1.1 Select the option below that is **NOT** equivalent to the following statement: `int *p = 20;`

(a) `int* p = 20;`

(b) `int * p = 20;`

(c) `int *p; p = 20;`

D. `int *p; *p = 20;`

(e) None of the above: All of the above statements are equivalent.

1.2 Consider the program below, has a function call missing from main(). Select the correct function call that would result in the program printing the string "Hello, world!".

```
void increment1(int count) {
        count++;
}
void increment2(int *count_ptr) {
        (*count_ptr)++;
}
int main(int argc, char **argv) {
        int count = 0;
        ______________________; // Correct function call (from the options below) goes here
        if(count == 1) printf("Hello, world!");
        return 0;
}
```

A. `increment1(&count);`

B. `increment1(*count);`

C. `increment2(count);`

D. `increment2(&count);`

E. `increment2(*count);`

1.3 Suppose we have the following declarations at the start of a program. Select the statement below that does **NOT** assign the value 4 to the variable q.

```
int x = 4;
int *y = &x;
int **z = &y;
int q;
```

A. `q = x;`

B. `q = *(&x);`

C. `q = *y;`

D. `q = &(*x);`

E. `q = **z;`

1.4 Consider the following code fragment, and assume (for the purposes of answering this question) that integers are 4 bytes and pointers are 8 bytes. Select the correct answer for how many bytes are allocated on the stack, and how many on the heap.

```
int *A = malloc(sizeof(int) * 20);
```

A. `Stack: 8 Heap: 160`

B. `Stack: 8 Heap: 80`

C. `Stack: 0 Heap: 168`

D. `Stack: 0 Heap: 88`

E. None of the above.

1.5 Consider the program below, and select the correct output that matches what the program will print.

```
struct StudentNode {
        char *name;
        int num; };
void updateStudent(struct StudentNode s) {
        s.num = s.num + 2;
        strcat(s.name, " v2");
}
int main() {
        struct StudentNode s1;
        s1.num = 12345;
        s1.name = malloc(256*sizeof(char));
        strcpy(s1.name, "Bob");
        updateStudent(s1);
        printf("%s; ", s1.name);
        printf("%d\n", s1.num);
}
```

A. `Bob; 12345`

B. `Bob; 12347`

C. `Bob v2; 12345`

D. `Bob v2; 12347`

E. This code will cause a segmentation fault, because `strcpy` is unsafe.

1.6 Select the statement below that is **TRUE** (or *All of the above*, if they are all true), concerning signals.

A. A process can only send a signal to its parent process or its child processes.

B. The kill system call is used only to terminate processes.

C. The default action of a SIGKILL signal cannot be modified.

D. A signal handler runs in a separate process from the program receiving the signal.

E. All of the above are true.

1.7 Recall: (1) The `tee` program, which reads from `stdin` and copies the stream to **both** `stdout` and to the file specified by the command-line argument; (2) The `sort` program, which sorts the lines of text contained in the file specified by the command-line argument, and outputs to `stdout`; and (3) The `head` program, which reads from `stdin` and outputs the first 10 lines to `stdout`. Consider the following shell command, and select the statement below that is TRUE.

```
sort input.txt | tee output.txt | head
```

A. The contents of `input.txt` and `output.txt` are identical.

B. Only the first 10 lines of `output.txt` will be displayed in the terminal.

C. The entire contents of `input.txt` followed by the first 10 lines of `output.txt` will be displayed in the terminal.

D. The entire contents of `output.txt` followed by the first 10 lines of `output.txt` will be displayed in the terminal.

E. The entire contents of both `input.txt` and `output.txt`, followed by the first 10 lines of `output.txt`, will be displayed in the terminal.

1.8 Select the statement below that is **TRUE**, concerning the invocation of the `read()` or `write()` system calls on a pipe.

A. A `read()` invocation on a pipe that has completely filled its buffer will return the number of bytes that were read, but only if at least one process still has an open write descriptor to the pipe.

B. A `read()` invocation on an empty pipe will block indefinitely until data is available, or until all processes close their write descriptors to the pipe.

C. A `read()` invocation on an empty pipe will immediately generate a `SIGPIPE` signal, if all processes have closed their write descriptors to the pipe.

