Queen's University
Faculty of Arts and Science
School of Computing
CISC 220
Final Exam, Fall 2017

Instructor: Margaret Lamb

- 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.
- 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 back page of this exam is blank and you may use it for rough work.
- The exam consists of 6 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 **60 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 anywhere 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 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 exam questions. Do your best to answer exam questions as written.

Question 1 (10 points): Bash Programming

Suppose there is a command on your system called `interesting`. It takes one file name as a parameter and outputs exactly one word – "yes" if it thinks the file is interesting and "no" if not.

Write a bash script that takes a directory name as a parameter and prints the total number of interesting files that exist in the directory and all of its sub-directories (and sub-sub-directories and so on). Directories themselves are never interesting and should not affect the count.

You may assume that the `interesting` script will always print either "yes" or "no" for any readable file. You may also assume that you have read permission for the directory and all of its sub-directories. However, it's possible that a file in the directory (or one of its sub-directories) is not readable; in that case, your script should print "`ERROR: <filename> is not readable`" to the standard error stream and then continue. (For example, if the unreadable file is named `mysterious` you should print "`ERROR: mysterious is not readable`".) Unreadable files should never count towards the total of interesting files.

Besides possible error messages for unreadable files, your script should just print a single number – the total number of interesting files inside the directory and all of its sub-directories. It should not print separate totals for the sub-directories or any other information.

Hint for including sub-directories: your script can call itself recursively. If you're uncomfortable with that don't skip this question. If your script ignores sub-directories but is correct in every other way you will get 7 out of the 10 points.

Question 2 (10 points): Bash Short-Answer Questions

Part A (5 points):

There is a Linux system running at the North Pole. One of the users on this system is called `rudolph`.

Rudolph is a member of a group called `reindeer`. There are lots of other users on this system, but this question is concerned only with Rudolph and two other users:

- Dasher (a member of the `reindeer` group)
- Santa (*not* a member of the `reindeer` group)

Rudolph, Dasher and Santa all belong to the `northPole` group.

None of the users has administrator priviledges (not even Santa!).

Here is a listing of a directory called `/usr/rudolph/flightPlan`:

```
-rw-rw-r-- 1 rudolph reindeer  123 Dec  24 10:00 sleigh
drwxr-x--- 2 rudolph reindeer 4096 Dec  24 10:00 presents/
-rw------- 1 rudolph reindeer  500 Dec  24 10:00 gps
```

And here is a listing of the `flightPlan` directory itself, made from the parent directory

```
drwxr-xr-x 3 rudolph northPole 4096 Dec  24 10:59 flightPlan/
```

Listed below are five actions a user might try to perform in the `flightPlan` directory. For each letter from A to E, you need to figure out which of the users have permission to perform the action. All you need to write in your answer book is the letters A through E, each followed by a list of the names of the users who can perform the action (one or more of Rudolph, Dasher and Santa, – or "nobody" if no one can perform the action).

A. view the contents of the `sleigh` file
B. view the contents of the `gps` file
C. change the contents of `sleigh` file (i.e. open it up with an editor and change it)
D. view the names of the files in the `presents` directory
E. add a file to the `presents` directory

For example, if you think that Rudolph and Santa but not Dasher can view the contents of the `sleigh` file, you'd start with:

A. Rudolph, Santa

Question 2, Part B (5 points):

You're in a directory containing only one file, called `holidayScript`. Here are the contents of `holidayScript`:

```
echo "dinner"
echo "pie" > meal
echo "turkey" 1>&2
echo "fruitcake" > dessert
echo "stuffing" >> meal
echo "cookies" >| dessert
```

In this directory, you execute the following command:

```
holidayScript >output1 2>output2
```

A. What will be written to the screen (i.e. not to a file)?

B. What will be written to the file called `output1`?

C. What will be written to the file called `output2`?

D. What will be written to the file called `meal`?

E. What will be written to the file called `dessert`?

If nothing will be written to a place, just write "nothing" for that item.

