PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

St. George Campus

DECEMBER 2015 EXAMINATIONS

CSC 209H1F
Instructor — Michelle Craig
Duration — 3 hours

Examination Aids: One double-sided 8.5x11 sheet of paper. No electronic aids.

Student Number: |_|_|_|_|_|_|_|_|_|_|

Last (Family) Name(s): _____

First (Given) Name(s): _____

---

*Do **not** turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below *carefully*.)

This final examination consists of 8 questions on 18 pages. A mark of at least 30 out of 75 on this exam is required to pass this course. *When you receive the signal to start, please make sure that your copy of the examination is complete.*
You are not required to add any `#include` lines, and unless otherwise specified, you may assume a reasonable maximum for character arrays or other structures. Error checking is not necessary unless it is required for correctness or specifically requested.

Good Luck!

MARKING GUIDE

# 1:	_____/14
# 2:	_____/ 5
# 3:	_____/10
# 4:	_____/11
# 5:	_____/10
# 6:	_____/10
# 7:	_____/10
# 8:	_____/ 5
TOTAL:	_____/75

Question 1. [14 MARKS]

Part (a) [1 MARK]

You have a text file `calendar.full` in your current directory. Write a shell command that will extract a copy of all the lines that contain the phrase `TODO` and put them into a new file called `my_TODO_list`. Do not change the original calendar file.

Part (b) [6 MARKS]

You are arguing with your friend over who has the longer to do list. You each have a file in your home directory named `my_TODO_list`. Your friend's login is **g4stud**. Write a shell fragment that will print "Mine is longer" if your file has more lines than your friend's file and "I'm not telling" otherwise. You shouldn't assume anything about the current directory when this script is run. Assume you already have all the permissions to access the files you need to run the script.

Part (c) [2 MARKS]

In A3 you wrote a `psort` program that was called as follows
`psort -n <number of processes> -f <input file name> -o <output file name>`. For this question, write a shell **loop** that calls your program with 1, 2, 4, 8, 16 and 32 processes on the inputfile `big_test.in` from the directory `/u/csc209h/testfiles`. The `psort` executable is in the current directory. You should save the output in files in this same directory named `testrun_X.out` where `X` is the number of processors for that run.

Part (d) [3 MARKS]

We have a file `tic_tac_toe.c` with a main function that depends on some functions in `game_stuff.c` and `general_stuff.c`. Prototypes for the functions are in `game_stuff.h` and `general_stuff.h` and these have been included in our program. Write a set of makefile rules that build the executable from tic_tac_toe.c and name it ttt. Your rules should only recompile files as needed.

Part (e) [2 MARKS]

Consider the following program that prints triathlon activities.

```
int main() {
    int j;
    int r = fork();
    if (r == 0) {
        fprintf(stderr, "swim\n");
        j = fork();
        fprintf(stderr, "bike\n");
    } else if (r > 0) {
        fprintf(stderr, "run\n");
    }
    fprintf(stderr, "eat\n");
}
```

In the table below, provide how many times each activity is printed assuming there are no errors from the system calls.

Activity	**How many times?**
swim	
bike	
run	
eat	

Question 2. [5 MARKS]

Each example below contains an independent code fragment. In each case a variable type has been replaced with XXX. In the boxes to the right of the code, give a value for XXX so that the code fragment would compile and run without warnings or errors assuming that the hidden code allocates space appropriately.

Code Fragment	Replacement for XXX
`XXX v1 = &argv[1];`	

`float **f;`
`// some hidden code`
`XXX v2 = *f;`	

`double **z;`
`// some hidden code`
`XXX v3 = **z;`	

`struct stat *s;` `XXX v4 = &s;`	

`struct stat *s;`
`// some hidden code`
`XXX v5 = *s;`	

Question 3. [10 MARKS]

Consider the following C program.

```
char *where_am_I(char *location) {
    char *result = malloc(strlen(location) * 2 + 1);
    strcpy(result, location);
    strcat(result, location);
    return result;
}

int found(int choice) {
    if (choice == 1) {
        char location[8] = "Iceland";
    } else {
        char location[8] = "Denmark";
    }
    return 0;
}

int main() {

    XXX statements from the table below go here
    // what memory is allocated at this point?
    return 0;
}
```

In the table below are six different code fragments that could go in this program at the point labelled XXX. In the box beside each fragment, indicate what memory would be currently allocated at the point when the XXX replacement had just finished executing. (Before the return statement.) You do not need to include the allocation of the string literals `Iceland` and `Denmark` which will be in read-only memory for every version of the program. The first fragment is completed for you as an example. Since the same code can allocate memory in more than one location, you may need to check more than one box per fragment.