D. A `write()` invocation on a full pipe will immediately generate a `SIGPIPE` signal, if another process has an open read descriptor to the pipe but never invokes `read()`.

E. A `write()` invocation on a pipe will block indefinitely if no process has an open read descriptor to the pipe.

1.9 Select the statement below that is **TRUE**, concerning the `fork()` system call.

A. After `fork()` returns, the child process will always execute first .

B. After `fork()` returns, the parent process will always execute first .

C. If a child process updates the value of a **static** variable, it will be updated in the parent process as well.

D. The child process will inherit the open file descriptor table of its parent.

E. The parent process will not terminate until all of its children have terminated.

1.10 Select the statement below that is **TRUE** (or *All of the above*, if they are all true), concerning the `exec()` family of system calls.

A. The `exec` system calls do not create a new process.

B. After `exec` is called, the new program inherits the open file descriptor table of the original program.

C. If a call to `exec` succeeds, it will not return a value.

D. After `exec`, the new program will retain the PID of the previously-running program.

E. All of the above are true.

1.11 Select the statement that is **TRUE**, considering the program consisting of the following two source files. Assume that the program is compiled with `gcc -o hello hello.c main.c`.

```
/* Complete contents of main.c */
void hello(void);
int main(void) {
        hello();
        return 0; }
/* Complete contents of hello.c */
#include <stdio.h>
void hello() {
        printf("Hello, world!\n");
}
```

A. The program will fail to compile, because `main.c` is missing the line `#include "hello.c"`.

B. The program will fail to compile, because `main.c` is missing the line `#include <stdio.h>`.

C. The program will fail to compile, because the `gcc` command is incorrect.

D. The program will compile successfully, but its behaviour is undefined (e.g., it may trigger a segmentation fault or print out garbage values).

E. The program will compile and print `Hello, world!` to the terminal.

1.12 Select the statement below that is **TRUE** (or *All of the above*, if they are all true), concerning threads and processes.

A. Processes do not share the same memory space, but threads belonging to the same process do.

B. Process creation with `fork` is slow, but thread creation is much faster.

C. Each thread has its own global `errno` variable.

D. Threads belonging to the same process share the same heap and global variables, but have separate function call stacks.

E. All of the above are true.

1.13 Select the statement that is **TRUE**, considering the program below.

```
struct my_struct {
        char *name;
};
void array_chief(struct my_struct *s) {
        char new[4] = {'a', 'b', 'c', '\0'};
        s->name = new;
}
int main(void) {
        struct my_struct s1;
        s1.name = "Bob";
        array_chief(&s1);
        printf("%s\n", s1.name);
        /* Do other things, call some other functions... */
        return 0;
}
```

A. The program will fail to compile, because `array_chief()` assigns an array to a pointer variable.

B. The program will fail to compile, because `s1.name` in `main()` is a read-only string literal, which `array_chief()` attempts to overwrite.

C. The program will compile without errors, but a segmentation fault will be triggered when `main()` assigns the return value of `array_chief()` to `s1.name`, since the latter is a read- only string literal.

D. The program will compile without errors, but its behaviour is undefined, e.g., it may print `abc` or it may print other garbage values or result in other unpredictable behaviour.

E. The program will compile without errors, and will always print out `abc`.

1.14 Select the file type that is most appropriate for opening with `fopen` using the `"rb"` flag.

A. A C program's header file(s).

B. A C program's object file(s).

C. A C program's source file(s).

D. A Makefile.

E. None of the above are appropriate to open using the `"rb"` flag.

2. **Structs and Dynamically Allocated Arrays**

The program below defines a struct to manage a *dynamically-allocated array*. Your job is to write two helper functions, `initialize` and `add`, to make the program work correctly. The requirements are as follows:

- `initialize` initializes an `array_list` struct, which is passed in as the single input parameter. By default, the array should have a capacity of 5.
- `add` appends one or more integers contained in the array passed in the first parameter (which may be either on the stack or the heap). The second parameter specifies the number of integers contained in the array being passed in the first parameter. The third parameter specifies the `array_list` to which the new integer(s) should be added. If there is not enough space in the `array_list`, a new array should be allocated that is big enough to hold **double** the new elements plus the existing elements (i.e. double the total of the two), and the contents of the old array should be moved into the new one (for full marks, do this without writing a loop) before appending the new integers to the list.

