B3 - C++ Pool

B-PAV-242

Day 02

Morning

KOALA

42.0

EPITECH.
L'ECOLE DE L'INNOVATION ET DE
L'EXPERTISE INFORMATIQUE

Day 02

binary name:	no binary
group size:	1
repository name:	cpp_d02m
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,...).

General Setpoints

READ THESE CAREFULLY
You will have no possible excuse if you end up with a 0 because you didn't follow one of these.

If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercices 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 for the exercise.

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

THINK. Please.

THINK. For Odin's sake.

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

Please note that 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.

This subject may be modified up to one hour before turn-in time!

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 0 - ADD MUL – BASIC POINTERS

	Exercise: 00	points : 3
	Add Mul - Basic Pointers	
Turn-in directory: `cpp_d02m/ex00`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `mul_div.c`		
Notes: `None`		
Forbidden functions: `None`		

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

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

Calculates the sum of the `first` and `second` parameters and stores the result in the integer `add` points to. Calculates the product of the `first` and `second` parameters and stores the result in the integer `mul` points to.

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

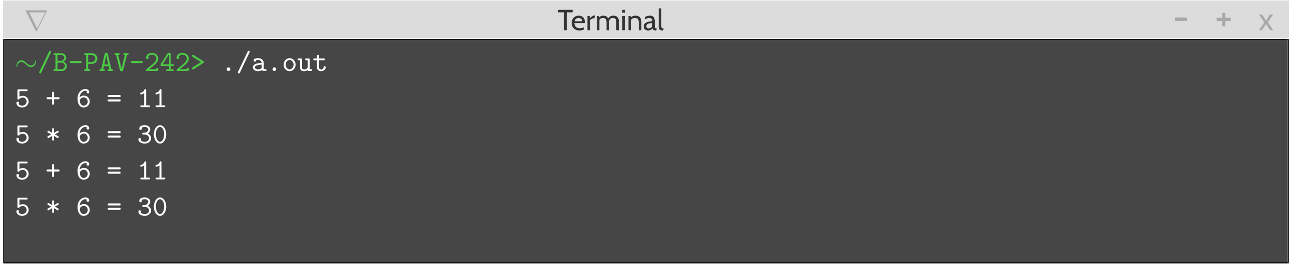
Calculates the sum and product of the `first` and `second` parameters.

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

Here is a sample main function with the expected output:

```c
int main(void)
{
  int first = 5;
  int second = 6;

  int add_res;
  int mul_res;

  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);
}
```

main.c



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

Exercise 1 - Mem Ptr – Pointers and memory

	Exercise: 01	points : 3
Mem Ptr - Pointers and memory		
Turn-in directory: `cpp_d02m/ex01`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `mem_ptr.c`		
Notes: `The ‘t_str_op‘ structure is in the provided ‘mem_ptr.h‘ file`		
Forbidden functions: `None`		

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

```
void add_str(char *str1, 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`.

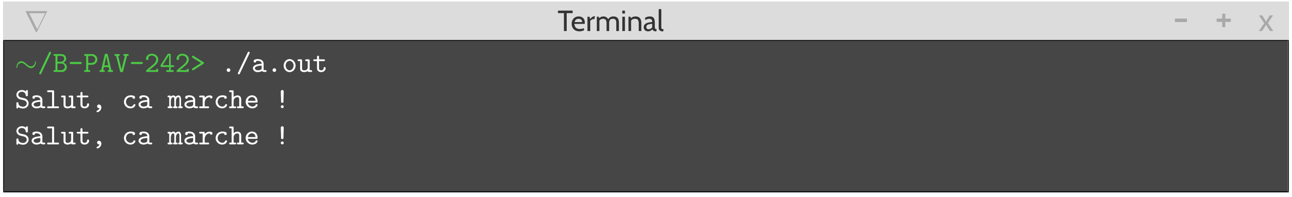
```
void add_str_struct(t_str_op *str_op);
```

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:

```
int main(void)
{
  char *str1 = "Salut,␣";
  char *str2 = "ca␣marche␣!";
  char *res;

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

  t_str_op str_op;

