Queen's University, Faculty of Arts and Science
School of Computing
CISC 220 Final Exam, Fall 2019
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 extremely 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 can 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 unless you're really out of space; let's not kill any more trees than we have to! And make sure your student number is on each answer book.
- 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 to be understood and marked. However, good style (indenting, meaningful variable names, etc) can make your intentions more clear and thus might result in more partial marks for an answer that isn't completely correct.
- 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.

This exam document is copyrighted and is for the sole use of students registered in CISC 220 and writing this exam. This material shall not be distributed or disseminated.

Question 1 (10 Points): Links

This question is about hard and symbolic links. While answering this question, remember that re-writing the contents of a file using ">|" when the target already exists does *not* create a new file. It merely overwrites the contents of the file. The file's inode number stays the same. For example:

```
-------$ cat myfile
hello
-------$ ls -i myfile
39190591 myfile
-------$ echo goodbye >| myfile
-------$ cat myfile
goodbye
-------$ ls -i myfile
39190591 myfile
```

(Reminder: ls -i displays the inode number of a file along with its name)

In the space below there is a series of bash commands. In your exam book, please write the output that this sequence of commands will produce.

```
-------$ echo "snow" >| file1
-------$ ln file1 file2
-------$ ln -s file1 file3
-------$ echo "sleet" >| file1
-------$ echo "file 2:"
-------$ cat file2
-------$ echo "file 3:"
-------$ cat file3
-------$ rm -f file1
-------$ echo "rain" >| file4
-------$ ln file4 file1
-------$ echo "sun" >| file1
-------$ echo "rain" >| file2
-------$ echo "dark" >| file3
-------$ echo "file 1:"
-------$ cat file1
-------$ echo "file 2: "
-------$ cat file2
-------$ echo "file 4:"
-------$ cat file4
```

In your answer book, write the output that this code would produce.

Question 2 (10 Points): Shell Scripting

Write a bash script that takes one argument, which should be the name of a directory. You may save time by assuming that the script will be called with exactly one argument and that the argument will be the name of a readable directory.

The script must add up the lengths of all of the file and sub-directory names in that directory and print the total. (Not the sizes of the files and sub-directories but the numbers of characters in their names.)

If there are sub-directories, don't look at the files inside them; just add the length of the sub-directory names to the total.

If the argument is not the name of a directory, your script should write "ERROR!" and return with a non-zero exit code. If the argument *is* the name of a directory, your script should return with an exit status of 0.

For example, if I call my script `countLengths`:

```
-------$ ls /cas/course/cisc220/files
marks.c  subdir
-------$ countLengths /cas/course/cisc220/files
13
```

Question 3 (10 Points): Arrays and Heap Use

Write a function called `addArrays` that takes three parameters: two arrays of floats and an integer. The integer should be the length of the arrays – meaning that the arrays should be of the same size, although there's no way your function can check this. Your function must create a new array of integers of the same size on the heap and return a pointer to the beginning of the array. Each element of this new array must be the sum of the corresponding elements of the two parameter arrays.

For example:

```
float one[] = {1,2,3};
float two[] = {4,5,9};
float *three = addArrays(one, two, 3);
```

After the call in the third line, the variable three should be the address of a new array on the heap containing the three floating-point numbers 5, 7 and 12.

Question 4 (10 Points): Structs and Pointers

In class we discussed an example of a linked list of integers implemented with C structs, using these definitions:

```
struct ListNode {
     int value;
     struct ListNode *next;
};

typedef struct ListNode *ListPtr;
```

The `next` attribute in the last node of the list will always have the value `NULL`, to show that there is no next node after it.

Write a function called **`deleteLast`** with this header:

```
ListPtr deleteLast(ListPtr *list);
```

The parameter will be a pointer to the first node in a linked list, or NULL for an empty list. The return value must be a pointer to the first node in the modified list – which in most cases will be the same as the first node of the original list. However, if the list had exactly one element the list will be empty after the deletion and in that case the return value of `deleteLast` should be NULL.

If the list was already empty, the function should print an error message and return NULL.

Question 5 (10 Points): C Programming

Write a complete C program that will prompt the user for a positive integer, read an integer, and print all the multiples of 7 that are less than or equal to the integer. If the user enters something that isn't a positive integer, the program should print an error message and exit immediately with a non-zero exit status. Note that if the user enters a number between 1 and 6 it's not an error but the program will print nothing.

Here's a transcript of three runs of my solution:

```
quiz2:242: cquestion
enter a positive integer: 28
7
14
21
28
quiz2:243: echo $?
0
quiz2:244: cquestion
enter a positive integer: abc
error: not an integer
quiz2:245: echo $?
1
quiz2:246: cquestion
enter a positive integer: -2
error: not a positive integer
quiz2:247: echo $?
1
quiz2:248:
```

Write your program in your answer book. Please note that we're asking for a complete program, not just a single function.

Question 6 (10 Points): Processes and Signals and Strings

This question is a very simple imitation of the way a Linux system can change the settings of a system process that is running continuously and can't be stopped safely without re-booting the system.

