Assessment test solutions

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SECTION 1 (basic knowledge) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**/////////////////////////////// FLOW OF CONTROL /////////////////////////////////////////**

Q1: What numbers are printed by the following loops?

for (int i = 0; i < 5; ++i)

printf("loop 1: counter %d\n", i);

for (int i = 1; i <= 5; ++i)

printf("loop 2: counter %d\n", i);

for (int i = 5; i > 0; --i)

printf("loop 3: counter %d\n", i);

for (int i = 5; i--;)

printf("loop 4: counter %d\n", i);

Q1 solution:

The output is as follows:

loop1: 0 1 2 3 4  
loop2: 1 2 3 4 5  
loop3: 5 4 3 2 1  
loop4: 4 3 2 1 0

It is not relevant whether we have pre-or post-increment/decrement, since the third expression in the for statement is executed as a separate statement. In loop 4, the printed value is one smaller than in the test; the last nonzero value to pass the test is i=1, so the last value printed is 0.

**Flow of control additional questions:**

Q2: What output does the following code produce?

int x, y;

int i = 3;

x = y = 0;

while (i >= 0) {

x += 1;

i--;

}

do {

y += 1;

i++;

} while (i <= 3);

printf("%d\n", (x - y));

Q2 solution:

The output is: -1

The while loop will add 1 to x four times. The do-while loop will add 1 to y five times.

**/////////////////////////////////////// POINTERS ////////////////////////////////////////**

Q1: What is the output of the following code?

int ary[4] = { 11, 22, 33, 44 };

int \*ptr;

ptr = ary;

for (int i = 0; i < 4; i++)

{

printf("%d %p\n", \*ptr, ptr);

ptr++;

}

Q1 solution:

The program prints the following:

11 0x7ffc3ae795e0

22 0x7ffc3ae795e4

33 0x7ffc3ae795e8

44 0x7ffc3ae795ec

Since the ptr variable is set to ary, the program will print the value of each element in the array followed by the element’s address in memory. For each iteration through the loop, the pointer is incremented to point to the next element of the array (in this example elements are 4 bytes each).

**Pointers additional questions**

Q2: What does the following program print when run with arguments: one two three ?

int main(int argc, char \*argv[])

{

for (char \*p; p = \*++argv;)

for (; \*p; ++p)

putchar(\*p);

}

Q2 solution:

The program will print onetwothree.

The array argv[] contains pointers (char\*) to the words on the command line, starting with the program name, then the command line arguments, and terminated by NULL. Each outer loop increments argv before dereferencing, testing and assigning to p, so the outer loop skips over the program name and assigns p in turn with each command line argument. The inner loop prints the current character \*p and increments the pointer p until reaching the \0 terminator for each string.

**//////////////////////////////////////// FUNCTIONS //////////////////////////////////////**

Q1: What happens in the following functions?

void swap1(int x, int y) {

int tmp = x;

x = y;

y = tmp;

}

void swap2(int &x, int &y) {

int tmp = \*x;

\*x = \*y;

\*y = tmp;

}

Q1 solution:

The function swap1 will swap the values of x and y, with effect only inside the function (the values placed on the stack are swapped). Since in C, arguments are passed by value, this has no effect when returning to the caller code. That is, the actual arguments (expressions) are evaluated, the resulting values are pushed on the stack and the function is called. On return, the stack is cleaned up, so the values of the arguments (modified or not) are discarded, they are not propagated back to the caller.

In swap2, the arguments are pointers. Therefore, the values at addresses x and y will be interchanged, and this change is persistent. If we have int z = 2, t = 3; swap2(&z, &t); after this call we’ll have z=3, t=2.

**Functions additional questions:**

Q2: What is the output of the following program?

#define SIZE 3

int sum(int a1[], int a2[]) {

int sum = 0;

for (int i = 0; i < SIZE; i++) {

a1[i] += a2[i];

sum += a1[i];

}

return sum;

}

int main() {

int array1[SIZE] = {1,2,3};

int array2[SIZE] = {1,2,3};

int x = sum(array1, array2);

for (int i = 0; i < SIZE; i++) {

printf("%d + %d", array1[i], array2[i]);

if (i < SIZE - 1) printf(" + ");

}

printf(" = %d\n", x);

}

Q2 solution:

The output is: 2 + 1 + 4 + 2 + 6 + 3 = 12.

When passing the arrays to the sum() function, we are passing pointers to the first element of the arrays. Therefore, any manipulation of the arrays in the sum() function will change the original arrays in main (it is not a copy). In this example we are adding the i:th element of each array and storing it in array a1[]. Since a1 is pointing to array1 in the main() function, the array elements of array1 are changed to 2, 4, 6.

