



B3 - C++ Pool

B-CPP-300

Day 02 Morning

pointers



2.0



Day 02 Morning

repository name: `cpp_d02m_$ACADEMICYEAR`
repository rights: `ramassage-tek`
language: `C`



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).

All your exercises will be compiled with the `-std=gnu11 -W -Wall -Wextra -Werror` flags, unless specified otherwise.

All output goes to the standard output, and must be ended by a newline, unless specified otherwise.



None of your files must contain a `main` function, unless specified otherwise.
We will use our own `main` functions to compile and test your code.

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.



Read the examples CAREFULLY. They might require things that weren't mentioned in the subject...



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](#).



EXERCISE 0 - ADD MUL – BASIC POINTERS

Turn in: add_mul.c

In a add_mul.c file, define the following functions:

1. `void add_mul_4param(int first, int second, int *sum, int *product);`

calculates the sum and product of the `first` and `second` parameters.

The sum is stored in the integer `sum` points to, and the product in the integer `product` points to.

2. `void add_mul_2param(int *first, int *second);`

calculates the sum and product of the `first` and `second` parameters.

The sum is stored in the integer `first` points to and the product is stored in the integer `second` points to.

Here is a sample main function with the expected output:

```
static void test_4_params(void)
{
    int first = 5;
    int second = 6;
    int sum;
    int product;

    add_mul_4param(first, second, &sum, &product);
    printf("%d + %d = %d\n", first, second, sum);
    printf("%d * %d = %d\n", first, second, product);
}

static void test_2_params(void)
{
    int first = 5;
    int second = 6;
    int add_res = first;
    int mul_res = second;

    add_mul_2param(&add_res, &mul_res);
    printf("%d + %d = %d\n", first, second, add_res);
    printf("%d * %d = %d\n", first, second, mul_res);
}

int main(void)
{
    test_4_params();
    test_2_params();
    return (0);
}
```

```
Terminal
~/B-CPP-300> ./a.out
5 + 6 = 11
5 * 6 = 30
5 + 6 = 11
5 * 6 = 30
```



EXERCISE 1 - MEM PTR - POINTERS AND MEMORY

Turn in: `mem_ptr.c`

Notes: The `str_op_t` structure is defined in the provided `mem_ptr.h` file.

In a `mem_ptr.c` file, define the following functions:

1. `void add_str(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 add_str_struct(str_op_t *str_op);`

behaves like the `add_str` function.

Concatenates the `str1` and `str2` fields of `str_op`, and stores the resulting string in its `res` field.

Here is a sample main and the expected output:

```
static void test_add_str(void)
{
    char *str1 = "Hey, ";
    char *str2 = "it works!";
    char *res;

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

static void test_add_str_struct(void)
{
    char *str1 = "Hey, ";
    char *str2 = "it works!";
    str_op_t str_op;

    str_op.str1 = str1;
    str_op.str2 = str2;
    add_str_struct(&str_op);
    printf("%s\n", str_op.res);
}

int main(void)
{
    test_add_str();
    test_add_str_struct();
    return (0);
}
```

```
Terminal
~/B-CPP-300> ./a.out
Hey, it works!
Hey, it works!
```



EXERCISE 2 - TAB TO 2DTAB - POINTERS AND MEMORY

Turn in: `tab_to_2dtab.c`

In a `tab_to_2dtab.c` file, define the following function:

```
void tab_to_2dtab(const int *tab, int length, int width, int ***res);
```

It takes an array of integers as its `tab` parameter, and uses it to create a bidimensional array of `length` 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.

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

```
int main(void)
{
    int **tab_2d;
    int tab[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};

    tab_to_2dtab(tab, 7, 6, &tab_2d);
    printf("tab2[%d][%d] = %d\n", 0, 0, tab_2d[0][0]);
    printf("tab2[%d][%d] = %d\n", 6, 5, tab_2d[6][5]);
    printf("tab2[%d][%d] = %d\n", 4, 4, tab_2d[4][4]);
    printf("tab2[%d][%d] = %d\n", 0, 3, tab_2d[0][3]);
    printf("tab2[%d][%d] = %d\n", 3, 0, tab_2d[3][0]);
    printf("tab2[%d][%d] = %d\n", 4, 2, tab_2d[4][2]);
    return (0);
}
```

```

Terminal
~/B-CPP-300> ./a.out
tab2[0][0] = 0
tab2[6][5] = 41
tab2[4][4] = 28
tab2[0][3] = 3
tab2[3][0] = 18
tab2[4][2] = 26
```



EXERCISE 3 - FUNC PTR – FUNCTION POINTERS

Turn in: `func_ptr.c`, `func_ptr.h`

Notes: The `action_t` type is defined in the provided `func_ptr_enum.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!

You must include the `func_ptr_enum.h` file in `func_ptr.h`.

Define the following function:

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)
{
    char *str = "I'm using function pointers!";

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

```
~/B-CPP-300> ./a.out | cat -e
I'm using function pointers!$
!sretniop noitcnuf gnisu m'I$
I'M USING FUNCTION POINTERS!$
42$
```




EXERCISE 4 - CAST MANIA

Turn in: `add.c`, `div.c`, `castmania.c`

Notes: All structures and enumerations are defined in the provided `castmania.h` file.