Code Fragment		location	number of bytes
`int waldo1;`	✓	stack	sizeof(int) bytes
		heap	
		read-only memory	
`char *waldo2 = "New York";`		stack	
		heap	
		read-only memory	
`char *waldo3 = malloc(10);`		stack	
		heap	
		read-only memory	
`char waldo4[15] = "Ottawa";`		stack	
		heap	
		read-only memory	
`char *waldo5 = where_am_I("Peru");`		stack	
		heap	
		read-only memory	
`int waldo6 = found(2);`		stack	
		heap	
		read-only memory	

Question 4. [11 MARKS]

Each of the code fragments below reveal a problem. In the box below each fragment, explain **briefly** what is wrong.

Part (a) [1 MARK]

```
char sport[6] = "hockey";
```

Part (b) [1 MARK]

```
char instrument[SIZE] = "bass";
strncat(instrument, " guitar", SIZE - 1);
```

Part (c) [1 MARK]

```
char **ta_names = malloc(sizeof(char *) * 4);
int i;
for (i=0; i<4; i++) {
    ta_names[i] = malloc(9);
}
strcpy(ta_name[0], "Elaine");   strcpy(ta_name[1], "Stathis");
strcpy(ta_name[2], "Julianna");   strcpy(ta_name[3], "Kyle");
// hidden code
free(ta_names);
```

Part (d) [1 MARK]

```
int *mystery(int param) {
    return &param;
}
```

Part (e) [1 MARK]

```
int *status_pt;
wait(status_pt);

if (WIFEXITED(*status_pt)) {
```

Part (f) [1 MARK]

```
void do_child_process(int fd[2]) {
    // get an int from reading end of pipe from parent
    int child_no;
    read(fd[0], &child_no, sizeof(int));
    // do something with this integer in hidden code
    // write the result back to the parent on the writing end of pipe
    write(fd[1], &child_no, sizeof(int));
    exit(0);
}
```

Part (g) [1 MARK]

```
// signal handler previously declared
void dance(int code) {
    printf("Dance dance dance\n");
}

// install signal handler in main
struct  sigaction sa;
sa.sa_handler = dance;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
sigaction(SIGKILL, &sa, NULL);
```

Part (h) [1 MARK]

```
// signal handler previously declared
void report_time(int reads) {
    printf("Since the last signal I have done %d reads.\n");
}

// install signal handler in main
struct  sigaction sa;
sa.sa_handler = report_time;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
sigaction(SIGUSR1, &sa, NULL);

int reads;
// start code that changes the value of reads and is interupted to report the value
// by a SIGUSR1 signal
```

Part (i) [1 MARK]

```
int result = execl("./program", "program", NULL);
if (result > 0) {
    printf("%d files processed\n", result);
}
```

Part (j) [1 MARK]

```
int find_id_match(int *pids, int pid_needed) {
    int i;
    for (i = 0; i < sizeof(pids); i++) {
        if (pids[i] == pid_needed) {
            return i;
        }
    }
    return -1;
}
```

Part (k) [1 MARK]

```
int *pt;
int i;
*pt = i + 2;
```

Question 5. [10 MARKS]

Part (a) [7 MARKS]

You have a binary file that consists of a header followed by a number of records. The first four bytes of the file hold an integer representing the length of the header (including those 4 bytes.) You may assume that you have the `get_file_size` function that you used in assignment 3 available in the file `helper.c`.

Your job is to complete a function below that dynamically allocates an array of just enough size, reads the records from that file into the array (ignoring the header), and returns the number of records that were read. The function is called with a pointer parameter that must point to the newly created array when the function returns.

```
                                    FROM helper.h
int get_file_size(char *filename);
struct rec {
    int value;
    char label[10];
};
```

```
int read_file(struct rec **array_pt, char *filename) {
    // open the file



    // read the size of the header



    // calculate the number of records in the array



    // allocate space for the array



    // read the records from the file to the array



    // finish whatever you need
```

Part (b) [1 MARK]

If you were to write this function in a file named `lab7.c` as part of a 209 lab, your instructor might provide both `helper.h` and `helper.c`.
Do you need `helper.h`? YES ☐ NO ☐ (check one). If you said yes, explain specifically how you make use of this file. If you answered no, explain why it is unnecessary for this problem.

