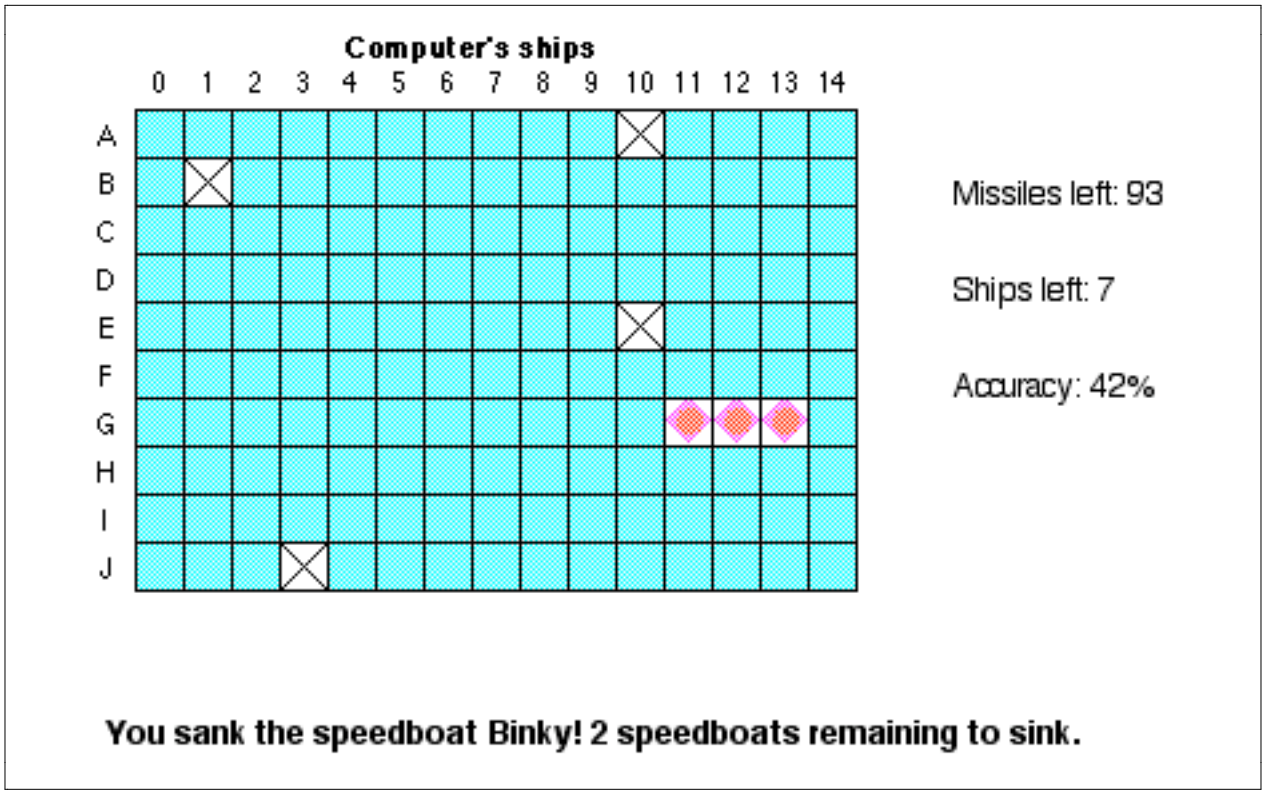
Assignment #2 - Battleship!

Due: Mon Jan 24th 2:15pm

Your next assignment and first substantive program will be to implement the game of Battleship™. This assignment will give you practice working in the procedural paradigm and using C++ pointers, arrays, structures, and file processing. The program also will exercise your ability to decompose a large problem into manageable pieces. The scope is comparable to the last assignment of a CS106A course and should serve as both a valuable refresher on programming and substantive practice with C++ fundamentals.

A program of this size requires up-front planning, so your first tasks are to carefully design your data structure and map out a sensible decomposition strategy. A design checkpoint with your section leader will give you feedback before you move on. In the implementation phase, incremental developing and testing are key to putting your plan into action. We highly recommend starting early to ensure that you have enough time to work through the C++ 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 the game of battleship, the computer positions a set of ships of various sizes on a two-dimensional board. Each square of the board is called a cell. Each ship occupies either a horizontal or vertical line of cells. The user attempts to guess where the ships are hidden by clicking on a cell on the board. If the user's guess hits a location occupied by a ship, the program indicates that a ship has been hit. Otherwise, the location is marked as a miss. If all of the ship's cells have been hit, the ship is marked as sunk. To make the game more interesting, the user will have a limited number of missiles to sink all of the ships. The game ends when the user sinks all

of the ships (the user wins) or runs out of missiles (the computer wins). For variety, each ship is of a particular model (battleship, aircraft carrier, rowboat, etc.) and you will draw different ship models in different colors and play different sounds when each is hit by missiles.

