1. A breakdown of how your script handles arguments and options.

The script handles arguments and options through a structured three-phase approach: initialization, parsing, and assignment/validation.

1. **Initialization:**
    - **The script first initializes variables that control behavior:**
        - **[ShowLineNumbers=0] — flag to determine if line numbers should be printed.**
        - **[InvertMatch=0] — flag to determine if matching should be inverted.**
        - **[SearchPattern=""] and [FileName=""] — placeholders for positional arguments.**
2. **Option Parsing (Bonus 2):**
    - **The script uses the getopts utility to parse optional flags -v and -n:**
        - **-v sets [InvertMatch=1], indicating that lines not matching the search pattern should be printed.**
        - **-n sets [ShowLineNumbers=1], indicating that output should include line numbers.**
    - **If an unknown option is provided, the script immediately calls the usage function to print usage information and exit.**
3. **Help Option (--help) (Bonus 1):**
    - **The script supports the --help flag, which, when passed as an argument, immediately triggers the display of the usage instructions.**
    - **The usage function provides the user with:**
        - **A description of the available options (-v, -n, and --help).**
        - **An example usage for clarity.**

        - **Information on the two positional arguments: <pattern> and <file>.**
    - **If the user runs the script with --help or an invalid option, the script will call the usage function and exit, ensuring the user knows how to properly use the script.**
4. **Argument Handling and Validation:**
    - **After parsing options, the script uses shift $((OPTIND - 1)) to skip over the processed flags, ensuring that $1 and $2 refer to the remaining positional arguments.**
    - **It checks if fewer than two arguments remain ($# -lt 2) and shows an error if necessary.**
    - **Then, it assigns:**
        - **SearchPattern=$1**
        - **FileName=$2**
    - **Additional validation is performed:**
        - **It verifies that the search pattern is not mistakenly a file by checking [-f "$SearchPattern"].**
        - **It ensures that the file to search exists by checking [-f "$FileName"].**
    - **If any validation fails, the usage function is called to guide the user.**
5. **Behavior Control During File Reading:**
    - **The flags set earlier (InvertMatch and ShowLineNumbers) control how each line is processed and displayed while reading the file.**
    - **Matching logic (case-insensitive in the updated version) and output formatting depend dynamically on the provided options.**

2. A short paragraph: If you were to support regex or -i/-c/-l options, how would your structure change?

If I were to support additional features such as regular expressions or options like -i, -c, and -l, there would be three main structural changes to the script: option parsing, input validation, and search logic implementation.

In the option parsing phase, I would extend the getopts section to recognize the new flags and update the usage function to document their behaviors clearly. During the validation phase, additional checks would ensure that options are used correctly, regular expressions are well-formed (if needed), and that the pattern and file inputs are appropriately distinguished to avoid user errors, such as mistakenly providing a filename where a search pattern is expected.

Finally, the search logic would become more modular to support different functionalities dynamically. For example, enabling case-insensitive matching via the -i flag would require conditionally converting the search pattern and file lines to lowercase only when the option is active, rather than doing so unconditionally. Similarly, supporting options like -c (count matches) or -l (list matching files) would involve adding new logic branches to handle those specific behaviors without affecting the core matching mechanism. Overall, the structure would become more flexible and extensible to accommodate new features cleanly.

3. What part of the script was hardest to implement and why?

The most challenging part of the script to implement was the validation section. This portion required careful consideration of multiple scenarios, such as handling cases where the user provides too few arguments, supplying an incorrect file path, or mistakenly swapping the positions of the search pattern and file name. Developing the necessary conditional checks to address these situations and ensure proper input validation was complex. Additionally, it was important to ensure the script could guide the user effectively in case of any errors, which added to the complexity of the implementation.

Screen shots + ([--help] Bonus):