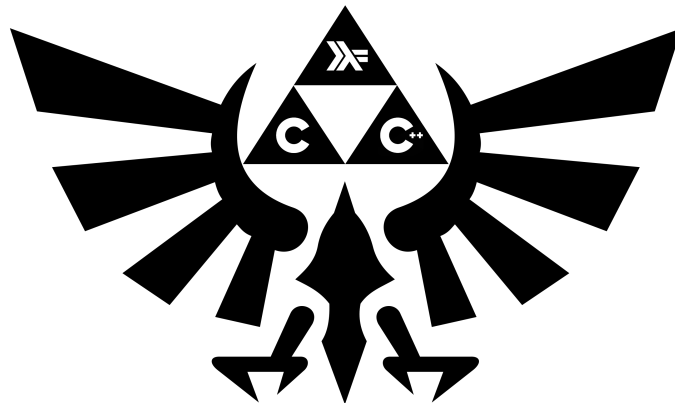


B3 - Paradigms Seminar

B-PDG-300

Day 04 AM

C, Life, the Universe and everything else





Day 04 AM

language: C



- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

All your exercises will be compiled with the `-std=gnu17 -Wall -Wextra` **flags**, unless specified otherwise.



Every function implemented in a header or any unprotected header leads to 0 for the exercise.



To avoid compilation problems during automated tests, please include all necessary files within your headers.

For each exercise, the files must be turned-in in a separate directory called **exXX** where XX is the exercise number (for instance `ex01`), unless specified otherwise.



UNIT TESTS

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the “**How to write Unit Tests**” document on the intranet, available [here](#).

Create a directory named `tests`. For each of the functions you turn in, create a file in that directory named `tests_FUNCTION_NAME.c` containing all the tests needed to cover all of the exercise's possible cases (regular or irregular).

Here is a sample set of unit tests for the **my_strlen** function:

```
#include <criterion/criterion.h>

Test(my_strlen, positive_return_value)
{
    cr_assert_eq(my_strlen("toto"), 4);
}

Test(my_strlen, empty_string)
{
    cr_assert_eq(my_strlen(""), 0);
}
```



EXERCISE O - Z IS FOR ZORGLUB

Turn in: Makefile, z.c in ex00/

Compilation: using your Makefile

Executable name: z

Our Lord and Genius Zorglub, master of Zorgland, wants you to design the software for his new brain-washing machine to zorgmanize his enemies into zorgmen.

The machine works by sending a specific character into the mind of its victim.

The character must be chosen carefully using an algorithm devised by our Lord and Genius Zorglub.

The algorithm uses the prisoner ID to determine which character should be used.

An ID is a hexadecimal number that fits in a `uint64_t`, given as parameter to your program.

You must then display the character to be used, following the rules defined by the Grand Z:

- If the binary notation of the ID is a palindrome, it means that the victim is just a dummy used for test purpose. You must display the default brain-washing byte `0172` (octal) and validation byte `012` (octal).
- If the ID is a multiple of 13 and 29 and 89 or 41 and 71 or 67 or 7 and 43 and 47 and 53, you must display the string `"z\n"` and immediately stop your program.
- If the ID is `0x12345678`, `0x87654321`, `0x01111010` or `0x01011010`, you must display the following bits :
`0111101000001010`.
- If one of the byte of the ID is worth `0x42`, it means that the victim is tough and the strongest character must be used : `'z'`. The byte `0x0A` is then displayed.
- If the last byte of the ID is null, the victim is just a minion of our enemies. As we don't care about them, we will brain-wash them using the last letter of the alphabet in lowercase. The output will then be flushed with a line feed.
- If the ID is a multiple of the sum of its 8 bytes, you must display the sixth character of the ASCII table starting from the end, followed by a `'\n'`.
- If the 8 bytes of the ID are identical, it means that the prisoner is one of our special guest. You will use the character `'z'` in lowercase, and your program will end with a new line. This case override all the other cases.
- If the parameter contains non-hexadecimal characters, this is an error. The program must handle this case by displaying the error character `'z'` followed by a line feed.
- If the parameter does not fit in a `uint64_t`, this is an error. In this very specific case, the very specific character `'z'` must be used to brain-wash the victim, followed by the tenth character of the ASCII table.
- If there is no argument, the ID to be used is the return value of the function `time(NULL)`.
- If there is too many argument, your program must use the first valid argument starting from the end. If no argument is valid, we consider there is no argument and use the preceding rule.

Your program must *always* return `0` as it will never fail, even in case of invalid input.

```
Terminal
~/B-PDG-300> ./z "0x42242112" | cat -e
z$
~/B-PDG-300> ./z "invalid_ID" ; echo $?
z
0
```



EXERCISE 1 - MUL DIV

Turn in: `mul_div.c` in `ex01/`

Define the following functions:

1. `void mul_div_long(int a, int b, int *mul, int *div);`

Calculates the product and the division of the `a` and `b` parameters, storing the results in the integers pointed by `mul` and `div` respectively.

