Question 1. [8 MARKS]

Each code snippet below contains a problem. Briefly state what the problem is and then modify the code to address the problem. Make minimal changes so that the code works as intended. The code snippets in each part of this question are independent of each other.

Part (a) [2 MARKS]

```
struct thing {
    struct thing *next;
};

struct thing *a = malloc(sizeof(struct thing));
a->next = malloc(sizeof(struct thing));
free(a);
free(a->next);
```

Problem	**Solution**
Attempting to access `a->next` after `a` has already been freed.	`free(a->next);` `free(a);`

Part (b) [2 MARKS]

```
char *return_s() {
    char *s = "hello world";
    return s;
}

int main() {
    char *r = return_s();
    free(r);
}
```

Problem	**Solution**
`r` in `main()` is not on the heap, so it cannot be freed.	`Make return_s() use malloc, or don't` `call free(r) in main.`

Part (c) [2 MARKS]

```
int fd[2];
int res = fork();   // error-checking is not shown, but that is not the problem
pipe(fd);

if (res == 0) {
    close(fd[0]);
} else if (res > 0) {
    close(fd[1]);
}
```

Problem	**Solution**

The pipe is not shared by the two processes. At present, they each have a separate pipe. Instead the pipe call has to be moved above the fork call.	`int fd[2];`
	`pipe(fd);`
	`int res = fork();`
	
	`...`

Part (d) [2 MARKS]

```
int *make_list() {
    int list[3] = {1, 2, 3};
    return list;
}

int main() {
    int *l = make_list();
    int i;
    for (i = 0; i < 3; i++) {
        printf("%d\n", l[i]);
    }

    return 0;
}
```

Problem	**Solution**
`list` in `make_list()` is memory on the stack.	`Modify the function to make use` `of dynamic memory instead.`

Question 2. [8 MARKS]

Each code snippet below contains a problem. Briefly state what the problem is and then modify the code to address the problem. Make minimal changes so that the code works as intended. The code snippets in each part of this question are independent of each other.

Part (a) [2 MARKS]

```
struct animal {
    char *name;
};

struct animal dog;
dog->name = "Fish";
```

Problem	**Solution**
`dog` is not a pointer, so we can't use `->` to get its name value.	`dog.name = "Fish";`

Part (b) [2 MARKS]

```
char *s = "hey";
s[1] = 'a';
printf("%s", s); //print "hay"
```

Problem	**Solution**

Can't mutate a string literal.	`char s[] = "hey";`
	`s[1] = 'a';`
	`printf("%s", s); //print "hay"`

Part (c) [2 MARKS]

```
char *word = malloc(6 * sizeof(char));
word = "hello";
```

Problem	**Solution**
Memory leak. Assigning the string literal `"hello"` to `word` causes you to lose your only reference to the memory originally allocated to the variable.	`char *word = malloc(6 * sizeof(char));` `strncpy(word, "hello", 6);`

Part (d) [2 MARKS]

```
int *a;
*a = 3;
```

Problem	**Solution**

`a` is not initialized and does not have memory assigned to it.	`int *a = malloc(sizeof(int));`
	`*a = 3;`
	`----- OR -----`
	`int *a, b = 3;`
	`a = &b;`

Question 3. [9 MARKS]

Match each term below to the description that best describes it. (Functions are indicated with parentheses.) Note that some descriptions will not be used and each description should be used at most once.

[I] `socket()`	[D] `bind()`	[K] `listen()`	[M] `accept()`	[O] `connect()`
[J] endianness	[G] `htons()`	[X] port	[R] `dup2()`	[L] `select()`
[U] pipe	[C] errno	[Y] `wait()`	[E] `sigaction()`	[A] zombie

A. a process that has completed execution, but is still waiting for its parent to accept its termination status

B. an active process whose parent has finished or terminated before it

C. an integer variable that is set by certain functions

D. assigns a name / address to a socket

E. changes action taken by a process on receipt of a specific signal

F. chooses a client to connect to

G. converts values from host to network byte order

H. converts values from network to host byte order

I. creates an endpoint for communication and returns a descriptor

J. describes byte order

K. establishes a queue for connections

L. examines descriptor sets to see if any are ready for reading, writing, etc.

M. gets a connection from the queue

N. gets data from a server

O. initiates a connection from the client

P. initiates a connection from the server

Q. listens for new activity on one existing client

R. makes a copy of an open file descriptor

S. monitors the activity of a signal

T. object that allows bidirectional data flow

U. object that allows unidirectional data flow

V. raises a signal

W. specifies a file descriptor to examine

X. specifies which process to interact with on a server

Y. suspends execution until a child has terminated

Z. suspends execution until all children have terminated

Question 4. [6 MARKS]