Part (c) [1 MARK]

Do you need `helper.c`? YES ☐ NO ☐ (check one). If you said yes, explain specifically how you make use of this file. If you answered no, explain why it is unnecessary for this problem.

Part (d) [1 MARK]

Did you make use of the specific fields inside `struct rec` to write your function? YES ☐ NO ☐ (check one).
If you said yes, could you have written the function without knowing the details of the struct within your own code? If you said no, could you have made use of the struct details to be more efficient? Why or why not?

Question 6. [10 MARKS]

Part (a) [5 MARKS]

Write a function `kill_parent_process` that will be used as a signal handler in the next part of this question. When run, this function should send a SIGKILL signal to the parent of the running process. So that you can demonstrate that you know how to do this, check the return value of your system call to kill and respond to the error in the conventional way..

Part (b) [1 MARK]

The program is supposed to send a SIGKILL signal to the parent. Sometimes this will kill the parent and other times it will not. Explain why this happens.

Part (c) [4 MARKS]

Here is the code fragment from main that installs the signal handler in your program dots.

```
//POINT A
struct  sigaction sa;
sa.sa_handler = kill_parent_process;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
//POINT B
sigaction(SIGINT, &sa, NULL);
//POINT C
while (1) {
    fprintf(stderr,".");
    sleep(1);
}
```

Here is a table showing different actions that the user could take at different times in the program. Complete the final column giving the behaviour of the program. Each row of the table is independent.

Action	**Point in program**	**Behaviour**
Control-Z	POINT A	
Control-C	POINT B	
User finds PID of dots then types `KILL -INT <PID>`	POINT B	
Control-C	POINT C	

Question 7. [10 MARKS]

In A4 you used the netcat program to act as the client for your poll server. In this question we will write our own simplified version of netcat. Your program should take two commandline arguments. The first is the name of the machine to which you want to connect and the second is the port.

Your program should open a socket to this port on this machine and then simply echo the data back and forth. It must listen to both the keyboard and the socket. Whenever the user types something on the keyboard, the program sends this to the socket and whenever information arrives on the socket, the program echoes this data to the user on standard output.

In assignment 4 we worried about buffering partial reads, but for this question you may ignore buffering and assume that each time you read, the data will fit into a buffer of size `MAXSIZE`. To make your code easier to read (and to write), please omit error checking for system call failures, error checking for user-error in providing arguments and #include statements. I've given some bits of code to help keep you organized and there are useful prototypes on the last page of the test for things you didn't need to memorize.

```
int main(int argc, char* argv[])
{
    char buf[MAXSIZE];




    // create the socket




    // get set up for connecting
    struct sockaddr_in peer;
    peer.sin_family = AF_INET;


    peer.sin_port =


    struct hostent *hp;
    hp = gethostbyname(argv[1]);
    peer.sin_addr = *((struct in_addr *)hp->h_addr);

    // connect the socket







    // anything else to do before the infinite loop (on next page)
```

```
    while (1) {


        // wait for someone to send me something









        //  if it is some new typing at the keyboard
        if                                                    {









        }
        // if it is new data returned from the peer
        if                                                    {









        }



    }
    return(0);
}
```

Question 8. [5 MARKS]

Given the following struct where the category field is either the string "novice" or the string "expert", complete the function find_winning_novice according to its comment.

```
typedef struct {
 char name[10];
 int score;
 char category[7]; //novice, expert
} rec;

/* Return the index of the element in array competitors that is the novice with the highest score.
   If competitors has no novices, return -1. If there is a tie, return any one of the tied indices. */

int find_winning_novice(rec *competitors, int n) {
```

Total Marks = 75

C function prototypes and structs:

```
int accept(int sock, struct sockaddr *addr, int *addrlen)
char *asctime(const struct tm *timeptr)
int bind(int sock, struct sockaddr *addr, int addrlen)
int close(int fd)
int closedir(DIR *dir)
int connect(int sock, struct sockaddr *addr, int addrlen)
char *ctime(const time_t *clock); int dup2(int oldfd, int newfd)
int execl(const char *path, const char *arg0, ...  /*, (char *)0 */);
int execvp(const char *file, char *argv[])
int fclose(FILE *stream)
int FD_ISSET(int fd, fd_set *fds)
void FD_SET(int fd, fd_set *fds)
void FD_CLR(int fd, fd_set *fds)
void FD_ZERO(fd_set *fds)
char *fgets(char *s, int n, FILE *stream)
int fileno(FILE *stream)
pid_t fork(void)
FILE *fopen(const char *file, const char *mode)
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
int fseek(FILE *stream, long offset, int whence);
      /* SEEK_SET, SEEK_CUR, or SEEK_END*/
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
pid_t getpid(void);
pid_t getppid(void);
unsigned long int htonl(unsigned long int hostlong) /* 4 bytes */
unsigned short int htons(unsigned short int hostshort) /* 2 bytes */
char *index(const char *s, int c)
int kill(int pid, int signo)
int listen(int sock, int n)
void malloc(size_t size);
unsigned long int ntohl(unsigned long int netlong)
unsigned short int ntohs(unsigned short int netshort)
int open(const char *path, int oflag)
       /* oflag is O_WRONLY | O_CREAT for write and O_RDONLY for read */
DIR *opendir(const char *name)
int pclose(FILE *stream)
int pipe(int filedes[2])
FILE *popen(char *cmdstr, char *mode)
ssize_t read(int d, void *buf, size_t nbytes);
struct dirent *readdir(DIR *dir)
int select(int maxfdp1, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
       /* actions include SIG_DFL and SIG_IGN */
int sigaddset(sigset_t *set, int signum)
int sigemptyset(sigset_t *set)
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
       /*how has the value SIG_BLOCK, SIG_UNBLOCK, or SIG_SETMASK */
unsigned int sleep(unsigned int seconds)
int socket(int family, int type, int protocol) /* family=PF_INET, type=SOCK_STREAM, protocol=0 */
int sprintf(char *s, const char *format, ...)
int stat(const char *file_name, struct stat *buf)
char *strchr(const char *s, int c)
size_t strlen(const char *s)
char *strncat(char *dest, const char *src, size_t n)
int strncmp(const char *s1, const char *s2, size_t n)
char *strncpy(char *dest, const char *src, size_t n)
long strtol(const char *restrict str, char **restrict endptr, int base);
```

```
int wait(int *status)
int waitpid(int pid, int *stat, int options) /* options = 0 or WNOHANG*/
ssize_t write(int d, const void *buf, size_t nbytes);

WIFEXITED(status)       WEXITSTATUS(status)
WIFSIGNALED(status)     WTERMSIG(status)
WIFSTOPPED(status)      WSTOPSIG(status)
```

Useful structs

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
struct hostent {
    char *h_name; // name of host
    char **h_aliases; // alias list
    int h_addrtype; // host address type
    int h_length; // length of address
    char *h_addr; // address
}
struct sockaddr_in {
    sa_family_t sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char pad[8]; /*Unused*/
}
```

```
struct stat {
    dev_t st_dev; /* ID of device containing file */
    ino_t st_ino; /* inode number */
    mode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of hard links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device ID (if special file) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t st_blocks; /* number of 512B blocks allocated */
    time_t st_atime; /* time of last access */
    time_t st_mtime; /* time of last modification */
    time_t st_ctime; /* time of last status change */
};
```

Shell comparison operators

Shell	Description
-d filename	Exists as a directory
-f filename	Exists as a regular file.
-r filename	Exists as a readable file
-w filename	Exists as a writable file.
-x filename	Exists as an executable file.
-z string	True if empty string
str1 = str2	True if str1 equals str2
str1 != str2	True if str1 not equal to str2
int1 -eq int2	True if int1 equals int2
-ne, -gt, -lt, -le	For numbers
!=, >, >=, <, <=	For strings
-a, -o	And, or.

Useful Makefile variables:

$@	target
$^	list of prerequisites
$<	first prerequisite
$?	return code of last program executed

Useful shell commands:

cat, cut, echo, ls, read, sort, uniq, wc
set (Note: with no arguments set prints the list of environment variables)
ps aux - prints the list of currently running processes
grep (returns 0 if match is found, 1 if no match was found, and 2 if there was an error)
grep -v displays lines that do not match
diff (returns 0 if the files are the same, and 1 if the files differ)

$0	Script name
$#	Number of positional parameters
$*	List of all positional parameters
$?	Exit value of previously executed command