Queens's University
Faculty of Arts and Science
School of Computing
CISC220
Final Exam, Fall 2018
Instructor: Samir Mohammad

- This exam is **THREE HOURS** in length.
- All programs in this exam have developed and tested.
- The exam includes the summary sheets for all topics in the exam.
- **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 to find. If you write more than one answer for the question, cross out the discarded answer(s). If it's not clear which answer you want me to mark, I'll choose random one and ignore the others.
- Please **do not write answers on this exam paper.** I will not see them and you will not receive marks for them. I will only mark what is written in the answer books.
- If you need more space for answers, the proctors will supply additional answer books. But please don't ask until you're really out of space; let's not kill any more trees than we have to!
- The exam consists of 8 questions. As long as you label your answer clearly, you may write your answers within the answer books in any order you wish. I encourage you to read all the questions and start with the question that seems easiest to you.
- The exam will be marked out of **80 points.** Each question is worth 10 points. Budget your time accordingly. Please make sure your **student ID number** is written very legible 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 anywhere on the answer book.
- Your answers to programming questions will be marked for correctness only, not style - as long as your code is legible and clear enough for me to understand in mark.
- 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 scripts.
- To save time on the C programming questions, **don't bother with #include's.** I will assume you have included all of the necessary ".h" files. Information about include files has been deleted from some of the summary sheets to save space.
- Queen's requires the following notice on all final exams:

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

This set of instructions plus summary sheets are written by Dr. M. Lamb

Question 1 (10 points): (bash)

When you develop a C program you have to compile the source code file, then you have to link it and give a name for the execution file. For security reasons, usually, the "umask" do not include an execution privilege for newly created files; so you have to change the permission if you want to run the program. Finally, your path may not include the current directory, from which you need to run (execute) your program; so you need to take care of this issue too in order to be able to run your program.

Every time you make a change to your program you have to go through all of these steps in order to test the result of your changes. "makefile" would be the right choice to use in such situations; only if you are developing a big project with multiple files. However, if you are developing only one single program, then writing a script would be your best choice.

Assume you are developing a program named "myProg.c" that contains only one file, and you named the execution file of this program "myProg"
Write a bash script that can carry out all of the above mentioned steps. If we assume that your bash script is "myBash" then the following bash statement will implement all steps for you:

```
./myBash.sh myProg
```

Question 2 (10 points): (shell scripting)

Write a shell script to manage a Telephone Directory. Your script should be able to implement only two tasks:

1) Search for someone's phone number. The search can be based in full or part of a name. Also, search can be based on full of part of a phone number.
   - If a match for the search exist(s), display it.
   - If no match exist(s), display a message, "search has no match", replace your "search" word with the actual search string. For example, if you search for "Sam" your message should be "Sam have no match" if there isn't any match for the search.
2) Enter a new entry:
   - If the user enter a name and phone number in one line, the script should add it to "PhoneDir.txt" file, which contains the directory of phone numbers.
   - If the user do not supply any information, then the script should message "You didn't enter any info"

Assume that your "PhoneDir.txt" contains the following:

```
-----$ cat PhoneDir.txt

Kim 613-111-1111
Joe 613-555-2222
Kim 613-777-7890
```

