Queen's University
Faculty of Arts and Science
School of Computing
CISC 220
Final Examination, Fall 2014

Instructor: M. Lamb

- This exam is **THREE HOURS** in length.
- The exam includes the summary sheets for all topics tested in the exam. To save paper, a few details which are not needed for this exam have been omitted from them.
- **No other aids are permitted** (no other notes, books, calculators, computers, etc.)
- The instructor **will not answer questions** during the exam except to clarify what is expected if a question is ambiguous.
- **This exam is printed double-sided.** Make sure you read both sides of each page.
- Please **write your answers in the answer books provided.** Label each answer clearly so it's easy for me to find. If you write more than one answer for a question, cross out the discarded answer(s). If it's not very clear which answer you want me to mark, I'll choose an answer at random and ignore the other ones.
- Please **do not write answers on this exam paper.** I will not see them and you will not receive marks for them.
- The exam consists of **8 questions.** As long as you label your answers clearly, you may write your answers in the answer book in any order you wish. I encourage you to read all of the questions and start with the question that seems easiest to you.
- The exam will be marked out of **45 points.** The number of points for each question is given with the question. Budget your time accordingly.
- Please make sure your **student ID number** is written clearly on the **front** of your answer book and on **all pages** of your answer book which contain final answers. Please **do not write your name** on the answer book.
- Your answers to the programming questions will be marked for correctness only, not style – as long as your code is legible and clear enough for me to understand and mark. However, comments sometimes help me understand what you're thinking so I can give you partial marks.
- You may use any C library functions and Linux/bash commands that you find useful unless noted otherwise in the question. You may also write your own helper functions or commands.
- To save time on the C programming questions, **don't bother with `#include`'s for libraries.**
- Queen's requires the following notice on all final exams:

  PLEASE NOTE: Proctors are unable to respond to queries about the interpretation of exam questions. Do your best to answer exam questions as written.

NOTE TO PROCTORS: The instructor doesn't need these question pages back. Please allow students to undo the staples if they find it useful to separate the pages for easier reference.

Question 1 (10 points): Linux & Bash

A (5 points): Write a bash script that prints the third character of each of its arguments, all in one line with no spaces between. If any of the arguments has fewer than three characters, print nothing for it. If there are no arguments or if none of the arguments have three or more characters, your script may either print nothing or print a blank line.

Example (if your script is called `thirdChars`):

```
-------$ thirdChars apple xx orange y "" banana
pan
```

B. (5 points): Suppose there is a program or script called `compute` which takes a file name as a parameter and prints an integer on the standard output, based on the contents of the file. In some cases, `compute` will encounter an error and fail. You don't have to write a `compute` script; just assume it exists.

Your job is to write a bash script that assumes each of its arguments is the name of a file and calls `compute` with each of the files. Your script must print two lines stating the sum of all of the values printed by successful calls to `compute` as well as the number of files for which `compute` failed. These two lines should be the only output from your script. If a call to the compute script prints a number and then fails, that number should not be included in the sum.n

Example: Suppose that your script is called `totals` and that `compute file1` would print 11, `compute file2` would print 5, and `compute file3` would print 99 and then fail. Then `totals` should behave like this:

```
-------$ totals file1 file2 file3
sum of all compute values: 16
number of failures: 1
-------$ totals file1 file2
sum of all compute values: 16
number of failures: 0
-------$ totals file3
sum of all compute values: 0
number of failures: 1
-------$ totals
sum of all compute values: 0
number of failures: 0
```

Question 2 (5 points): Grep/Find/Sed

A **(3 points)**. Write a sed substitute command that will read a file and look for C expressions in which a variable is added to itself and change it to a multiplication by 2. For example, `x+x` should be changed to `2*x` and `value123 + value123` should be changed to `2*value123`. To be more specific:

- A C variable may contain upper- and lower-case letters, digits and underbars, but must start with a letter. To make your command a little easier for you to write and for me to read, you may forget about underbars if you wish, but you must allow for letters and digits.
- You must recognise any expression that consists of a variable, zero or more spaces, a plus sign, zero or more spaces, and the same variable again -- as long as it all occurs on a single line. I don't expect you to deal with an expression like that which extends over multiple lines.
- Your substitute command must change the expression into a multiplication by two. I don't care whether there are spaces around the "*" or not.
- Your command must be capable of recognising and changing two or more such expressions in the same line -- for example: `y=x+x;  z=y+y;  w = z  +  z;`

I'm just asking for a single substitute command that you could put into a .sed file, not a command you could type into bash on its own. For example, if you put your substitute command into a file called `timesTwo.sed`, we could do the following:

```
-------$ cat example.c
int main() {
  int x = 42+42; // won't be changed because 42 isn't a variable
  int xTimes2 = x + x;
  int y = (xTimes2 +    xTimes2) * (y+y-4);
} // end main
-------$ sed -i -rf timesTwo.sed example.c
-------$ cat example.c
int main() {
  int x = 42+42; // won't be changed because 42 isn't a variable
  int xTimes2 = x*2;
  int y = (xTimes2*2) * (y*2-4);
} // end main
```

B (2 points): For this part of the question, assume that the `timesTwo.sed` file exists and works correctly as described above.

