

1. **(34 points) Quick Hits.** Please read the questions carefully. For each question, if you answer the question correctly, you get +2 points. If you do not answer, you get 0 points. If you answer the question incorrectly, you get -2 points.

1.1. `sizeof(9) > sizeof(1L)` is true.

- a. True
- b. False

1.2. `-sizeof(int) > 1` is true.

- a. True
- b. False

1.3. Given the following declarations, what is the value of the following expression?

```
int a = 8;
int b = 2;
int c = 4;

a + b / c * a - c / b
```

- a. 10.0
- b. 14
- c. 6
- d. 10

1.4. Suppose `i` is a double. After the statements

```
i = 7.7;
j = (int) i;
```

are executed, what is the value of `i`?

- a. 7.7
- b. 7.0
- c. 8.0
- d. 7

1.5. When the following code snippet runs, what is printed?

```
#define A 1

#ifdef A
    #if A == 0
        printf("1\n");
    #else
        printf("2\n");
    #endif
#endif
```

```
        #endif
    #else
        #if A == 0
            printf("3\n");
        #else
            printf("4\n");
        #endif
    #endif
#endif
```

- a. 1
- b. 2
- c. 3
- d. 4
- e. Error

1.6. What is the 8-bit 2's complement of 5?

- a. 11111011
- b. 00000101
- c. 11111010
- d. 10000101

1.7. Freeing memory in the hash table is faster than in linked lists.

- a. True
- b. False

1.8. What is $11100011 \ll 3$?

- a. 11100110
- b. 00011000
- c. 00011100
- d. 00011111

1.9. What is `sizeof("xyz\0")`?

- a. 3
- b. 4
- c. 5
- d. 8

1.10. What does the following represent?

```
int (*p)[10];
```

- a. Array of ten integers
- b. Array of ten integer pointers
- c. Points to pointers to ten integers
- d. Points to integer array of size ten

1.11. When the following code snippet runs, what is printed?

```
p=&i;
i=1;
*p=2;
printf("%d, %d\n", i, *p);
```

- a. 1, 1
- b. 1, 2
- c. 2, 1
- d. 2, 2

1.12. When the following code snippet runs, what is printed?

```
int i, j;
int *p = &i, *q = &j;
int **k = &p;

*p = 1;
*q = 2;
*k = q;
k = &q;
*k = &i;
*p = 3;
**k = 4;

printf("%d\n", i);
```

- a. 1
- b. 2
- c. 3
- d. 4

1.13. When the following code snippet runs, what is printed?

```
int x = 3;
int *y;
int *z;

y = &x;
z = y;
(*z)--;

printf("Result: %d\n", *y + *z);
```

- a. Result: 4
- b. Result: 5

- c. Result: 6
 - d. None of the above
- 1.14. The memory block allocated by a calloc function is always initialized to zero.
- a. True
 - b. False
- 1.15. Which of the following statements regarding the selection of a data type for a variable is FALSE?
- a. The operations that can be performed on a value are limited by its type.
 - b. The amount of memory necessary to store a value depends on its type.
 - c. How a value is stored in the memory of the computer depends on its type.
 - d. None of the above.
- 1.16. When the following program runs, what will it print?

```
#include <stdio.h>

int myFunc(int i) {
    i=1;
    return 5;
}

int main(void) {
    int i;
    i=0;
    i=10;
    i=myFunc(i);
    printf("i=%d", i);
    return 0;
}
```

- a. i=0
 - b. i=1
 - c. i=5
 - d. i=10
- 1.17. What is the value of sx after the following statements?

```
unsigned char ux = 255;
char sx = ux;
```

- a. 255
- b. -1
- c. 0

d. error

2. (15 points) Debugging

2.1. **(5 points)** The following function has a bug and doesn't work as expected. What is the issue with this function and how will you fix it?

```
/* If x is greater than y, this function should return 1.
   Else, this function returns 0. */

int is_greater(unsigned int x, unsigned int y)
{
    if (x - y > 0)
        return 1;
    else
        return 0;
}
```