Following are all possible scenarios that the script should be able to handle. Write your script accordingly:

```
-----$ ./PhoneDir.sh
        —TELEPHONE DIRECOTRY
   please chose from the following menu:
 1- Enter 1 to search for a phone number
 2- Enter 2 to add a person phone number
 0- to EXIT enter 0 (zero)
-----$ 1
Enter the name to search for
-----$ Kim
Kim 613-111-1111
Kim 613-777-7890
Hit Enter to continue
-----$ (the user hits Enter key)
         TELEPHONE DIRECOTRY
   please chose from the following menu:
 1- Enter 1 to search for a phone number
 2- Enter 2 to add a person phone number
 0- to EXIT enter 0 (zero)
-----$ 1
Enter the name to search for
-----$ Sam
 Sam has no match
Hit Enter to continue
-----$ (the user hits Enter key)
         TELEPHONE DIRECOTRY
   please chose from the following menu:
 1- Enter 1 to search for a phone number
 2- Enter 2 to add a person phone number
 0- to EXIT enter 0 (zero)
-----$ 2
Enter the new name and No. e.g. Sam Jim 613-444-1234
```

```
-----$ (the user hits Enter key, with no entry at all)
You didn't enter any info
 Hit Enter to continue
-----$ (the user hits Enter key)
         TELEPHONE DIRECOTRY
   please chose from the following menu:
 1- Enter 1 to search for a phone number
 2- Enter 2 to add a person phone number
 0- to EXIT enter 0 (zero)
-----$ 2
Enter the new name and No. e.g. Sam Jim 613-444-1234
(the user enters the following)
-----$ Mike Sam 613-531-7777
(then the user hits enter)
The entry has been added to the phone directory
 Hit Enter to continue
-----$ (the user hits Enter key)
         TELEPHONE DIRECOTRY
   please chose from the following menu:
 1- Enter 1 to search for a phone number
 2- Enter 2 to add a person phone number
 0- to EXIT enter 0 (zero)
-----$4
you must choose only 1, 2, or 0 (zero)
 Hit Enter to continue
-----$ (the user hits Enter key)
   please chose from the following menu:
 1- Enter 1 to search for a phone number
 2- Enter 2 to add a person phone number
 0- to EXIT enter 0 (zero)
-----$0
closing Telephone Directory
-----$
```

To see the updates to the PhoneDir.txt after the addition of the above new phone number.

```
-----$ cat PhoneDir.txt
Kim 613-111-1111
Joe 613-555-2222
Kim 613-777-7890
Mike Sam 613-531-7777
-----$
```

Actual prompt during the script execution is BLANK line, but instead, I used "-----$" prompt in this scenarios just to distinguish user input (at the prompt -----$) from the script echoes and actions.

You are not asked to check if the format of a given name and phone number is correct, in option 2; However, you are required to verify at least that a string is entered regardless of the format and the contents.

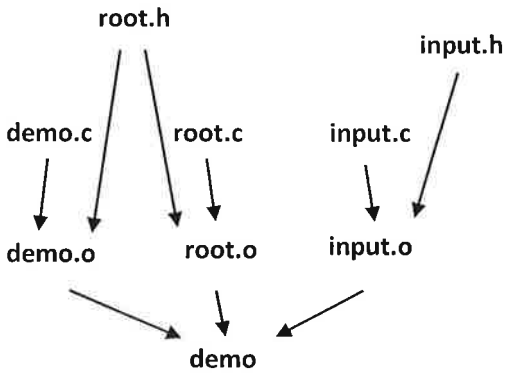
Question 3 (10 points): (Basic C and Pointers)

Write the output of the following C program. Your output should match the format and the details given in the program.

```
#include <stdio.h>
#include <stdlib.h>
void pointers(int a, int b, int *c) {
  int *ptr1 = &a;
  int *ptr2 = &b;
  printf("%d, %d, %d, %d, %d\n", a, b, *c, *ptr1, *ptr2);
  a = 3;
  b = 4;
  printf("%d, %d, %d, %d, %d\n", a, b, *c, *ptr1, *ptr2);
  *ptr1 = 5;
  *ptr2 = 6;
  printf("%d, %d, %d, %d, %d\n", a, b, *c, *ptr1, *ptr2);
  ptr2 = ptr1;
  ptr1 = c;
  *ptr1 = 7;
  *ptr2 = 8;
  *c = 9;
  printf("%d, %d, %d, %d, %d\n", a, b, *c, *ptr1, *ptr2);
} // end pointers
int main() {
  int x = 1;
  int y = 2;
  pointers(x, y, &x);
  printf("%d, %d\n", x, y);
  return 0;
} // end main
```

Question 4 (10 points): (makefile)

"demo" is a program that computes square roots. Many files are required to run the program. The required files and their dependency are depicted in the graph below. These files are needed to build the "domo" file.



Based on the dependency diagram, write the related "makefile" using the variables given below. Assume we have the following variables defined as follows:

```
OBJS = demo.o  root.o  input.o
C_OPTIONS = -Wall
```

Use these variable definitions in writing the "makefile" based on the dependency graph above.