Looking at the sample run on the first page, we can see that the user incorrectly guessed (among others) B1, A10, and J3. The user just sunk the ship "Binky" which occupied the horizontal line G11 through G13. The message at the bottom tells us that Binky was of model "speedboat" and there are two more ships of that model hiding on the board.

Graphics are provided

As you can see from the screen shot, 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 file has comments describing each of the functions in more detail.

- There is a new struct `coord` for a row-col pair.
- There are #define-d constants for the number of rows and columns in the board.
- The `DrawEmptyBoard` function is used at the start of the game to draw the empty board.
- The `GetLocationChosenByUser` function is for the user's turn. It waits for the user to click in the graphics window and returns by reference the location of the cell chosen.
- The `MarkMiss` function exposes and draws a black 'X' over a cell for a missed shot.
- The `MarkHit` function fills a red circle over a cell to identify a hit.
- The `MarkLineAsSunk` function dims out a line of cells. It is used to mark sunken ships.
- The `DrawMessage` function prints a message on the display during game play.

You will not need to edit the `gbattle.h/.cpp` files, however, you are welcome to poke around in the code if you're curious or feel inspired to change anything about the display. The module is implemented using the 106 graphics library described in reader section 5.3.

Designing a data structure

One of your most important tasks for this program comes before you do any coding, when designing your data structures. Think carefully through the information you need to store: the ships, their models, their locations on the board, and so on. A well-designed approach for storing and accessing the data is essential.

As a small example, just consider the problem of storing the ship location information. At times you need to go quickly from a board location to the ship that occupies it. In other situations, you have a ship and need to quickly access its placement on the board. Although you could always determine one given the other, it is worthwhile to build a bit of redundancy into your data structures in order to facilitate both types of access. A two-dimensional array is a good choice for representing the board and provides direct access to each cell. Each cell could keep a pointer to the ship that occupies that cell, or `NULL` to indicate the cell is empty. This gives quick access to the ship at a given location. In order to go in the other direction (from a given ship to its location), each ship structure could store the start and end coordinates of that ship on the board. This would be useful when that ship is sunk and you need to mark all of its cells on the board.

There is other information that you need in various data structures (whether each cell has been fired upon, number of missiles remaining, what model a ship is, and so on) and it will be up to you to work out the appropriate organization for this data. There are many interesting issues to consider: choices between static (fixed-size) and dynamic allocation, where to use structures, where pointers are appropriate, where an enum might be useful, and so on. All of these are valuable opportunities for becoming familiar with the C++ tools for managing data.

Some of the important goals you are trying to achieve:

sensible	Data should not be haphazardly thrown together. If you can't think of a good name for a particular structure, this might be a sign it isn't a logical grouping.
complete	Related pieces are kept together. If you find yourself always passing the same int along with a struct, maybe that int belongs in the struct itself.
compact	Keep wasted space to a minimum and avoid unnecessary redundancy. Don't introduce extra levels of indirection (i.e. pointers) where not beneficial.
efficient	Provide direct and straight-forward access. Consider caching often-needed results or setting up pointers to facilitate quick access rather than repeatedly searching or rearranging data.

Sometimes these goals come in conflict with one another — providing efficient access may require additional storage or an extra level of pointers. You are trying to strike the right overall balance. A decision that's right for this program might go the other way in a different context.

Taking care when designing your structure makes the entire program much easier to develop and debug. Don't shortchange this important step! One of your assigned tasks (discussed later in the handout) is to get feedback from your section leader on the appropriateness of your design before you do any implementation to make sure you are on the right track.

The ship data files

The ship information is stored in a text file. The file begins with the number of ship models followed by the info for each ship model. Next is the number of ships and the info for each ship. The file formatted exactly like this:

```
5                           <- Number of ship models to follow
battleship                  <- Name of this ship model
4                           <- Length of ships of this model
Magenta                     <- Color for drawing this model of ship
Little Explosion.wav        <- Filename of sound played when this model is hit
                            (blank line follows each model)
aircraft carrier            info for next model
5
Red
Big Klaxon.wav
          ... so on for all ship models ...
7                           <- Number of ships that follow
USS Enterprise              <- Name of this ship
battleship                  <- Name of this ship's model
                            (blank line follows each ship)
Pequod                      info for next ship
aircraft carrier
               ... so on for all ships...
```

This file is designed to be very easy to read using the **`getline`** function, which retrieves the file contents line-by line. You can assume the files will be properly formatted with no errors.

Placing the ships

Once the user's ships have been read in, the program randomly places them on the board. The ship needs to occupy a horizontal or vertical line of cells, without extending past the board edges.