Question 3 (10 points): Make

Your current directory contains the following files and no others:

```
computeMain.c  format.c       javaToC*       makefile
mathHelp.h     mathHelp.java  rawData.txt
```

Here are the contents of `makefile`:

```
CC=gcc
CFLAGS=-g -Wall

results.txt: data.txt compute
        compute data.txt >| results.txt

data.txt: rawData.txt format
        format rawData.txt data.txt

format: format.o
        gcc -o format format.c

mathHelp.c: mathHelp.java
        javaToC mathHelp.java

mathHelp.o: mathHelp.h

computeMain.o: computeMain.c

compute: computeMain.o mathHelp.o
        gcc -o compute computeMain.o mathHelp.o
```

A. If you type "`make results.txt`" to the command prompt, what output will you see from make? (In other words, what commands will make execute?)

B. What new files will have been added to your directory?

C. If after the above you change `rawData.txt` and type "make" again, what output will you see from make? (In other words, what commands will make execute?)

Question 4 (10 points): Processes & Signals

You're writing a program that will create a very large, unsorted array. After creating the array, program will need to search to see if a particular value is present in the array. The objective of the program is simply to write "found" if the value is found and "not found" otherwise. It doesn't need to write any information to indicate where the value was found in the array or how many times it occurs in the array.

You've had a bright idea: since your computer has multiple processors you should be able to get a faster result if you use two processes, each searching one half of the array. In the space below I've written a skeleton version of a program. You must fill in the missing part so that the program will search the array using two parallel processes, write "found" or "not found", and exit.

Your initial process should create two child processes, one to search the first half of the array and one to search the second half. If a child finds the target value in its half of the array, it should immediately send a signal (you can choose what kind of signal) to the parent process and stop searching. If the parent process gets a signal of that type from either child it should kill both of its children, write a "found" message to the standard output, and exit.

If a child process does not find the target value in its half of the array, it should send a different kind of signal (your choice again) to the parent process. If the parent gets a signal of this type from both children it should write a "not found" message to the standard output, kill both of its children, and exit.

To save time, you may skip error-checking and assume that your `fork` calls succeed in creating child processes. You may use a simple linear search in each half of the array; you don't have to spend time writing a binary search.

You *must* solve the problem with two child processes as described above.

In the space below is an outline of the program. I have labeled three different locations in it, where you might need to add code. Write all your code in your answer booklet, labeling each section to tell me where it belongs.

```
// Assume all the necessary #includes are here

// Assume the following sets ARRAY_SIZE to a very large number
#define ARRAY_SIZE ...

// LOCATION A

int main() {
  int numbers[ARRAY_SIZE];
  int target;
  // Assume there's some code here that initializes target and
  // the numbers array.
  //
  // LOCATION B
} // end main

// LOCATION C
```

Question 5 (10 points): Structures and Unions

You are writing some code to use in an online card game. This game uses an unusual kind of cards. Every card contains either a number between 1 and 10 or a letter between 'A' and 'D'. Every card is either red or green. When scoring, every card has a point value, determined as follows:

- If the card contains a number, that number is its "base value"
- If the card contains a letter, that card's base value is 5 for 'A', 10 for 'B', 15 for 'C' or 20 for 'D'
- The value of a green card is its base value. The value of a red card is two times its base value.

Examples:

- A card containing a green 7 has a value of 7.
- A card containing a red 5 has a value of 10.
- A card containing a green 'B' has a value of 10.
- A card containing a red 'D' has a value of 40.

Here is some type definitions you have written:

```
struct cardStruct {
  // Every card is either red or green
  int isRed; // non-zero means it's read, zero means it's green

  // Every card contains either a number or an upper-case letter
  int hasLetter; // 0 means this card has a number,
                 // non-zero means it has a letter
  union {
    int intValue;
    int charValue;
  } value;
};// end struct cardStruct

typedef struct cardStruct cardType;
```

Your job is to write two functions. The descriptions of these functions are on the next page. You may assume that the definitions above have been included in your program.

