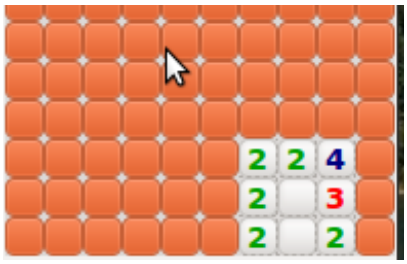
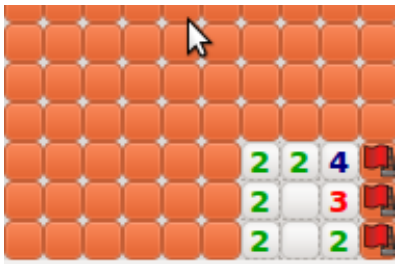
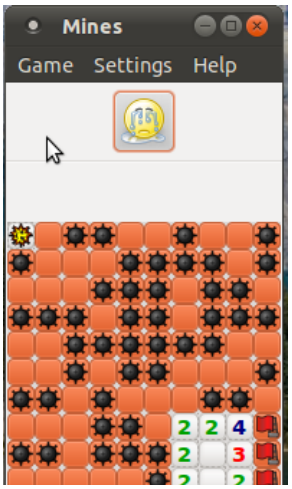
The goal of this project is to write programs that solve a minesweeper game. If you've never played Minesweeper[1] (or Mines in Linux), your first task is to play the game for hours (as important research). In Minesweeper, there is an **8**x**8** minefield of squares, originally all unopened. Hidden mines are buried under **m** squares in the field. Normally, if you step on a mine by opening one of these squares, Game Over ... except in this class project! In ours, it is not fatal to step on a mine if you are making a "necessary guess" when no local inferences are possible, as explained below. As you open squares, numbers are revealed (unless you open a mined square, in which case a bomb is shown). A number on a square indicates how many mines are in the immediately adjacent squares surrounding it. Based on this information, the object of the game is to infer where the mines are and to place flags on those squares.



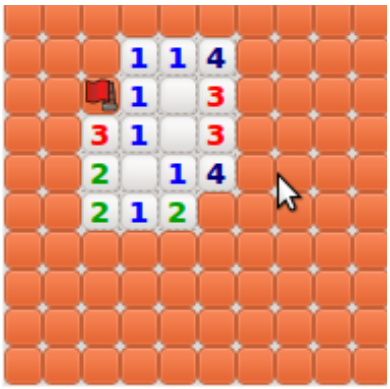
Opening squares, revealed these counts (the blank squares have a count of 0), indicating the number of mines in immediately surrounding neighbors.



From the "3" count, it can be inferred that all of that square's unopened neighbors are mines, so they have been flagged.



Opening the upper left corner revealed a mine, but was it a *necessary guess*? If so, we are safe and game continues. If not, Game Over!



Since the "1" count in the upper left corner is accounted for by the flag, it is safe to open all other squares around it.

Result of opening the surrounding squares.

Note: If you make an unnecessary guess, the game is lost, even if a mine was not buried under the square that was guessed.

There is an element of chance in the game because it is not always possible to infer all the positions of mines. In those cases, a *necessary guess* must be made, which might uncover a mine. In this project, if you make a necessary guess of a mined square, it will not blow up to lose the game. However, your program can still fail if unnecessary guesses are made. *A guess is unnecessary if there is at least one square P for which an inference can be made based solely on P and its immediate neighbors (a one-square inference).*

1 If you do not have Minesweeper, you can find it online at: http://minesweeperonline.com/#intermediate

Your program will be able to perform one of three different actions on a square in a given minefield:

1. Open – indicating that a square is clear of mines; this will reveal a count of surrounding mines (unless the square is mined).
2. Flag – indicating that a square holds a mine.
3. Guess – a cautious open, indicating that there is uncertainty about whether the square is clear (when your guess is necessary, the move is safe even if a mine is buried there); this will reveal either a count or a mine.

To solve the puzzle, your program must take these actions on squares to infer where the mines are and flag them.

Your program succeeds if it:

1. makes no unnecessary guesses,
2. opens no square containing a mine,
3. does not overwrite (open, flag, or guess) any square that was already opened, flagged, or guessed,
4. flags only squares that actually hold mines, and
5. accounts for all **m** mines either by flagging them or making necessary guesses that reveal them.

If all guesses made by your program are necessary, they are correct *even if they reveal a mine*. However, your program fails if it makes an unnecessary guess, *even if there is no mine buried at the square guessed*.

Strategy: Unlike many "function only" programming tasks where a solution can be quickly envisioned and implemented, this task requires a different strategy.