2.2. **(10 points)** The following program finds the common elements in two different integer arrays (fibArray and primeArray) and stores them in another array called commonArray. At the end of the program, it prints out how many common elements there are. There are five bugs in the code. Identify them and then fix them.

```
1 int main()
2 {
3     int fibArray[] = { 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
4     int primeArray[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
5     int commonArray[];
6     int i, j;
7
8     for (i = 0; i < 10; ++i)
9     {
10        for (j = 0; j < 10; ++j)
11        {
12            if (fibArray[i] = primeArray[j])
13            {
14                commonArray[j] = primeArray[j];
15                ++n;
16            }
17        }
```

```
18 }
19
20 printf("The total number of common elements is %n\n", n);
21 return 0;
22 }
```

3. (15 points) Recursion

(a) Read the following code.

```
#include <stdio.h>

int fun(int *arr, int start, int end) {
    if (end - start == 1)
        return arr[start];

    int middle = (start + end) / 2;
    int left = fun(arr, start, middle);
    int right = fun(arr, middle, end);
    return left > right ? left : right;
}

int main() {
    int arr[] = { 4, 7, 8, -1, 2, 5 };
    int array_length = sizeof(arr) / sizeof(*arr);
    printf("ans=%d\n", fun(arr, 0, array_length));
}
```

(a-1) (3 points) What is the output of the following code?

Your answer: ans =

(a-2) (4 points) Briefly explain what the fun does

Your answer:

(b) Write a **recursive** function that prints a given string reversely.

For example, if we call `print_reverse("abcd")`, it should print "dcba". Your answer can include multiple statements.

```
#include <stdio.h>

void print_reverse(const char *str) {
    if ((1))
        return;

    (2)
}

int main() {
    print_reverse("abcd");
}
```

(b-1) **(4 points)** Fill (1)

Your answer:

(b-2) **(4 points)** Fill (2)

Your answer:

4. **(12 points)** Dynamic Memory Allocation

4.1. **(6 points)** Does the following code run successfully to return 0 or does it generate a segmentation fault? If it runs fine, then what is the output? Otherwise explain why it segfaults.

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

void populate(int *a)
{
    int *parray = malloc(2 * sizeof(int));
    parray[0] = 37;
    parray[1] = 73;
    a = parray;
}

int main()
{
    int *a = NULL;
    populate(a);
    printf("a[0] = %d and a[1] = %d\n", a[0], a[1]);
    return 0;
}
```

4.2. (6 points) Point out the problem with the following code and fix it.

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

int main()
{
    int *p = malloc(sizeof(int));
    *p = 42;
    p = malloc(sizeof(int));
    free(p);
}
```

5. (24points) Hash Tables

Here are source code files that implement a hash table for a key-value store.

table.h

```
#ifndef TABLE_H_
#define TABLE_H_
/* table.h */
typedef struct Table Table;

Table* Table_create(void);
void Table_add(Table* t, const char *key, int value);
int Table_search(Table* t, const char *key, int *value);
int Table_remove(Table* t, const char *key); // remove a node whose key matches

unsigned int hash(const char *x);
#endif
```

table.c

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

#include "table.h"

enum {BUCKET_COUNT = 1024};
```



```

struct Node {
    char *key;
    int value;
    struct Node *next;
};

struct Table {
    struct Node *array[BUCKET_COUNT];
};

unsigned int hash(const char *x) {
    int i;
    unsigned int h = 0U;
    for (i=0; x[i]!='\0'; i++)
        h = h * 65599 + (unsigned char)x[i];
    return h % BUCKET_COUNT;
}

struct Table *Table_create(void) {
    struct Table *t;
    t = (struct Table*)calloc(1, sizeof(struct Table));
    return t;
}

void Table_add(struct Table *t, const char *key, int value)
{
    struct Node *p = (struct Node*)malloc(sizeof(struct Node));
    int h = hash(key);
    p->key = malloc(strlen(key) + 1);
    strcpy(p->key, key);
    p->value = value;
    p->next = t->array[h];
    t->array[h] = p;
}

int Table_search(struct Table *t, const char *key, int *value)
{
    struct Node *p;
    int h = hash(key);
    for (p = t->array[h]; p != NULL; p = p->next)
        if (strcmp(p->key, key) == 0) {
            *value = p->value;
            return 1;
        }
    return 0;
}

```

(a) **(12 points)** Please implement the missing Table_remove() based on comments. Your answer can include multiple statements.