2. `void mul_div_short(int *a, int *b);`

Calculates the product and the division of the integers pointed by `a` and `b` and stores the results in `a` and `b` respectively.



In case of division by zero, the result is the Answer to Life, the Universe and Everything.



Here is a sample main function with the expected output:

```
static void test_long(void)
{
    int a = 13;
    int b = 4;
    int mul;
    int div;

    mul_div_long(a, b, &mul, &div);
    printf("%d * %d = %d\n", a, b, mul);
    printf("%d / %d = %d\n", a, b, div);
}

static void test_short(void)
{
    int a = 12;
    int b = 0;
    int mul_res = a;
    int div_res = b;

    mul_div_short(&mul_res, &div_res);
    printf("%d * %d = %d\n", a, b, mul_res);
    printf("%d / %d = %d\n", a, b, div_res);
}

int main(void)
{
    test_long();
    test_short();
    return (0);
}
```

```
Terminal
~/B-PDG-300> ./a.out
13 * 4 = 52
13 / 4 = 3
12 * 0 = 0
12 / 0 = 42
```



EXERCISE 2 - CONCAT

Turn in: `concat.c` in `ex02/`

Notes: The `concat_t` structure is defined in the provided `concat.h` file.

Define the following functions:

1. `void concat_strings(const char *str1, const char *str2, char **res);`

Concatenates `str1` and `str2`. The resulting string is stored in the pointer pointed by `res`. The required memory WILL NOT be preallocated in `res`.

2. `void concat_struct(concat_t *str);`

Concatenates the `str1` and `str2` fields of `str`, and stores the resulting string in its `res` field. The required memory WILL NOT be preallocated in the `res` field.



Here is a sample main and the expected output:

```
static void test_concat_strings(void)
{
    char str1[] = "I find your lack of faith...";
    char str2[] = " disturbing.";
    char *res = NULL;

    concat_strings(str1, str2, &res);
    printf("%s\n", res);
    free(res);
}

static void test_concat_struct(void)
{
    char str1[] = "These aren't the Droids";
    char str2[] = " you're looking for.";
    concat_t str = {
        .str1 = str1,
        .str2 = str2,
        .res = NULL
    };

    concat_struct(&str);
    printf("%s\n", str.res);
    free(str.res);
}

int main(void)
{
    test_concat_strings();
    test_concat_struct();
    return (0);
}
```

```
~/B-PDG-300> ./a.out
I find your lack of faith... disturbing.
These aren't the Droids you're looking for.
```




EXERCISE 3 - 1D ARRAY TO 2D ARRAY

Turn in: array_1d_to_2d.c in ex03/

Define the following functions:

1. `void array_1d_to_2d(const int *array, size_t height, size_t width, int ***res);`

It takes an array of integers as its `array` parameter, and uses it to create a bidimensional array of `height` lines and `width` columns. This new array must be stored in the pointer pointed to by `res`. The necessary memory space will **not** be allocated in `res` beforehand.

2. `void array_2d_free(int **array, size_t height, size_t width);`

Free a 2D array created by your `array_1d_to_2d` function.



Here is a sample `main` function and its expected output:

```
int main(void)
{
    int **array_2d;
    const int array_1d[42] = {
        0, 1, 2, 3, 4, 5,
        6, 7, 8, 9, 10, 11,
        12, 13, 14, 15, 16, 17,
        18, 19, 20, 21, 22, 23,
        24, 25, 26, 27, 28, 29,
        30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40, 41
    };

    array_1d_to_2d(array_1d, 7, 6, &array_2d);
    printf("array_2d[%d][%d] = %d\n", 0, 0, array_2d[0][0]);
    printf("array_2d[%d][%d] = %d\n", 6, 5, array_2d[6][5]);
    printf("array_2d[%d][%d] = %d\n", 4, 4, array_2d[4][4]);
    printf("array_2d[%d][%d] = %d\n", 0, 3, array_2d[0][3]);
    printf("array_2d[%d][%d] = %d\n", 3, 0, array_2d[3][0]);
    printf("array_2d[%d][%d] = %d\n", 4, 2, array_2d[4][2]);
    array_2d_free(array_2d, 7, 6);
    return (0);
}
```

```
~/B-PDG-300> ./a.out
array_2d[0][0] = 0
array_2d[6][5] = 41
array_2d[4][4] = 28
array_2d[0][3] = 3
array_2d[3][0] = 18
array_2d[4][2] = 26
```



EXERCISE 4 - FUNCTION POINTERS

Turn in: `print.c` in `ex04/`

Notes: The `action_t` type is defined in the provided `print.h` file.

