Queen's University
Faculty of Arts and Science
School of Computing
CISC 220, Section 1
Final Exam, Fall 2016
Instructor: Margaret Lamb

****If you are enrolled in Prof. Shadi Khalifa's section (i.e. if SOLUS and OnQ show you his section) you are in the wrong place. You will not receive credit for writing this exam. Please talk with a proctor immediately and tell them you need to be at the Section 2 exam.****

- This exam is **THREE HOURS** in length.
- The exam includes the summary sheets for all topics tested 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 a question, cross out the discarded answer(s). If it's not clear which answer you want me to mark, I'll choose one at random 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.
- 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 **70 points**. Each question is worth 10 points. 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.
- 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 (10 points): Shell Scripting

Suppose there is a program on your path called `score`, which takes the name of a file as a parameter and prints an integer score for that file, which will always be between 0 and 100.

Write a script called `maxScore` that takes the name of a directory as its only parameter and prints the highest score reported by the `score` program for any file in that directory.

Your script may assume that there is at least one file in the directory. If there is a tie for the highest score it's not a problem; your script should just print that highest score.

Please note that I am *not* asking you to write the `score` program. Just assume that it exists, so that whenever you type

`score` *filename*

and *filename* is the name of an existing file, the command will print an integer between 0 and 100. to the standard output. Your job is just to write a `maxScore` script.

For example, suppose my current directory has a sub-directory called `myfiles` which contains four files, called `fileA`, `fileB`, `fileC` and `fileD`. Here is what happens when I invoke the `score` program with each of these four files:

```
-------$ score myfiles/fileA
34
-------$ score myfiles/fileB
57
-------$ score myfiles/fileC
44
-------$ score myfiles/fileD
56
```

The command `score myfiles` should print 57, since that is the largest score of any of the files in the `myfiles` directory.

Question 2 (10 points): Linux and Links

In your answer booklet, write the output of the commands below. There should be exactly 10 lines of output, one for each of the "cat" lines.

If you think a command would produce an error message, just write "error". I don't care about the exact text of the error message. Continue on to the end of the session after the error.

```
echo "Coulson" > file1
ln file1 file2
ln -s file1 file3
echo "Skye" >| file1
cat file3
mv file1 file4
echo "Fitz" >| file4
cat file1
cat file2
cat file3
cat file4

echo "Simmons" > fileA
ln fileA fileB
ln -s fileA fileC
rm fileA
echo "Ward" >| fileA
cat fileB
cat fileC
echo "Fury" >| fileA
cat fileA
cat fileB
cat fileC
```

Question 3 (10 points): Pointers

Read the following short program:

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

int main() {
  int a = 1;
  int b = 2;
  int c = 3;
  int *p, *q, *r;
  p = &a;
  q = &b;
  r = &a;

  printf("line 1: %d, %d, %d\n", *p, *q, *r);

  p = q;
  q = &c;

  printf("line 2: %d, %d, %d\n", *p, *q, *r);

  *p = *q;
  *r = *q + 1;
  *q = 5;

  printf("line 3: %d, %d, %d\n", *p, *q, *r);
  printf("line 4: %d, %d, %d\n", a, b, c);

  int numbers[] = {10,20,30,40,50,60,70,80,90,100,110,120};

  printf("line 5: %d, %d\n", *numbers, *(numbers+2));


  exit(0);
} // end main
```

Write the output of this program in your answer book.

Question 4 (10 points): I/O and Strings

Write a complete C program that reads a text file and decides if the lines in the file are in alphabetical order. The program should take one command line argument, the name of the input file. (Save time by assuming there will be exactly one argument.) If the lines of the input file are in order, the program should print a confirmation message to the standard output and exit successfully. If the lines are not in order, it should print a message to the standard output identifying the first pair of lines that are out of order and exit with a failure status.

Your program should compare lines as strings, in dictionary order with a case-sensitive comparison. (Hint: use the `strcmp` function.) It's okay if the input file contains duplicate lines. Lines may contain spaces.

If any line in the input file contains more than 100 characters, your program should immediately write an error message and exit with a failure status.

You may assume that there is at least one line in the input file.

Besides looking for lines over 100 characters long, the only I/O error you need to check for is not being able to open the input file. In that case, your program should write an error message and exit with a failure status.

Here is a transcript showing the correct behavior (assuming that your program is called `ordered`):