```
int Table_remove(struct Table *t, const char *key)
{
    struct Node *p, *prev = NULL;
    int h = hash(key);
    /* loop a linked list in the bucket */
    for ((1))
        if (strcmp(p->key, key) == 0) {
            if (prev == NULL) (2) /* if p is the first element of the list, update the bucket */
            else (3) /* update the linked list to remove p */
            /* free all memory occupied by p */
            (4)
            return 1;
        }
    return 0;
}
```

(a-1) Fill (1)

Your answer:

(a-2) Fill (2)

Your answer:

(a-3) Fill (3)

Your answer:

(a-4) Fill (4)

Your answer:

(b) **(12 points)** Here is code for a client. Assume that assertions are never triggered (i.e., conditions in assertions are true). Fill out the blanks.

client.c

```
/* client.c */
#include <string.h>
#include <stdio.h>
#include <assert.h>

#include "table.h"

int main ( ) {
    Table* t = Table_create(); // assume it's successful.
```

```

int value = 0, res = 0;

// assume they are correct
assert(hash("aaaa") != hash("bbbb"));
assert(hash("aaaa") == hash("iiii"));

char k[] = "aaaa";
Table_add(t, k, 10);

res = Table_search(t, "iiii", &value);
printf("1. res=%d, value=%d\n", res, value);

strcpy(k, "bbbb");
Table_add(t, k, 20);

res = Table_search(t, "aaaa", &value);
printf("2. res=%d, value=%d\n", res, value);

res = Table_search(t, "iiii", &value);
printf("3. res=%d, value=%d\n", res, value);
}

```

1. res= _____ , value = _____
2. res= _____ , value = _____
3. res= _____ , value = _____

6. (Bonus, 6 points) Overflow

Saturate addition is a special addition that returns a value in a fixed range between the minimum and maximum value. If the result of the addition is greater than the maximum, it returns the maximum. If the result of the addition is smaller than the minimum, it returns the minimum. Otherwise, the saturate addition should return the addition result without changing it. For example, if we implement saturate addition for the 'long' type, whose maximum is LONG_MAX and minimum is LONG_MIN, these are the expected outcomes.

```

saturate_add(1, 2) = 3
saturate_add(1, -2) = -1
saturate_add(1, -1) = 0

saturate_add(LONG_MAX, 1) = LONG_MAX
saturate_add(LONG_MAX, -1) = LONG_MAX - 1
saturate_add(LONG_MIN, -1) = LONG_MIN
saturate_add(LONG_MIN, 1) = LONG_MIN + 1
saturate_add(LONG_MIN, 1) = LONG_MIN + 1

```

```
saturate_add(LONG_MIN, LONG_MAX) = -1
```

To make the saturate addition for the 'long' type, I found this code from the internet.

```
long saturate_add(long a, long b) {  
    if (a > 0 && b > 0 && a+b <= a)  
        return LONG_MAX; // overflow (positive)  
    else if (a < 0 && b < 0 && a+b >= a)  
        return LONG_MIN; // overflow (negative)  
    else  
        return a + b;  
}
```

I found that this code behaves differently depending on the environment. In particular, it works well on my Linux machine, but it gives me incorrect results on Mac.

- (a) (2 points)** Why does it happen? Assume that compilers in different platforms have no bugs and are compliant with the C specification. This code triggers signed integer overflow, which is an undefined behavior in C. Therefore, its result could be different in different environments.

FYI, this code gives us incorrect results if we compile it with clang (a default compiler in Mac) with optimization (e.g., -O2).

Make a correct version of saturate_add

```
long saturate_add(long a, long b) {  
    if ( (1) )  
        return LONG_MAX; // overflow (positive)  
    else if ( (2) )  
        return LONG_MIN; // overflow (negative)  
    else  
        return a + b;  
}
```

- (b) (2 points)** Fill (1)

Answer:

(c) (2 points) Fill (2)

Answer: