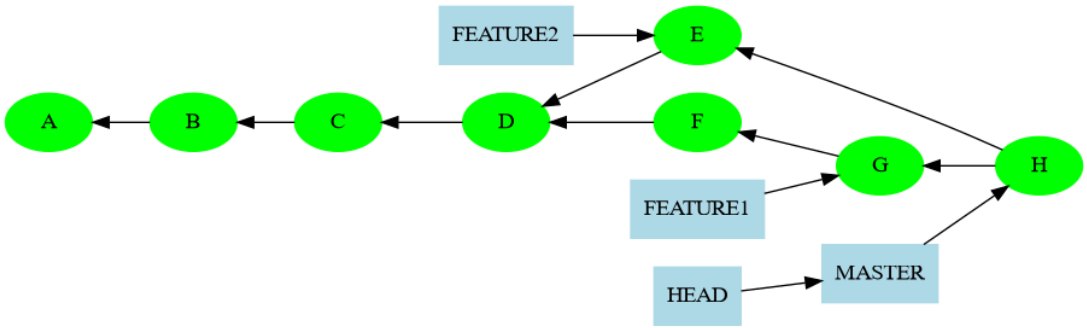
CS 1XC3 Lab 3: Branches, Merging and Conflicts in Git Repositories

Feb. 6^{th} - 10^{th}, 2023

Contents

1 Introduction: Git Branching and Merging



In real-world development, it is common to use a **Production Branch** (generally the master branch) which is kept in working condition for use by the general public. Any active development takes place in other branches. Consider the following case study.

You are working on a project with a team of developers, and you decide to add a new features while your team members are also working on their own independent features.

1. You create a branch, **feature1 off of the master** branch.
2. You proceed to work in this branch, committing as you go like a good programmer.
3. While you are working, your team member creates a new branch, `feature2`, by branching `master`.
4. Your team member commits their work and merges `feature2` back into master.
5. after testing your new feature, you also merge your branch back into `master`. Effectively, you are also merging with **feature2**.

If you need some additional reference for understanding branches and merging, take a look at the following:

- `https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging`

2 Activity: Fixing a Very Broken Chess Game

Some hapless developer has been working on a command-line chess program, but has been somewhat overzealous with branching his repository. It is up to you to fix it!

If you are unfamiliar with chess, you don't have to be able to play it to do this lab, but a general familiarity with the rules will probably be helpful: `https://www.youtube.com/watch?v=IU6k-4rKf-g`

You objective is to do everything required to merge *everything* back into `main` (github's equivalent to `master`). Of course, anyone can resolve merge conflicts randomly (for example, by rejecting all changes which do not merge nicely). So, however you choose to approach this problem, the following conditions must be met by the final chess program.

- When the algorithm displays the game board:
  - Numbers should exist around the border of the board, which are used to determine the coordinate positions of pieces on the board.
  - The border must have double, not single, lines.
  - The board should have grid lines.
  - The board should display black and white spaces differently.
- The function checking for valid moves should:
  - check to make sure the coordinates entered are actually on the game board
  - Include logic for all piece movement
  - Parse the input string inside of the function itself.
- The game's main loop should :
  - Alternate between white and black players.
  - Acquire input from the player, check that the input constitutes a valid move, and then make that move if it is valid.
  - If a move is not valid, the game should prompt the player for another, less invalid move without displaying the board again.

This program is not a complete chess game. The author would prefer if you thought of this as "a program exhibiting chess-like behaviour."

The following are things the game is NOT intended to do. This list is also for the benefit of people who know the rules of chess well already, so if you haven't heard of something in this list DON'T WORRY ABOUT IT!

- There are no black pieces on the board.
- It doesn't matter which player's turn it is, the only thing each player can do is move white chess pieces.
- There is no castling rule implemented.
- There is no en passant rule implemented.
- There is no pawn promotion implemented.
- The game does not know when a check or checkmate occurs.
- The game does not keep track of previous moves, and can not determine if a draw has occured.
- Players of the game can not resign, but a quit option is provided.
- White can capture White's own pieces.
- This is not an exhaustive list of the things this program can't do.

Pretty much the only thing this program does do is display a chess board, and allow white pieces to move around the board according to their standard logic.

You do not have to correct errors in logic pre-existing within the code (if you find any), you just have to perform the merges successfully, and with some degree of intelligence.

The point of this exercise is that you will not have to create any new code. *All the code you need to complete this exercise is already in the repository, you just have to merge it together correctly!*

2.1 Procedure

- To complete this activity, you will need to make a mirrored clone of the following github repository.
  - `https://github.com/Zed-devp/Lab3`
- Simply forking this repository will not allow you to make it private. Please follow the following steps from whatever bash prompt you have set up:
- Create a bare clone of the repository.

```
$ git clone --bare git@github.com:1XC3/L03.git
```

- Create a new private repository on Github, and call it "1XC3_L03".
- Mirror-push your bare clone to your new private repo.

```
$ git push --mirror git@github.com:<your username>/1XC3_L03.git
```