Consider the following code.

```
/* Precondition: strlen(orig) == strlen(new) */
void change_airline(char** flights, int size, char *orig, char *new){

    for (int i = 0; i < size; i++) {
        if (strncmp(flights[i], orig, strlen(orig)) == 0) {
            strncpy(flights[i], new, strlen(orig));
        }
    }
}

int main(int argc, char**argv) {

    char first[8] = {'A', 'C', ' ', '1', '2', '3', '4', '\0'};
    char *flights[3];

    flights[0] = first;
    flights[1] = malloc(sizeof(char) * 8);
    strcpy(flights[1], "AC 5555");
    flights[2] = "WS 9876";

    change_airline(flights, 3, "AC", "UN");

    /* HERE */

    for (int i=0; i < 3; i++) {
        printf("%s\n", flights[i]);
    }

    return 0;
}
```

Part (a) [1 MARK]

The code above runs without error. What does it print to `stdout`?

```
UN 1234
UN 5555
WS 9876
```

Part (b) [5 MARKS]

Complete the memory model diagram to reflect the state of memory at the point in the program marked by the comment `HERE`.

Section	Address	Value	Label
Read-only	0x100		
	0x104		
	0x108		
	0x10c		
	0x110		
	0x114		
	0x118		
	0x11c		
	⋮	⋮	
Heap	0x23c		
	0x240		
	0x244		
	0x248		
	0x24c		
	0x250		
	0x254		
	0x258		
	0x25c		
	⋮	⋮	
Stack	0x454		
	0x458		
	0x45c		
	0x460		
	0x464		
	0x468		
	0x46c		
	0x470		
	0x474		
	0x478		
	0x47c		
	0x480		
	0x484		
	0x488		
	0x48c		
	0x490		
	0x494		
	0x498		
	0x49c		

Question 5. [8 MARKS]

Part (a) [5 MARKS]

The `sum_ith_integers` program takes an integer `i` and one or more binary file names as arguments. Each binary file contains a sequence of integers, and the program prints to `stdout` the sum of all the i^{th} integers in each file.

For example, if you have files that contain binary representations for these numbers:

```
   file1                    file2                    file3
-----------              -----------              ------------
1 2 3 4 5 6              3 -2 400 384             8 21 8 0 ...
```

`sum_ith_integers 0 file1 file2 file3` would print 12. (12 is the sum of 1, 3, and 8, the integers at position 0 of `file1`, `file2`, and `file3`, respectively.) `sum_ith_integers 2 file1 file2` would print 403.

You may assume that the user calls the program with valid arguments, the each file exists, is readable and contains enough integers, and that the files were created on the same machine as the one that will run your program. You should not assume that integers are 4 bytes long.

Write C code to implement the `sum_ith_integers` program as specified above.

```
int main(int argc, char **argv) {
    if (argc < 3) {
        fprintf(stderr,"Usage: sum_ith_integers i filename[s]\n");
        exit(1);
    }
    int sum = 0;
    int i;
    int offset = strtol(argv[1], NULL, 10);

    for (i = 2; i < argc; i++) {
        FILE *fp = fopen(argv[i], "rb");  // "r" is fine too, b is optional
        fseek(fp, offset * sizeof(int), SEEK_SET);

        int val;
        fread(&val, sizeof(int), 1, fp);
        sum += val;
    }

    printf("%d\n", sum);
    return 0;
}
```

Part (b) [2 MARKS]

Why did we need the assumption that the files were created on the same machine as the one running the program? Give two reasons.

The following may differ between machines:

- sizeof(int)
- endianness

Part (c) [1 MARK]

The program prints the result, but could have been designed to return the result instead. What is the advantage of printing rather than returning the result?

If the result is printed, the return value (exit code) can be used to indicate whether the program terminated successfuly or due to some sort of error.

Question 6. [9 MARKS]

Consider the following struct definition:

```
struct town {
  int population;
  char name[MAXNAME+1];
};
```

Part (a) [4 MARKS]

Write a function `grow` that will triple the population of a town represented by a `struct town` and append "-city" to the name. If this would cause the population to exceed the value stored in the pre-defined constant `MAXINT`, set the population to `MAXINT`. If there is not enough room for the new name, do not change it at all. If you encounter only one of these problems return 1, both of these problems return 2, and otherwise return 0.

```
int grow(struct town *t) {
    int result = 0;
    if (t->population * 3 > MAXINT) {
        t->population = MAXINT;
        result = 1;
    } else {
        t->population *= 3;
    }
    if (strlen(t->name) + 5 <= MAXNAME) {
        strcat(t->name, "-city");
    } else {
        result = 1;
    }
    return result;
}
```

Part (b) [5 MARKS]

Write the main function for a program that will read a town name from the first command-line argument, a population from the second argument, create a struct to represent this town and then call `grow` on the struct. Do not use any dynamically-allocated memory. If the original name provided by the user is too long, just truncate it to take what will fit safely in the struct. You may assume that the population provided by the user is less than `MAXINT`.