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

```
Before you begin, every cell on user's board must be initialized to indicate
that location is empty

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
       Select random start column between 0 and upper bound
             (note the upper bounds depend on direction and ship length)
       If all cells from start to end are empty
          found good location, assign ship here!
```

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.

The player's attack

The provided function `GetLocationChosenByUser` waits for the user to click in the graphics window and returns by reference the coordinates of the cell that was clicked by the user. You check the board to see if that cell is a hit. If so, update the board to show the new hit and congratulate the user. If the cell is empty, ridicule the user and mark the cell to indicate a miss. If that cell had been previously fired upon, tell the user they're a bozo for wasting a missile on a location they've already chosen.

If this hit is one that sinks a ship, mark the board to show that ship has been sunk. You also inform the user of the name and model of ship that was just sunk ("You sunk the aircraft carrier USS Mississippi") and tell the user how many more of that model of ship remain to sink ("There are 3 aircraft carriers still out there"). Anytime you need to give a message to the user, call the provided function `DrawMessage` to write the message at the bottom of the graphics window.

For each turn, you update the scoreboard to indicate how many missiles are left, how many total ships are yet to sink, and what the user's accuracy is (number of hits divided by the number of total missiles fired). Call the provided function `DrawScore` with the appropriate values and it will nicely draw this information in the graphics window.

If the player sinks all the ships, she wins. If the player runs out of missiles, the computer wins.

A suggested development strategy

Complex programs are always easier 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 sketched below:

Task 0	Play the demo
Task 1	Design your data structure and review with your section leader
Task 2	Read files, place ships
Task 3	One player turn, game control
Task 4	Graphics and sounds
Task 5	Deallocation

- *Task 0—Play the demo*. Especially if you aren't familiar with the Battleship game, it is a good idea to start by running our demo to make sure you know what the goal is. Think about what's happening behind the scenes so you get an understanding of what data is being manipulated.

- *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 models 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 model of ship?
  - What has the user's accuracy been so far?
  - How many ships of this model are still active on the board?
  - Is the game over yet?

  You might want to try writing the expressions to obtain the above information out of your data structure to learn how to use your data structure. Drawing diagrams of the various memory allocations and storage locations may also be helpful.

  You can assume that board is 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 models and number of ships is unbounded, so do not introduce any hard upper limit on the number of models or ships. You must dynamically allocate an array of exactly the size needed after you have read the counts from the data file. Although you might feel tempted, you should not use any global variables for this assignment. Global variables can be convenient but they have a lot of drawbacks. In this class we will use them only sparingly and with good reason. For any of our assignments, you can assume that globals are verboten unless we explicitly tell you otherwise.

  Because it's so essential that you start with a good data structure, we require you to run your proposed design by your section leader as an early "checkpoint." You can do this either in email or in person. Email or bring a printout of your data structures. All that's needed is a quick review to verify your choices are appropriate and to offer suggested improvements. We recommend doing this as soon as you get a chance, and certainly should be done before you start implementing the program. This step is a requirement for the assignment and is graded pass/fail on whether you completed it.

- *Task 2—Read data file, place ships, draw board*. The assignment folder contains several files with ship information, formatted as described above. The files are titled “ShipData1”, “ShipData2”, and so on. There are **`NUM_DATA_FILES`** files. When choosing a file to read, concatenate an appropriate random number to the string "ShipData". Feel free to create your own files and update the constant to the new count.

  After making sure you understand the file format, write the code to initialize your data structure with the information read from a file. Examine the structures in the debugger or add some temporary **`cout`** statements to print out the data to confirm all is well before going on.

  Next randomly place those ships on the board. Verify that each ship is placed in a valid location with no overlaps, perhaps by drawing the ships on the board using the functions in the **`gbattle`** module to mark cells.

- *Task 3—One turn at a time, game control*. Get the user's click, determine the result, and update your data structure in response. Verify that your program can correctly detect hits and misses. Once single hits and misses are working, add the necessary handling for ships being sunk completely. Don't yet worry about updating the graphics yet, just use temporary cout statements to report what is happening. Finish the control flow for the entire game by

stringing together a sequence of turns until the game ends, either by sinking all the ships or running out of missiles. The user should be allowed to play as many games as desired.

You may find it helpful to work on this step without randomization— for example, always read data file #1 and assign ships across the first few rows. This gives you reproducible results from run to run which can be quite helpful for debugging. Turn on randomization (and don't forget the call to Randomize!) once you know the basic algorithms are working.