Question 5 (10 points): (Arrays)

"isdigit" is a C function that verifies if a given character is of digit type or not. Similarly, write a function that verifies if a given string is a number or not. Note that your program should accept numbers with decimal points. Numbers that are separated by point delimiter are accepted too such as an IP address (e.g. 152.256.100.0)
Following is different scenarios of the function run. Assume the name of the program is "IsItNumber.c" Use the given details in the scenarios to write your function. For example, you have to get a number to be tested from the user.

```
-----$ ./IsItNumber
Enter number or string to test
```
(The user enters)
```
12
```
(then the user hit Enter key)
```
 It is a number
```

```
-----$ ./IsItNumber
Enter number or string to test
```
(The user enters)
```
12.3
```
(then the user hit Enter key)
```
 It is a number
```

```
-----$ ./IsItNumber
Enter number or string to test
```
(The user enters)
```
12a
```
(then the user hit Enter key)
```
It is not a number
```

Question 6 (10 points): (Structure and Pointers)

A C program called "Cars.c" produces the following output when it is compiled and executed:

```
C-----$ ./Cars
Car Make  : Toyota
Car Model : Corolla
Car Year  : 2015
Car Make  : BMW
Car Model : Sedan
Car Year  : 2017
```

If this is the output of "Cars.c" program, write the program that can produce such an output. Your program must satisfy the following requirements:

1- You have to use "Structure" to define cars
2- The program must have a printing function that is called to print by reference-using pointer. Printing function that called and prints by value is not accepted.
3- Collecting the data about the two cars can be done either twice, once for every car, or by using "for" or "while" that loops twice.
4- The program enters the given output data internally. You do not have to ask users for this information.

Question 7 (10 points): (Processes, Signals, & Pipes)

Write a program showing the use of Process, Forks, Piping, and Communication between processes. The program should do the following:

1- Create a pipe.
2- Fork off the first child process which writes two numbers into the pipes (the number that it writes is 1 and 2, one and two).
3- Fork off a second child process that reads these two numbers from the pipe add them up and display the total:

```
The total of the two number is 3
```

4- Error handling should be included in the program.
5- The Parent process makes sure the child processes are terminated before it stopes.

Question 8 (10 points): (Multiple Choices)
Write your answers in the Answer Book.

Q.1 What is the output of the following C program?

```
# include <stdio.h>
main ( )
{
   int a, b=0;
   static int c [10]={1,2,3,4,5,6,7,8,9,0};
   for (a=0; a<10;++a)
   if ((c[a]%2)== 0) b+=c[a];
   printf ("%d", b);
}
```

(A) 45
(B) 25
(C) 20
(D) 90

Q.2 If `a, b` and `c` are integer variables with the values `a=8, b=3` and `c=-5`. Then what is the value of the arithmetic expression:

```
2 * b + 3 * (a-c)
```

(A) 117
(B) 6
(C) -16
(D) 45

Q.3 What is the output of this progam?

```
#include <stdio.h>
int main ()
{
  float x;
  x =  7/5;
  printf("%f \n",x);
}
```

(A) 1.400000
(B) 1.000000
(C) 1.4
(D) 1

Q.4 What is the output of the following?

```
#include<stdio.h>
void increment(void);
int main()
{ increment();
  increment();
  increment();
  return 0;
}
void increment(void)
 {
   int i = 0 ;
   printf ( "%d", i ) ; i++;
 }
```

(A) 00
(B) 000
(C) 3
(D) 2

Q.5 What is the output of the following program?

```
#include<stdio.h>
int x = 10 ;
int main( )
{
  extern int y;
  printf("The value of x is %d \n",x);
  printf("The value of y is %d",y); return 0;
}
int y=50;
```

(A) The value of x is 10
    The value of y is 50
(B) The value of x is 10
    The value of y is 0
(C) Error message
(D) None of the above

Summary Page: Basic Linux & Bash
CISC 220, Fall 2018

Make sure you're running bash:

Right after you log in, type this command: `ps` (This command lists all the processes you're currently running. Don't worry about the details for now.)

If you're running bash, the output will include a process called "bash". If not, send an e-mail to Samir (sm373@queensu.ca) to get your login shell changed to bash. For now, you can type `bash` to switch to that shell.