You may assume that the user enters exactly two arguments and that the second one is an integer.

```
int main(int argc, char **argv) {

    struct town t;

    // ok to have MAXNAME + 1 here, just gets overwritten below
    strncpy(t.name, argv[1], MAXNAME);

    t.name[MAXNAME] = '\0';

    t.population = strtol(argv[2], NULL, 0);

    // also fine to just have grow(&t); and not save return value
    int result = grow(&t);

    return 0;
}
```

Question 7. [5 MARKS]

Consider the following code:

```
int main(int argc, char **argv) {
    int i;
    int pid;
    for (i = 1; i < argc; i++) {
        pid = fork();

        if (pid == 0) {
            int n = 0;
            int j;
            for (j = 0; j < strlen(argv[i]); j++) {
                if (argv[i][j] == 'a') {
                    n++;
                }
            }
            exit(n);
        }
    }

    for (i = 1; i < argc; i++) {
        int status;
        wait(&status);
        if (WIFEXITED(status)) {
            printf("%d ", WEXITSTATUS(status));
        }
    }
    printf("***\n");
    return 0;
}
```

Part (a) [1 MARK]

Briefly describe what the program does.

Prints the number of times the character 'a' appears in each commandline argument, followed by ***.

Part (b) [1 MARK]

List all possible outputs generated by the program if it is compiled as `do_something` and run like so:
`./do_something orange blue aardvark`

1 0 3 ***
1 3 0 ***
0 1 3 ***
0 3 1 ***
3 1 0 ***
3 0 1 ***

Part (c) [3 MARKS]

Make changes to this copy of the original program so that the same number of processes are forked, but the order of the output is guaranteed to match the order of the input. Also, change it so that the output now goes to `stderr` instead of `stdout`.

```
int main(int argc, char **argv) {
    int i;
    int pid;

    for (i = 1; i < argc; i++) {
        pid = fork();

        if (pid == 0) {
            int n = 0;
            int j;
            for (j = 0; j < strlen(argv[i]); j++) {
                if (argv[i][j] == 'a') {
                    n++;
                }
            }

            exit(n);
        } else if (pid > 0) {
            int status;
            wait(&status);
            if (WIFEXITED(status)) {
                fprintf(stderr, "%d\n", WEXITSTATUS(status));
            }
        }
    }

    fprintf(stderr, "***\n");
    return 0;
}
```

Question 8. [9 MARKS]

Write a program in which the parent process forks a single child, and there is a pipe set up between the two processes that the child will use to write to the parent.

The child process should have a `SIGUSR1` signal handler that writes the integer 1 into the pipe whenever `SIGUSR1` is received.

The parent process should keep track of the number of times 1 has been read from the pipe. Whenever it reads a 1, the parent prints the updated count to standard output.

The program should continue to run indefinitely. You do not need to perform error checking for system calls, but you should close unneeded pipe ends.

Part (a) [8 MARKS]

Write the program as described above.

```
int fd[2];
int write_val = 1;

void send_one(int signal) {
    write(fd[1], &write_val, sizeof(int));
}

int main() {
    pipe(fd);
    int pid = fork();

    if (pid == 0) {
        close(fd[0]);

        struct sigaction sa;
        sa.sa_handler = send_one;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);
        sigaction(SIGUSR1, &sa, NULL);

        while (1);
    } else if (pid > 0) {
        close(fd[1]);

        int count = 0;

        while (1) {
            // Read from the pipe:
            int num = 0;
            int num_read = read(fd[0], &num, sizeof(int));

            if (num_read == sizeof(int) && num == 1) {
                count++;
                printf("%d\n", count);
            }
        }
    }
    return 0;
}
```

Part (b) [1 MARK]

Assume the program has begun execution, and the child's pid is 21483. Write a shell command to send the `SIGUSR1` signal to the child process.

```
kill -USR1 21483
```

Question 9. [5 MARKS]