You have a directory which contains many C programs -- in the directory itself and in its sub-directories and sub-sub-directories, and so on -- and you'd like to make the above change in all of these programs. Unfortunately, not every file whose name ends in .c is part of a C program; some files with the .c extension are written in a different language and should not be modified. Fortunately, every single .c file in your directory tree that really is part of a C program contains a "`#include`" directive somewhere inside it.

Write a bash script which uses find and grep to run the sed command from the example above on every C program file in the current directory and its whole directory tree, changing only files whose names end in .c and whose contents do not include "`#include`".

Question 3 (5 points): Basic C

Write a C program which prompts the user to enter one or more integers. After each integer except the first, the program must report the difference between the last two integers entered. When the user enters a zero, the program must stop successfully (i.e. with a zero exit code).

You may ignore the possibility of I/O errors or input in non-integer format.

For example:

```
enter a number: 2
enter a number: 5
difference = 3
enter a number: 11
difference = 6
enter a number: 20
difference = 9
enter a number: 18
difference = -2
enter a number: 25
difference = 7
enter a number: 0
```

Please write a whole C program, except that you may leave out the `#includes` at the beginning.

Question 4 (5 points): Pointers, Arrays & Strings

Recall that the `strcat` function in C takes two string parameters and replaces the first string with a concatenation of the two strings. For example:

```
char string1[20] = "Happy ";
char string2[10] = "Holidays";
strcat(string1,string2);
```

After executing the above code, `string1` now contains `"Happy Holidays"`. This works out because `string1` was declared with length 20, so it had plenty of room for the concatenated string. The danger of `strcat` is that if `string1` had been declared with length 10 (more than enough room for its initial value, but not enough room for the concatenated string) it could have caused a segmentation fault or other kinds of hard-to-debug problems.

For this question, write a C function called `stringMerge` that can serve as an alternative to `strcat`. It should not change either of its parameters, but instead should return a result which is a concatenation of its two parameters. This new result string **must be allocated on the heap.** You may assume that the heap has enough space to store your new concatenated string.

It's OK if your new function calls C string functions such as `strcat`.

For example, the following code:

```
char string1[7] = "Happy ";
char string2[9] = "Holidays";
char *greeting = stringMerge(string1, string2);
printf("%s\n", greeting);
```

will print "Happy Holidays" with no problems, even though that string is longer than the space allocated for either `string1` or `string2`.

Question 5 (5 points): Make

You are working in a directory which initially contains the following files and no others:

```
createH (this file is an executable program)
makefile
numberCrunch.c
numberCrunch.h
numbers.txt
statistics.c
```

Here are the contents of `makefile`:

```
CFLAGS=-Wall -ansi
CC=
OBJS=numberCrunch.o statistics.o

numbers.h: numbers.txt createH
        createH numbers.txt >| numbers.h

numberCrunch.o: numbers.h

statistics.o: numberCrunch.h

statistics: $(OBJS)
        gcc -o statistics $(OBJS)
```

A (2 points): If you type `make statistics` in this directory, what commands will be run? Show the commands exactly as `make` would display them, in the order in which `make` would display them (or one of the correct orders in which `make` could display them).

B (1 point): After the commands are run, what new files will have been created?

C (2 points): After this, you edit `numbers.txt` (and no other files) and then type `make statistics` again. What commands will be run? Show the commands exactly as `make` would display them, in the order in which make `would` display them.

Question 6 (5 points): I/O in C

Write a C program called `lineCount` that takes one command-line argument, which should be the name of a file, and prints a message on the standard output saying how many lines the file contains. Your program may assume that this program will be a text file in the usual format, which includes the fact that every line in the file will end with `'\n'`. However, you may not assume anything else. The file could be empty (no characters at all) or could have a huge number of lines. You may not assume any limit on the length of the lines in the file.

Your program must also be prepared to deal with errors. It must exit with a non-zero status and write an informative error message if any of the following occur:

- the program is called with no command-line arguments, or more than one
- the argument is not the name of an existing, readable file
- a error occurs while attempting to read from the file

Imagine that you're writing this program to be a `bash` utility. Your users don't want to see your program crash; if they call it with bad arguments, or if the disk containing their file crashes while the program is running, they want to see a nice error message.

