Queen's University
Faculty of Arts and Science
School of Computing
CISC 220
Final Examination, Fall 2015

Instructor: M. Lamb

- This exam is **TWO 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**. It is too disruptive. If you think there's an error or a missing detail in a question, make your best guess about what the question means and state your assumptions.
- **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 7 **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 **35 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 **very legibly** 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.

Question 1 (5 points): Linux Scripting

Suppose that on your Linux system there is a command called `legal`, which takes the name of a file as an argument and checks whether the file is "legal" according to some set of conditions. The command never prints any output, either to the standard output or the standard error stream. It finishes with exit status zero if the file is "legal"; otherwise it finishes with a non-zero status.

There is another command called `rating`, which takes the name of a file as an argument and prints a rating (a number between 1 and 4) on the standard output. It never prints anything besides this number, either on the standard output or the standard error.

Write a Linux script that takes the name of a directory as its only argument. It must print the number of files in the directory that are "legal" and get a rating of 4 from the `rating` command. Your script should print nothing but a number. It should be looking at files in the directory named by the argument, but not in sub-directories.

For example, suppose there are 19 files in the current directory. Ten of them are legal and nine are not. Of the ten legal files, six get a rating of 4. Your script should print 6 and nothing else. It's possible that some of the illegal files also get a rating of 4, but that should not increase the count.

If there are no legal files in the directory that get a rating of 4, your script should print 0.

Your script doesn't have to do any error checking. You may assume that it will be called with exactly one argument, and that this argument will be the name of an existing, readable directory.

Question 2 (5 points): Bash Output and Links

Read the following sequence of shell commands. Assume that the current directory has no files in it before these commands are executed. Write the output of this sequence of commands in your answer booklet. If a line would produce an error message, just write "ERROR"; you don't have to know the exact error message. If a line would produce an error message, continue on to the next line regardless.

```
echo xmas >| one.txt
ln one.txt two.txt
ln -s one.txt three.txt
cp one.txt four.txt
mv one.txt five.txt
echo kwanzaa >> one.txt
echo hanukkah >> two.txt
echo newyear >> three.txt
echo boxingday >> four.txt
echo yule >> five.txt
cat one.txt
echo "---"
cat two.txt
echo "---"
cat three.txt
echo "---"
cat four.txt
echo "---"
cat five.txt
echo "---"
```

Question 3 (5 points): Linux Text Tools

You have a directory called `ChristmasFiles`. That directory contains many sub directories and sub-sub directories and so on. There are many files in these directories. You believe that at some time you created a file in that directory tree that had information about Rudolph the Red Nosed Reindeer, but you don't remember where it is. All you're sure about is that its extension was `.txt`.

Write a command or series of commands that will help you locate this information. It must look through all of the files in that directory tree with extension .txt. For each .txt file, it must print each line inside that file that contains the string "Rudolph". "Rudolph" can appear anywhere in a line and doesn't have to be a separate word; a line containing a word like "Rudolphs" or "SirRudolph" should show up in the output. Print the whole line for each match, not just the word "Rudolph".

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

Write a C function called `firstChars` that takes just one parameter, an array of strings. The last element in the array of strings will be NULL. The `firstChars` function must return a string containing the first letter of each of the strings in the parameter array. The result of the `firstChars` function must be allocated on the heap. You may not make any assumptions about how many strings will be in the parameter array.

For example, here is a simple main program using the completed `firstChars` function:

```
int main() {
  char *holidays[] = {"Xmas","Kwanzaa","Hanukkah","NewYears",NULL};
  char *result = firstChars(holidays);
  printf("%s\n", result);
  free(result);
  return 0;
} // end main
```

The output of this program should be

```
XKHN
```

Question 5 (5 points): Make

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

```
format.sed frenchWords.txt makefile translate.c
```

Here are the contents of `makefile`:

```
CC=gcc
CFLAGS=-Wall

translate: translate.o
	gcc -o translate translate.o

englishWords.txt: frenchWords.txt translate
	translate frenchWords.txt >| englishWords.txt

dictionary.txt: englishWords.txt format.sed
	sed -rf format.sed englishWords.txt >| dictionary.txt
```

A (3 points). If you type `make dictionary.txt` in this directory, what commands will be run and in what order? Write the exact commands that make would display and run; don't write dependency lines from the makefile (i.e. don't write lines such as "`translate: translate.o`") and don't write an essay about what make will do. Just write the commands that make would execute. If there's more than one legal order for the commands, any legal order is fine.

