### UNIVERSITY OF TORONTO MISSISSAUGA JULY 2022 MIDTERM EXAMINATION

### CSC209 Software Tools and Systems Programming Instructors: K. Abbasi Duration: 100 minutes Examination Aids: One piece of double-sided A4 paper

Student Number:	
UtorID:	
Family Name(s):	
Given Name(s):	

The University of Toronto Mississauga and you, as a student, share a commitment to academic integrity. You are reminded that you may be charged with an academic offence for possessing any unauthorized aids during the writing of an exam. Examples are including but not limited to: cell phones, smart watches, smart glasses, tablets, laptops, desktops, calculators of any kind, headsets or earphones (visible or invisible), microphones, walkie talkies, GPS trackers, radars, gaming consoles, electronic readers, electronic pens, wireless erasers, laser or infrared sensors, cameras, carbon papers, and flashlights. Please turn off all devices, seal them in your backpack, and place the it under your desk for the duration of the examination. You will not be able to touch the bag or its contents until the exam is over.

If, during an exam, any of these items are found on your person or in the area of your desk other than in the clear, sealable, plastic bag, you may be charged with an academic offence. A typical penalty for an academic offence may cause you to fail the course.

Please note, once this exam has begun, you **CANNOT** re-write it.

# Do **not** turn this page until you have received the signal to start. In the meantime, please read the instructions below carefully.

This paper consists of 5 questions on 17 pages (including this one). To obtain the full marks, you need to get at least 100 points out of 114 from this even	
fun marks, you need to get at least 100 points out of 114 from this exam.	# 1:/ 20
Write the answer to each question in the space immediately follows the question. If you need more space, use the blank pages at the end of the booklet. However,	# 2:/ 28
write a <b>clear note</b> at the end of the current space.	# 3:/ 24
Error checking is required throughout the test.	# 4:/ 32
In <b>all questions</b> , assume that include statements are already provided.	# 5:/ 10
You may assume the following type sizes throughout the exam: char, bool: 1B; int, float: 4B; long long, double and pointer: 8B, (B stands for bytes)	TOTAL:/114

# Question 1. Concepts [20 MARKS]

At each part, you are given a statement about concepts in C programming. Indicate if each statement is true or false. In either cases, provide a **very brief** explanation about your choice. Do not write detailed or verbose answers.

### Part (a) [2 MARKS]

The gcc compiler translates C code into the assembly language, which is executable by the operating system.

Part (b) [2 MARKS]

stdio.h is a file written in C, and the .h suffix in its name does not matter to the gcc compiler.

Part (c) [2 MARKS]

A string is not the same as an array of characters.

Part (d) [2 MARKS]

The statement  $1 \le a \le 3$  evaluates to true if a = 10.

Part (e) [2 MARKS]

The space allocated as the result of malloc is monitored by the garbage collector, so it will be reclaimed once it is not referenced anymore.

Part (f) [2 MARKS]

It is impossible to change the content of a variable that is defined with the **const** modifier.

Part (g) [2 MARKS]

All types of variables are passed to functions using the call by value method.

Part (h) [2 MARKS]

A struct cannot have an attribute of its own type.

Part (i) [2 MARKS]

The strlen function is unsafe because the input can point to an invalid address to which the program does not have access.

Part (j) [2 MARKS]

The **sizeof** operator of a dynamically allocated array returns the total number of allocated bytes, not the total number of elements in the array.

# Question 2. Code execution [28 MARKS]

At each part, you are asked to determine what happens when the given code is executed. There are three possible cases: a **compilation error**, an **undefined behavior** potentially leading to a crash or unknown outcomes, or a **deterministic output**. First, determine the appropriate case. In the first two cases, explain the issue that led to that choice. If you chose the final case, explain what is printed out to the standard output. That is, if you believe the code does not have any issues, there is no need to explain why you think the code is okay; just describe the output.