1. Before writing any code, reflect on the task requirements and constraints. *Mentally* explore different approaches and algorithms, considering their potential performance and costs. The metrics of merit are **static code length**, **dynamic execution time**, and **storage requirements**. There are often trade-offs between these parameters. Sometimes *back of the envelope* calculations (e.g., how many memory accesses will be performed) can help illuminate the potential of an approach.
2. Once a promising approach is chosen, a high level language (HLL) implementation (e.g., in C) can deepen its understanding. The HLL implementation is more flexible and convenient for exploring the solution space and should be written before constructing the assembly version where design changes are more costly and difficult to make. For P1-1, you will write a C implementation of the program.
3. Once a working C version is created, it's time to "be the compiler" and see how it translates to MIPS assembly. This is an opportunity to see how HLL constructs are supported on a machine platform (at the ISA level). This level requires the greatest programming effort; but it also uncovers many new opportunities to increase performance and efficiency. You will write the assembly version for P1-2.

P1-1: High Level Language Implementation:

In this section, the first two steps described above will be completed. It's fine to start with a simple implementation that produces an answer; this will help deepen your understanding. Then experiment with your best mentally-formed ideas for a better performing solution. Use any instrumentation techniques that help to assess how much work is being done. Each hour spent exploring here will cut many hours from the assembly level implementation exercise.

The back-end of the game has already been written for you; you are responsible for writing the code that will decide which moves to perform. The following files are provided in a zip file:

`minefield.h`, `minefield.o`, `Makefile`, and `P1-1-shell.c`

Be sure to download the zip file (P1-1-32bit or P1-1-64bit) that is appropriate for your platform.

You can only modify `P1-1-shell.c` (and rename it to `P1-1.c`). Inside `P1-1.c`, you must complete the solver function such that your program can successfully complete the game.

The solver function has the following prototype:

```
void solver(int total_mine_num);
```

The backend will initialize the minefield randomly, and then call `solver(…)` with one parameter: the total number of mines in the field.

To solve the game, you will use functions defined in `minefield.h`:

- `open(r,c)`: Call this function if you are certain a square is clear. One of 2 events will occur:
  1. If you were correct, it will return the number of mines in adjacent squares.
  2. If you were incorrect, Game Over – the game will end (in failure).
- `flag(r,c)`: Call this function if you are certain a square contains a mine. Nothing happens until you have flagged, or uncovered by necessary guesses, all **m** mines. As soon as you flag the **m**th mine, the backend will 'open' all the remaining squares. The game will end (in failure) if any of those remaining squares have mines (i.e. you have flagged some squares incorrectly).
- `guess(r,c)`: Call this function if it is impossible to make a move based solely on logic involving single squares (1-square inferences). If the square has a mine, the game will pretend you flagged it; if the square is clear, it will function as an `open()` call. (Refer to minefield.h to see how to tell which happened.) *You may only call `guess` if there is no certain move; if there is any way to logically use `flag` or `open` when `guess` is called, the game will end (in failure).*
- `display_field_discovered_so_far()`: textual print out of minefield; not technically needed to complete the project, but useful for debugging.

The input parameters (`r` and `c`) of `open`, `flag`, and `guess` indicate the following:

`r`: row number of the square to be opened (0 is top row).

`c`: column number of the square to be opened (0 is leftmost column).

The game will end in one of the following scenarios:

1. a mine is opened
2. an unnecessary guess is made

3. total_mine_num flags are marked

Scenarios 1 and 2 are failures. For Scenario 3, after the last mine is flagged, the backend will immediately 'open' all the remaining unflagged and unopened squares. If any of these squares has a mine, the game ends in failure; otherwise, the game is successfully solved.

Compiling/Running your code:

In a terminal window, run the "`make`" command, which will look for a `Makefile` in the current directory and execute it, compiling and/or linking any files that are necessary for your project to run. For example, if you type the following commands (the text in black), the results of your commands are shown in blue:

```
> cd /mnt/c/Users/gburdell/2035/P1
> mv P1-1-shell.c P1-1.c
> ls
Makefile  minefield.h  minefield.o   P1-1.c
> make
gcc -Wall -O3 -c P1-1.c
gcc -Wall -O3 P1-1.o minefield.o -o msweep
```

Once you use this to create an executable, called "`msweep`", you can run it at the command line:

```
./msweep p s v
```

where `p, s, v` are the following:

- `p` is the probability of any square being occupied by a mine 0<p<1. For example, 0.1. Note: it is only a probability, so do not expect the actual number of mines to be exactly `8*8*probability`. The actual number of mines is passed to your `solver()` function as its second argument.
- `s` is the seed that determines the placement of mines in the field (e.g., a number such as 2046 or 5787). You may find it helpful to reuse a seed to recreate the same minefield and debug specific scenarios. However, your program will ultimately be expected to handle any seed thrown at it. Enter -1 to test with a different random seed each time.
- `v` specifies whether or not you want the "verbose" print out: 1 is yes, 0 is no.