Question 5, continued

A (5 points) Write a function called `cardValue` that takes a pointer to a card as a parameter and returns the value of the card that the parameter is pointing to. If the parameter is `NULL`, the function should return 0. The header of this function should be:

```
int cardValue(cardType *card)
```

B (5 points): A "deck" in this game consists of one or more cards. The value of a deck is the total of the values of all of the cards in the deck. Write a function called `deckValue` that takes an array of pointers to cards as a parameter and returns the total of the values of all of the cards in the deck. This function does not include a second parameter to tell you the length of the array. Instead, the array will contain a NULL as its last value.

For example, if your deck consists of a red A, a green 6 and a red 7, the array describing the deck would contain four elements:

1. a pointer to a structure describing a red A
2. a pointer to a structure describing a green 6
3. a pointer to a structure describing a red 7
4. NULL

The header of this function should be:

```
int deckValue(cardType *cards[])
```

or equivalently:

```
int deckValue(cardType **cards)
```

This function may call `cardValue` as a helper. If your `cardValue` function is incorrect it will not affect your mark for `deckValue`.

Question 6 (10 points): Files and Memory

The following function opens and reads several files in turn and computes and prints statistics about the contents of each file. Its parameters are an array of file names and the size of that array. Each of the files it reads is expected to contain an integer followed by floating point numbers. The integer should be a count of how many floating point numbers come after it in the file.

Assume that the `statistics` function is defined elsewhere in the program; the details of what it does are unimportant for this question. It does not use any heap space.

```
1 void manyFiles(char *filenames[], int numFiles) {
2   int i;
3   int result;
4   for (i = 0; i < numFiles; i++) {
5     FILE *infile = fopen(filenames[i], "r");
6     int size;
7     result = fscanf(infile, "%d", &size);
8     float *numbers = malloc(size * sizeof(float));
9     int j;
10    for (j = 0; j < size; j++) {
11      result = fscanf(infile, "%f", &numbers[j]);
12    } // end for
13    printf("statistics for file %s:\n", filenames[i]);
14    statistics(numbers, size);
15  } // end for
16 } // end manyFiles
```

This function works well when the input is correct, but it lacks error checking. It also runs out of heap space if given a lot of very large files.

You must add lines to this function to do the following:

- Get rid of "memory leaks" so that the function will not run out of memory regardless of the number of files or the sizes of the files.
- Abort – call exit(1) – if a file can't be opened.
- Abort if a file's input is not in the right format (i.e. you're trying to read a number and you get non-numeric characters).
- Abort if a file does not contain enough numbers (i.e. you hit the end of the file while trying to read a number).

To save time, no error messages are necessary; just call exit(1) if an error occurs. You also do *not* have to check whether a file contains extra characters at the end.

You **may not change** any of the existing lines of the function; just add lines.

Please save yourself time and don't write out a complete new version of this function. You can just write your additions by referring to the line numbers. For example, your answer could look like this:

Add this line after line 3:
some new line
Add these two lines after line 15:
two new lines
and so on

Summary Page: Basic Linux & Bash

CISC 220, Fall 2017

(last update: 09/08/17)

Navigating directories:

`cd` (*change directory):* moves you to another directory

`cd otherDir`: moves you to `otherDir`

`cd` with no arguments: moves you back to your home directory

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

Listing file information: the `ls` command

arguments should be names of files & directories

for each file: lists the file

for each directory: lists the *contents of* the directory (unless –d flag)

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

some flags for `ls`:

`-a`: include files & directories starting with "."

`-d`: for directories, show directory itself instead of contents

`-l`: (lower-case L) long format: lots of information about each entry

`-R`: list sub-directories recursively

`-1`: (one) list each file on separate line (no columns)

Displaying the contents of a short file:

`cat <filename>`

Wildcards in file names:

`?`: any single character

`*`: any sequence of zero or more characters

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

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>`

Summary Page: More Shell Skills

CISC 220, Fall 2017

(last update 09/17/17)

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 7. December 2017)

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 2017

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 2017

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 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

(strtol deleted to save space, not needed for final exam)

Summary Page: Make
CISC 220, fall 2017

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: Structs, Unions and Typedefs
CISC 220, Fall 2017

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 2017

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(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 2017

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);
```

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:
  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)`.

This page left empty. You may use it for rough work, but remember that nothing written on this exam paper will be marked.