Define the following functions:

1. `void print_normal(const char *str);`

Prints `str`, followed by a newline.

2. `void print_reverse(const char *str);`

Prints `str`, reversed, followed by a newline.

3. `void print_upper(const char *str);`

Prints `str` with every lowercase letter converted to uppercase, followed by a newline.

4. `void print_42(const char *str);`

Prints "42", followed by a newline.



Use `printf` OR `write` to display the strings, but not both at the same time!

5. `void do_action(action_t action, const char *str);`

Executes an action according to the `action` parameter:

- if the value of `action` is `PRINT_NORMAL`, the `print_normal` function is called with `str` as its parameter,
- if the value of `action` is `PRINT_REVERSE`, the `print_reverse` function is called with `str` as its parameter,
- if the value of `action` is `PRINT_UPPER`, the `print_upper` function is called with `str` as its parameter,
- if the value of `action` is `PRINT_42`, the `print_42` function is called with `str` as its parameter.



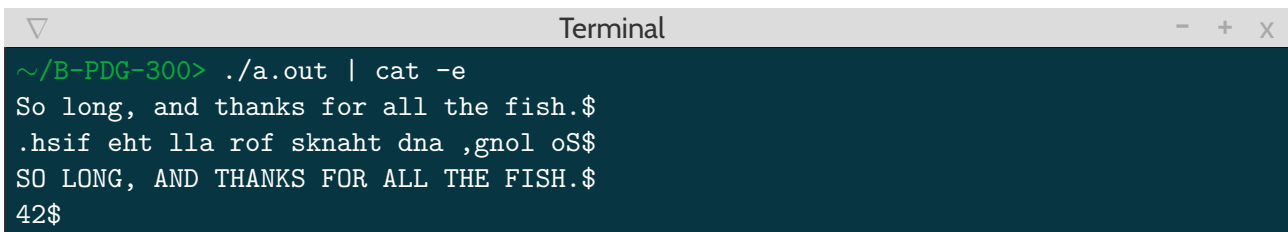
Of course, you **HAVE** to use function pointers. Chained `if ... else if ...` expressions or `switch` statements are **FORBIDDEN**.



Here is an example of a main function with the expected output:

```
int main(void)
{
    const char *str = "So long, and thanks for all the fish.";

    do_action(PRINT_NORMAL, str);
    do_action(PRINT_REVERSE, str);
    do_action(PRINT_UPPER, str);
    do_action(PRINT_42, str);
    return (0);
}
```

A terminal window titled "Terminal" with standard window controls (minimize, maximize, close). The prompt is ~/B-PDG-300>. The command ./a.out | cat -e is entered. The output is displayed on four lines: "So long, and thanks for all the fish.\$", ".hsif eht lla rof sknaht dna ,gnol oS\$", "SO LONG, AND THANKS FOR ALL THE FISH.\$", and "42\$".

```
~/B-PDG-300> ./a.out | cat -e
So long, and thanks for all the fish.$
.hsif eht lla rof sknaht dna ,gnol oS$
SO LONG, AND THANKS FOR ALL THE FISH.$
42$
```



EXERCISE 5 - ARRAYS

Turn in: `disp.c`, `sort.c`, `uniq.c` in `ex05/`

Define the following functions:

1. `void sort_int_array(int *array, size_t nmemb);`

Sorts an array of integer in ascending order.

2. `size_t uniq_int_array(int *array, size_t nmemb);`

Removes duplicates in an integer array. Only the first occurrence must remain. It returns the number of elements remaining.

3. `void disp_int_array(const int *array, size_t nmemb);`

Displays each integer in the array followed by a `'\n'` (including the last one).

4. `void sort_array(void *array, size_t nmemb, size_t size, int(*compar)(const void *, const void *));`

Sorts an array of `nmemb` elements of size `size` using given comparison function in ascending order. The comparison function returns a negative/null/positive value when the first object is inferior/equal/superior to the second object.

5. `size_t uniq_array(void *array, size_t nmemb, size_t size, int(*compar)(const void *, const void *));`

Removes duplicates in an array. Only the first occurrence must remain. Objects are equals if comparator return 0. It returns the number of elements remaining.

6. `void disp_array(const void *array, size_t nmemb, size_t size, void (*print)(const void *));`

Displays each element of the array using the display function given as parameter.



