array, mapping each location to the ship that occupies it. For each cell in the board, you will keep a pointer to the ship that occupies that cell, or NULL to indicate the cell is empty. This will give you quick access to the ship at a given location.

In order to go in the other direction (from a given user's ship to its location), in each ship structure you will store the start and end coordinates of that ship on the board. This information will be useful when that ship is sunk and you need to mark all of the cells to show the ship is out of commission.

For the opponent's board, you need to store much less information, in fact, you may find that you don't need to store anything explicitly at all. You don't know what ships the opponent has and where they are placed, figuring that out is the object of the game! When you fire on the opponent's board, the opponent responds with the hit/miss status which you will use to update the graphics window, but given that you don't need to refer to it later, you may find that the graphical display is all you need to track the opponent's board.

There is other information that you will need in various data structures (whether each cell has been fired upon, number of ships remaining, what class a ship is, and so on) and it will be up to you to work out the appropriate organization for this data. You're encouraged to ask questions of the course staff if you need feedback on your choices. Having a good design for your data structure is essential to successful development, so don't shortchange this important step.

The ship data files

The ship information is stored in a text file. The file begins with the number of ship classes followed by the info for each ship class. Next is the number of ships and the info for each ship. You can assume the file will have no errors and will be formatted exactly like this:

```
Num Classes: 5                  <- Number of ship classes to follow
battleship                      <- Name of this class
4                               <- Length of ships of this class
Magenta                         <- Color for drawing this class of ship
                                (blank line follows each class)
aircraft carrier                info for next class
5
Green
        ... so on for all ship classes ...
Num Ships: 7                    <- Number of ships that follow
USS Enterprise                  <- Name of this ship
battleship                      <- Name of this ship's class
                                (blank line follows each class)
Pequod                          info for next ship
aircraft carrier
          ... so on for all ships...
```

The assignment folder contains several ship data files formatted as above that are titled “ShipData1”, “ShipData2”, and so on. At the start of each game, each player randomly chooses one of the data files to use (the players don’t have to agree on which file, that's part of the fun). To pick a file, choose a number randomly from 1 to the number of data files and append it to the string "ShipData". Feel free to create your own files if you like and include them as options if you like.

Placing the ships

Once the user's ships have been read in, the program randomly places them on the board. (Although not required, an excellent extension worthy of a super-scooby-snack would be to allow the user to place the ships by clicking on the board). Placing a ship on the board can be a little tricky. You need to position each ship in a row or column of the board, without extending past the board edges.

Also, you need to make sure that at most one ship occupies any cell. To help you out, consider this pseudocode:

```
Before you begin, every cell on board must be initialized to NULL to
indicate that location is empty and available

For each ship to place
   Loop while not found a good location for this ship
      Select random direction (either horizontal or vertical)
      Select random start row between 0 and upper bound
             (bound is NUM_ROWS-1 if horiz, NUM_ROWS-ShipLength if vert)
      Select random start column between 0 and upper bound
              (bound is NUM_COLS-1 if vert, NUM_COLS-ShipLength if horiz)
      If all cells from start to end are empty (i.e. NULL)
         found good location! assign ship, update board & ship data
```

This is not the world's most efficient code. As the board becomes full, it will take more tries to find an open position and this code will slow down, but it will do just fine for our purposes.

One round of the game

A round of the game consists of both players getting to take a turn. The convention in a network game is that the player who hosted the game goes first in each round, the client who connected is second. If playing against the computer, the player goes first, the computer second.

When it is the user's turn, use `GetLocationChosenByUser` to wait for the user to click on the graphics window. The function returns the coordinates of the clicked cell or the coordinates `{-1,-1}` if the mouse click was not on a valid board cell. Once you have a valid choice, send that shot information to the opponent using the `SendShotToOpponentAndWaitForResponse` function (how's that for a descriptive name?). Once you receive the opponent's response supplied by this function, you can then update the graphical display to show the hit/miss status.