**Note**: In all parts, assume that the include statements are provided, so do not point out to them as compilation errors.

**Note**: a deterministic output does **not** mean that the output is the same in all scenarios. Rather, it means that it can be **determined** from a certain set of criteria and does not depend on the internal implementation of the compiler or the operating system.

```
Part (a) [3 MARKS]
int f(int *a){
    (*a)++;
}
int main() {
    int a = 1;
    int *b = &a;
    f(b);
    printf("%d\n", a);
}
Part (b) [3 MARKS]
int main() {
    int a = 21;
    const long long b = -2;
    printf("%d\n", a & b);
}
```

```
July 2022
```

```
Part (c) [3 MARKS]
int main() {
    char **a = 200;
    printf("%llu\n", &a[1]);
}
```

```
Part (d) [3 MARKS]
int main() {
    int a = 1, b = 2, c[10] = {}, d = 4;
    printf("%d\n", c[-1] + c[10]);
}
```

```
Part (e) [4 MARKS]
typedef struct linked {
    struct linked *next;
    int val;
} linked;
int main(){
    linked *node = malloc(sizeof(linked)), *head;
    node->val = 1;
    node->next->val = 2;
    node->next->next = NULL;
    head = node;
    while (head){
        printf("%d ", head->val);
        head = head->next;
    }
```

}

```
Part (f) [4 MARKS]
void strcpy2(char *dst, const char *src){
    dst = malloc((strlen(src) + 1) * sizeof (const char));
    if (dst == NULL){
        printf("Was not able to allocate memory...aborting\n");
        return;
    }
    while (*src){
        *(dst++) = *(src++);
    }
    *dst = 0;
}
int main() {
    char s[] = "hello", t[20];
    strcpy2(t, s);
    printf("%s\n", t);
}
```

# Part (g) [4 MARKS]

Assume that the computer is little-endian, meaning the least significant byte of an integer is stored at the smallest memory address.

```
int main(){
    long long a = (1 << 8) + (1 << 16);
    char *tmp = &a;
    for (int i=0; (tmp + i) < &a + 1; i++){
        printf("%d", tmp[i]);
    }
}</pre>
```

**CSC209** 

```
Part (h) [+4 BONUS MARK]
int *fill(int a[], int x){
    for (int i = 0; i < sizeof(a); i++){</pre>
        a[i] += x;
    }
    return a;
}
int main (){
    int arr[10] = {};
    for (int i = 0; i < sizeof(arr); i++){</pre>
        if (arr[i] = i){
            fill(arr + i, i + 1);
        }
        else
            break;
    }
    for (int i=0; i < 10; i++){</pre>
        printf("%d ", arr[i]);
    }
}
```

# Question 3. Troubleshooting [24 MARKS]

A student was asked to simulate python's string replace method in C. That is, a function that inputs three constant strings *orig*, *old*, *new*, and returns a new string in which all non-overlapping occurrences of the non-empty string *old* in *orig* are replaced with *new*. For example, replace("hello", "llo", "H") should return "heH", and replace("AAAA", "AAA", "bye") should return "byeA".

The student's code can be find on the next page. The code has at least 8 issues, and you are asked to find them out. For each issue, explain the reason you believe it is a problem, as well as the steps to fix it. Note that *problems* are considered issues that could lead to compilation errors, undefined behaviors, memory leaks, or incorrect results. In this question, we do **not** consider issues that merely result in a compiler warning.

Note: If you list more than 8 issues, only the first 8 will be graded.

Note: The man pages of strstr and strncat can are annexed to the end of this booklet.

