CS 253: Introduction to Systems Programming
The match Utility

Learning Objectives

1. Makefiles: Create a Makefile with multiple targets including the default, `all` and `clean`
2. Makefiles: Use `CFLAGS` to control compilation
3. Makefiles: Make rebuilds *only* those artifacts that are out-of-date with recent changes
4. C Programming: Character strings and functions, `fgets()`, and command-line arguments

In this assignment, you will develop a makefile and a simple pattern matching utility akin to `fgrep`. While a makefile is a bit of an overkill for such a simple program, this is a preferable first encounter with makefiles than is the Linux kernel's makefiles appearing in CS453. We will introduce more complexity and features into our CS253 makefiles as the semester progresses. The specification for the `match` utility and its makefile appear below.

The match Utility Specification

NAME

```
match -- simple pattern matcher
```

SYNOPSIS

```
match [-i] pattern
```

DESCRIPTION

The `match` utility reads lines of text from stdin and prints only those lines of text containing a string matching the specified *pattern* (similar to fgrep) to stdout. A line whose text includes a matching string is said to *match* the pattern. Lines that do not match are not printed. Lines containing one or more matches are are printed only once. A *pattern* is a simple string (akin to `fgrep`); regular expressions are not supported. Unlike fgrep, `match` always reads from stdin; there is no support for filenames as command-line arguments.

By default, `match` is case-sensitive; the pattern "foo" will match lines containing the string "foo" but not lines containing "Foo" nor "FOO". However, use of the `-i` option enables case-insensitive matching in which the patterns "foo" and "FOO" and "Foo" will all match a line containing the string "Foo".

ERRORS

If the user does not supply a *pattern*, or if the command line contains an unsupported option or an extraneous argument, `match` prints the message "usage: match [-i] pattern" on stderr and exits.

EXIT STATUS

The `match` utility exits with one of the following exit status values (akin to `fgrep`):

0 One or more lines were matched

1 No lines matched

2 An error occurred

EXAMPLES

```
#Print lines containing the lower-case string, foo
match foo <testData.txt
I went to the food market.

#Case-insensitive matching
match -i FOO <testData.txt
I went to the food market.

#User error (missing pattern)
match <testData.txt
```

The match Makefile Specification

Your implementation of the `match` utility must build with GNU `make` on onyx, and your Makefile must support the *targets* illustrated below.

```
#Rebuild the executable program, match, with the default target (all)
make
```

```
#Rebuild only those artifacts (e.g. match.o and match) that are out-of-date
make all

#Remove all the build artifacts (binary files)
make clean
```

Your Makefile should arrange for `make` to rebuild the executable program if it is out-of-date with any of its prerequisite object files (i.e. *.o). Likewise, an object file must be rebuilt if its associated source (i.e. *.c) files are out-of-date (don't worry about the header files for the runtime libraries — we'll discuss header file dependancies later in the semester). Your Makefile should only rebuild artifacts that are out-of-date. For example, `make` should do nothing the second time if you enter the command, `make all`, twice in a row.

Recall that **make** examines the timestamps of a *target* and each of its *prerequisite* files. A *target* file is out-of-date (and must be re-built) if its timestamp is older than that of any of its *dependencies* . You can examine the time-stamps of all files in the current directory with the command, `ls -l *`. A Makefile rule defines the *dependencies* for a *target*. For example, the rule,

```
match:  match.o
        gcc match.o -o match
```

...specifies the executable file, `match`, must be rebuilt using gcc if it's out-of-date with `match.o`. Recall that `make` recursively follows dependencies, bringing each up-to-date before checking the current target.

Note: Your `Makefile` must arrange for the compiler to print all warning messages, and compile your C sources using the C99 version of the language. You can do this by including a statement, `CFLAGS = -Wall -std=c99`, near the top of your `Makefile`.

Recommended Procedure

1. Download the project's starter files from Blackboard. Confirm that you can build and execute the starter code (it runs but doesn't do anything except flag usage errors).

2. Add code to `match.c` to implement the match specification.

3. Test thoroughly! This and most product specifications focus on a program's *happy path*, its behavior when used as expected. When creating test cases, you must consider the program's *exceptions*, its behavior when processing unexpected or otherwise unusual conditions whose examples include:

- Single character patterns

- Patterns matching a string at the end of a line
- Invalid command line arguments
- Blank lines
- Zero-length lines
- Very long lines
- Empty files

4. Implement the project `Makefile` with targets `all` and `clean` discussed above (these are the minimum required targets). The default target should be `all`.

5. Test your `Makefile` carefully. Will `make clean` remove all binary artifacts of the build? Will `make all` rebuild the executable if it doesn't exist? Is the executable really named, `match`? Did you remember to compile with the `-Wall -std=c99` options? Does your code compile without warnings? If the executable already exists and is up-to-date, will `make` simply note that there's nothing to do?

Submitting

Submit this project using the onyx `submit` tool:

1. Login to onyx and change into your project's directory.
2. Verify all of your project's files are in this directory (not a subdirectory). At a minimum, these must include `match.c` and your `Makefile`).
3. Delete any binary artifacts left over from previous builds. Your binaries will not necessarily work on other developers' hardware and operating system. Many developers consider submission of binaries to your project's repository to be unprofessional.
4. `submit username cs253 p2` where *username* is your instructor's onyx username.

Hints, Assumptions and References

The grader for this project exercises features of your `Makefile`. Read Mecklenburg[1] Chapters 1 and 2 before starting. You may also wish to review the example makefiles in ExampleMakefile1, ExampleMakefile2, and ExampleMakefile3 available from the "CS253 Files, Slides and Examples" content on Blackboard.

The `strstr` function searches for the location of a substring within a string. See, `man strstr`, for its documentation. The `strcasestr` function is similar to `strstr` but case-insensitive.

The `strcmp` function will compare two strings and return 0 if they are equal. See `man strcmp`, for documentation.

The `fgets` API will safely read a line of text from `stdin`. See, `man fgets`, for its documentation. The `fgets` API returns a `NULL` pointer if either an end-of-file or an I/O error occurs. You may handle both cases identically by assuming, if anything went wrong, an end-of-file condition arose. We'll learn how to handle I/O errors properly in a later project.

You will need to declare an array of char where `fgets` will store each line of text read from `stdin`. K&R Ch1 has a section on arrays and another on character arrays. You may assume valid text files will never contain more than 1023 bytes in a line, but what about an invalid file? Expect the grader's test programs will deliberately try to crash your program by storing data in non-existing array elements. No matter what you decide to do when this happens, your program should not crash. Your program should never crash!

The starter code uses the `exit()` API to abruptly exit your program with a specified exit status code, bypassing the need to return from main. You can find the documentation with, `man 3 exit`.

[1] https://www.oreilly.com/openbook/make3/book/index.csp

The `fprintf` API is similar to `printf` but will allow you to write to a specified location (rather than always writing to stdout). To output a non-descriptive error message to stderr, you could use: `fprintf(stderr,"Oopsie!")`. See `man fprintf` for more details.