We have not provided you with the function signatures for `initialize` and `add`: You need to determine these yourself, in a way that satisfies both the requirements given above and the correctness of the program below.

Both `initialize` and `add` should perform any necessary error checking: They should return `0` on success, and `-1` on failure.

```
struct array_list {
        int *contents;
        size_t capacity;      // Current capacity of the array
        size_t curr_elements; // Number of elements currently occupied in the array
};
int main(void) {
        struct array_list list;
        initialize(&list);
        int a[11] = {2, 0, 9, 4, 5, 6, 7, 8, 9, 10, 11};
        add(a, 11, &list);
        // The loop below should print "2 0 9 4 5 6 7 8 9 10 11 "
        for(int i = 0; i < list.curr_elements; i++)
        printf("%d ", list.contents[i]);
        /* The program does some fancy stuff with the array here,
        * generates some output, and performs any remaining
        * cleanup before terminating.
        */
        return 0; }
```

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

struct array_list {
    int *contents;
    size_t capacity;      // Current capacity of the array
    size_t curr_elements; // Number of elements currently occupied in the array
};

int initialize(struct array_list *list) {
    if ((list->contents = (int *) calloc(5, sizeof(int))) == NULL) {
        perror("calloc");
        return -1;
    }
    list->capacity = 5;
    list->curr_elements = 0;
    return 0;
}

int add(int *int_list, int number, struct array_list *list) {
    if (number + list->curr_elements > list->capacity) {
        int *new_list;
        if ((new_list = (int *) calloc((2 * (number + list->curr_elements)), sizeof(int)))
 == NULL) {
            perror("calloc");
            return -1;
        }
        // Copy old contents to new array
        memcpy(new_list, list->contents, list->curr_elements * sizeof(int));
        // Free old declared array
        free(list->contents);
        // Copy new contents to new array
        memcpy((new_list + list->curr_elements), int_list, number * sizeof(int));
        list->contents = new_list;
        list->capacity = (2 * (number + list->curr_elements));
        list->curr_elements += number;
        return 0;
    }
    memcpy(list->contents + list->curr_elements, int_list, number * sizeof(int));
    list->curr_elements += number;
    return 0;
}

int main(void) {
    struct array_list list;
    initialize(&list);
    int a[11] = {2, 0, 9, 4, 5, 6, 7, 8, 9, 10, 11};
    add(a, 11, &list);
    // The loop below should print "2 0 9 4 5 6 7 8 9 10 11 "
    for(int i = 0; i < list.curr_elements; i++)
        printf("%d ", list.contents[i]);
    /* The program does some fancy stuff with the array here,
     * generates some output, and performs any remaining
     * cleanup before terminating.
     */
return 0; }
```

3. **Signals**

Study the following program that installs a signal handler.

```
int turn = 0;
void handler(int code) {
        if(turn == 0) {
                fprintf(stderr, "First\n");
                turn = 1;
                /* D */
        }
        else {
                fprintf(stderr, "Second\n");
                kill(getpid(), SIGQUIT);
        }
        fprintf(stderr, "Here\n");
}
int main(void) {
        struct sigaction sa;
        sa.sa_handler = handler;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);
        sigaddset(&sa.sa_mask, SIGINT);
        /* A */
        sigaction(SIGTERM, &sa, NULL);
        /* B */
        fprintf(stderr, "Done\n");
        /* C */
        return 0; }
```

On the next page, provide the output of the above program when the events described in each subquestion occurs, assuming that the code runs correctly, i.e., no undefined behaviour or other unspecified events occur. Treat each subquestion as if the program were restarted. Each event is described as a signal that is delivered to the program just before the program executes the line of code following the specified comment line (i.e., A, B, C, or D). Give the **TOTAL** output of the program in each case.

Note: When a process exits due to a `SIGTERM`, `SIGQUIT`, or `SIGKILL`, the shell process prints “Terminated”, “Quit”, or “Killed”, respectively, after the program terminates. Include these messages in your answers where applicable.

3.1 Two `SIGTERM` signals arrive one after the other at A.

> Because `sigaction` has not been used as yet, `SIGTERM` will not be handled by the installed handler, so shell would print "Terminated" for the first `SIGTERM`, and since the program is terminated after the first signal, sending the second `SIGTERM` would do nothing.

3.2 `SIGTERM` arrives at B and `SIGTERM` arrives again at C.

> `First`, then `Here`, then `Done` gets printed to `stderr` when `SIGTERM` arrives at B.
> `Second` gets printed to `stderr` and then shell would print "Quit".

3.3 `SIGTERM` arrives at B and `SIGINT` arrives at D.

> `First` would be printed due to `SIGTERM` arriving, and since `SIGINT` is in masked in `sa`, `SIGINT` at D would cause no interruption, therefore `Here` then `Done` would be printed to `stderr`.

3.4 SIGTERM arrives at B and SIGKILL at D

> `First` would be printed due to `SIGTERM` arriving, and since `SIGKILL` cannot be handled, shell would print "Killed".

3.5 True or False: `fprintf` is async-signal-safe, assuming it is used in a single-threaded program.

> False

4. **Forking**

4.1 Consider the program below, and enter the correct numbers in the table on the right-hand side.