For example:

```
> ./msweep 0.25 -1 1
program runs w/ 25% mine density, a random seed, verbose printing
> ./msweep 0.50 5302 0
program runs w/ 50% mine density, reusable seed, non-verbose
```

When you run your program, the backend will tell you whether you correctly solved the field or whether you made an error and what it was. In this version, we are enforcing these additional constraints:

1. You may only make one action per square. In other words, you may not unflag or open a square that you have flagged.
2. Your code does not need to do anything special when it has finished the game; the game code will detect and take care of the end of the game
3. Unlike the normal game, large sections of squares will not be cleared with a single command – you must manually open all squares, including squares adjacent to ones with '0' adjacent mines.

When have completed the assignment, submit the single file `P1-1.c` to Canvas. Although it is good practice to employ a header file (e.g., `P1-1.h`) for declarations, external variables, etc., in this project please include this information at the beginning of your submitted program file. For your solution to be properly received and graded, there are a few requirements:

1. The file must be named **`P1-1.c`**. *Do not submit the executable* **`P1-1`**.
1. Your program must compile and run with the provided Makefile under Linux.
2. Your solution must be properly uploaded to Canvas before the scheduled due date, **5:00pm on Friday, 28 September 2018.**

During grading, your C code will be compiled and subjected to

- 50 random runs with mine probability 0.25 and
- 50 random runs with mine probability 0.50.

It will be evaluated according to the following criteria: your program

- has good coding style (well-formatted and commented),
- compiles without any errors or warnings using the `-Wall` flag,
- does not crash, and
- successfully solves all 100 games.

P1-2 Assembly Level Implementation: In this part of the project, you will write the performance-focused assembly program that solves minesweeper. A shell program is provided to get you started (`P1-2-shell.asm`). Rename it to `P1-2.asm`.

In this version, execution performance and cost is often equally important. The assessment of your submission will include functional accuracy during 100 trials and performance and efficiency. The code size, dynamic execution length, and operand storage requirements are scored empirically, relative to a baseline solution. The baseline numbers for this project are static code size: **120** instructions, dynamic instruction length: **22,195** instructions (avg.), storage required: **40** words (not including dedicated registers $0, $31). *The dynamic instruction length metric is the maximum of the baseline metric and the average dynamic instruction length of the five fastest student submissions.*

Your score will be determined through the following equation:

$$\text{PercentCredit} = 2 - \frac{\text{Metric}_{\text{YourProgram}}}{\text{Metric}_{\text{BaselineProgram}}}$$

Percent Credit is then used to determine the number of points for the corresponding points category. Important note: while the total score for each part can exceed 100%, especially bad performance can earn *negative credit*. The sum of the combined performance metrics scores (code size, execution length, and storage) will not be less than zero points. **Finally, the performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn performance credit if your implementation fails ten or more of the 100 trials.**

Library Routines: There are two library routines (accessible via the `swi` instruction).

SWI 567: Bury Mines: This routine internally creates a hidden representation of an **8x8** minefield and randomly places mines in it. The mine density is set to **25%**.
INPUTS: There are no inputs to this routine.
OUTPUTS: The number of mines buried is returned in register $1.

A visualization of the minefield is also produced and displayed in which all squares are unopened (shown as lightblue). To help with debugging, the positions of hidden mines are indicated by a small dashed rectangle within each square in which a mine is buried.

SWI 568: Open, Flag, or Guess Square: A square position is given to this routine along with a command to either, open, flag, or guess it.
INPUTS: Register $2 gives a square position as a linear index between 0 and 63 (one of 8x8 = 64 positions).
Register $3 gives the command:

-1: Guess
0: Open
1: Flag

OUTPUTS: Register $4 gives the result:

-1: mine is uncovered
n: number of mines in immediate neighbors (integer from 0 to 8)
9: flag

The -1 or n is returned regardless of whether the square was opened or guessed and regardless of whether a guess was necessary or not.
This routine also detects **Overwrite Errors** which occur if the square specified has already been previously opened/flagged/guessed. These are reported in the lower left window in MiSaSiM.

