Changes announced on September 12th to make nmake more make-like:

The complete rules for comments are:

A. Always ignore lines beginning with #.

B. Until the first rule begins, ignore lines beginning with whitespace whose
   first non-whitespace character is #.

C. When a # appears in a TARGET:PREREQUISITE line, the remainder of the line
   is ignored.

~~~~~~

Changes announced on September 4th to make nmake more make-like:

1. The comment convention only applies to lines containing the
     TARGET : [PREREQUISITES]*.
   part of a rule.

2. Write the message
      nmake: 'TARGET' is up to date.
   to stdout if command line TARGET is up to date.

3. If nmakefile (or NMAKEFILE) does not exist, issue a warning and continue as
   if it was empty.

4. If a file is a target in more than one rule with an associated command,
   issue a warning and use the command in the later rule.

~~~~~~

P R E L I M I N A R Y S P E C I F I C A T I O N

Due 2:00 AM, Friday, 16 September 2016

CS-323 Homework #1 Make

(60 points) Write in C a (new) make utility

nmake [-f NMAKEFILE] [target]*

that "executes" shell commands from the file nmakefile (or the file NMAKEFILE specified by the -f option) to update the targets specified in the command. (For a description of the real make, see Matthew & Stones, pp. 377-392.)

An nmakefile contains a set of target rules of the form

```
TARGET : [PREREQUISITE]*
  COMMAND
```

Each rule specifies a relationship where the TARGET file depends on each of the PREREQUISITE files and can be "updated" by executing the associated shell COMMAND. The name of the TARGET file must begin in the first column; the whitespace surrounding the colon is optional; the list of PREREQUISITE files is delimited by whitespace and may be empty; and the shell COMMAND (if present) must be preceded by whitespace.

When a # (number sign) is encountered, the remainder of the line is treated as a comment and ignored. Lines containing nothing other than whitespace (or with nothing other than whitespace preceding a #) are also ignored.

A file may appear in the target list of more than one rule, in which case it

depends on every file in the prerequisite list of any of those rules. However, only one of those rules may be followed by a shell command.

For example, the following, very simple nmakefile says that `pgm' depends on three files, a.o, b.o, and c.o, which in turn depend on the corresponding .c files and a common file incl.h:

```
pgm: a.o b.o c.o
     cc -g a.o b.o c.o -o pgm
a.o: incl.h a.c
     cc -c a.c
b.o: incl.h
b.o: b.c
     cc -c b.c
c.o:
     cc -c c.c
c.o: c.c
c.o: incl.h
```

Note that the last three rules have the same effect as the single rule

```
c.o: c.c incl.h
     cc -c c.c
```

A target file is considered to be out-of-date (and in need of updating) if
* it does not exist (see stat(), i.e., "man 2 stat" or M&S, pp. 106-108); or
* at least one of its prerequisite files has been updated during the execution of nmake (which can only be determined AFTER all of its prerequisite files have been checked); or
* its last modification time is earlier than that of at least one of its prerequisite files (see stat() and difftime(), i.e., "man 2 stat" and "man 3 difftime").

To update an out-of-date target file, nmake "executes" the shell command (if any) associated with the file by ANY rule (not just one that forces an update).

nmake decides which commands to execute by performing post-order, depth-first searches beginning from each target file specified on the command line (in left-to-right order). When it discovers a file that needs to be updated, it writes the associated shell command (if any) to stdout with leading whitespace removed and marks that file as up to date (so that files are only updated once). In practice each command would be executed by its own shell, using the system() function; and if a command returned a nonzero status (i.e., system() returned a nonzero value), then nmake would write a one-line error message to stderr and terminate. However, we will dispense with this step. Note that the commands can be output in any order that is consistent with the rules.

Additional Functionality (each invoked by AT MOST 6 of the 60 tests)
~~~~~~

a. Treat the FIRST ; (semicolon) if any in the list of prerequisites as a newline, and treat the rest of the line (if nonblank) as a command even if it does not begin with whitespace. E.g.,

```
nmake : nmake.o ; cc -g -o nmake nmake.o
```

is equivalent to

```
nmake : nmake.o
  cc -g -o nmake nmake.o
```

b. Allow rules to have multiple targets that are delimited by whitespace. The effect is equivalent to replicating the rule for each target. E.g.,

```
nmake.o table.o : table.h
```
~~~~~~

is equivalent to

```
nmake.o : table.h
table.o : table.h
```

c. Allow rules to have multiple associated commands, each on a separate line beginning with whitespace and immediately following the first command. The commands are to be executed sequentially in order of appearance. E.g.,

```
nmake : nmake.o
  cc -g -o nmake nmake.o
  rm nmake.o
```

causes both the cc and rm commands to be printed when nmake is updated.

d. Check for circular dependencies (e.g., A depends on B, B depends on C, and C depends on A) that arise during the post-order, depth-first search, and if one is detected, write a one-line warning message to stderr and ignore that (last) dependency. A circular dependency may have any number of links. E.g.,

```
a : b
b : c
c : a
d : e
```

will generate an error message if you try to nmake a (or b or c), but not if you try to make d (assuming that e exists).

e. When the built-in macro $@ appears within a command, replace it by the name of the target being updated. E.g.,

```
nmake.o :
  cc -g -o $@ nmake.c
```

is equivalent to

```
nmake.o :
  cc -g -o nmake.o nmake.c
```

f. When the built-in macro $^ appears within a command, replace it by a string containing the names of all prerequisites of the target separated by blanks and with duplicates removed (order is not significant). E.g.,

```
nmake : nmake.o table.o
  cc -g -o nmake $^
```

is equivalent to

```
nmake : nmake.o table.o
  cc -g -o nmake nmake.o table.o
```

g. When the macro $? appears, replace it by a string containing the names of all prerequisites of the target that are _newer_ than or have been updated, separated by blanks and with duplicates removed (order is not significant).

Use the submit command (see below) to turn in the C source file(s) for nmake, a Makefile, and your log file (see below) as assignment 1; e.g.,

```
% /c/cs323/bin/submit 1 Makefile nmake.c log.nmake
```

YOU MUST SUBMIT YOUR FILES (INCLUDING THE LOG FILE) AT THE END OF ANY SESSION WHERE YOU HAVE SPENT AT LEAST ONE HOUR WRITING OR DEBUGGING CODE, AND AT LEAST ONCE EVERY HOUR DURING LONGER SESSIONS. (All submissions are retained.)

Notes
~~~~~

1. The behavior of nmake is similar to that of "make -nr", but may not be the same in all cases.

2. To avoid any limit on the size of an input line, nmake should use the function getLine() which is defined in Hwk1/getLine.h and implemented in Hwk1/getLine.c. Your source file(s) should #include /c/cs323/Hwk1/getLine.h rather than the name of a local copy; similarly your Makefile should specify Hwk1/getLine.o rather than a local copy.

3. As nmake reads the nmakefile, it should parse the rules read into targets, prerequisites, and commands and store this information in a data structure of your choice. No further reading of the file or parsing of the rules is permitted. Note that efficiency is not important, so your structure need not use hash tables or balanced binary search trees to achieve fast search.

4. nmake should assume that filenames do not contain colons, semicolons, whitespace, null characters (i.e., '\0'), or any characters such as $, {, and } that have special meaning to the shell.

5. nmake should be relatively robust but not bombproof (e.g., you may assume that malloc() and realloc() never fail) and should detect the following errors:
   a. There is no nmakefile.
   b. There is no NMAKEFILE argument following -f on the command line.
   c. A rule is malformed.
   d. A file is a target in more than one rule with an associated command.
   e. A command precedes the first rule.
   f. A prerequisite does not exist and is not itself the target of some rule.
      (Checked only if the target of the rule is being checked.)
   g. A filename specified on the command line neither exists nor is the target of a rule in the nmakefile.

   writing a one-line message written to stderr, rather than stdout, and causing nmake to exit immediately.

6. When deciding whether to update a target, nmake may check its prerequisites in any order. Thus commands may be output in any order that is consistent with the rules. For example, for the invocation

```
% nmake a
```

   with the nmakefile

```
a : b c ; echo a b c
b : d   ; echo b d
c : d   ; echo c d
```

   if d is the newest file then either

```
echo b d
echo c d
echo a b c
```

   or

```
echo c d
echo b d
echo a b c
```

   is acceptable, but

```
echo a b c
echo b d
echo c d
```
~~~~~

is not.

7. To detect certain errors associated with the use of malloc(), calloc(), realloc(), and free(), all tests will use valgrind, a system that detects uninitialized variables, malloc() errors, etc. To use it yourself, type

   ```
   % /usr/bin/valgrind -q ./nmake ...
   ```

   See http://valgrind.org/ for details. (There are links to this page and an explanation of common valgrind errors on the class web page.)

8. nmake need not free() all storage before exiting, but all heap storage (i.e., any blocks allocated by malloc(), calloc(), or realloc()) must be reachable at that time.

   Note: When main() returns, all stack storage (where storage for arguments and automatic locals is allocated) is freed before valgrind checks for unreachable heap storage. Thus if an argument or automatic local variable contains the only pointer to a block on the heap, valgrind will report that block as unreachable. One solution is to call exit() instead of returning, so that stack storage is not freed. Another is to use a static local or global variable (which is not stored on the stack) to hold the pointer.

9. The touch command is useful for changing the modified date of a file while debugging nmake (e.g., to create test cases). However, it should not be called by nmake.

A. Hwk1/nmake.h contains the #include files and some macros from my solution.

B. Keep track of how you spend your time in completing this assignment. Your log file should be of the general form (that below is fictional):

   ```
   ESTIMATE of time to complete assignment: 10 hours

         Start  Time
   Date  Time   Spent Work completed
   ----  -----  ----  --------------
   9/03  10:15  0:50  Read assignment and documentation for make
   9/05  20:15  1:00  Played with Hwk1/nmake to discover how it works
   9/08  12:45  0:30  Sketched solution using recursion
   9/08  14:00  0:10  Recursive solution cannot handle all cases
   9/10  21:20  1:00  Sketched new solution using two stacks and a queue
   9/12  09:00  5:00  Wrote/typed-in program; eliminated compile-time errors
   9/13  20:00  1:30  Debugged program; it passes every public test
                -----
                10:00  TOTAL time spent

   I discussed my solution with: Peter Salovey, Ben Polak, Tamar Gendler,
   and Jonathan Holloway.

   <A brief discussion of the major difficulties encountered>
   ```

   but MUST contain

   * your estimate of the time required (made prior to writing any code),

   * the total time you actually spent on the assignment,

   * the names of all others (but not members of the teaching staff) with whom you discussed the assignment for more than 10 minutes, and

   * a brief discussion (100 words MINIMUM) of the major difficulties that you encountered in developing/debugging (and there will always be some).

This log will generally be worth 5-10% of the total grade.

N.B. To facilitate analysis, the log file MUST be the only file submitted whose name contains the string "log" and the estimate / total MUST be on the only line in that file that contains the string "ESTIMATE" / "TOTAL".

C. The submit program can be invoked in eight different ways:

```
    /c/cs323/bin/submit  1  Makefile nmake.c time.log
```

submits the named source files as your solution to Homework #1;

```
    /c/cs323/bin/check  2
```

lists the files that you submitted for Homework #2;

```
    /c/cs323/bin/unsubmit  3  error.submit bogus.solution
```

deletes the named files that you submitted previously for Homework #3 (which is useful if you accidentally submit the wrong file);

```
    /c/cs323/bin/makeit  4  nmake
```

runs "make" on the files that you submitted previously for Homework #4;

```
    /c/cs323/bin/testit  5  nmake
```

runs the public test script for nmake using the files that you submitted previously for Homework #5;

```
    /c/cs323/bin/protect  6  nmake.c time.log
```

protects the named files that you submitted previously for Homework #6 (so they cannot be deleted accidentally);

```
    /c/cs323/bin/unprotect  7  nmake.c time.log
```

unprotects the named files that you submitted previously for Homework #7 (so they can be deleted); and

```
    /c/cs323/bin/retrieve  8  nmake.c time.log
```

and

```
    /c/csXYZ/bin/retrieve  8  -d"2012/09/11 20:00" nmake.c
```

retrieve copies of the named files that you submitted previously for Homework #8 (in case you accidentally delete your own copies). The day and hour are optional and request the latest submission prior to that time (see the -d flag under "man co" for how to specify times).

D. Prudence (and a 5-point penalty for code that does not make) suggests that you run makeit ("makeit 1 nmake") after submitting your source files (see above). Better yet, run testit ("testit 1 nmake").

E. When available, the public grading script will be /c/cs323/Hwk1/test.nmake and my solution will be /c/cs323/Hwk1/nmake.

To run your program on the file for Test #01, type

```
  % /c/cs323/Hwk1/Tests/t01
```

or

```
  % /c/cs323/Hwk1/test.nmake 01
```

(you may specify more than one test here). To compare the output from your program with the expected output, type

```
% /c/cs323/Hwk1/Tests/t01 | cmp - /c/c3223/Hwk1/Tests/t01.N
```

(cmp outputs the first character where the files differ) or

```
% /c/cs323/Hwk1/Tests/t01 | diff - /c/cs323/Hwk1/Tests/t01.N
```

(diff outputs the lines where they differ but uses a looser definition for "identical").

If your output looks the same as what is expected, but your program still fails the test, there are probably some invisible characters in your output. To make all characters visible (except blanks), type

```
% /c/cs323/Hwk1/Tests/t01 | cat -vet
```

or

```
% /c/cs323/Hwk1/Tests/t01 | od -bc
```

You can use the script /c/cs323/bin/tester to run your own tests.

F. (Optional, No Extra Credit)
   a. Include macro variables such as CC and CFLAGS.
   b. Allow explicit wildcard rules such as

```
%.o: %.c
     cc -c %.c
```

   which specifies that FILE.o depends on FILE.c and can be updated with the command "cc -c FILE.c" for every value of FILE.

CS-323-08/31/16