```
int main(void)
{
        int i = 0;
        printf("Broccoli\n");
        int r = fork();
        printf("Cucumbers\n");
        if (r == 0) {
                printf("Kale\n");
                int k = fork();
                if (k >= 0) {
                        printf("Peppers\n");
                }
        } else if (r > 0) {
                wait(NULL);
                printf("Cabbage\n");
                while(fork() == 0) {
                        printf("Carrots\n");
                        i++;
                        if(i == 3) break;
                }
                i = 0;
                while(fork() > 0) {
                        printf("Spinach\n");
                        i++;
                        if(i == 2) break;
                }
        }
        return 0;
}
```

Fruit name	Times printed
Broccoli	1
Cucumbers	2
Kale	1
Peppers	2
Cabbage	1
Carrots	3
Spinach	8

Assuming that the above program runs without errors (e.g., `fork` always returns successfully, and the program is not terminated by a signal such as `SIGKILL`):

4.2 How many distinct processes print "Spinach"?

> 6?

4.3 How many distinct processes print "Carrots"?

> 3

4.4 True or False: The second line of output should ALWAYS be "Cucumbers".

> True, Broccoli should only print one time.

4.5 True or False: The last line of output should ALWAYS be "Spinach".

> False, race conditions (I think)

4.6 True or False: "Peppers" is ALWAYS printed before "Cabbage".

> True (?), the `fork` process with `r` waits for child process

4.7 List all the vegetable(s) that will NEVER be printed after the bash prompt re-appears.

> Broccoli, Cucumbers, Kale

4.8 List all the vegetable(s) that MIGHT be printed after the bash prompt re-appears.

> Peppers, Cabbage, Carrots, Spinach

5. **Pipes**

Write a program that forks two children. We refer to the first child (i.e. the child that is created first) as Child A, and the second as Child B.

Child A must be able to send a stream of bytes to Child B over a pipe. All processes must close file descriptors at the earliest appropriate point in the program, and perform any necessary error checking. Right after this is done, Child A must invoke `foo_a()` and Child B must invoke `foo_b()` (you may assume that neither of these functions will return back to main). You do not need to worry about writing any data to the pipe — simply ensure that the pipe is correctly set up to allow Child A to send data over it to Child B.

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

int main() {
    int my_pipe[2];

    if ((pipe(my_pipe)) == -1) {
        perror("pipe");
        exit(1);
    }
    int a, b, a_status, b_status;
    switch (a = fork())     // A process
    {
        case -1:
            perror("fork");
            exit(1);
            break;
        case 0:
            close(my_pipe[0]); // Won't be reading from pipe
            foo_a();
            break;
        default:
            waitpid(a, &a_status, 0);
            break;
    }
    switch (b = fork())     // B process
    {
        case -1:
            perror("fork");
            exit(1);
            break;
        case 0:
            close(my_pipe[1]); // Won't be writing to pipe
            foo_b();
            break;
        default:
            waitpid(b, &b_status, 0);
            break;
    }
}
```