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.

False. The gcc compiler translates C code into machine code.

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.

True. The gcc compiler does not care about file extensions.

Part (c) [2 MARKS]

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

True. Strings have to contain the null terminator character, while arrays of characters do not.

Part (d) [2 MARKS]

The statement `1 <= a <= 3` evaluates to true if `a = 10`.

True. This expression is equivalent to `((1 <= a) <= 3)`. First, the left-most inequality is evaluated (`1 <= a`), which is true or 1. Then, `1 <= 3` is evaluated, which is true.

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.

False. There is no garbage collector in C.

Part (f) [2 MARKS]

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

False. A pointer to this variable can be defined and cast to the pointer of the non-const type. Then, you can dereference the second pointer and modify its content.

Part (g) [2 MARKS]

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

False. Arrays are passed using the call by reference method.

Part (h) [2 MARKS]

A `struct` cannot have an attribute of its own type.

True. Otherwise, it would be impossible for the compiler to calculate the sizeof of that struct..

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.

False. strlen is unsafe because the string can be malformed (without a null terminator), which makes it loop indefinitely until it breaches the memory bounds of the process.

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.

False. The `sizeof` operator of a dynamically allocated array returns 8.

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);
}
```

Prints 2. The function takes a pointer to the original variable and increments it with dereferencing the pointer.

Part (b) [3 MARKS]

```
int main() {
    int a = 21;
    const long long b = -2;
    printf("%d\n", a & b);
}
```

Prints 20. The 2's compliment representation of -2 is 11111111111111111111111111111110, and 21 is 10101. So the bitwise and returns 10100.

Part (c) [3 MARKS]

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

Prints 208. `a[1]` is the same as `a + 1`. Pointers advance according to the size of the variable they point to, which is `char *` in this case. (Note: since we never dereferenced this pointer, it did not matter that the address is invalid)

Part (d) [3 MARKS]

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

Undefined behavior: We cannot know that what is stored immediately before or after the array.

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;
    }
}
```

Undefined behavior: `node->next` is uninitialized, so accessing `node->next->val` entails dereferencing an invalid pointer, which leads to undefined behavior.

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);
}
```

Undefined behavior. The `strcpy2` function is implemented incorrectly. The results of `malloc` is stored in the local variable `dst` which is gone after the function returns. In other words, it ignores the original space allocated for `dst`, so the array `t` from the main function remains untouched.

Since `t` is uninitialized, it is filled with garbage. Therefore, the print function prints all the bytes starting from the beginning of `t` until it reaches the null terminator, which is very well an undefined behavior as it may go beyond the original boundaries of the array.

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]);
    }
}
```

Prints 01100000. The second and third bytes of `a` are 1, and the rest are zero.

Part (h) [+4 Bonus mark]

```
int *fill(int a[], int x){
    for (int i = 0; i < sizeof(a); i++){
        a[i] += x;
    }
    return a;
}

int main (){
    int arr[10] = {};
    for (int i = 0; i < sizeof(arr); i++){
        if (arr[i] = i){
            fill(arr + i, i + 1);
        }
        else
            break;
    }
    for (int i=0; i < 10; i++){
        printf("%d ", arr[i]);
    }
}
```

Prints 0 0 0 0 0 0 0 0 0 0. The if condition is an assignment (`arr[i] = i` instead of `arr[i] == i`). The assignment operator returns the assigned value.

Therefore, at the first iteration, it assigns zero to `arr[0]` and returns zero as the output. Since zero is considered `false`, it goes through the `else` block and breaks from the for loop.

That is, the `fill` function never gets called, and the initial values of `arr` will be printed (which are all zero).

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 page-turning 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**.

- Line 4: the variable `count` is uninitialized. Should be `int count = 0;`.
- Line 8: The while loop never ends because `cpy` will point to the first occurrence forever. A new line should be added after the if statement (between lines 11 and 12) with the following statement: `cpy += strlen(old);`
- Line 17: the space for the null terminator is not allocated. Should be `(new_len + 1) * sizeof(char)`.
- Line 20: checking the output of `malloc` is incorrect. It should be `if (out == NULL)` instead. (In fact, `out` is a stack variable and its address will never be `NULL`).
- Line 24: `out` should have the null terminator, or it is not a C string (even if it is meant to be an empty string). Otherwise, all subsequent calls to `strcat` will lead to undefined behavior. A new line should be added there: `out[0] = 0;`
- Line 30: the last segment of the original string (after the last occurrence) is not copied to `out`. Inside the if statement, a new statement should be added: `strncat(out, orig, strlen(orig));`
- Line 34: segment length calculation is wrong. It copies the first character of old every time. Should be `int segment_len = occ - orig;`.
- Line 39: `orig` should advance to the end of the occurrence. Otherwise, it would replace overlapping occurrences as well. This line should be `orig = occ + strlen(old);`
- Line 42: freeing `out` makes it inaccessible for all future users. This line should be deleted altogether, and the caller of this function is responsible for reclaiming the memory once it no longer uses it.

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

```
typedef struct {
    char code[9];
    char *name;
    unsigned int year;
    unsigned int grade;
} course;

typedef struct {
    char utorid[11];
    unsigned int year;
    unsigned int course_count;
    course *course_list;
} student;
```

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){
    strcpy(st->utorid, utorid);
    st->year = year;
    st->course_count = 0;
    st->course_list = NULL;
}
```

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){
    printf("%s %u\n", st->utorid, st->year);
    for (int i=0; i<st->course_count; i++){
        const course *c = &st->course_list[i];
        printf("%s, %u, %u\n", c->code, c->year, c->grade);
    }
}
```

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.

```
void add_course (student *st, const char *course_code, const char *course_name,
                 unsigned int course_year, unsigned int course_grade){

    // initializing the course object
    course new_course = {.year = course_year, .grade = course_grade};
    strcpy(new_course->code, course_code);
    new_course.name = malloc((strlen(course_name) + 1) * sizeof(char));
    strcpy(new_course.name, course_name);

    // appending it to the array
    st->course_count++;
    st->course_list = relloc(st->course_count * sizeof(course));
    st->course_list[st->course_count - 1] = course;
}
```

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";
```

The first line allocates the string (6 characters) in a data segment of the program's memory, and stores a pointer to its first character in the stack variable `s`. However, the second line allocates the entire array (6 characters) in stack, and no additional pointer is created anywhere.

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";
```

String literals are created in the data segment of the process's memory, which is a read-only segment. That is, any attempt to mutate it (e.g., `p.name[0] = 'a'`) will result in a run-time error.

See https://en.wikipedia.org/wiki/Data_segment for more information.

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$_t$ 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;
}
```

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	Attribute	Value
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++) {
                     if ((j % 10000) == 0)
                            printf("%d %ld", j, (long) (time(NULL) - base));
                     strcat(p, "a");
                 }
             }

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