**///////////////////////////////////////// STRUCTURES ////////////////////////////////////**

Q1: What is the output when running the following program?

struct Date

{

int day;

int month;

int year;

};

int main()

{

struct Date d = { 1, 1, 2020 };

struct Date \*d2;

d2 = &d;

printf("%d/%d/%d\n", d2->day, d2->month, d2->year);

return 0;

}

Q1 solution:

The output is: 1/1/2020

The structure declared has three member variables day, month, and year. In the main function the member variables are initialized and printed.

**Structures additional questions:**

Q2: What does the following code print?

struct {

char s[3];

int i;

} x;

printf("%d\n", (char \*)&x.i - x.s);

printf("%d\n", sizeof(x) - sizeof(x.s) - sizeof(x.i));

Q2 solution:

Structure fields are placed in memory in the same order as declared. However, there may be unused bytes so each field satisfies the alignment constraints for the type (integers and pointers are typically aligned at word boundaries. The first print statement casts &i to a char\*, so the pointer difference represents the number of bytes between the start of fields i and s (the structure start). This could be 4 if integers have 4-byte alignment. The second statement prints the difference between the total structure size and the sizes of its fields. Assuming 4-byte alignment for integers, this would be 1.

**///////////////////////////////// OPERATORS + CASTING //////////////////////////////////**

Q1: Explain the difference in result between program1 and program2?

Program1:

int main() {

int x, y, z;

x = y = z = 1;

x += (y += z);

printf("%d %d %d\n", x, y, z);

}

Program2:

int main() {

int x, y, z;

x = y = z = 1;

(x += y) += z;

printf("%d %d %d\n", x, y, z);

}

Q1 solution:

Program 1 outputs: 3 2 1

Program 2 outputs: Compiling error, lvalue required as left operand of assignment.

An lvalue in C is an object that occupies some identifiable location in memory. In program 2, the left operand is not an l-value, it is an assignment statement.

**Operators additional questions**

Q2: What output does the following code produce?

void initArray() {

int array[3];

int i = 3, x = 3;

while (--i && (array[i] = --x))

printf("%d\n", array[i]);

}

Q2 solution:

The output is:

2

1

In C zero evaluates to false and all non-zero values evaluate to true. The assignment statement assigns the value on the right side to the l-value on the left side. In addition, it returns the value assigned. The while loop will execute the body of the loop twice since i is pre-decremented (in this example the loop termination will depend on i due to short-circuit evaluation). If both i and x would have been post-decremented, the loop would have executed three times resulting in output 3,2,1.

Q3:What do the following functions do?

void counti(int n) {

while (n-- >= 0)

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

}

void countu(unsigned n) {

while (n-- >= 0)

printf("%u\n", n);

}

Q3 solution:

The loop in counti executes while n is nonnegative. The value printed is one smaller than in the test. The code will print values from n-1 down to -1.  
In function countu, n is unsigned, so can never become negative. When n is 0, decrementing will wrap aroud to UINT\_MAX. The loop never stops.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SECTION 2 (focused knowledge) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**/////////////////////////////////// STRINGS /////////////////////////////////////////////**

Q1: What is the output of the following program?

int main() {

char \*t;

const char \*str = "item1 item2 item3 item4 item5";

char \*tmp = (char \*)malloc(strlen(str) + 1);

if (tmp == NULL) {

exit(1);

}

strcpy(tmp, str);

t = strtok(tmp, " ");

printf("%s,", t);

while (t = strtok(0, " ")) {

printf("%s,", t);

}

free(tmp);

tmp = NULL;

}

Q1 solution:

The output is: item1,item2,item3,item4,item5,

The strcpy function copies one string to another. The strtok splits the string by some delimiter. In this example the string is split by the space character “ “.

**Strings additional questions**

Q2: What does the call to len2() produce?

void len1(char s[]) {

printf("size: %zu, len: %zu\n", sizeof(s), strlen(s));

}

void len2(void) {

char s[10] = "foo";

printf("size: %zu, len: %zu\n", sizeof(s), strlen(s));

len1(s);

}

Q2 solution:

The printf call in len2 will print 10 and 3: sizeof returns the declared size of the array and strlen the length of the string initializer (excluding the \0 terminator).  
In the call to len1, s is a char \* (a pointer with no length information is passed). Thus, sizeof(s) is sizeof(char \*) (possibly 4 or 8, on 32-bit/64-bit machines); the value of strlen() is the same, 3.