Consider the man page of an imaginary system call `office_hours`. Type `ps_set` is a defined type, similar to type `fd_set` that you used with `select`.

```
OFFICE_HOURS(2)             BSD System Calls Manual            OFFICE_HOURS(2)

NAME
     PS_CLR, PS_ISSET, PS_SET, PS_ZERO, office_hours

SYNOPSIS

     void PS_CLR(ps, ps_set *psset);

     int PS_ISSET(ps, ps_set *psset);

     void PS_SET(ps, ps_set *psset);

     void PS_ZERO(ps_set *psset);

     int officehours(ps_set *prof, struct timeval window);

DESCRIPTION
     office_hours() examines the schedules for the professors in the
professor set prof to see which have office hours scheduled within the
given window from the current time.  office_hours() replaces the
professor set with the subset of professors who have office hours
in the given window.

RETURN VALUE
     office_hours() returns the number of professors from the ps_set
who have office hours in the window, or -1 if an error occurs.
If office_hours() returns with an error, the descriptor sets will be
unmodified and the global variable errno will be set to indicate the error.
```

Suppose that (like file descriptors), professors are represented by small integers and there are professors defined as follows:

```
#define MICHELLE 1
#define ANDREW 2
#define JEN 3
#define ALAN 4
...   (there are more)
```

You are only interested in office hours held by Jen or Michelle in the next 5 hours. Finish the program on the next page, so that it calls `office_hours` and then prints either the message `"Jen has office hours"` or the message `"Michelle has office hours"` or both messages as appropriate.

Demonstrate that you know how to properly check for errors on a system call by writing the code to give the conventional behaviour if `office_hours` fails.

```
int main() {

    struct timeval window;
    window.interval = 5 * 60;
```

Jen

Jen &M

M

0 10

```
    // set up first argument to office_hours
    ps_set profs;
    PS_ZERO(&profs);
    PS_SET(MICHELLE, &profs);
    PS_SET(JEN, &profs);

    // call office_hours
    if (office_hours(&profs, window) == -1) {
        perror("office_hours");
        exit(1);
    }

    // print the appropriate message
    if (PS_ISSET(JEN, &profs)) {
        printf("Jen has office hours\n");
    }
    if (PS_ISSET(MICHELLE, &profs)) {
        printf("Michelle has office hours\n");
    }
    return 0;
}
```

Question 10. [9 MARKS]

Part (a) [7 MARKS]

Suppose that students in your course have been submitting an executable named `myprog.c` and for each student, the path to each student's file is `/209repos/<UTORID>/myprog.c` where `<UTORID>` is replaced by the student's UTORID.

Your current directory contains the following 200 files: `test1.in` ... `test100.in` and `expected1.out` ... `expected100.out` where `expected*.out` is the output that should be printed to stdout from a correct implementation of `myprog.c` when called with redirected input from `test*.in`. The current directory contains other files, but none that end in `.out` or `.in`.

On the next page, write a shell program that takes a single UTORID as a command-line argument and does the following:

- creates a new subdirectory named the same as the UTORID from the command-line argument
- changes into that subdirectory
- copies the 100 test input files, the 100 expected output files, and the student's submission into that subdirectory
- attempts to compile the student's code into an executable named `myprog`
- if the compilation is successful:
  - awards the student 5 marks for compiling successfully
  - runs the student's submission on each of the 100 tests producing `actual*.out`.
  - awards 1 additional mark for each test where the student's actual output is identical to the expected output.
  - prints to stdout, the UTORID and the total marks the student earned
- if the student is not successful:
  - prints to stdout, the UTORID and a message that the code did not compile
- changes back to the original directory

Your program should not print anything other than what is listed here. In particular, it should not print compilation messages or any other output about individual files matching or not. Note that `gcc` returns 0 when compilation is successful and a non-zero value otherwise. Complete Part (a) on the next page.

Part (b) [1 MARK]

Suppose your program from Part (a) is called `myscript.sh`. Give the command to set its permissions so that anyone can read it or run it but nobody can change it.

```
chmod myscript 555  (or many variations)
```

Part (c) [1 MARK]

Suppose your script has been run on many UTORIDs and the current directory now contains many subdirectories with files. Give a single shell command that from your current directory will set the permissions on all the student output files (those called `actual*.out`) so that nobody can execute or change any of the files, but TAs (who are members of the group) can copy the files. Accounts that are not members of the group should not be able to copy the files. No directory permissions need to be changed.

```
chmod 440  */actual*.out OR   chmod 040  */actual*.out
```

```
utorid=$1
mkdir $utorid
cd $utorid
cp ../*.in .
cp ../*.out .
cp /209/repos/$utorid/myprog.c

# other options for redirection to /dev/null
if gcc -Wall -std=gnu99 -o myprog myprog.c > /dev/null 2 /dev/null
then

points=5
for i in `seq 1 100`
do
    ./myprog < test$i.in > actual$i.out
    if ! diff actual$i.out expected$i.out >/dev/null
    then
        points=`expr $points + 1`
    fi
done
echo $utorid $points
else
echo $utorid did not compile
fi
cd ..
```

Write your script for Part (a) here.