B3- C++ Pool

B-PAV-242

Day 02 - Morning

Pointers and Memory

v1.7

Day 02 - Morning

Pointers and Memory

repository name:	cpp_d02m
repository rights:	ramassage-tek
language:	C
group size:	1

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- 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).

If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercises because you're lazy, and leave at 2PM, you WILL have problems. Do NOT tempt the devil.

Every function implemented in a header, or unprotected header, leads to 0 to the exercise.
Every class must possess a constructor and a destructor.

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

Any use of "friend" will result in the grade -42 no questions asked.

To avoid compilation problems, please include necessary files within your headers.

Please note that none of your files must contain the main function except when explicitly asked. We will use our own main function to compile and test your code.

Exercise 00

Add Mul - Basic Pointers

repository subdir:	/ex00
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	mul_div.c
forbidden functions:	none
points:	3

Create the **add_mul_4param** function with the following prototype:

```
void add_mul_4param(int first, int second, int *add, int *mul);
```

This function calculates the sum of the '**first**' and '**second**' parameters and stores the result in the integer pointed to by '**add**'. This function also stores the result of multiplying '**first**' and '**second**' in the integer pointed to by '**mul**'.

Create the **add_mul_2param** function with the following prototype:

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

This function adds the integers pointed to by '**first**' and '**second**'. It also multiplies the integers pointed to by '**first**' and '**second**'.
The result of the addition is stored in the integer pointed to by '**first**'.
The result of the multiplication is stored in the integer pointed to by '**second**'.

Here is an example of code using these functions:

```
int    main(void)
{
    int   first;
    int   second;
    int   add_res;
    int   mul_res;

    first = 5;
    second = 6;

    add_mul_4param(first, second, &add_res, &mul_res);
    printf("%d + %d = %d\n", first, second, add_res);
    printf("%d * %d = %d\n", first, second, mul_res);

    add_res = first;
    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);

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
5 + 6 = 11
5 * 6 = 30
5 + 6 = 11
5 * 6 = 30
```

Exercise 01

Mem Ptr - Pointers and Memory

repository subdir:	/ex01
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	mem_ptr.c
forbidden functions:	none
points:	3

The **t_str_op** structure is in the provided **mem_ptr.h** file.
Create the **add_str** function with the following prototype:

```
void add_str(char *str1, char *str2, char **res);
```

This function concatenates the **'str1'** and **'str2'** strings. The resulting string is stored in the pointer pointed to by **'res'**. The required memory WILL NOT be preallocated in 'res'.

Create the **add_str_struct** function with the following prototype:

```
void add_str_struct(t_str_op *str_op)
```

This function has the same behavior as the **add_str** function. It concatenates the **'str1'** and **'str2'** strings that are contained in the structure pointed to by **str_op**. The resulting string has to be stored in the **'res'** field of the structure pointed to by **'str_op'**.

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

```
int     main(void)
{
    char    *str1 = "Hi, ";
    char    *str2 = "It works!";
    char    *res;
    t_str_op     str_op;

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

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

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
Hi, it works!
Hi, it works!
```

Exercise 02

Tab to 2dTab - Pointers and Memory

repository subdir:	/ex02
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	tab_to_2dtab.c
forbidden functions:	none
points:	4

Create the **tab_to_2dtab** function with the following prototype:

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

This function takes an array of integers as its first parameter.
It creates a bi-dimensional array of '**length**' lines and '**width**' columns from '**tab**'.
The bi-dimensional array must be stored in the pointer pointed to by '**res**'. The required memory will not be allocated in '**res**' before calling the function.

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

```c
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-PAV-242> $>./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 03

Func Ptr - Function Pointers

repository subdir:	/ex03
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	func_ptr.h, func_ptr.c
forbidden functions:	none
points:	5

t_action is defined in the provided 'func_ptr_enum.h' file.

Write the following functions:

```c
void print_normal(char *str);
```

Displays the **'str'** string, which is the first parameter, followed by a newline.

```c
void print_reverse(char *str);
```

Displays the **'str'** string reversed, followed by a newline.

```c
void print_upper(char *str);
```

Displays the **'str'** string with every lowercase letter being converted to uppercase, followed by a newline.

```c
void print_42(char *str);
```

Displays the **"42"** string, followed by a newline.

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

Create the **do_action** function with the following prototype:

```c
void do_action(t_action action, char *str);
```

This function executes an action, depending on the **'action'** parameter.

- If the value of **'action'** is **'PRINT_NORMAL'**, the **'print_normal'** function must be called with **'str'** as its parameter.
- If the value of **'action'** is **'PRINT_REVERSE'**, the **'print_reverse'** function must be called with **'str'** as its parameter.
- If the value of **'action'** is **'PRINT_UPPER'**, the **'print_upper'** function must be called with **'str'** as its parameter.
- If the value of **'action'** is **'PRINT_42'**, the **'print_42'** function must be called with **'str'** as its parameter.

Of course, you MUST use function pointers. **'if/else if'**, **'switch'**,.. are forbidden.

You must include the **'func_ptr_enum.h'** file in your **'func_ptr.h** file.

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

```
int     main(void)
{
    char    *str = "I'm calling function pointers!";

    do_action(PRINT_NORMAL,  str);
    do_action(PRINT_REVERSE, str);
    do_action(PRINT_UPPER,   str);
    do_action(PRINT_42,      str);

    return (0)
}
```

Terminal

```
~/B-PAV-242> ./a.out | cat -e
I'm calling function pointers!$
!sretniop noitcnuf gnillac m'I$
I'M CALLING FUNCTION POINTERS!$
42$
```

Exercise 04

Cast Mania - Understanding and Mastering Casts

repository subdir:	/ex04
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	add.c, div.c, castmania.c
forbidden functions:	none
points:	5

Implement the following functions:

DIVISIONS (file 'div.c')

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

Performs an Euclidian division between **'a'** and **'b'** and then returns the result.
If the value of **'b'** is 0, the function returns 0.

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

Performs a decimal division between **'a'** and **'b'** and then returns the result.
If the value of **'b'** is 0, the function returns 0.

```
void exec_div(t_div *operation);
```

The first and only parameter of this function is a pointer to a **t_div** structure. Depending on the **'div_type'** field value, the function performs either integral or decimal division. The **'div_op'** field is a generic pointer. It points to a **t_integer_op** structure if the **'div_type'** value is **'INTEGER'**, or to a **t_decimale_op** structure if the value of **'div_type'** is **'DECIMALE'**.

The **'a'** and **'b'** integers to be used are the fields of the structure pointed to by **'div_op'**. In both cases, the result must be stored in the **'res'** field of the structure pointed to by **'div_op'**.

ADDITIONS (File 'add.c')

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

Calculates the sum of **'a'** and **'b'** and then returns the result.

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

Calculates the sum of the absolute values of **'a'** and **'b'** and then returns the result.

```
void exec_add(t_add *operation);
```

Takes a pointer to a **t_add** structure as its only argument, and depending on the value of **'add_type'**, will either perform normal or absolute addition.
The **'a'** and **'b'** integers are both of the **'add_op'** structure's fields. In both cases, the result must be stored in the **'add_op'** structure's **'res'** field.

CASTMANIA (File 'castmania.c')

```
void exec_operation(t_instruction_type instruction_type, void *data);
```

Executes an addition or division, depending on the value of '**instruction_type**'. In each case, '**data**' points to a '**t_instruction**' structure.

If the value of '**instruction_type**' is '**ADD_OPERATION**', you must perform an addition by using the '**exec_add**' function. In this specific case, the structure pointed to by '**data**''s '**operation**' field points to a '**t_add**' structure.

If the value of '**instruction_type**' is '**DIV_OPERATION**', you must perform a division by using the '**exec_div**' function. In this case, the structure pointed to by '**data**''s '**operation**' field points to a '**t_div**' structure.
For any other '**instruction_type**' value, nothing must be done.

If '**data**''s '**output_type**' has the '**VERBOSE**' value, the result of the operation must be displayed.

```
void exec_instruction(t_instruction_type instruction_type, void *data);
```

Executes an action depending on the value of '**instruction_type**'
If the value of '**instruction_type**' is '**PRINT_INT**', '**data**' points to an '**int**' that must be displayed.
If the value of '**instruction_type**' is '**PRINT_FLOAT**', '**data**' points to a '**float**' that must be displayed.
Otherwise, the '**exec_operation**' function should be called with '**instruction_type**' and '**data**' as parameters.

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

```
int                 main(void)
{
    int             i = 5
    float           f = 42.5;
    t_instruction   inst;
    t_add           add;
    t_div           div;
    t_integer_op    int_op;

    printf("Display i: ");
    exec_instruction(PRINT_INT, &i);
    printf("Display f: ");
    exec_instruction(PRINT_FLOAT, &f);
    printf("\n");

    int_op.a = 10;
    int_op.b = 3;
    add.add_type = ABSOLUTE;
    add.add_op    = int_op;
    inst.output_type = VERBOSE;
    inst.operation    = &add;
    printf("10 + 3 = ");
    exec_operation(ADD_OPERATION, &inst);
    printf("Indeed, 10 + 3 = %d\n\n", add.add_op.res);

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

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
Display i:  5
Display f:  42.500000

10 + 3 = 13
Indeed, 10 + 3 = 13

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

Exercise 05

Pointer Master - [Achievement] Pointer Steamroller

repository subdir:	/ex05
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	ptr_tricks.c
forbidden functions:	none
points:	2

An example **'ptr_tricks.h'** file is provided

Write the **get_array_nb_elem** function with the following prototype:

```
int get_array_nb_elem(int *ptr1, int *ptr2);
```

This function takes 2 pointers to int as parameters. Each of these pointers points to a different location within the same int array. The function returns the number of elements that are located between the two pointers.

Write the **get_struct_ptr** function with the following prototype:

```
typedef struct  s_whatever
{
  ...   // whatever
  int           member;
  ...   // whatever
}               t_whatever;

t_whatever      *get_struct_ptr(int *member_ptr);
```

An example of the **'s_whatever'** structure is given in the **'ptr_tricks.h'** file.
The **'get_struct_ptr'** function takes a pointer to a **'s_whatever'** structure's **'member'** field as parameter, and simply returns a pointer to this structure.

Here are examples of main functions 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 between the 666th element and 708th\n", nb_elem);

   return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
There are 42 elements between the 666th element and the 708th
```

```c
int main(void)
{
    t_whatever test;
    t_whatever *ptr;

    ptr = get_struct_ptr(&test.member);

    if (ptr == &test)
        printf("It works!\n");

    return (0);
}
```

Terminal

```
~/B-PAV-242> /a.out
It works!
```