When it is the opponent's turn, call `WaitForOpponentShot` to receive the incoming shot from the opponent. Check what is at that location, update your data structure and the graphical display to show it has been fired on, and inform the opponent of the result via `SendResponseToOpponent` function. There are five possible responses to a shot: the cell was already previously fired on, it was a miss, it was a hit, it was the final hit to sink a ship, or it was the final hit to sink the last remaining ship. If the cell was previously fired upon, respond by sending a message to the opponent that they're a bozo for wasting a precious shot. If the cell is empty, respond with a taunting message and mark the cell to indicate a miss. If the cell contains a ship, update for the new hit and respond with a congratulatory message. If that hit sinks the ship, mark it as sunk on your board and respond to the opponent with the name and class of the ship (“You sunk the gunboat USS Parlante”). You also will respond with the position of the sunk ship so the opponent can mark it on their board.

Note that when responding to a shot, the user can pass an arbitrary string message to the opponent along with the result code. Similarly, the opponent can return messages to the user as part of its response. This message-passing allows the players to "talk trash" to each other and provide chatty information such as the name and type of the ship that was just sunk. Experiment with the demo program to get a sense of how this message is used with the `DrawPrintfMessage` function of the graphics module to provide a running commentary throughout the game.

Development strategy

In a project of this size, it is extremely important that you get an early start and work consistently toward your goal. Complex programs are always more manageable to develop if you evolve the

program incrementally, adding one task at a time, testing and debugging each piece before moving on. We suggest that you write the program in stages, as indicated below:

- *Task 1—Design your data structure.* Diagram out on paper the data structure you will be using to represent an entire game, the board, the ships, the classes of ships, etc. You want sensible structures that group pieces of related data together. You should not have unnecessary copies of data, so where appropriate, use pointers to share references to the same data. Ensure your data structure can support the necessary operations for playing the game—here are a few examples to think about:
    - Can I find out what ship, if any, is at location A8?
    - This ship was just sunk. How can I mark all of its cells on the board with the sunk symbol?
    - What color do I use to draw this class of ship?
    - Is the game over yet?

  The type `coord` defined in our graphics module will probably be useful to you. You can assume that board will always be of the constant size `NUM_ROWS` by `NUM_COLS` as #defined in `gbattle.h`, so you can use these constants to create a fixed size 2-dimensional array for the board. The number of ship classes and number of ships is unbounded, do not introduce any hard upper limit on the number of classes or ships in your program. You must dynamically allocate an array of exactly the size needed after you have read the counts from the data file. You should not use any global variables.
- *Task 2—Read data file, place ships, draw user's board.* Write the code to initialize your data structures with ships read from a data file. I highly encourage you to put in some printf statements to write out the data as you read it in to verify this is working before going on. Now randomly place these ships onto the user's board. Use the graphics routines to draw the user's board and verify that each ship is placed in a valid location with no overlaps.
- *Task 3—Process one turn for the user*. The user should be able to click on a location on the opponent's board, send that shot to the opponent, and process and display the opponent's response. At this point, ignore networked players and focus on playing against the computer player. Since the computer player "hides" its ships so predictably, it will simplify testing.
- *Task 4—Process one turn for the opponent*. Wait for the opponent's shot, determine the result, update your data structure, and send a response. Verify that your program can correctly detect hits and misses, and only then go on to add the necessary handling for ships being sunk completely. Add calls to the graphics routines to display the results on the graphical display.
- *Task 5—Loop until the game is over.* String together the sequence of user and opponent turns until the game ends when one player's ships are all sunk. Make sure the graphics are properly updated on each turn. Fine-tune the running commentary of messages being exchanged between the players. Add sounds.
- *Task 6—Network head-to-head.* Last but not least, graduate from the simple computer opponent to true network play. Run two versions of the program on the same computer at first to simplify testing. Then try it out with a friend on another computer. One thing to be careful of is that network flakiness and timeouts may cause moves to get lost. You don't need to implement any error handling or recovery other than detecting when a send/response operation failed (as evidenced by its return value) and quitting the game at the first sign of trouble.