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

  return (0);
}
```

main.c



```
~/B-PAV-242> ./a.out
Salut, ca marche !
Salut, ca marche !
```

Exercise 2 - Tab to 2dTab – Pointers and memory

	Exercise: 02	points : 4
Tab to 2dTab - Pointers and memory		
Turn-in directory: `cpp_d02m/ex02`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `tab_to_2dtab.c`		
Notes: `None`		
Forbidden functions: `None`		

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

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

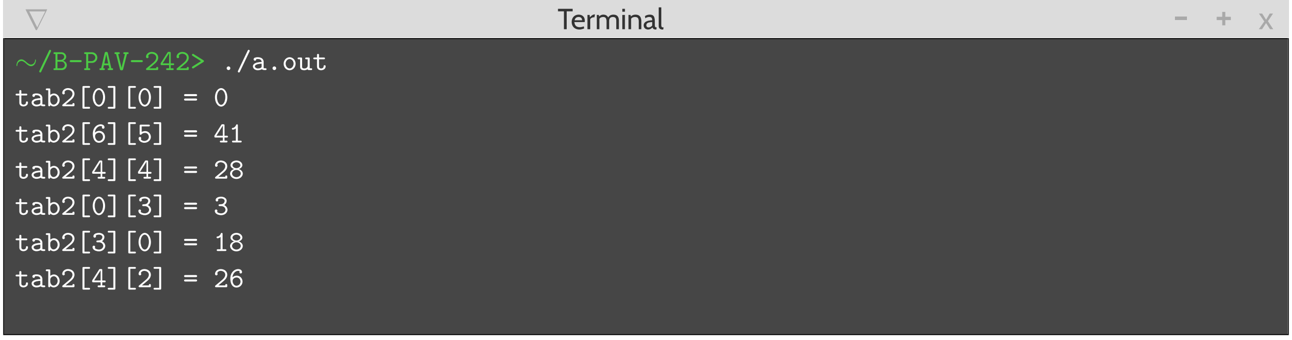
It takes an array of integers as its `tab` parameter, and uses it to create a bidimensional array of `length` lines and `width` columns.

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:

```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);
}
```

main.c



```
∼/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 3 - Func Ptr – Function pointers

Exercise: 03	points : 5
Func Ptr - Function pointers	
Turn-in directory: cpp_d02m/ex03	
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror
Makefile: No	Rules: n/a
Files to turn in: func_ptr.c, func_ptr.h	
Notes: ‘t_action‘ is defined in the provided ‘func_ptr_enum.h‘ file	
Forbidden functions: None	

Define the following functions:

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

Prints str, followed by a newline.

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

Prints str, reversed, followed by a newline.

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

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

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

Prints “42”, followed by a newline.

Use printf OR write to display the strings

You must include the `func_ptr_enum.h` file in `func_ptr.h`.
Define the following function:

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

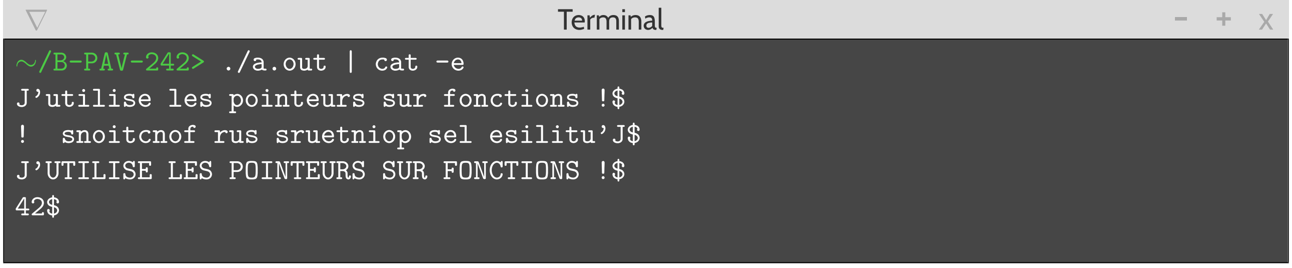
Executes an action according to the `action` parameter:

- 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:

```c
int main(void)
{
  char *str = "J'utilise␣les␣pointeurs␣sur␣fonctions␣!";

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

  return (0);
}
```

main.c



```
~/B-PAV-242> ./a.out | cat -e
J'utilise les pointeurs sur fonctions !$
! snoitcnof rus sruetniop sel esilitu'J$
J'UTILISE LES POINTEURS SUR FONCTIONS !$
42$
```

Exercise 4 - Cast Mania – Understanding and mastering casts

	Exercise: 04	points : 5
Cast Mania - Understanding and mastering casts		
Turn-in directory: `cpp_d02m/ex04`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `add.c, div.c, castmania.c`		
Notes: `All structures and enumerations are defined in the provided ‘castmania.h‘ file`		
Forbidden functions: `None`		

Implement the following functions in `div.c`:

```
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.

```
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.

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

Performs a 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 `t_integer_op` structure.
If the value of `div_type` is `DECIMALE`, it points to a `t_decimale_op` 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:

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

Calculates the sum of a and b and returns the result.

```
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.