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

C (1 point). Now you edit `format.sed` and type `make dictionary.txt` again. What commands will be run and in what order? (Once again, any legal order is fine.)

Question 6 (5 points): C I/O and structs

Here is the definition of a `struct` type. You may assume it's included in your program; you don't have to write it in your answer book.

```
typedef struct {
  char firstChar;
  char lastChar;
} initials;
```

Write a function that will read the first five lines of a file and record the first and last characters of these lines. By the last character in the line I mean the last character before the end of line character (`'\n'`). It's OK to assume that every line except the last one in the file ends with a `'\n'`, but the last line in the file might not.

The function should take two parameters:

1. The name of the input file (a string)
2. An array of five structs of type `initials`.

Put the first and last characters of the first line into `initials[0]`, the first and last characters of the second line into `initials[1]`, and so on.

Your program must abort the program with an informative error message if any of the following things happen:

- The file does not exist or you don't have read permission for it
- The file contains less than 5 lines
- A line in the file has less than 2 characters (not counting the end-of-line character)

You don't have to worry about any other sorts of I/O errors.

You may assume that no line in the file is more than 50 characters long and that all of the necessary C libraries for I/O have been included in your program.

Here is a header for your function:

```
void firstAndLast(char *filename, initials structArray[]);
```

Question 7 (5 points): Processes & Signals

You need to solve a difficult problem. You have thought of 10 different algorithms, but you're worried that some of them might take a long time and some of them might even get into an infinite loop. You're sure that any of the algorithms that don't get into an infinite loop will eventually return the right answer.

You have put all of your algorithms into one function, called `solution`, which takes an integer as a parameter. It expects the parameter n to be between 1 and 10 and a call to `solution(n)` will attempt your nth algorithm and return the result as long as it doesn't get into an infinite loop. You have written a multi-process C program which will run all 10 of your algorithms in parallel. Each one will print its result when it is finished. Your program is shown on the next page.

Your task for this question is to improve the program so that as soon as one of the algorithms returns with an answer the processes running the other 9 algorithms will be killed off.

The program is printed with line numbers. You can re-write the whole program if you really want to, but to save time you can just tell me very clearly what changes you want to make, with reference to the line numbers. If you want to delete line N, write "delete line N". If you want to change line N, write "change line N to....". If you want to add something after line N, write "add the following after line N".

Question 7, continued

```
1.    #include <unistd.h>
2.    #include <signal.h>
3.    #include <sys/wait.h>
4.    #include <sys/types.h>
5.    #include <stdio.h>
6.    #include <stdlib.h>
7.
8.    // The solution function is defined in another file that will
9.    // be linked with this one.
10.   int solution(int option);
11.
12.   #define NUM_OPTIONS 10
13.
14.   int main() {
15.     int i;
16.
17.     // The process IDs of the child processes
18.     // (element zero not used)
19.     pid_t childProcesses[NUM_OPTIONS+1];
20.
21.     // Create forks to run solution with argument 1-10
22.     for (i = 1; i <= NUM_OPTIONS; i++) {
23.       pid_t childProcesses[i] = fork();
24.       if (childProcesses[i] == 0) {
25.         int answer = solution(i);
26.         printf("solution(%d) returned %d\n", i, answer);
27.         exit(0);
28.       }
29.       else if (childProcesses[i] < 0) {
30.         printf("error: could not fork off child process\n");
31.         exit(1); // exiting the parent process
32.       }
33.       else {
34.         // This is the parent process after a child has been
35.         // forked off
36.         printf("started solution(%d)\n", i);
37.       }
38.     } // end if
39.
40.     // Only the parent process gets to this point.
41.     // The parent exits and lets its children continue running.
42.
43.     return 0;
44.   } // end main
```

This is the end of the last question of this exam. The remaining pages are summary sheets.

Summary Page: Basic Linux & Bash

CISC 220

(last update: 09/17/15)

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)

Wildcards in file names:

`?`: any single character

`*`: any sequence of zero or more characters

File/directory protections:

`chmod <who>=<what> <list of files and folders>`

`<who>` is `u` for owner, `g` for group, `o` for other users

`<what>` is `r` for read, `w` for write, `x` for execute

can use + or – instead of =, to add or subtract permissions

Example:

`chmod u+w *`: gives owner write permission to all files in current directory

`umask <who>=<what>`: sets the default protections for new files you create

`umask -S`: displays your current set of default protections

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*

Summary Page: More Shell Skills

CISC 220

(last update 09/26/15)

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`**

Summary Page: More Shell Skills (continued)

More Useful Predefined Shell Variables:

`$HOME`: your home directory (same as ~)
`$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
(last update 7. December 2015)

Links

`ln file1 file2`

file1 should be an existing file. This command creates a "hard link" to file1, called file2.