**Note**: The entire code is printed on the next page, so you could inspect it all at once with reduced pageturning and confusion. List all issues in the space below (current page) by referring to their line numbers. Do **not** make in-place annotations to the code as they will **not be graded**.

```
char *replace(const char *orig, const char *old, const char *new){
1
            const char *cpy = orig;
2
3
            // finding no. of occurrences of old in orig
4
            int count;
5
            while (*cpy){
6
                // find the next occurrence and move the pointer there
7
                cpy = strstr(cpy, old);
8
9
                if (!cpy)
10
                     break;
11
                count++;
12
            }
13
14
            // calculating the length of the output string
15
            int new_len = strlen(orig) + (strlen(new) - strlen(old)) * count;
16
            char *out = malloc(new_len * sizeof(char));
17
18
            // error handling
19
            if (&out == NULL){
20
                fprintf(stderr, "Failed to allocate memory; aborting...");
21
                return NULL;
22
            }
23
24
            while (true){
25
                // replace the occurrences one by one
26
                char *occ = strstr(orig, old);
27
28
                // there is no more occurrence
29
                if (occ == NULL)
30
                    break;
31
32
                // the length that must be copied from orig
33
                int segment_len = occ - orig + 1;
34
                strncat(out, orig, segment_len);
35
                strncat(out, new, strlen(new));
36
37
                // advancing orig to immediately after the occurrence
38
                orig = occ + 1;
39
            }
40
41
            free(out);
42
            return out;
43
       }
44
```

# Question 4. Design and Program [32 MARKS]

You are asked to implement a C program that stores simplified records of CS students at UTM. Each *student* has the following information:

- UTorID, a string whose length is at most 10 characters
- Year of study (non-negative integer)
- Number of courses passed (non-negative integer)
- An array of *course* objects as the list of passed course

A course object has the following fields:

- Course code, a string whose length is at most 8 characters
- Course name, a string of unknown size
- Year offered (non-negative integer)
- Final grade of the corresponding student at the course (integer)

### Part (a) [8 MARKS]

Use the space below on define two structs **student** and **course** that fulfills the above requirements.

Part (b) [6 MARKS]

On the next page, complete the implementation of the initialization function. It inputs a pointer to a student object, UTorID, and year of study, and populates the given student object with the inputs. Initially, the number of passed courses should be zero, hence the list of course objects must be set to NULL. Assume that the given utorid is a valid input. void initialize (student \*st, const char \*utorid, int year){

### }

Part (c) [6 MARKS]

Complete the implementation of the following print function. It should print the student's UTorID and year of study on the first line, and then print each passed course in a separate line. For each course, print the course code, year, and the student's final grade.

void print (const student \*st){

# }

### Part (d) [12 MARKS]

Complete the implementation of the add\_course function. It takes a pointer to student object, as well as all the relevant info of the new course. You should create a new course object out of the given inputs and append it to the list of passed courses of the student. Assume all inputs are valid.

}

Bonus Bonus

Question 5. [+10 Bonus Mark]

Part (a) [4 MARKS]

What is the difference between the following two lines? Explain it in terms of what is done behind the scenes (by the compiler or the CPU).

```
char *s = "hello";
char t[] = "hello";
```

Part (b) [6 MARKS]

Assume that **person** is a struct that has a **name** field of type **char** \*. What is wrong with the following approach to setting the name to John?

struct person p; p.name = "John"; [Use the space below for rough work. This page will **not** be marked, unless you clearly indicate the part of your work that you want us to mark.]

[Use the space below for rough work. This page will **not** be marked, unless you clearly indicate the part of your work that you want us to mark.]

### Appendix: man pages

STRSTR(3) from Linux Programmer's Manual

### NAME

strstr, strcasestr - locate a substring

### SYNOPSIS

include <string.h>

char \*strstr(const char \*haystack, const char \*needle);

define \_GNU\_SOURCE /\* See feature\_test\_macros(7) \*/

include <string.h>

char \*strcasestr(const char \*haystack, const char \*needle);

### DESCRIPTION

The strstr() function finds the first occurrence of the substring needle in the string haystack. The terminating null bytes ('') are not compared.

The strcasestr() function is like strstr(), but ignores the case of both arguments.

#### RETURN VALUE

These functions return a pointer to the beginning of the located substring, or NULL if the substring is not found.

#### ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value	
strstr()	Thread safety	MT-Safe	
strcasestr()	Thread safety	MT-Safe locale	

### CONFORMING TO

strstr(): POSIX.1-2001, POSIX.1-2008, C89, C99.

The strcasestr() function is a nonstandard extension.

#### SEE ALSO

index(3), memchr(3), memmem(3), rindex(3), strcasecmp(3), strchr(3), string(3), strpbrk(3), strsep(3), strspn(3), strtok(3), wcsstr(3)

### COLOPHON

This page is part of release 5.05 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/. STRCAT(3) from Linux Programmer's Manual

#### NAME

strcat, strncat - concatenate two strings

#### SYNOPSIS

include <string.h>

char \*strcat(char \*dest, const char \*src);

char \*strncat(char \*dest, const char \*src, size<sub>t</sub>n);

### DESCRIPTION

The strcat() function appends the src string to the dest string, overwriting the terminating null byte ('') at the end of dest, and then adds a terminating null byte. The strings may not overlap, and the dest string must have enough space for the result. If dest is not large enough, program behavior is unpredictable; buffer overruns are a favorite avenue for attacking secure programs. The strncat() function is similar, except that

\* it will use at most n bytes from src; and

\* src does not need to be null-terminated if it contains n or more bytes.

As with strcat(), the resulting string in dest is always null-terminated.

If src contains n or more bytes, strncat() writes n+1 bytes to dest (n from src plus the terminating null byte). Therefore, the size of dest must be at least strlen(dest)+n+1.

A simple implementation of strncat() might be:

```
char *
strncat(char *dest, const char *src, size_t n)
{
    size_t dest_len = strlen(dest);
    size_t i;
    for (i = 0 ; i < n src[i] != 0; i++)
        dest[dest_len + i] = src[i];
    dest[dest_len + i] = 0;
    return dest;</pre>
```

```
}
```

#### RETURN VALUE

The strcat() and strncat() functions return a pointer to the resulting string dest.

#### ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

						_
Interface		I	Attribute	I	Value	Ι
strcat(),	strncat()		Thread safety		MT-Safe	

-----

#### CONFORMING TO

POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD.

#### NOTES

Some systems (the BSDs, Solaris, and others) provide the following function:

size\_t strlcat(char \*dest, const char \*src, size\_t size);

This function appends the null-terminated string src to the string dest, copying at most size-strlen(dest)-1 from src, and adds a terminating null byte to the result, unless size is less than strlen(dest). This function fixes the buffer overrun problem of strcat(), but the caller must still handle the possibility of data loss if size is too small. The function returns the length of the string strlcat() tried to create; if the return value is greater than or equal to size, data loss occurred. If data loss matters, the caller must either check the arguments before the call, or test the function return value. strlcat() is not present in glibc and is not standardized by POSIX, but is available on Linux via the libbsd library.

#### EXAMPLE

Because strcat() and strncat() must find the null byte that terminates the string dest using a search that starts at the beginning of the string, the execution time of these functions scales according to the length of the string dest. This can be demonstrated by running the program below. (If the goal is to concatenate many strings to one target, then manually copying the bytes from each source string while maintaining a pointer to the end of the target string will provide better performance.)

Program source

```
#include <string.h>
#include <time.h>
#include <stdio.h>
int
main(int argc, char *argv[])
#define LIM 4000000
    int j;
    char p[LIM + 1]; /* +1 for terminating null byte */
    time_t base;
    base = time(NULL);
    p[0] = 0;
    for (j = 0; j < LIM; j++) {</pre>
        if ((j % 10000) == 0)
               printf("%d %ld", j, (long) (time(NULL) - base));
        strcat(p, "a");
    }
}
```

bcopy(3), memccpy(3), memcpy(3), strcpy(3), string(3), strncpy(3), wcscat(3), wcsncat(3)

SEE ALSO