- *Task 4—Graphics and sound*. Adding graphics and sounds is an easy last step. Read over the interface provided in the `gbatte.h`. Call our functions to update the scoreboard on each turn, mark cells for hits and misses, and write status messages on the display. Our CS106 `sound.h` header file exports the function `PlayNamedSound(string filename)`. When a ship is hit, call this function to play the named sound for ships of that model. There is a random smattering of other sound files we provide you can sprinkle about to liven up the game. The function `SetSoundOn(bool onOrOff)` gives you a global control for enabling or disabling sound to avoid annoying your long-suffering roommate during your late night coding marathons.
- *Task 5—Deallocate memory*. At the end of each game, you should free all dynamically allocated memory before starting another. 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!).

Strategies for success

A high-quality solution to this program will take time. It is not recommended that you wait until the night before to hack something together. Here are some of thoughts on key strategies for a successful outcome:

Decomposition: This program is an important exercise in learning how to put decomposition to work in practice. Decomposition is not frosting to spread on the cake when it's done, it's the tool that helps you get the job done. **Decompose problems, not programs**! With the right decomposition, this program is much easier to write, to test, to debug, and to comment. With the wrong decomposition, all of the above will be a real chore.

Incremental Development and Testing: 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 program when you're not sure if you even properly read in the data file is just asking for trouble. Take it step-by-step and thoroughly test and debug each piece before moving on to the next milestone. If you build and test your program in stages, rather than all at once, you will have an easier time finding and correcting your errors.

Watch for sharp edges: C++ is certainly is less forgiving about some common errors that Java is all over. A couple of things in particular to be attentive to:

— C++ does not do array bounds-checking, so it's up to you make sure that are you accessing legitimate indexes. A little off-by-one access here and there can introduce some pretty nasty bugs, so be careful! For example, you may find it valuable to funnel all access to elements of the board array through a function you write which first does bounds-checking and halts with an error if you attempt to access a location that is out of bounds.

— C++'s doesn't force you to initialize variables. If you depend on the starting contents of any variable, struct field, or array element, it is your job to ensure that you properly make the initial assignment.

Coding standards, commenting, style: Getting the program to work is only half the challenge. We're looking for a **well-crafted program with good style— sensible decomposition, clear readability, and intelligent comments.** Your program should be easy to read and stylistically elegant. Be conscious of the style you are developing and aim for consistency. Comment thoughtfully. Take care to avoid any unnecessary code repetition—if you need something more

than once, write one function and call it from both places. Functions should represent small, meaningful subtasks and be named and parameterized appropriately.

To give you some general idea of the scope of the assignment: our solution contains about 250 lines of source (disregarding comments and blank lines). It has been decomposed into about 30 functions. As you can calculate, each function is fairly short, averaging 8 lines. With blank lines and comments, the completed solution prints in just over 10 pages.

Seek help: Our friendly and knowledgeable section leaders staff the LaIR most afternoon and evening hours and offer all kinds of useful help. What a great resource to have a cool, clued-in person to ask for an explanation about a C++ feature, a clarification about an assignment requirement, or advice about how to go about debugging a difficult problem! We are fortunate that we can staff a lot of hours (what other class offers 40 office hours a week?!) but you still outnumber the staff, so we do appreciate your efforts to make efficient use of our resources. A little studying, experimentation, or analysis on your own often allows you to ask a much more targeted question than "my program crashes, fix it." During crunch time (typically the night before assignments are due) things can get a little hectic in the LaIR, so you may find it beneficial to start early and take advantage of some of the lesser-trafficked times (afternoons, early evenings).

Accessing files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Be warned: the project folder is unusually large because of the sounds. Here is a listing of its contents:

battlemain.cpp	Shell of the main program
gbattle.h	Interface file for the graphics module
gbattle.cpp	Source for the graphics module
Sounds	Directory of sound files
ShipData1, etc.	Data files with ship info
Battle Demo	A working program that demonstrates how the game is played

To get started, create your own starter 106 C++ project and add the two sources files gbattle.cpp and battlemain.cpp.

Deliverables

Checkpoint: You must review your data structure with your section leader (preferably by the end of this week, but at the very least, before you start implementing).

Program: You are required to submit both a printed version of your code in lecture, as well as an electronic version via ftp. Both are due before the **beginning of lecture** on the due date. Please submit only those source files you edited (we don't need project, sound, library, etc. files). See the electronic submission handout for details on how to submit the electronic version. The printed version is handed in in class. Be sure the pages are firmly stapled together and that the printout is clearly marked with your name, CS106B, and your section leader's name! By the way, this is a good time to remind you to keep a current backup copy of your work and save a copy of what you submit. Computers are notorious for devouring your most precious work at the perfect time to cause maximum pain.

Remember you may use at most two late days for any assignment in this class. Thus absolutely **no submissions will be accepted after Friday Jan 28th 2:15pm**.

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