```
void exec_add(t_add *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:

```
void exec_operation(t_instruction_type 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 t_instruction 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 t_add 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 t_div structure.
- If the value of the output_type field of the data structure is VERBOSE, the result of the operation has to be displayed.

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

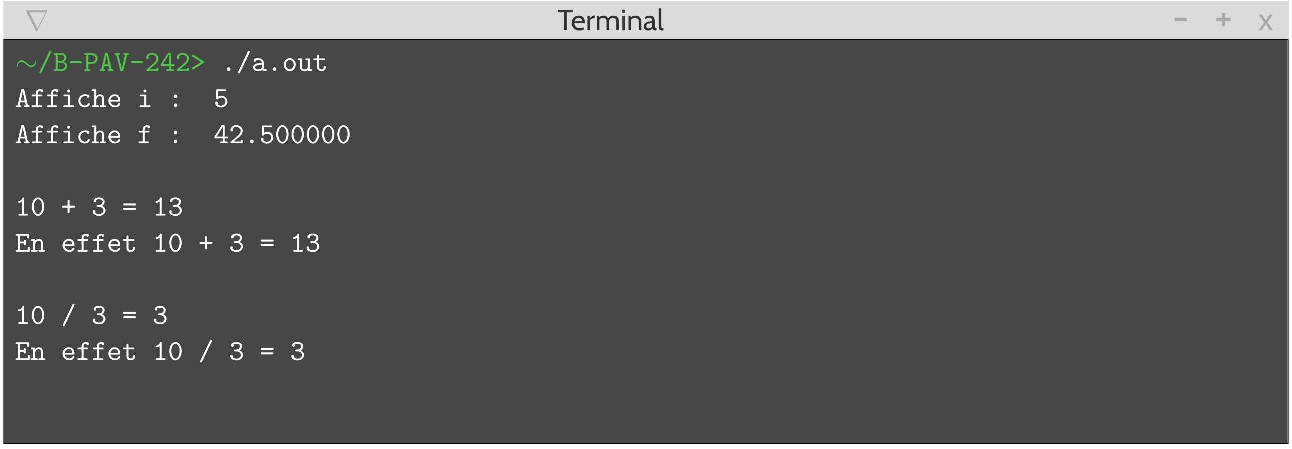
Executes an action depending on the value of instruction_type.

- If the value of instruction_type is PRINT_INT, data 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:

```c
int main(void)
{
  int i = 5;
  printf("Affiche␣i␣:␣");
  exec_instruction(PRINT_INT, &i);

  float f = 42.5;
  printf("Affiche␣f␣:␣");
  exec_instruction(PRINT_FLOAT, &f);

  printf("\n");

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

  t_add add;
  add.add_type = ABSOLUTE;
  add.add_op = int_op;

  t_instruction inst;
  inst.output_type = VERBOSE;
  inst.operation = &add;

  printf("10␣+␣3␣=␣");
  exec_instruction(ADD_OPERATION, &inst);

  printf("En␣effet␣10␣+␣3␣=␣%d\n\n", add.add_op.res);

  t_div div;
  div.div_type = INTEGER;
  div.div_op = &int_op;

  inst.operation = &div;

  printf("10␣/␣3␣=␣");
  exec_instruction(DIV_OPERATION, &inst);

  printf("En␣effet␣10␣/␣3␣=␣%d\n\n", int_op.res);

  return (0);
}
```

main.c



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

10 + 3 = 13
En effet 10 + 3 = 13

10 / 3 = 3
En effet 10 / 3 = 3
```

Exercise 5 - Pointer Master – [Achievement] Pointer Steamroller

KOALA	Exercise: 05	points : 2
Pointer Master - [Achievement] Pointer Steamroller		
Turn-in directory: cpp_d02m/ex05		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in: ptr_tricks.c		
Notes: An example ‘ptr_tricks.h‘ file is provided		
Forbidden functions: None		

Define a get_array_nb_elem function with the following prototype:

```c
int get_array_nb_elem(int *ptr1, 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:

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

t_whatever is defined like so:

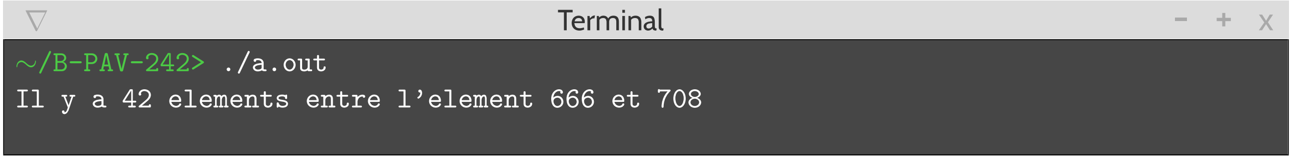
```c
typedef struct s_whatever
{
    ...
    int member;
    ...
} t_whatever;
```

“...” means that any field could be inserted in the s_whatever structure before and after the member field. A sample s_whatever 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 s_whatever structure. It must return a pointer to the structure itself.

Here is a sample main function with the expected output:

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

  printf("Il␣y␣a␣%d␣elements␣entre␣l'element␣666␣et␣708\n", nb_elem);

  return 0;
}
```

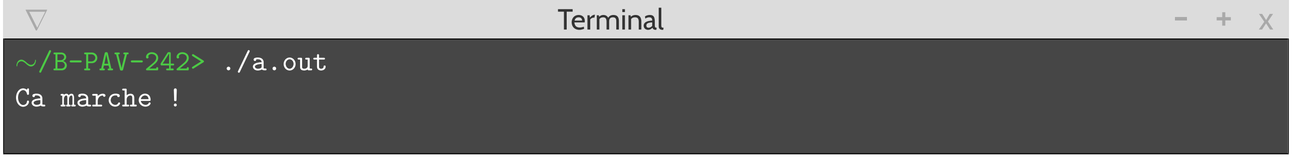
main.c



```
~/B-PAV-242> ./a.out
Il y a 42 elements entre l'element 666 et 708
```

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

  if (ptr == &test)
    printf("Ca␣marche␣!\n");

  return 0;
}
```

main.c



```
~/B-PAV-242> ./a.out
Ca marche !
```