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>`

Reading a longer file one screen at a time:

`less <filename>`

While running the less program, use these single-character commands to navigate through the file:

- f or space: forward one window
- b: backward one window
- e or return: forward one line
- y: backward one line
- h: display help screen which describes more commands
- q: exit

Wildcards in file names:

`?` : any single character

`*` : any sequence of zero or more characters

Information about commands (Linux "manual"):

`man ls` : information about the `ls` command

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

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

`umask -S`: displays your current set of default protections in symbolic (not binary) form

Examples of File/directory protection commands:

`chmod g+rx myprogram`: gives group members read permission and execute for `myprogram`
`chmod u+w *`: gives owner write permission to all files in current directory
`umask -S` shows permissions of current user in symbolic form
Example using `umask`:

```
-------$ umask -S
u=rwx,g=,o=
-------$ umask g+rx
-------$ umask -S
u=rwx,g=rx,o=
-------$ umask g-x
-------$ umask -S
u=rwx,g=r,o=
-------$ umask g+w
-------$ umask -S
u=rwx,g=rw,o=
-------$ umask g=
-------$ umask -S
u=rwx,g=,o=
-------$
```

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 extra 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 & modification times to the present time. If not, creates an empty file with that name.*

Log out of the Linux system:

`Logout`

End current shell:

`Exit`

See a list of your current jobs:

`Jobs`

Run a command in the background:

`<command> &`

Change to a background job:

`%n`, where `n` is number of job as shown by `jobs`
`%pre`, where `pre` is a prefix of the job name

To terminate a job:

`kill <job number>` or `kill <prefix of job name>`

To stop a foreground job:

control-c *(hold down the control key and type c)*

Text editors:

`emacs`: Start emacs with the `emacs` command. To go to a tutorial, type control-h followed by "t". To exit emacs, type control-x following by control-c. To suspend emacs (put it in the background), type control-x following by control-z.

`vim`: The shell command `vimtutor` will start a tutorial to teach you how to use vim. To exit `vim`, type `:q!`. If that doesn't work, you're probably in "insert mode", so type Escape to go back to "edit mode" and try again. To start vim without the tutor, just use the `vim` command.

`nano`: a scaled-down version of emacs, with a menu showing to help you get started

There are other editors on CASLab Linux as well, but most use graphics (not just plain characters) so they don't work in a shell window, which means you can't use them over putty or with Vagran

Summary Page: More Shell Skills
CISC 220, Fall 2018

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 file name of script
to execute script in current shell: source + name of script

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`

`$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)

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, appends 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, Fall 2018

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 full path name of "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 (if suffix matches)

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
------$ DIR=$(dirname $FILENAME)
------$ DIR is now /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

```
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

```
X=1
while ((X<=4))
do
      echo $X
      ((X=X+1))
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
      ((SUM = SUM + 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)

User Interaction During a Shell Script

`read myvar` Waits for user to type a line of input and assigns the input line to variable `myvar`.

`read var1 var2` Same as above, but assigns the first word of the input to `var1` and the rest of the line to `var2`. May be extended to as many variables as you want.

`read -p prompt <variables>` Types the prompt before waiting for input

`echo -n ...` *Same as normal echo, but doesn't add a newline character. Useful for prompts.*

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: Basic C
CISC 220, Fall 2018

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
```

ommands to compile & link this program:

```
gcc  c root.c float
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 `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 imput file before it could read anything

Summary Page: Make
CISC 220, fall 2018

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
```

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: Pointers, Arrays & Strings
CISC 220, Fall 2018

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 `type`

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 line or end of file. The 20 characters don't include 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 before s2 in a dictionary)
- positive if `s1` > `s2`

Converting from string to integer:

`atoi(str)`: returns the integer represented by str OR zero if not in integer format

```
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 use

Summary Page: Structs, Unions and Typedefs
CISC 220, Fall 2018

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: File I/O Using the C Library
CISC 220, fall 2018

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: Signals & Processes in C
CISC 220, fall 2018

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	sent by system when user hits control-Z
SIGTSTP	stops process	
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);
```

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.

"exec" functions. general:

Each of these functions runs another program or script. This program or script takes over the current process and doesn't return. If an exec function returns, it means there was an error in calling the other program. You can't use wildcards, I/O redirection, or other shell features; the arguments are passed directly to the program.

execl:

Takes a variable number of parameters:
- path name of the program to run
- "base name" of the program (without the directory name)
- first argument to the program
- second argument to the program
- ....
- last argument to the program
- `NULL` (to mark the end of the list of parameters)

Example (runs "`ls -l -F`")