Other random notes and suggestions

- A useful debugging tip: funnel all access to elements of the board array through a function you write which first does bounds-checking and halts with an error when you attempt to access a location outside the allocated array. Given that C itself has no such array bounds-checking, a little off-by-one access here and there can introduce some pretty nasty bugs and you want to find and fix those problems earlier rather than later.
- Don't fall into the trap of attempting to write all the code first, then compile, then test, then debug. Trying to debug the entire game when you're not sure if you can even read in the ship data file is just asking for trouble. Go back and re-read our development strategy for suggestions about how to stage your work.
- Definitely don't worry about networking until you have completely worked through handling the simpler computer player. Our network library tries to shield you from the more messy parts, but networks are unreliable beasts by nature and you don't want to get involved with them until you have a fully working program against the computer player.
- One neat thing about the networking is that you should be able to play your program against itself, against our demo, or against other student programs and communicate Mac-to-Mac, Mac-to-PC and so on without any changes. The lower level details of the protocol are handled by our library and as long as you correctly call out functions and pass the appropriate data, all combinations should work.
- To get the IP address of a Macintosh, go to Control Panels->TCP/IP and read it off from the panel. On a PC, open up Start->Run and type `winipcfg` (95/98) or `ipconfig` (NT) and read it off from the box that comes up. On a UNIX machine, the command `hostname` will tell you the hostname of the local machine and the `arp` command can be used to translate between hostname and IP address.
- For now, we're still going easy on you in terms of freeing memory. You should take care not to allocate any unneeded memory, but we don't ask that you free memory when you're done with it. Since you play one game and quit, there's little reason to put in the work to free the game structure when the game ends, since exiting a program deallocates all program memory anyway. However, for those of you interested in tackling this extra challenge, your section leader is sure to provide some brownie points for your efforts. Remember that correctly freeing things can be a little tricky. You should get your program working without any deallocation first, and only then should you go back and tidy up your use of memory (carefully!).
- There is a random smattering of sounds you can sprinkle about to liven up the game and annoy the poor people working next to you. Supporting sound is not required, but it can be fun and it is easy to play prerecorded sounds. If you are using CodeWarrior, add the Macintosh `sounds.rsrc` file to your project, (just like a .c file). If you are using Visual C++, there are individual .wav files for each sound. You do not need to add the .wav files to your project, just make sure the files are in the same folder as your project. Then (in either environment), add `#include "sound.h"` to your code, and use the function `PlayNamedSound` to play sounds. For example, if you have a sound called "Whoops", you can play it using the function call `PlayNamedSound("Whoops")`.

  The available sounds by name are: `"Missile"`, `"Little Explosion"`, `"Medium Explosion"`, `"Big Explosion"`, `"Little Klaxon"`, `"Big Klaxon"`, `"You Sank My Battleship"`, `"That's Pathetic"`, `"Whoops"`, `"Moo"`, Fanfare", "Song of Joy"

Accessing files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each contains a pre-configured project because there are various special options needed that we didn't want you to have to muck with. For this assignment, please use our starting project rather trying to build your own!

Be warned: the project folder is unusually large because of the sounds and networking support. Here is a listing of its contents:

battlemain.c	Shell of the main program
gbattle.h	Interface file for the graphics module
gbattle.c	Source for the graphics module
netbattle.h	Interface file for the networking module
netbattle.lib	Compiled library for networking module
sounds.rsrc	Resource file containing all sounds (Mac)
moo.wav, etc.	Individual sound files (PC)
ShipData1,2, etc.	Data files with ship and class info
NetBattle Demo	A working program that demonstrates how the game is played (can play against as a networked player, too)

Deliverables

As usual, submit an envelope with a printout of your well-structured, well-commented code, along with a disk containing your source files (do not include the entire project -- it won't even fit on a floppy, just submit the source files you edited). The envelope, printout, and disk should all be clearly marked with CS106X, your name and your section leader's name.

Special note: This time you hand in 3 copies!

For this assignment, there is a new wrinkle in the deliverables: we want you to turn in two additional printed copies of your battlemain.c. These copies should not have your name anywhere on them and should be marked only with your Stanford ID number and your section leader's name for identification.

Next week, each student will be given two randomly-chosen anonymous programs to blindly review, along with a review form. You must complete the reviews, but the results will not impact your grade (unless you don't do the review at all). Think of this as an industry “peer review.” This will provide you with a little more feedback on your programs, and also show you how other students write their programs. There is not one “right way” to do an assignment and it's interesting to learn how other students approach the same program as well as seeing how individual code styles differ. Given this upcoming peer review, it's doubly important that you organize and comment your program so someone else can follow what you did!

If somebody had told me that I would become Pope one day,
I would have studied harder.
— Pope John Paul I, on becoming Pope

If I had only known,
I would have been a locksmith.
—Albert Einstein, on research that led to the A-bomb