The minefield visualization displays the results of this routine:

1. Necessary Guess of S: turns the square green and either a mine or a count is revealed (blank if the count is 0);
2. Unnecessary Guess of S: turns the square red and puts a diagonal line through it and either a mine or a count is revealed (blank if the count is 0);
3. Flag of S: square stays blue, but flag is displayed;
4. Open of S: If a mine is buried at S, square turns red and mine is revealed; Otherwise, square turns white and the count is revealed (blank if the count is 0).

Differences from P1-1 (C implementation):

- Your MIPS code must keep track of the number of mines found or flagged and end the program when this count reaches the number of mines originally buried (the result returned from swi 567). This is unlike the C code which automatically ends when the last mine is found/flagged.
- The C backend automatically ends your program when a mined square is opened or a guess is unnecessary. MiSaSiM will not halt your program under these conditions. Instead, the visualization indicates these conditions so that you can move through the trace to see where they occur and what caused them.
- The mine density (25%) is a constant in the MIPS version, while it is an input parameter to the C program.
- The `open`, `flag`, `guess` functions in the C program take the row and column position as input, while the MIPS swi 568 takes a square position as a linear index between 0 and 63 (one of 8x8 = 64 positions).

In order for your solution to be properly received and graded, there are a few requirements.

1. The file must be named **`P1-2.asm`**.
1. Your program must return to the operating system via the **`jr`** instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.
2. Your solution must be properly uploaded to Canvas before the scheduled due date, **5:00pm on Friday, 12 October 2018**.

Implementation Evaluation:

In this project, the functional implementation of MineSweeper in C (**P1-1**) will be evaluated on whether the correct answer is computed. Although proper coding technique (e.g., using the proper data types, operations, control mechanisms, etc.) will be evaluated, parametric performance will not be considered.

The performance implementation of the MineSweeper program in MIPS assembly (**P1-2**) will be evaluated both in terms of correctness and performance. The correctness evaluation employs the same criterion described for the functional implementation. The performance criteria include static code size (# of instructions), dynamic instruction length (# executed instructions), and storage requirements (total number of words in registers or memory, including stack memory). All of these metrics are used to determine the quality of the overall implementation.

Once a candidate algorithm is selected, an implementation is created, debugged, tested, and tuned. In MIPS assembly language, small changes to the implementation can have a large impact on overall execution performance. Often trade-offs arise between static code size, dynamic execution length, and operand storage requirements. Creative approaches and a thorough understanding of the algorithm and programming mechanisms will often yield impressive results.

One final note on design strategy: while optimizing MIPS assembly language might, at times, seem like a job best left to automated processes (i.e., compilers), it underscores a key element of engineering. Almost all engineering problems require multidimensional evaluation of alternative design approaches. Knowledge and creativity are critical to effective problem solving.

Hints and Warm-up Exercises:

- Play the game yourself until you are familiar with it.
- Attempt to write out your logic for playing as though you were instructing a friend on how to play the game. Translate this logic into computer code.
- It is helpful to create a working field array that contains the status of each square (e.g., unopened, mine, mine count of neighbors, etc.). A warm-up exercise is to run through each square position and call `guess` on it and then keep track of the status as information is revealed. (You won't be able to scan the entire field, of course, because you'll soon make an unnecessary guess, but this will allow you to create and test some of the infrastructure you need.)
- Another useful warm-up exercise is to write a function that counts the number of adjacent neighbors each square has, based on its position (it doesn't matter whether they are open/not). This is useful for testing whether you have the boundary conditions correct in accessing the immediate neighbors of a square. This function should take about 20 lines of C code. Once you get this working correctly, it is easier to modify it to count other

things, like how many flagged neighbors a given square has, or how many unopened neighbors it has.

- Think about this: for a given square, under what conditions can all the unopened adjacent squares be opened? Under what conditions should all the unopened adjacent squares be flagged?
- The baseline program does *not* use recursion.
- Be sure to start early! This project may appear deceptively easy.

Project Grading: The project weighting will be determined as follows:

part	*description*	*due date 5:00pm*	*percent*
P1-1	Minesweeper (C implementation)	Fri., 28 September 2018	25
P1-2	Minesweeper (Assembly)	Fri., 12 October 2018	
	correct operation, technique & style		25
	static code size		15
	dynamic execution length		25
	operand storage requirements		10
	total		100

Honor Policy: In all programming assignments, you should design, implement, and test your own code. Any submitted assignment containing non-shell code that is not fully created and debugged by the student constitutes academic misconduct. You should not share code, debug code, or discuss its performance with anyone. Once you begin implementing your solution, you must work alone.

Good luck and happy coding!