CSC 209H1 Y 2015 Midterm Test
Duration — 50 minutes
Aids allowed: none

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

Last Name: _____ **First Name:** _____

Lecture Section: L5101 **Instructor: McCormick**

---

*Do **not** turn this page until you have received the signal to start.*
(Please fill out the identification section above, **write your name on the back of the test**, and read the instructions below.)
Good Luck!

This midterm consists of 5 questions on 8 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.* Comments are not required, although they may help us mark your answers. They may also get you part marks if you can't figure out how to write the code. Answers that contain both correct and incorrect or irrelevant statements will not get full marks.
If you use any space for rough work, indicate clearly what you want marked.

1: _____/ 4
2: _____/ 2
3: _____/ 7
4: _____/ 3
5: _____/ 9

TOTAL: _____/25

SOLUTIONS

LEC 5101

Question 1. [4 MARKS]

Part (a) [1 MARK]

Assume that a file named `README.txt` exists in the current working directory. Using a pipe, give a shell command that will output the number of words in this file:

Solution:

```
cat | wc -w README.txt
```

Part (b) [1 MARK]

Here is the output of running `hexdump -C input.txt` from the shell:

```
00000000  54 75 65 73 64 61 79 20  4a 75 6e 65 20 32 33 0a  |Tuesday June 23.|
00000010  49 20 62 65 6c 69 65 76  65 20 69 6e 20 79 6f 75  |CSC209H1Y Summer|
00000020  0a                                                |.|
00000021
```

What will the output of `wc -l input.txt` be?

Solution:

```
2 input.tx
```

Part (c) [1 MARK]

What is the effect of running the following piece of code?

```
char *s = "UofT";
s[0] = 'V';
printf("%s\n", s);
```

Solution: Program crashes because string literal arrays are read-only.

Part (d) [1 MARK]

What kind of error message are you likely to see if you declare a global variable inside of a header file?

Solution: Multiple definitions / symbol redefinition

Question 2. [2 MARKS]

Assume the current working directory contains three files: `Makefile`, `wc209.c` and `untar.c`. The contents of `Makefile` are as follows:

```
all: untar wc209


wc209: wc209.c
    gcc -Wall $< -o $@


untar: untar.c
    gcc -Wall -g -o $@ $^
```

Part (a) [1 MARK]

Give the exact action commands in the order that they are executed when you run `make` without any arguments.

Solution:

```
gcc -Wall -g -o untar untar.c
gcc -Wall wc209.c -o wc209
```

Part (b) [1 MARK]

Assume that there exists an executable shell script named `runalltests.sh` in the CWD. Modify the `Makefile` above so that, once the individual executables are built, this script will be executed.

Solution: Change the first rule to the following:

```
all: untar wc209
    ./runalltests.sh
```

Question 3. [7 MARKS]

For each of the subquestions below, fill in the box with an appropriate prototype declaration for the mystery function such that the code will compile without error. The subquestions are independent from one another.

Part (a) [2 MARKS]

```
double **matrix;
int *column;
double sum;
// Assume these variables are appropriately initialized.

mystery1(matrix[0], column, &sum);
```

Solution: Note that *any* return type would be accepted since the value is discarded anyways.

```
void mystery1(double *, int *, double *);
```

Part (b) [2 MARKS]

```
char *cats[] = { "Chelsea", "Buster Brown" };
int weights[] = { 10, 15 };
char vet[100];

strncpy(vet, mystery2(cats[1], weights[1]), sizeof (vet));
```

Solution:

```
char *mystery2(char *, int);
```

Part (c) [3 MARKS]

```
short *p;
void **rest;
// Assume these variables are appropriately initialized.

p = mystery3(rest + *p, &rest, *rest);
```

Solution:

```
short *mystery3(void **, void ***, void *);
```

Question 4. [3 MARKS]

Consider the following piece of code:

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

int main()
{
    union U {
        char full[7];
        struct S {
            char department[3];
            char code[3];
        } part;
    } course;

    strncpy(course.full, "CSC209", sizeof (course.full));


    if (strcmp(course.part.department, "CSC") == 0) {
        printf("Welcome to Computer Science!\n");
    }

    if (strcmp(course.part.code, "209") == 0) {
        printf("Welcome to 209!\n");
    }

    return 0;
}
```

Part (a) [1 MARK]

When the original author wrote this code, they were initially puzzled to see that the only output it produced was the message `Welcome to 209!`. Why did the other `printf` statement not get executed?

Solution: Because of the `union` type definition for `course`, `course.part.department` shares the same 3 initial `char` (byte) values with `course.full` because of their coinciding memory addresses.

Despite being an array of size 3, `course.part.department[3]` does not contain a NUL/zero terminator as a proper C style string *must*, but rather the `char` value `'2'` from the initialization of `course.full`. Thus the `strcmp` comparison is in fact comparing the C style strings `CSC209` and `CSC`, which are not equal, thus it returns a non-0 (not equal) value, and the first `printf` is never executed

Part (b) [2 MARKS]

Fix the above code so that the message `Welcome to Computer Science!` is correctly printed.

Solution: Change the first `strcmp` to a length limited `strncmp`:

```
    if (strncmp(course.part.department, "CSC", sizeof (course.part.department)) == 0) {
```

Question 5. [9 MARKS]

On the next page are questions that deal with the following piece of code:

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

// The definitions for these are not given here...
void cleanup_dupv(char **dupv);
int check_dupv(int argc, char **argv, char **dupv);

char **duplicate_argv(int argc, char **argv)
{
    // A)
    char **dupv = (char **) malloc(                                    );

    int i;
    for (i = 0; i < argc; i++) {
        // B)


    }

    // Last entry will be NULL to signify the end
    dupv[argc] = NULL;

    return dupv;
}

int main(int argc, char *argv[])
{
    if (!check_dupv(argc, argv, argv)) {
        printf("check_dupv is broken.");
        return -1;
    }

    char **dupv = duplicate_argv(argc, argv);

    if (dupv[argc] != NULL || !check_dupv(argc, argv, dupv)) {
        printf("duplicate_argv is broken.");
        return -1;
    }

    cleanup_dupv(dupv);

    return 0;
}
```

Explanation: This code is intended to create a duplicate copy of `argv`, the array through which program arguments are passed into the `main` function. Instead of using an explicit array length like `argc`, however, the end of a `dupv` array is indicated by one extra `char *` entry that is set to the `NULL` pointer.

Part (a) [2 MARKS]

Fill in the argument of the `malloc` call after comment `A` in the code.

Solution: Allocate an array with enough space to store `argc + 1` values of type `char *` (the +1 is for the final NULL array terminating value.)

```
char **dupv = (char **) malloc((argc + 1) * sizeof (char *));
```

Part (b) [2 MARKS]

Provide the body of the `for` loop (comment `B`) to fill `dupv` with copies of the elements of `argv`.

Solution: Create a duplicate copy of each string in `argv`:

```
dupv[i] = strdup(argv[i]);
```

A slightly more involved but equivalent solution would involve an explicit `malloc` and `strncpy` of the correct sizes:

```
size_t len = strlen(argv[i]);
dupv[i] = (char *) malloc(len + 1);
strncpy(dupv[i], argv[i], len + 1);
```

Part (c) [3 MARKS]

Provide an implementation of the function `cleanup_dupv` which releases all the dynamically allocated memory associated with a `dupv` array.

Solution:

```
void cleanup_dupv(char **dupv)
{
    int i = 0;
    while (dupv[i] != NULL) {
        // Release memory allocated by each ‘strdup‘
        free(dupv[i]);
        i++;
    }

    // Release the memory for the array of pointers itself
    free(dupv);
}
```

An alternate solution would be to iterate using a pointer instead of an index variable:

```
void cleanup_dupv(char **dupv)
{
    char **cur = dupv;
    // The loop condition ‘while (*cur) {‘ would also work to check for a non-NULL value

    while (*cur != NULL) {
        free(*cur);
        cur++;
    }

    free(dupv);
}
```

Part (d) [2 MARKS]

As a sanity check, write a function `check_dupv` which compares a `dupv` array against the original `argv`. Return a *false* value if any of the corresponding elements of the two arrays differ, and a *truth* value otherwise.

Solution: In C, the *false* value is represented by the integer `0` while any non-zero value is consider *true* for the purposes of conditional expressions.

```
int check_dupv(int argc, char **argv, char **dupv)
{
    int i;
    for (i = 0; i < argc; i++) {
        // Is dupv shorter in length than argv?
        if (dupv[i] == NULL) {
            return 0;
        }

        // Compare corresponding elements for string equality
        if (strcmp(argv[i], dupv[i]) != 0) {
            return 0;
        }
    }

    // Is dupv longer in length?
    if (dupv[argc] != NULL) {
        return 0;
    }

    return 1;
}
```

Full marks were awarded even if the extra `dupv` checks were missing.