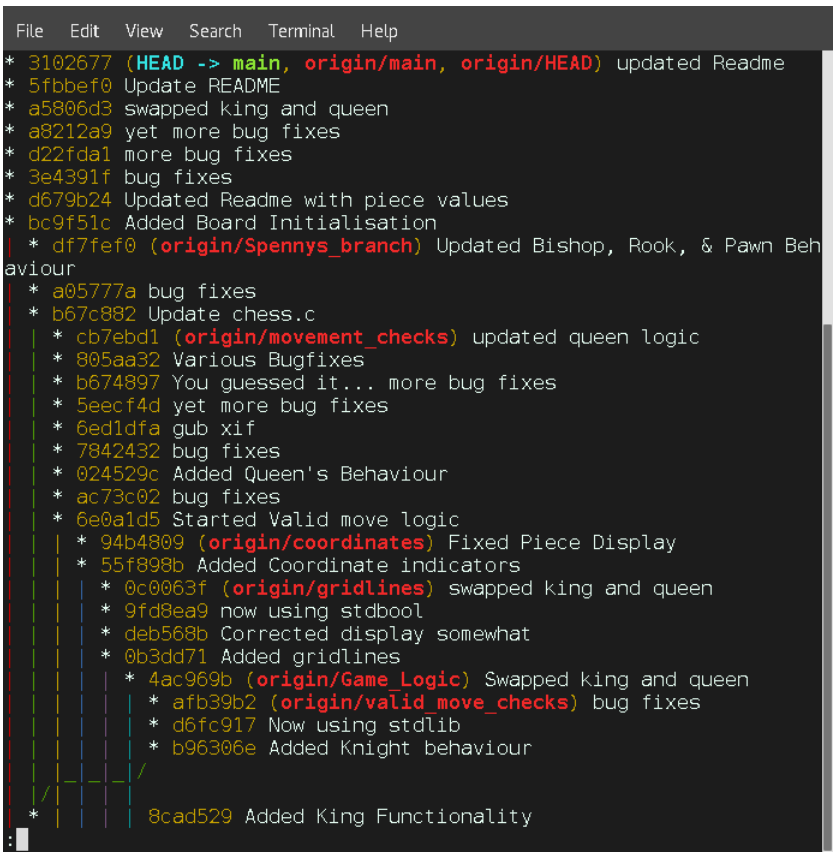
- Remove the temporary local repository you set up in step 1.
- You can now clone your 1XC3_L03 repository on your local machine.

```
$ git clone <URL of your private 1XC3_L03 repo>
```

- Next, add 1XC3_L03 as a submodule to your CS1XC3 master submission repository in a new "L03" directory.

```
$ git submodule add <URL of your private 1XC3_L03 repo>
```

- If you pull the latest changes, you should be able to see four files:
  - `chess.c`
    - This is the source code file for our chess program.
  - `README`
    - In GitHub, if a `README` file exists in the root directory of your repository, the contents are rendered in HTML text on the front page of your repository.
    - The point of the README is to explain the code project, contributor policies, usage information, and other good stuff.
  - `makefile`
    - Makefiles are used to be able to build code quickly and easily. This one is blank because we will have plenty of time to talk about makefiles in another lab.
  - `example_output.txt`
    - This file contains the example output of the correctly merged terminal chess program in operation. You are trying to match this output, especially in parts of the lab concerning board display.
- If you invoke `git log -graph -all -oneline` in the submodule repository, you'll know you've done the previous steps correctly if you get something that looks like this:



From here, use the various commands in git, as well as file editing in your favourite text editor (emacs of course) (or nano if you're on the pascal server), to merge those branches as described above! Here are some hints:

- `git log -graph -all -oneline` is the most space efficient way to view all the branches in the repository.
- `git status` should always be used before `git commit`.
- `git checkout <branch>` switches branches
- `git merge <branch>` merges the active branch into the specified branch.
- It is a very, very good idea to take some time after each merge to fix any syntax errors, and test out the program.
  - Sometimes the program will give warnings indicating that certain functions haven't been defined. Don't worry, these functions are definitely defined in some other branch.
  - Each branch post merger should minimally compile.
- To re-iterate, this lab does not require you to write any new code. You must use the code you've been given.
- You can also use checkout to roll back the repo if you've made an error during merging.

Once you're finished, commit your work for evaluation.

3 Midterm Practice Question

3.1 Question A

An integer which when added to 100 is a perfect square number, and then added to 168 is another perfect square number. Please write a C program to find all the integers that satisfy the above condition.
Hint: suppose the number is x.

1. x + 100 = n^2, x + 100 + 168 = m^2
2. $m^2 - n^2$ = (m + n)(m - n) = 168

3. Let m + n = i, m - n = j, then i × j = 168, at least one of i and j is even
4. Since m = (i + j) / 2, n = (i - j) / 2, i and j are either both even or odd
5. From step 3 and 4, we know that both i and j are even numbers greater than or equal to 2
6. Since i * j = 168 and j >= 2, then 1 < i < 168 / 2 + 1
7. All the numbers of i can be found by iterating in a loop.

3.2 Question B

A number is said to be "perfect" if it is exactly equal to the sum of its factors (except itself). For example, 6 = 1 + 2 + 3. Please write a C program to all the perfect numbers up to 1000.
Hint: please google the procedures of how to solve this problem, and then implement the algorithm based on the descriptions by your own.

4 Grading

Your work will be given a score out of 20 for Steps 1 and 2, 10 for Step 3 (5 for each question) based on:

- Whether your produced code compiles, and is free of syntax errors and compiler warnings.
- The degree to which your game logic matches the logic as demonstrated in the example file.
- You do not have to match the source code the example file was generated from (minor variations are permissible), but the logic itself should have only minor variations or deviations.
- For the midterm practice questions, the codes are executable and the results (output) are correct.