Implement the following functions in `div.c`:

1. `int integer_div(int a, int b);`

performs a euclidian division between `a` and `b` and returns the result.
If the value of `b` is 0, the function returns 0.

2. `float decimale_div(int a, int b);`

performs a decimal division between `a` and `b` and returns the result.
If the value of `b` is 0, the function returns 0.

3. `void exec_div(division_t *operation);`

performs an euclidian or a decimal division, depending on the value of the `div_type` field of `operation`.
The `div_op` field is a generic pointer.
If the value of `div_type` is `INTEGER`, it points to a `integer_op_t` structure.
If the value of `div_type` is `DECIMALE`, it points to a `decimale_op_t` structure.

The operands for the division are the fields of the `div_op` structure.
The result of the division must be stored in the `res` field of the `div_op` structure.

Implement the following functions in `add.c`:

4. `int normal_add(int a, int b);`

calculates the sum of `a` and `b` and returns the result.

5. `int absolute_add(int a, int b);`

calculates the sum of the absolute value of `a` and the absolute value of `b` and returns the result.

6. `void exec_add(addition_t *operation);`

performs a normal or an absolute addition, depending on the value of the `add_type` field of `operation`.

The operands for the addition are the fields of the `add_op` structure.
The result of the addition must be stored in the `res` field of the `add_op` structure.

Implement the following functions in `castmania.c`:

7. `void exec_operation(instruction_type_t instruction_type, void *data);`

executes an addition or a division according to the value of `instruction_type`.
In either case, `data` will point to a `instruction_t` structure.

- if the value of `instruction_type` is `ADD_OPERATION`, the `exec_add` function should be called.
The `operation` field of the structure pointed to by `data` will point to a `addition_t` structure.



- if the value of `instruction_type` is `DIV_OPERATION`, the `exec_div` function should be called. The `operation` field of the structure pointed to by `data` will point to a `division_t` structure.
- if the value of the `output_type` field of the `data` structure is `VERBOSE`, the result of the operation has to be displayed.

8. `void exec_instruction(instruction_type_t instruction_type, void *data);`

executes an action depending on the value of `instruction_type`.

- if the value of `instruction_type` is `PRINT_INT`, `data` will point to an `int` that must be displayed.
- if the value of `instruction_type` is `PRINT_FLOAT`, `data` will point to a `float` that has to be displayed.
- otherwise, `exec_operation` must be called with `instruction_type` and `data` as parameters.

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

```
static void test_print(void)
{
    int i = 5;
    float f = 42.5;

    printf("Print i : ");
    exec_instruction(PRINT_INT, &i);
    printf("Print f : ");
    exec_instruction(PRINT_FLOAT, &f);
}

static void test_add_op(integer_op_t *int_op, instruction_t *inst)
{
    addition_t add;

    add.add_type = ABSOLUTE;
    add.add_op = *int_op;
    inst->operation = &add;
    printf("10 + 3 = ");
    exec_instruction(ADD_OPERATION, inst);
    printf("Indeed 10 + 3 = %d\n\n", add.add_op.res);
}

static void test_div_op(integer_op_t *int_op, instruction_t *inst)
{
    division_t div;

    div.div_type = INTEGER;
    div.div_op = int_op;
    inst->operation = &div;
    printf("10 / 3 = ");
    exec_instruction(DIV_OPERATION, inst);
    printf("Indeed 10 / 3 = %d\n\n", int_op->res);
}

static void test_operations(void)
{
    integer_op_t int_op;
    instruction_t inst;

    int_op.a = 10;
    int_op.b = 3;
```



```
        inst.output_type = VERBOSE;
        test_add_op(&int_op, &inst);
        test_div_op(&int_op, &inst);
    }

    int main(void)
    {
        test_print();
        printf("\n");
        test_operations();
        return (0);
    }
```

```
Terminal
~/B-CPP-300> ./a.out
Print i : 5
Print f : 42.500000

10 + 3 = 13
Indeed 10 + 3 = 13

10 / 3 = 3
Indeed 10 / 3 = 3
```



EXERCISE 5 - [ACHIEVEMENT] POINTER MASTER

Turn in: `ptr_tricks.c`

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

Define a `get_array_nb_elem` function with the following prototype:

```
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.

Define a `get_struct_ptr` function with the following prototype:

```
typedef struct s_whatever
{
    ...
    int member;
    ...
} whatever_t;
```

```
2. whatever_t *get_struct_ptr(const int *member_ptr);
```



“...” means that any field could be inserted in the `whatever_s` structure before and after the `member` field.

A sample `whatever_s` structure is provided in the `ptr_tricks.h` file.

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



Here is a sample main function with the expected output:

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

    printf("There are %d elements bandween elements 666 and 708\n", nb_elem);
    return (0);
}
```

```
Terminal
~/B-CPP-300> ./a.out
There are 42 elements bandween elements 666 and 708
```

```
int main(void)
{
    whatever_t test;
    whatever_t *ptr = get_struct_ptr(&test.member);

    if (ptr == &test)
        printf("It works!\n");
    return (0);
}
```

```
Terminal
~/B-CPP-300> ./a.out
It works!
```