Consider the following short C program:

```
#include <stdio.h>
char name[21] = "Patrick Deane";
int main() {
     for (;;) { // infinite loop
          printf("Hello, %s!\n", name);
          sleep(300);
     } // end for
     return 0;
} // end main
```

If you run this program in the background, it will print

```
Hello, Patrick Dean!
```

every 300 seconds (5 minutes). We'd like to be able to change the name in the greetings while the program is running, without having to stop and re-start it. In your answer book, write a new version of the program to add the following behavior:

- If the program receives a SIGUSR1 signal, it reads a new name from the first line of the file `name.txt` (in the directory in which the program is running) and starts using that name in its greetings.
- If the program receives a SIGUSR2 signal, it goes back to the default name of "Patrick Deane".

If the program can't open `name.txt`, it should write an error message to `stderr` and continue running without changing the name in its greetings.

Your new version must be the original version with lines added, not a complete re-write.

The program should read only the first line of `name.txt`. If the first line contains more than 20 characters, the program should use only the first 20 (plus an ending `'\0'`) as the name, to avoid overrunning the space allocated for the name. No error message is needed in that case. For example, if the first line of `name.txt` is "William Lyon Mackenzie King", your program would just print "William Lyon Mackenz".

If your method of reading the name would put a '\n' character at the end of the name, make sure your program gets rid of it, so that the exclamation point in the greeting comes immediately after the name and not on the next line.

Don't bother adding #include's to your program.

On the next page is a sample session using my solution. (I've called my program `signalQuestion`.) I paused long enough between typing the input lines that multiple messages were displayed between some pairs of input lines. Remember that bash won't automatically print its prompt every time a background program prints something.

Question 6, continued:

```
-------$ signalQuestion &
[2] 18268
-------$ Hello, Patrick Deane!
Hello, Patrick Deane!
Hello, Patrick Deane!
echo "Albus Dumbledore" >| name.txt
-------$ Hello, Patrick Deane!
Hello, Patrick Deane!
kill -SIGUSR1 18268
-------$ Hello, Albus Dumbledore!
Hello, Albus Dumbledore!
echo "John Jacob Jingleheimer Smith" >| name.txt
-------$ kill -SIGUSR1 18268
-------$ Hello, John Jacob Jinglehei!
Hello, John Jacob Jinglehei!
kill -SIGUSR2 18268
-------$ Hello, Patrick Deane!
```

Summary Page: Basic Linux & Bash

CISC 220, Fall 2019

(last update: 12/02/19)

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 the bash shell for the current session only.

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 .` ("." means 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 permissions:

`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 permissions for new files you create

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

Note: Regardless of your umask, Linux doesn't give people execute permission to a new file unless it's in a format that looks like an executable file. This has the annoying consequence that you have to call `chmod` when you create shell scripts because they look like plain text.

Examples of File/directory protection commands:

`chmod g+rx myprogram`: gives group members read and execute permissions for `myprogram`

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

`umask` sets the default permission for new files you create (for the rest of the current session)

Example using `umask`:

```
-------$ umask -S
u=rwx,g=,o=     (Give me all permissions for new files and don't give anyone else any
          permissions)
-------$ umask g+rx  (Give my group read & execute permission for new files)
-------$ umask -S
u=rwx,g=rx,o=
-------$ umask g-x (No longer give my group execute permission for new files)
-------$ umask -S
u=rwx,g=r,o=
-------$ umask g+w  (Give my group write permission for new files)
-------$ umask -S
u=rwx,g=rw,o=
-------$ umask g=  (Don't give my group any permissions for new files)
-------$ umask -S
u=rwx,g=,o=
-------$ echo > newfile
-------$ ls -l newfile
-rw-r----- 1 lambm student 1 Apr 16 13:06 newfile
```

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 is similar to `cp`, but it gives files new names and/or locations instead of making extra copies. After this command the old file names will be gone. If last arg is a directory, moves all previous args to that directory.

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` *(also logs you out if this is the login shell)*

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 Vagrant.

Summary Page: More Shell Skills

CISC 220, Fall 2019

(last update 08/18/19)

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 the login shell)*

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)*

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 (not including $0)%
to execute script in a sub-shell, type file name of script
to execute script in current shell: `source` + name of script

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)

Displaying Values

`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)*

Quoting

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="${today}, 2019"
------$  echo $today
Tuesday Jan 13, 2019
------$  echo the price is \$2.97
the price is $2.97
```

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 2. December 2019)

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)

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 2019

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 2019

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

`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

Using Command-Line Arguments:

Give your main function two parameters:

`int main(int argc, char *argv[])`

When you run the program, `argc` will be set to the number of command-line arguments the user typed and argv will be set to their values. Note that argc will include the name of the program in the count, so if you call the program with 3 arguments argc will be 4 and the first element of argv will be the name of the program.

Summary Page: Structs, Unions and Typedefs
CISC 220, Fall 2019

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 2019

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 2019

(This summary sheet has been condensed to save paper. It still includes all the information you need for this exam.)

Types of Signals:

name	default action	notes
SIGALRM	terminates process	Used by Linux "alarm clock" timer
SIGCHLD	ignored	Sent by system when a child process terminates or stops
SIGINT	terminates process	Sent by system when user hits control-C
SIGTERM	terminates process	Programs can catch SIGTERM
SIGKILL	terminates process	Programs can't catch SIGKILL.
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));
```

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);
```