```
-------$ cat good
apple
banana
grape
grape
orange
-------$ ordered good
lines in input are in order
$: cat bad
apple
grape
banana
peach
orange
-------$ ordered bad
lines out of order:
  grape
  banana
-------$
```

Question 5 (10 points): Structs and Storage Allocation

In Assignment 5 you created linked lists using structs and pointers. As a reminder, here is the definition of the types used in that assignment:

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

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

The "next" pointer in the last node of a list should be null.

In your answer book, write a function called `deleteSecond` with a parameter of type `LinkedList`. The function should delete the second node of the linked list. Its return value should be the number that was inside the node that was deleted. The function must also free the heap space used for the node being deleted, to prevent a "memory leak".

If the list is empty or has only one element, your function should do nothing and return 0. (These cases aren't shown in the sample code on the next page in order to save paper, but please be sure to include them in your function.)

The header for your `deleteSecond` function should be:

```
int deleteSecond(LinkedList list)
```

On the next page there is an example of some code using `deleteSecond`, also using the `addToEnd` and `heapReport` functions from Assignment 5.

Question 5, continued:

Example code using deleteSecond:

```
printf("start of program: "); heapReport();
LinkedList list = NULL;

// create a singleton list
list = addToEnd(list, 10);
printf("heap report for singleton list: "); heapReport();

// Add five more values to the list
list = addToEnd(list, 20);
list = addToEnd(list, 30);
list = addToEnd(list, 40);
list = addToEnd(list, 50);
list = addToEnd(list, 60);
printf("full list: "); printList(list);
printf("heap report for full list: "); heapReport();

// Test deleteSecond
int i;
for (i = 0; i < 5; i++) {
  int val = deleteSecond(list);
  printf("deleted %d, list is ", val); printList(list);
} // end for

printf("heap report after deletions: "); heapReport();
```

Output:

```
start of program: heap use 0
heap report for singleton list: heap use 32
full list: [10,20,30,40,50,60]
heap report for full list: heap use 192
deleted 20, list is [10,30,40,50,60]
deleted 30, list is [10,40,50,60]
deleted 40, list is [10,50,60]
deleted 50, list is [10,60]
deleted 60, list is [10]
heap report after deletions: heap use 32
```

The above example was run on one of the CASLab Linux machines. The exact amount of heap space used might be different on a different computer, but the important thing is that the heap space reported in the two boldfaced lines is the same, which demonstrates that the space used to store all of the deleted nodes was released.

Question 6 (10 points): Make

You have a directory containing files `file1`, `file2`, `file3`, `file4`, `file5` and `makefile`. There are no other files in your directory. The commands `cmdA`, `cmdB`, `cmdC`, `cmdD` and `cmdE` are on your path. Each of these commands takes two parameters, both names of files, and writes to the standard output.

Here are the contents of `makefile`:

```
file8: file7 file9
     cmdC file7 file9 > file8
file6: file1 file2
     cmdA file1 file2 > file6
file10: file4 file5
     cmdE file4 file5 > file10
file7: file3 file6
     cmdB file3 file6 > file7
file9: file3 file10
     cmdD file3 file10 > file9
```

There are no other files in your directory.

A (5 points). You type the command `make file8`. Show the commands that make will execute. The order of the commands is important. There may be more than one correct order, in which case you will receive full credit for any correct order.

B (5 points). After executing Part A, all of your files are up to date. You make a change to `file1`, then type `make` again. Show the commands that make will execute. Once again, the order is important and there may be more than one correct order.

Question 7 (10 points): Signal Handling

You have written a program that modifies several important files. It includes a function called `closeFiles` and it's extremely important that this function is called before the program ends or some of the important files may be left in an inconsistent state. You are worried about the possibility of a user typing control-C while the program is running and causing it to stop suddenly without calling `closeFiles`. Fortunately, you remember that when a user types control-C into a program it results in a SIGINT signal being sent to the program. All you have to do is modify the program so that when it receives a SIGINT signal it will call `closeFiles` before it exits.

Below is a skeleton version of your program, with irrelevant details omitted and two locations marked in the comments:

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

// LOCATION A:

void closeFiles() {
  // closes the important files
}

int main() {
  // LOCATION B:

  // a whole bunch of code that modifies the important files

  closeFiles();
  return 0;
}
```

Show how you would modify the program so that it would always call `closeFiles` before exiting, even if it is interrupted by a SIGINT. You may not delete or change any of the lines above; you must change the program only by adding lines of code immediately after the "LOCATION A" and "LOCATION B" comments. Your answer (in your answer book) should list the code that you would add to the program in each of these locations, labeled clearly as "after location A" and "after location B".

Summary Page: Basic Linux & Bash

CISC 220, Fall 2016, Section 1

(last update: 2 September 2016)

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 Margaret (malamb@cs.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:

```
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

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 %n` *or* `kill %pre`

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 in a Vagrant window.

Summary Page: More Shell Skills

CISC 220, Fall 2016, Section 1

(last update 9/22/2016)

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
```

(example continued on next page)

```
------$  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

(last update 6. December 2016)

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 (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

```
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
   ((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 2016

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 2016

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 2016

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: Structs, Unions and Typedefs
CISC 220, Fall 2016

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 2016

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 2016

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