You can't create hard links to a file on a different physical device or to a directory.

`ln -s file1 file2`

file2 becomes a symbolic link to file1 – a special file containing the name "file1". No restrictions.

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: +, -, *, /, %, ++, --, !, &&, ||

(Summary Page: Shell Scripting, continued)

Arithmetic Substitution:

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

If Command

```
if test-commands; then
      consequent-commands;
[elif more-test-commands; then
      more-consequents;]
[else
      alternate-consequents;]
fi
```

Example:

```
if [[ $X == $Y ]]
then
      echo "equal"
else
      echo "not equal"
fi
```

While Command

```
while test-commands; do consequent-commands; done
```

Example

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

For Command

```
for name [in words ...]; do commands; done
```

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
```

Summary Page: Text Tools (Grep, Find and Sed)

CISC 220, fall 2015

(last updated 7. Dec. 2015)

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.

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
-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.

Summary Page: Text Tools (Continued)

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)*
(other tests omitted to save space)

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)

Summary Page: Basic C
CISC 220, fall 2015

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
```

Summary Page: Basic C (Continued)

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 `chars`.
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 2015

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 2015

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 2015

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: Structs, Unions and Typedefs
CISC 220, fall 2015

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 2015

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
SIGINT	terminates process	Sent by system when user hits control-C
SIGKILL	terminates process	Programs can't "catch" SIGKILL.
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 of the above signals except `SIGKILL` can be caught by user programs.

Sending Signals Using Bash:

`kill -signal pid`

`signal` should be a signal name or number. The initial "`SIG`" may be left off signal names.

`pid`: a process id or job id

Examples:

`kill -SIGSTOP 4432` sends a `SIGSTOP` signal to process 4432.

`kill -INT %2` sends a `SIGINT` signal to job number 2

If no signal is specified, sends a `SIGTERM`.

`kill -l` (lowercase L) prints a list of all the signal names and numbers, if you're interested.

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));
```

Predefined catchers:

`SIG_IGN`: ignore the signal

`SIG_DFL`: use the default action for the signal

Waiting For Signal:

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

Sending a Signal To Yourself:

```
int raise(int signal);
```

Summary Page: Signals & Processes in C (Continued)

Using the Alarm Clock:

```
int alarm(int secs);
/* generates a SIGALRM in that many seconds. */
/* alarm(0) turns off the alarm clock */
```

Sending a Signal To Another Process:

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

Observing Processes From bash:

`jobs -l`: show your jobs with pid numbers
`ps`: info about your own processes
`ps a`: info about every process on computer coming from a terminal
`ps -e`: info about **every** process on computer (terminal or not)
add "-f" to any "ps" command: full listing format
`top`: interactive display of processes on computer

fork: Splits the current process into two concurrent processes.

```
pid_t pid = fork(); /* pid_t is an integer type */
```

For child process, result of fork is zero. For parent process, result of fork is the process id of the child. A negative result means error – couldn't create a new process.

(Information about exec functions omittted to save space)

wait: Waits until *any* child of the current process exits. Return value is the process id of the child that exited. Parameter is a pointer to an integer, which will get the exit status of the child.
Example:

```
int status;
pid_t child_pid = wait(&status);
if (status == 0)
   printf("child process %d was successful!\n", child_pid);
 else
   printf("child process %d exited with status = %d\n",
     child_pid, status);
```

If you don't care about the child's exit status, call `wait(NULL)`.

waitpid: Waits for a child to exit, or checks on its status without waiting. Parameters are:

- the process id of the child to wait for (or -1, meaning any child)
- pointer to an integer, which will get the exit status of the child
- an int containing options. Useful values are:
    8. 0: no special options; wait for child to exit
    9. WNOHANG: Check without waiting. If the child process is still running, the function will return a value of zero immediately.
- return value is the process id of the child if it has exited, or zero if the child is still running and we're using the WNOHANG options.

`waitpid(-1,&status,0)` is equivalent to `wait(&status)`.