Here is a sample main function with the expected output:

```
static int test_comp(const void *a, const void *b)
{
    return (strcmp(*(char **)a, *(char **)b));
}

static void test_disp(const void *str)
{
    printf(" %s", *(char **)str);
}

int main(int argc, const char **argv)
{
    printf("argv:");
    disp_array(argv, argc, sizeof(*argv), &test_disp);
    printf("\n");
    argc = uniq_array(argv, argc, sizeof(*argv), &test_comp);
    printf("uniq:");
    disp_array(argv, argc, sizeof(*argv), &test_disp);
    printf("\n");
    sort_array(argv, argc, sizeof(*argv), &test_comp);
    printf("sort:");
    disp_array(argv, argc, sizeof(*argv), &test_disp);
    printf("\n");
    return (0);
}
```

```
~/B-PDG-300> ./a.out mi fa sol la mi re re mi fa sol sol sol re do
argv: ./a.out mi fa sol la mi re re mi fa sol sol sol re do
uniq: ./a.out mi fa sol la re do
sort: ./a.out do fa la mi re sol
```



EXERCISE 6 - RECURSIVE PYRAMID

Turn in: `pyramid.c` in `ex06/`

You are stuck at the top of a pyramid. Each room inside it leads to two neighboring rooms on the lower floor.

```
0
1 2
3 4 5
6 7 8 9
```

Thus, from room 0, one can access rooms 1 and 2. From room 2, one can reach rooms 4 and 5 and from room 4, we can go to rooms 7 and 8. The only thing in your possession is the map of the pyramid you're stuck in. It indicates the distance between rooms.

```
0
7 4
2 3 6
8 5 9 3
```

There are 7 meters between the top level and the left room, and only 4 between the top level and the right one.

Your goal is to find the **shortest path** to the pyramid's exit. In our example, that would be:

$$0 + 4 + 3 + 5 = 12$$

Write a `pyramid_path` function with the following prototype:

```
int pyramid_path(unsigned int size, const unsigned int **map);
```

The function returns the total distance traveled to get out of the pyramid. Its parameters are:

- `size`: the height of the pyramid
- `map`: a two-dimensional array containing the distances between rooms

In the previous example, the `map` parameter could be declared as follows:

```
int *pyramid[] = {
    { 0 },
    { 7, 4 },
    { 2, 3, 6 },
    { 8, 5, 9, 3 }
};
```



It doesn't really work, isn't it? Try to understand why.



Here's a more interesting pyramid:

```
      00
     95 64
    17 47 82
   18 35 87 10
  20 04 82 47 65
 19 01 23 75 03 34
 88 02 77 73 07 63 67
99 65 04 28 06 16 70 92
41 41 26 56 83 40 80 70 33
41 48 72 33 47 32 37 16 94 29
53 71 44 65 25 43 91 52 97 51 14
```




EXERCISE 7 - STRUCT AND UNION

Turn in: ping_pong.h in ex07/

Create the ping_pong.h file needed for the following code to compile and generate the expected output.

```
#include <stdlib.h>
#include <stdio.h>
#include "ex05.h"

int main(void)
{
    ping_t ping;

    ping.pong = 0;
    ping.ping.ping = 0xCAFE;
    printf("%d\n", sizeof(ping) == sizeof(ping.ping));
    printf("%d\n", sizeof(ping.ping.pong.ping) == sizeof(ping.ping.ping));
    printf("%d\n", sizeof(ping.pong) == 2 * sizeof(ping.ping.pong));
    printf("%d\n", sizeof(ping.ping.ping) == sizeof(ping.ping.pong.pong));
    printf("%08X\n", ping.pong);
    return EXIT_SUCCESS;
}
```

```
~/B-PDG-300> ls
ping_pong.h main.c
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 main.c
~/B-PDG-300> ./a.out
1
1
1
1
0000CAFE
```



EXERCISE 8 - PGM (POINTER GRAND MASTER)

Turn in: `whut.c` in `ex08/`

Notes: An example `whut.h` file is provided.

Define the following functions:

1. `int get_array_nb_elem(const int *ptr1, const int *ptr2);`

Each of the two pointers passed as parameters point to a different location of the same array of integers. This function returns the number of elements of the array between both pointers.

```
typedef struct whut_s
{
    ...
    int member;
    ...
} whut_t;
```

2. `whut_t *get_whut_ptr(const int *member);`



"..." means that any field could be inserted in the `whut_s` structure before and after the `member` field. A sample `whut_s` structure is provided in the `whut.h` file.

The `get_whut_ptr` function has a single parameter: a pointer to the `member` field of a `whut_s` structure. It must return a pointer to the structure itself.

Here is a sample main function with the expected output:

```
int main(void)
{
    const int tab[1000] = {0};
    int nb_elem = get_array_nb_elem(&tab[666], &tab[708]);
    whut_t whut = { 0 };

    printf("There are %d elements between elements 666 and 708\n", nb_elem);
    if (&whut == get_whut_ptr(&whut.member))
        printf("You hacked reality!\n");
    return (0);
}
```

```
Terminal
~/B-PDG-300> ./a.out
There are 42 elements between elements 666 and 708
You hacked reality!
```