```
execl("/bin/ls", "ls", "-l", "-F", NULL);
fprintf(stderr, "error: execl returned\n");
exit(1);
```

execv:

Takes two parameters:
- path name of the program to run
- an array of strings, consisting of:
  - the "base name" of the program
  - arguments for the program
  - a NULL at the end

The second parameter will be passed to the program as the `argv` array.

Example (runs "`ls -l -F`")

```
char *args[4];
args[0] = "/bin/ls";
args[1] = "-l";
args[2] = "-F";
args[3] = NULL;
execv("/bin/ls", args);
fprintf(stderr, "error: execv returned\n");
exit(1);
```

execlp and execvp

Just like `execl` and `execv`, but the first parameter doesn't need to be a full name; the function will search your `PATH`.

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:
  - 0: no special options; wait for child to exit
  - `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)`

Libraries:

This topic involves some new libraries:

`<sys/types.h>`:
`pid_t` (the integer type for process ids)

`<unistd.h>`:
`fork`
the "exec" functions
`sleep`
`pause`
`getpid`
`getppid`

`<sys/wait.h>`:
`wait`
`waitpid`
`WNOHANG`

`<signal.h>`:
names for the signals
`signal`
`kill`
`SIG_IGN`
`SIG_DFL`

Summary Page: Low-Level I/O (I/O System Calls)
CISC 220, fall 2018

Note: the text advises using `creat` *instead of* `open` *to create a new file, a practice which is a bit outdated. Use* `open` *instead.*

Representing Files:

In low-level I/O, a file is represented by a non-negative integer.

Headers To Include For Low-Level I/O:

`<fcntl.h>` and `<unistd.h>`

Opening Or Creating a File:

```
int open(char *filename, int flags, [mode_t mode]);
```

Possible values for the flags:

`O_RDONLY`: open for reading only
`O_WRONLY`: open for writing only
`O_RDWR`: open for both reading and writing
`O_CREAT`: for writing, creates the file if it doesn't exist
`O_TRUNC`: for writing, erase the old contents of the file if it exists
`O_APPEND`: for writing, append to the old contents of the file if it exists
`O_EXCL`: for writing, if the file already exists don't write to it (return a negative integer)

You can combine these values with the bitwise or operator "|" – for example, `O_WRONLY|O_CREAT`.

`mode` is an optional parameter, needed only if we're creating a new file. It gives the octal permissions for the new file. A common permission is 0644, which gives the owner of the file permission to read and write, while other users can only read it. Another is 0600, which gives the owner read and write permission while other users can't read or write it.

A return value less than zero signifies an error.

Closing a File:

```
int close(int filedes)
```

where

`<filedes>` is a number obtained from a call to `open` or `creat`.

Non-zero return value signifies an error.

Reading Bytes From a File:

```
ssize_t read(int filedes, void *buffer, size_t n);
```

`filedes` is a file number obtained from a call to `open`.

`buffer` is the address we want to read into.

`N` is the maximum number of bytes to read.

`ssize_t` & `size_t` are integer types.

The return value is the number of bytes actually read – usually the parameter n, but less if we hit the end of the file before we could read n bytes. If `read` returns -1, this signifies an error.

Writing Bytes To a File:

```
ssize_t write(int filedes, void *buffer, size_t n);
```

`filedes` is a file number obtained from a call to `open`.

`buffer` is the address of the start of the data we want to write
`n` is the number of bytes to write.
The return value is the number of bytes actually written, or -1 in case of an error.

Standard Input, Output and Error Files

The standard input, output and error files are files 0, 1, and 2. For better readability, use the macros `STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO, defined in <unistd.h>`.

Summary Page: Pipes
CISC 220, fall 2018

Creating a Pipe in a C Program:

```
int pipe(int filedes[2]);
```

`filedes[0]` is set to a descriptor for reading from the pipe
`filedes[1]` is set to a descriptor for writing to the pipe
Return value is -1 if there was an error.

Non-Blocking I/O:

Open a file with the `O_NONBLOCK` flag or call:

```
fcntl(file, F_SETFL, O_NONBLOCK);
```