Q3: What is the result of running the following program?

int main() {

char arr[] = "abcc";

char \*str = "abcc";

\*(arr + 3) = 'd';

puts(arr);

\*(str + 3) = 'd';

puts(str);

return 0;

}

Q3 solution:

Running the program would result in undefined behavior. More specifically, printing the array arr[] would result in the output abcd while trying to print the string str would result in undefined behavior. Since str is a string literal, attempting to modify it, may result in access violations since it is typically stored in read-only memory.

**//////////////////////////////////// INPUT / OUTPUT /////////////////////////////////////**

Q1: What does the following program do?

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

FILE \*fp;

int x = 0;

int c;

fp = fopen(argv[1], "r");

if (!fp) {

/\*handle error and exit\*/

}

for (c = getc(fp); c != EOF; c = getc(fp)) {

if (c == '\n')

x++;

}

fclose(fp);

printf("%d\n", x);

return 0;

}

Q1 solution:

The program is counting the number of lines in a file.

It opens the file specified as a command line argument argv[1]. It goes through the file content character by character until it reaches the end of the file (EOF). If a new line character ‘\n’ is encountered while reading the file, the x variable is incremented by 1. Finally, x is printed to standard output.

**I/O additional questions**

Q2: Explain the behavior of the three code fragments. In each case the input is

line one\nline two\n

char s1[10];

fgets(s1, 10, stdin);

printf("%zu", strlen(s1));

char s2[10];

scanf("%9s", s2);

printf("%zu", strlen(s2));

char s3[10];

scanf("%10c", s3);

printf("%zu", strlen(s3));

Q2 solution:

In each case the input is:

line one\nline two\n

Fragment 1 (fgets) reads a line, until reaching \n, EOF, or the limit (10-1 chars).  
In this case, the string is “line one\n” (total 9 characters).  
The second fragment (scanf) reads a word, until reaching whitespace or 9 characters.  
With the given input, the string read is: “line”.  
The third fragment reads exactly 10 characters of any kind (including spaces or terminators), stopping only at EOF. Thus, s3[] will contain “line one\nl” (ending with the first ‘l’ character of line two), with no \0 terminator.

**//////////////////////////////////// POINTERS ARITHMETIC ///////////////////////////////**

Q1: What does the following function do?

char \*foo(char \*restrict d, const char \*restrict s)

{

const unsigned char \*s\_tmp = s;

unsigned char \*d\_tmp = d;

while ((\*d\_tmp++ = \*s\_tmp++));

return d;

}

Q1 solution:

This functions copies one string to another. The loop will iterate through the string assigning the source string to the destination string character by character. Since the assignment statement returns the value assigned, the loop will terminate once it reaches the end of the string (the null terminating character ‘\0’).

**/////////////////////////////////// MEMORY MANAGEMENT ///////////////////////////////////**

Q1: Draw str1 and str2 as they appear in memory. What is the output of this program?

int main()

{

char str1[5] = "Hello";

char str2[5] = "cStr";

printf("%s\n", str1);

return 0;

}

Q1 solution:

l

l

e

H

o

str1

c

S

t

r

\0

str2

The output of this program is undefined. Since str1 was declared with size 5 and initialized to a string of 5 characters, the string will not end with the null terminating character ‘\0’. If the null terminating character is missing, attempting to print the string might result in unexpected behavior.

**Memory management additional questions:**

Q2: Explain the memory layout for:

char days[][3] = { "mon", "tue", "wed", "thu", "fri" };

char \*days[] = { "mon", "tue", "wed", "thu", "fri" };

Q2 solution:

The first definition is for a bidimensional array (character matrix). The initializer strings have the same length as the number of columns (3). The array has 5x3 = 15 bytes, the strings are placed contiguously in memory, without terminators or separators: montuewedthufri .  
The second line defines an array of five pointers. Each of these points to a string literal as given by the initializer. Each string literal has a \0 terminator. The strings need not be placed contiguously in memory, although the compiler will likely do so. Assuming “mon” is at address 0xfce3 and the other strings follow, we’ll have  
char \*days[] = { 0xfce3, 0xfce7, 0xfceb, 0xfcef, 0xfcf3 }.

Q3: Explain the memory layout in the following two cases:

int \*\*a = malloc(3 \* sizeof(int \*));

if (a)

for (int i = 0; i < 3; ++i) {

a[i] = malloc(5 \* sizeof(int));

if (a[i])

for (int j = 0; j < 5; ++j)

a[i][j] = 3\*i + j;

}

int (\*m)[5] = malloc(3 \* 5 \* sizeof(int));

for (int i = 0; i < 3; ++i) {

for (int j = 0; j < 5; ++j)

m[i][j] = 3\*i + j;

The first fragment declares an array of three pointers. Each pointer is allocated for a block of five integers. Assuming sizeof(int) = 4, sizeof(int \*) = 8, and that allocated memory blocks have 4 bytes overhead, rounded to multiples of 16 bytes, we could have:  
a = 0x7ce0, a[0] = 0x7d10, a[1] = 0x7d30, a[2] = 0x7d50.|  
The second fragment allocates a matrix of 3x5 integers, contiguously allocated. The declaration int (\*m)[5] states that m is a pointer to an int[5], i.e., a matrix “line” of 5 integers. Thus, m[0], m[1] and m[2] are pointers to such lines (5 integers apart), and can be indexed once more to yield the integer matrix elements.