Question 7 (5 points): Structs

Here is a reminder of the linked list structure used in Assignment 6:

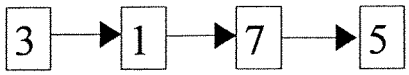
```
// A single node in a linked list
struct ListNode {
  int data; // the data in this node
  struct ListNode *next; // points to next node, NULL in last node
};

// Refer to a linked list with a pointer to the first node
typedef struct ListNode *LinkedList;
```

The `next` pointer in the last list element should be `NULL`.

Write a function called `listTotal` which takes a `LinkedList` as a parameter and returns the sum of all the data elements in the list.

For example, if `numList` looks like this:



`listTotal(numList)` should return 16.

Question 8 (5 points): Processes and Signals

Suppose you compile and run the following program:

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void catch2(int signo);

void catch1(int signo) {
  printf("one\n");
  signal(SIGINT, catch2);
} /* end catch1 */

void catch2(int signo) {
  printf("two\n");
  signal(SIGINT, catch2);
} /* end catch2 */

int main() {
  signal(SIGINT, catch1);
  pause();
  printf("three\n");
  pause();
  printf("four\n");
  pause();
  printf("five\n");
  pause();
  printf("six\n");
  return 0;
} // end main
```

While the program is running, you hit control-C four times.

A. What output do you see before you hit the first control-C?
B. What output do you see between the first and second control-Cs?
C. What output do you see between the second and third control-Cs?
D. What output do you see between the third and fourth control-Cs?
E. What output do you see after the fourth control-C?

For each part above, write "nothing" if there will be no output.

Summary Page: Basic Linux & Bash
CISC 220

Navigating directories:

`cd` *(change directory):* moves you to another directory
- `cd otherDir`: moves you to `otherDir`
- `cd` with no arguments: moves you back to your home directory

`pwd` *(print working directory):* shows the name of your current directory

Listing file information: the `ls` command

arguments should be names of files & directories
- for each file: lists the file
- for each directory: lists the *contents of* the directory (unless –d flag)

`ls` with no arguments: equivalent to `ls .` (list the current directory)

some flags for `ls`:
- `-a`: include files & directories starting with "."
- `-d`: for directories, show directory itself instead of contents
- `-l`: (lower-case L) long format: lots of information about each entry
- `-R`: list sub-directories recursively
- `-1`: (one) list each file on separate line (no columns)

Displaying the contents of a short file:

`cat <filename>`

Wildcards in file names:

`?`: any single character

`*`: any sequence of zero or more characters

Copying files:

`cp oldFile newFile`

`oldFile` must be an existing file. Makes a copy and calls it `newFile`.

`cp file1 file2 file3... fileN dir`

`file1` – `fileN` must be existing files and `dir` must be a directory. Puts copies of files in directory `dir`

Moving/renaming files:

`mv oldFile newFile`

`mv file1 file2 file3... fileN dir`

This command command is similar to `copy`, but it gives files new names and/or locations instead of making new copies. After this command the old file names will be gone.

Deleting files:

`rm <list of files>` *careful – no recycle bin or un-delete!*

`rm -i <list of files>` *interactive mode, asks for confirmation*

`rm -f <list of files>` *suppresses error message if files don't exist*

Creating & deleting directories:

`mkdir dir` *creates new directory called* `dir`

`rmdir dir` *deletes* `dir`, *providing it is empty*

`rm -r dir` *removes* `dir` *and all of the files and sub-directories inside it – use with great caution!*

Create a file or change a timestamp:

`touch filename` *If filename exists, changes its last access time to the present time.*
If not, creates an empty file with that name.

Summary Page: More Shell Skills
CISC 220

Shell Variables

`today=Tuesday`
(sets value of `today` *to* `"Tuesday"`*; creates variable if not previously defined)*
(important: no spaces on either side of equals sign)
`set` *(displays all current variables & their values)*
`echo $today`
(displays value of `today` *variable; output should be* `"Tuesday"`*)*
`export today`
(sets property of `today` *variable so it is exported to sub-shells)*

Echo & Quoting

`echo <args>` *(prints its arguments to the standard output)*
`echo -n <args>` *(doesn't start a new line at the end – useful for prompts in interactive scripts)*

backslash (\) protects literal value of the following character
single quotes protect the literal value of every character
double quotes protect the literal value of every character with a few exceptions:
dollar sign ($), back quote (`), and exclamation point (!)
examples:

```
------$  today=Tuesday
------$  echo Today    is      $today
Today is Tuesday
------$  echo "Today     is      $today"
Today     is      Tuesday
------$  echo 'Today is $today'
Today is $today
------$  echo "${today}s child is fair of face."
```

(Note the braces in the line above – they tell the shell that the s is not part of the variable name.)

```
Tuesdays child is fair of face.
------$  today="$today Jan 13"
------$  echo $today
Tuesday Jan 13
------$  today="${today}, 2009"
------$  echo $today
Tuesday Jan 13, 2009
------$  echo the price is \$2.97
the price is $2.97
```

Useful Predefined Shell Variables:

`$PS1`: your shell prompt. May include special values:
`\d`: the current date
`\h`: the name of the host machine
`\j`: number of jobs you have running
`\s`: the name of the shell
`\w`: current working directory
`\!`: history number of this command
note: these special character sequences only work as part of **`$PS1`**

Useful Predefined Shell Variables (continued)

`$HOME`: your home directory (same as ~)
`$PATH`: list of directories in which to find programs – directory names separated by colons
`$PS2`: secondary prompt for multi-line commands
`$?`: the exit status of the last command (0 means successful completetion, non-zero means failure)
`$SHLVL`: shell level (1 for top-level terminal, larger for sub-shells)

Sub-Shells

`bash` *(starts up a sub-shell, running bash)*
`exit` *(exits from the current shell back to parent – or logs out if this is login shell)*

Shell Scripts

script = file containing list of bash commands
comments start with "#" (to end of line)
`$0` is name of command
`$1`, `$2`, ... are the command-line arguments
`$#` is number of command-line arguments (not counting $0)
`$*` is all command-line arguments in one string, separated by spaces
to execute script in a sub-shell, type filename of script
to execute script in current shell: `source` + name of script

Initialization Files

When you log onto Linux (directly to a shell), it executes `~/.bash_profile`
When you start a sub-shell, Linux executes `~/.bashrc`

Redirection & Pipes

`cmd < inputFile`	*(runs cmd taking input from inputFile instead of keyboard)*
`cmd > outputFile`	*(sends normal output to outputFile instead of screen)*
`cmd >| outputFile`	*(if outputFile already exists, overwrites it)*
`cmd >> outputFile`	*(if outputFile already exists, append to it)*
`cmd 2> errFile`	*(sends error messages to errFile instead of screen)*
`cmd 1>outFile 2>&1`	*(sends both normal and error output to outFile)*
`cmd 2>outFile 1>&2`	*(does same as previous)*
`cmdA | cmdB`	*(a "pipe": output from cmdA is input to cmdB)*

Special file name for output: `/dev/null`. Text sent here is thrown away.

Aliases

`alias newcmd="ls -l"`	*(typing* `newcmd <args>` *is now equivalent to typing* `ls -l <args>`*)*
`unalias newcmd`	*(removes alias for* `newcmd`*)*
`alias rm="rm -i"`	*(automatically get –i option with rm command)*
`'rm' or "rm"`	*(the original rm command, without the alias)*

Summary Page: Shell Scripting
CISC 220

Exit Status:

Every bash command has an exit status: 0 = success, non-zero = failure

exit Command

`exit n` ends a script immediately, returning n as its exit status

Useful commands: Getting Parts of Filenames

`dirname filename` prints the folder part of `filename` minus base name
`basename filename` prints `filename` without its folder
`basename filename suffix` prints `filename` without its folder and suffix

Examples:

```
------$ basename /cas/course/cisc220/pride.txt
pride.txt
------$ basename /cas/course/cisc220/pride.txt .txt
pride
------$ basename /cas/course/cisc220/pride.txt .cpp
pride.txt
------$ dirname /cas/course/cisc220/pride.txt
/cas/course/cisc220
```

Command Substitution

When you write `$(cmd)`, Bash runs `cmd` and substitutes its output. Example:

```
------$ FILENAME=/cas/course/cisc220/somefile.txt
------$ ls $(dirname $FILENAME)
```

... lists contents of `/cas/course/cisc220`

Conditional Statements:

`[[ expression ]]`: evaluates `expression` using string comparisons – not arithmetic!
You must include a space after "[[" and before "]]"!
You must put "$" before variable names.
Statement succeeds (exit status 0) if the condition is true. No output; just used for exit status.
May use string comparisons with "=", "!=", "<" and ">"; there are no <= or >= operators.
File query operators:

`-e file`: `file` exists
`-d file`: `file` exists and is a directory
`-f file`: `file` exists and is a regular file
`-h file`: `file` exists and is a symbolic link
`-r file`: `file` exists and is readable
`-s file`: `file` exists and has size > 0
`-w file`: `file` exists and is writeable
`-x file`: `file` exists and is executable
`file1 -nt file2`: `file1` is newer than `file2`

Arithmetic Conditional Statements:

`((expression))`
The expression can contain assignments and arithmetic with syntax like Java or C.
No need to use "$" before variable names

```
((X > Y+1))
((Y = X + 14))
((X++))
```

The expression succeeds (exit status 0) if the value of the expression is true or a non-zero number
Operators: +, -, *, /, %, ++, --, !, &&, ||

Arithmetic Substitution:

```
-------$ sum=$((5+7))
-------$ echo $sum
12
```

If Command

Example:

```
if [[ $X == $Y ]]
then
      echo "equal"
elif [[ $X > $Y ]]
then
      echo "X is greater"
else
      echo "Y is greater"
fi
```

While Command

Example

```
while ((X<=4))
do
    echo $X
    ((X++))
done
```

For Command

Example:

```
for ARG in $*
do
    ls $ARG
done
```

Useful command with `for` loops: `seq x y` prints all integers from x to y

Example:

```
for X in $(seq 1 10)
do
   echo $X
done
```

`shift` Command

`shift`

"shifts" arguments to the left

Example:

if command-line arguments are `"a"`, `"b"` and `"c"` initially:
after `shift` command they are `"b"` and `"c"` (and `$#` becomes 2)

Shell Variables: Advanced Features

`${var:-alt}`

expands to `alt` if `var` is unset or an empty string; otherwise, expands to value of `var`

`${var:offset:length}`

Expands to a substring of `var`. Indexes start at 0. If length is omitted, goes to end of string. Negative offset starts from end of string. *You must have a space between the colon and the negative sign!*

`${#var}`

expands to the length of `$var`

`${var/pattern/string}`

expands to the value of `$var` with *the first match* of `pattern` replaced by `string`.

`${var//pattern/string}`

expands to the value of `$var` with *all matches* of `pattern` replaced by `string`.

Summary Page: Text Tools (Grep, Find and Sed)
CISC 220, fall 2014

Extended Regular Expressions:

^: matches the beginning of the line (at beginning of pattern only)
$: matches the end of the line (at end of pattern only)
.: matches any single character
[..]: matches any single character from the characters inside the brackets
for example: [aA]: matches an upper or lower case A
[a-z]: matches any lower-case character
[a-zA-Z]: matches any upper- or lower-case character
[^..]: matches any single character that is *not* one of the characters after the ^
for example: [^abc]: matches any character that is not a or b or c
[^a-zA-Z]: matches any character that is not a letter
\<: matches the start of a word (words consist of letters, digits and underscores)
\>: matches the end of a word
\ before any character that normally has a special meaning matches that character literally
for example: \. matches a dot, not any single character
(..): parentheses for grouping sub-expressions

Regular Expressions With Repetition and Alternatives:

R*: zero or more occurrences of R
R?: zero or one occurrence of R (in other words, an optional R)
R+: one or more occurrences of R
R{n}: exactly n occurrences of R
$R_1 | R_2$: either R_1 or R_2
group expressions with (and)
Examples: (cat)|(dog) matches either cat or dog
(cat)+ matches cat, catcat, catcatcat, etc.
Important to Remember: regular expression syntax is not the same as wildcard syntax for files!

Character Classes:

[:alnum:]: any letter or decimal digit
[:alpha:]: any letter (upper or lower case)
[:blank:]: space or tab
[:digit:]: the decimal digits (0-9)
[:lower:]: any lower-case letter
[:punct:]: any punctuation symbol (. , " ' ? ! and so on)
[:upper:]: any upper-case letter
Note: Character classes are always used inside square brackets, as in [[:digit:]] or [[:digit:]ABCDEF]

Backreferences:

\n where n is in [1..9]: matches an occurrence of the same string matched by the nth regular expression surrounded by (and), including nesting.
Example: (\<[a-z]*\>) versus \1 matches
"spy versus spy", "cat versus cat", "computer versus computer", etc. -- but not "spy versus computer"

The grep command

Basic form:

```
grep <string> <filenames>
```

Looks in each file for the string. Prints each line of each file that contains the string.
Options:
-E: use extended regular expression syntax (used in all class examples)
(egrep is the same as grep -E)
-i: search is case-insensitive
continued on next page....

Grep command options, continued:

-h: never show files name with matches
-H: always show file names with matches
(default: show file name when called with more than one file)
-l: (lower case L) shows names of files that contain matches instead of the matching lines
-o: shows only the matching parts of lines (instead of the entire line)
-q: quiet – no output (using grep just for its exit status)
-v: show lines that do *not* match, instead of lines that do match
-w: matches string only when it's a whole word
-num: show *num* lines of context before and after each match

If you don't give grep any file names, it reads the standard input.
Exit status: 0 if at least one match is found, 1 if no matches found, 2 if an error occurred.

The `find` command

Form of command:

```
find dir [test...] [action...]
```

`[dir...]` is the name of one or more directory. `find` searches those directories and all of their subdirectories. `[test...]` is zero or more tests to perform on the files. `[action..]` is zero or more commands to perform on the files that survive the tests.

Numeric arguments to tests: 5 means exactly 5, +5 means more than 5, -5 means less than 5.

Tests:

`-name pattern` *(remember to quote pattern if it contains wildcards!)*
`-iname pattern` *(same as* `-name`*, but not case sensitive)*
`-atime n` *(means file was accessed n days ago)*
`-amin n` *(means file was accessed n minutes ago)*
`-mtime n` *(means file was modified n days ago)*
`-mmin n` *(means file was modified n minutes ago)*
`-size nc` *(means file size is n bytes)*
`-size nk` *(means file size is n kilobytes)*
`-size nM` *(means file size is n megabytes)*
`-size nG` *(means file size is n gigabytes)*
`-type d` *(means file is a directory)*
`-type f` *(means file is a regular file)*

Note: Sizes are rounded UP. Times are rounded DOWN.

Actions:

`-print` *(prints the file's name – happens automatically if no actions specified)*
`-delete` *(deletes the file – use this with caution!)*

Examples of find:

```
find . -size +100k (shows names of all files of size > 100 kilobytes)
find . -name '*junk*' -delete (deletes all files with "junk" in their names)
chmod ugo+rx $(find website -type d)
        (sets read&execute permission for all subdirectories of website)
for FILE in $(find . ); do
        ls $FILE
        cp $FILE ../backup
done
        (copies all files from current directory. and its subfolders to ../backup, printing
         their names first)
```

The sed command:

Command-line options:

`-f <filename>`: get sed commands from <filename> (a "sed script")

`-e script`: script is a string containing a sed command

`-i`: make changes to input file in place (instead of output going to standard output)

Caution: the "-i" flag is very picky about where it occurs in lists of options!

(more sed options on the next page)

More sed command-line options:

`-r`: use extended regular expression syntax (used in all class examples)

If there is no `-e` or `-f` flag, the first non-option argument is interpreted as a command.

Remaining non-option arguments are input file names. If no input file names, reads from the standard input.

Output goes to the standard output stream, except when using the `-i` flag

Examples of Running sed:

```
sed -re 's/Mary/Martha/g' poem >newPoem
sed -rf script somefile
```
(`script` must exist and contain a list of sed commands)

Substitutions:

`s/pattern/replacement/`: replace first occurrence of pattern with replacement

`s/pattern/replacement/N`: replace Nth occurrence of pattern with replacement

`s/pattern/replacement/g`: replace all non-overlapping occurrences of pattern with replacement (a *global* replacement)

Special characters in replacement strings:

`&`: refers to the entire string matched by the pattern

Example: `s/^([a-zA-Z]+)/& &/` makes the first word in a line repeat.
So the line "linux is fun" becomes "linux linux is fun".

`\n` where n is in [1..9]: matches an occurrence of the string matched by the nth regular expression in the pattern surrounded by `(` and `)`, including nesting

Example: `s/^([a-zA-Z]+) ([a-zA-Z]+)/\2 \1/` reverses the order of the first two words in a line, so "linux is fun" becomes "is linux fun".

Summary Page: Basic C
CISC 220, fall 2014

Sample Program:

File 1: `root.c`:

```
float absValue(float x) {
  if (x < 0)
    return -x;
  else
    return x;
} // end absValue

float squareRootGuess(float n, float guess) {
  return (guess + n / guess) / 2;
} // end squareRootGuess

float squareRoot(float n) {
  // Using Newton's method
  float guess = n/2;
  int done = 0; // false (not done yet)
  float newGuess;
  while (!done) {
    newGuess = squareRootGuess(n, guess);
    if (absValue(guess-newGuess) < .001)
      done = 1; // true
    else
      guess = newGuess;
  } // end while
  return newGuess;
} // end squareRoot
```

Commands to compile & link this program:

```
gcc -c root.c
gcc -c sqrt.c
gcc -o sqrt root.o sqrt.o
```

Executable program is now in `sqrt` file.

File 2: `root.h`:

```
float squareRoot(float n);
```

File 3: `sqrt.c`

```
#include <stdlib.h>
#include <stdio.h>
#include "root.h"

int main() {
  printf("enter a number: ");
  float num;
  int result = scanf("%f", &num);
  if (result != 1) {
    printf("Error: input is not a number\n");
    exit(1);
  }
  else {
    float root = squareRoot(num);
    printf("The square root of %f is %f\n", num, root);
    printf("%f squared is %f\n", root, root*root);
  } // end if

  return 0;
} // end main
```

Types:

`char`: a single character
`int`: an integer
`float`: a single-precision floating point number
`double`: a double-precision floating point number

There is no special string type: a string is simply an array of `char`s.
There is no boolean type: use `int`, with 0 meaning false and any non-zero value meaning true.

Pre-Processor:

```
#define SIZE 10
#if SIZE<20
                ....
     #else
        ....
#endif
```

other tests:

```
#ifdef SIZE
#ifndef SIZE
```

to remove a definition:

```
#undef SIZE
```

printf:

```
int printf(char *format, arg1, arg2, ....);
```

Conversion specifications which may be used in format string:

`%d`: integer
`%f`: floating-point number
`%c`: single character
`%s`: string

Minimum field width: a number directly after the "%" -- for example, `%8d`. If the output would not be 8 characters long, pads with spaces on the left (right justified). If the number is negative (`%-8d`), the output is left justified (spaces on the right).

`%8.2f`: minimum of 8 characters total, with *exactly* 2 digits after the decimal point
`%10.8s`: 8 = maximum length; extra characters cut off. Displayed using 10 characters -- so 2 spaces added on the left.

Result of `printf` is the number of items printed.

scanf:

```
int scanf(char *format, address1, address2, ....);
```

Format string contains `%d`, `%f`, etc. as for `printf`.

To read a value into a variable, put "&" before the variable name to get its address:

```
float number;
scanf("%f", &number);
```

Result of `scanf` is the number of items successfully read, or `EOF` if it hit the end of the input file before it could read anything.

Summary Page: Pointers, Arrays & Strings
CISC 220, Fall 2014

To create an array of 10 integers:

```
int nums[10];
```

To declare a pointer to an integer:

```
int *ptr;
```

Operators related to pointers:

`&x` = the address of `x`

`*ptr` dereferences `ptr` (finds the value stored at address `ptr`)

Using the heap:

`malloc(n)`: returns a pointer to n bytes on the heap

`free(ptr)`: releases heap space, where `ptr` is the result of a call to `malloc`

`calloc(num, typeSize)`: like `malloc(num*typeSize)`, but also clears the block of memory before returning (i.e. puts a zero in each byte)

Type sizes:

`sizeof(typ)`: returns the number of bytes used by a value of type `typ`

Creating strings:

```
char abbrev[] = "CISC"; // array containing 5 characters (for CISC plus '\0')
char abbrev[10] = "CISC"; // 10 characters, the first 5 initialized to CISC plus '\0'
```

`printf` conversions for printing strings:

`%s`: print the whole string, no padding

`%20s`: print the whole string with a minimum length of 20, padding on the left if necessary

`%-20s`: print the whole string with a minimum length of 20, padding on the right if necessary

`%.5s`: print the whole string with a maximum length of 5, truncating if necessary

Printing strings with `puts`:

`puts(str)`: writes `str` to the standard output, followed by `'\n'`

Reading strings:

`scanf("%20s", str)`: skips white space, then reads characters until either it reaches 20 characters or it gets to more white space or the end of the file. The 20 characters doesn't count the ending `'\0'` – so `str` must have room for at least 21.

`fgets(str, 21, stdin)`: reads into `str` until it has 20 characters or it reaches the end of the line. `str` will have `'\n'` at the end unless there were 20 or more characters in the line.

More useful string functions:

`strlen(str)`: returns the length of `str` (not counting the ending `'\0'`)

`strcpy(s1,s2)`: copies contents of `s2` to `s1`

`strncpy(s1,s2,n)`: copies at most n characters from `s2` to `s1` (no guarantee of ending `'\0'`)

`strcat(s1,s2)`: concatenates `s2` to end of `s1`

`strcmp(s1,s2)`: returns an integer:

- 0 if `s1` and `s2` are equal
- negative if `s1` < `s2` (i.e. `s1` would come first in a dictionary)
- positive if `s1` > `s2`

Converting from string to integer:

```
long int strtol(char *string, char **tailptr, int base)
```

returns `string` converted to an integer

`base` should be the radix, normally 10

`tailptr` should be the address of a pointer

`strtol` will set `*tailptr` to the address of the first character in `string` that wasn't used

Summary Page: Make
CISC 220, fall 2014

Command-line arguments:

`-f <filename>` means to use `<filename>` as the "make file", containing rules and dependencies. Without `-f`, `make` looks for a file called `makefile` or `Makefile`.

`-n`: don't actually make the target, just print the commands that `make` would execute to make the target

`-k`: keep going in spite of errors (default behavior is to stop if a command fails)

`-B`: assume all files have changed (rebuild the target from scratch)

`<var>=<value>`: sets the value of a make variable, overriding its definition in the make file, if any. For example, `make CFLAGS='-ansi -Wall'`

Specifying a target: An argument that is not a flag starting with "-" or a variable assignment is a target to be made. For example, `make lab5.o` makes the `lab5.o` file.
If you don't specify a target, uses the first target in the `make` file.

Rule Syntax:

```
target: prerequisites ...
        command
        ..
```

or:

```
target: prerequisites ; command
        command
        ...
```

In multi-line syntax, command lines must start with a tab. Beware of editor settings that replace tabs with spaces!

Variables:

To define and set a variable in a make file:

```
<varname> = <value>
```

To use the variable:

```
$(<varname>)
```

or:

```
${<varname>}
```

Implicit Rule For Compiling C Programs:

```
xxx.o: xxx.c
$(CC) -c $(CFLAGS) -o xxx.o xxx.c
```

Sample make file:

```
CC=gcc
CFLAGS=-Wall
sim: input.o random.o alienSim.o
        gcc -o sim input.o random.o alienSim.o
random.o: random.h
input.o: input.h
alienSim.o: input.h random.h
```

Summary Page: File I/O Using the C Library
CISC 220, fall 2014

Opening a File:

`FILE* fopen(char *filename, char *mode)`
`mode` can be: `"r"` (read), `"w"` (write), `"a"` (append)
`fopen` will return `NULL` and set `errno` if file can't be opened

Closing a File:

`int fclose(FILE* file)` *returns 0 if successfully closed*

Predefined File Pointers:

`stdin`: standard input
`stdout`: standard output
`stderr`: standard error

Reporting Errors:

`char *strerror(int errnum)`: Returns a string describing an error.
`void perror(char *msg)`: Prints an error message based on current value of `errno`, with `msg` as a prefix

Character Input:

`int getc(FILE *stream)`: reads a character and returns it (or EOF if at end of file)
`int getchar()`: equivalent to `getc(stdin)`
`int ungetc(int c, FILE *stream)`: "pushes" `c` back onto input stream

Character Output:

`int putc(int c, FILE *stream)`: writes `c` to the file, returns c if successful
`int putchar(c)`: equivalent to `putc(c, stdout)`

String Output:

`int fputs(char *s, FILE *stream)`: writes `s` to the file, returns EOF if error
`puts(char *s)`: writes `s` *plus* *'\n'* to `stdout, returns EOF if error`

String Input:

`char* fgets (char *s, int count, FILE *stream)`
Reads characters from stream until end of line OR count-1 characters are read.
Will include an end of line character (`'\n'`) if it reaches the end of the line
On return, `s` will always have a null character (`'\0'`) at the end.
Returns `NULL` if we're already at the end of file or if an error occurs.

Formatted I/O:

`int fscanf(FILE *stream, char *format,` *more args...*`)`: Works like `scanf`, but reads from the specified file.
`int fprintf(FILE *stream, char *format,` *more args...*`)`: Works like `printf`, but writes to the specified file

Checking for end of file

`int feof(FILE *stream)` returns non-zero if we're at the end of `stream`
`int eof()` returns non-zero if we're at the end of the standard input

Summary Page: Typedefs, Structs and Unions
CISC 220, fall 2014

Typedefs:

With these typedefs:

```
typedef int idType
typedef char *String
```

The following definitions:

```
idType id;
String name;
```

mean the same thing as:

```
int id;
char *name;
```

Examples of Structs:

```
struct personInfo {
     char name[100];
     int age;
};

// this line:
struct personInfo mickey = {"Mickey", 12};

// does the same as these three lines:
struct personInfo mickey;
strcpy(mickey.name, "Mouse");
mickey.age = 12;
```

Combining Structs and Typedefs:

```
typedef struct {
     char name[100];
     int age;
} Person;

Person mickey;
mickey.age = 15;
```

Unions:

```
union identification {
     int idnum;
     char name[100];
};
union identification id;
// id may contain an integer (id.idnum) or a name (id.name)
// but not both.
```

Summary Page: Signals & Processes in C

CISC 220, fall 2014

(heavily abridged version for final exam)

Types of Signals:

name	default action	notes
SIGALRM	terminates process	Used by Linux "alarm clock" timer
SIGCHLD	ignored	Sent by system when a child process terminates or stops
SIGCONT	re-starts stopped process	Sent by system when stopped process re-starts
SIGINT	terminates process	Sent by system when user hits control-C
SIGKILL	terminates process	Programs can't "catch" SIGKILL.
SIGSTOP	stops process	a stopped process can be re-started later Programs can't "catch" SIGSTOP.
SIGTERM	terminates process	
SIGTSTP	stops process	sent by system when user hits control-Z.
SIGUSR1	terminates process	Not used by system; user programs may use for any purpose
SIGUSR2	terminates process	Not used by system; user programs may use for any purpose

All signals except `SIGKILL` and `SIGSTOP` can be caught or ignored by user programs.

Setting Up a Signal Catcher in a C Program:

`signal` function establishes a catcher for a particular signal type

```
signal(int signum, void (*catcher) (int));
```

Waiting For Signal:

```
pause(); /* suspends until you receive a signal */
```

Sending a Signal To Another Process:

```
kill(int pid, int signal);
```

END OF EXAM

Have a great holiday! See you in January.