Q4: Explain the resulting memory layout after running the code:

char \*\*t = calloc(5, sizeof(char \*));

if (t)

for (int i = 0; i < 5; ++i) {

t[i] = malloc(80);

if (t[i] && fgets(t[i], 80, stdin))

t[i] = realloc(t[i], strlen(t[i]) + 1);

} else break;

}

Q4 solution:

We have an array of pointers, like in the first part of Q3. By using calloc(), the five pointers are initialized to NULL. The following loop allocates each pointer to a block of 80 chars, and, if the allocation succeeds, reads a line (until \n, EOF, or 79 chars). Then the allocated block is shrunk to precisely the length needed, which may result in the block being moved to another address, thus the assignment to t[i]. Since we don’t request more memory, the call should normally succeed (unless the allocator is buggy or corrupted). If the input has fewer than five lines, or there is not enough memory, the remaining pointers in the array stay initialized to NULL.

**////////////////////////////////////// CONCURRENCY //////////////////////////////////////**

Q1: Assuming that the function printString prints the string it receives as an argument one character at a time. What is the output of the following program?

int main() {

pid\_t pid;

pid = fork();

if (pid) {

printString("Hello\n");

}

else {

printString("World!\n");

}

}

Q1 solution:

The output is some permutation of the strings Hello and World. For example, HeWorllold (excluding the new line character ‘\n’). The fork() system call is used to create a new process. This process (child process) will run concurrently with the process that called the fork function (parent process). The string Hello will be printed by the parent process and the string World will be printed by the child process since the fork function returns pid 0 in the child process. The order in which the letters will be printed to standard output will depend on the CPU scheduling controlled by the operating system.

**Concurrency additional questions:**

Q2: What does the following program output?

int main() {

pid\_t childpid;

int x = 0;

childpid = fork();

if (childpid == 0)

x = x + 4;

childpid = fork();

if (childpid == 0)

x = x + 2;

childpid = fork();

if (childpid == 0)

x = x + 1;

printf("%d", x);

}

Q2 solution:

The output is some permutation of the numbers 0,1,2,3,4,5,6 and 7. For example, 42160537.

The fork() system call is used to create a new process. This process (child process) will run concurrently with the process that called the fork function (parent process). The child process will have a copy of the parent’s address space, identical but separate. Any modifications to the variable x will be independent of the parent process. The below figure represents the spawning of processes for each call to the fork() function and the value of x in each process.

X=0

X=0

X=4

X=4

X=6

X=2

X=0

X=7

X=0

X=1

X=2

X=3

X=4

X=5

X=6

The final output (permutation of the values of x) will depend on the CPU scheduling controlled by the operating system.

**/////////////////////////////////////// SYSTEM CALLS ////////////////////////////////////**

Q1: Explain what the program below does?

#define MAX\_BUF\_SIZE 256

int main() {

const char\* home;

char command[MAX\_BUF\_SIZE];

int len = 0;

home = getenv("HOME");

if (home == NULL){/\*handle error and exit\*/ }

len = snprintf(command, sizeof(command),

"%s%s%s", "ls -l ", home, " | head -n 5");

if (len >= MAX\_BUF\_SIZE || len < 0) {/\*handle error and exit\*/ }

system(command);

return 0;

}

Q1 solution:

The program lists the first five files in the home directory. The program uses the getenv() function to get the path of the home directory. It then concatenates the following commands: ls -l (long listing of files), home directory path, pipe, and head -n 5 (lists the first 5 lines of a file). In other words, the output from ls -l of the home directory is piped to the head -n 5 command.

**System calls additional questions**:

Q2: Explain what the program below does?

extern char \*\*environ;

int main() {

if (setenv("PATH", "/some/path/bin", 1) != 0) {

/\* Handle error and exit \*/

}

if (environ != NULL) {

for (size\_t i = 0; environ[i] != NULL; ++i) {

puts(environ[i]);

}

}

return 0;

}

Q2 solution:

The program sets the environment variable PATH to /some/path/bin. It uses the global variable environ to access the environmental variables. It prints all of them to standard output after the modification of the PATH variable.