B3 - C++ Pool

B-CPP-300

Day 02

Afternoon

1.3

{EPITECH.}

Day 02

group size: 1
repository name: cpp_d02a
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 `-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...

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.

THINK. Please.

T.H.I.N.K., by Odin!

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:

```c
#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 - Simple List

Turn in: simple_list.c
Notes: The simple_list.h file is provided. You must use it without modifying it.

The purpose of this exercise is to create a set of functions to manipulate a list.
We will consider a list as the following:

```c
typedef struct node
{
        double value;
        struct node *next;
} node_t;

typedef node_t *list_t;
```

An empty list is represented by a NULL pointer.
Let's define the following type, representing a boolean:

```c
typedef enum BOOL
{
        FALSE,
        TRUE
} bool_t;
```

Implement the following functions:

```c
01. unsigned int list_get_size(list_t list);
```

returns the number of elements in the list.

```c
02. bool_t list_is_empty(list_t list);
```

returns TRUE if the list is empty, FALSE otherwise.

```c
03. void list_dump(list_t list);
```

displays every element in the list, separated by new-line characters.
Use the default display of printf (%f) with no particular precision.

```c
04. bool_t list_add_elem_at_front(list_t *front_ptr, double elem);
```

adds a new node at the beginning of the list with elem as its value.
The function returns FALSE if it cannot allocate memory for the new node, TRUE otherwise.

```c
05. bool_t list_add_elem_at_back(list_t *front_ptr, double elem);
```

adds a new node at the end of the list with elem as its value.
The function returns FALSE if it cannot allocate memory for the new node, TRUE otherwise.

```
06. bool_t list_add_elem_at_position(list_t *front_ptr, double elem, unsigned int
    position);
```

adds a new node at the `position` position with `elem` as its value; returns `FALSE` if it cannot allocate memory for the new node or if `position` is out of bounds, `TRUE` otherwise.
If the value of `position` is 0, a call to this function is equivalent to a call to `list_add_elem_at_front`.

```
07. bool_t list_del_elem_at_front(list_t *front_ptr);
```

deletes the first node of the list; returns `FALSE` if the list is empty, `TRUE` otherwise.

```
08. bool_t list_del_elem_at_back(list_t *front_ptr);
```

deletes the last node of the list; returns `FALSE` if the list is empty, `TRUE` otherwise.

```
09. bool_t list_del_elem_at_position(list_t *front_ptr, unsigned int position);
```

deletes the node at the `position` position; returns `FALSE` if the list is empty or if `position` is out of bounds, `TRUE` otherwise.
If the value of `position` is 0, a call to this function is equivalent to a call to `list_del_elem_at_front`.

```
10. double list_get_elem_at_front(list_t list);
```

returns the value of the first node in the list; returns 0 if the list is empty.

```
11. double list_get_elem_at_back(list_t list);
```

returns the value of the last node in the list; returns 0 if the list is empty.

```
12. double list_get_elem_at_position(list_t list, unsigned int position);
```

returns the value of the node at the `position` position; returns 0 if the list is empty or if `position` is out of bounds.
If the value of `position` is 0, a call to this function is equivalent to a call to `list_get_elem_at_front`.

```
13. node_t *list_get_first_node_with_value(list_t list, double value);
```

returns a pointer to the first node of `list` having `value` as its value.
If no node matches `value`, the function returns `NULL`.

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

```c
static void populate_list(list_t *list_head)
{
        list_add_elem_at_back(list_head, 5.2);
        list_add_elem_at_back(list_head, 42.5);
        list_add_elem_at_back(list_head, 3.3);
}

static void test_size(list_t list_head)
{
        printf("There are %u elements in the list\n", list_get_size(list_head));
        list_dump(list_head);
}

static void test_del(list_t *list_head)
{
        list_del_elem_at_back(list_head);
        printf("There are %u elements in the list\n", list_get_size(*list_head));
        list_dump(*list_head);
}

int main(void)
{
        list_t list_head = NULL;

        populate_list(&list_head);
        test_size(list_head);
        test_del(&list_head);
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
There are 3 elements in the list
5.200000
42.500000
3.300000
There are 2 elements in the list
5.200000
42.500000
```

Exercise 1 - Simple BTree

Turn in: simple_btree.c
Notes: The simple_btree.h file is provided. You must use it without modifying it.

The purpose of this exercise is to create a set of functions to manipulate a binary tree.
We will consider a binary tree as the following:

```
typedef struct node
{
        double value;
        struct node *left;
        struct node *right;
} node_t;

typedef node_t *tree_t;
```

An empty tree is represented by a NULL pointer.

Implement the following functions:

```
01. bool_t btree_is_empty(tree_t tree);
```

returns TRUE if tree is empty, FALSE otherwise.

```
02. unsigned int btree_get_size(tree_t tree);
```

returns the number of nodes in tree.

```
03. unsigned int btree_get_depth(tree_t tree);
```

returns the depth of tree.

```
04. bool_t btree_create_node(tree_t *node_ptr, double value);
```

creates a new node with value as its value and places it at the location pointed to by node_ptr.
Returns FALSE if the node could not be added, TRUE otherwise.

```
05. bool_t btree_delete(tree_t *root_ptr);
```

deletes the tree pointed to by root_ptr in its entirety, including its children.
The function returns FALSE if the tree is empty, TRUE otherwise.

```
06. double btree_get_max_value(tree_t tree);
```

returns the maximal value in tree; returns 0 if the tree is empty.

```
07. double btree_get_min_value(tree_t tree);
```

returns the minimal value in tree; returns 0 if the tree is empty.

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

```
static void populate_left(tree_t tree)
{
        tree_t left_sub_tree = tree->left;

        btree_create_node(&(left_sub_tree->left), 30);
        btree_create_node(&(left_sub_tree->right), 5);
}

static void populate_tree(tree_t *tree)
{
        btree_create_node(&tree, 42.5);
        btree_create_node(&(tree->right), 100);
        btree_create_node(&(tree->left), 20);
        populate_left(*tree);
}

static void test_size(tree_t tree)
{
        unsigned int size = btree_get_size(tree);
        unsigned int depth = btree_get_depth(tree);

        printf("The tree's size is %u\n", size);
        printf("The tree's depth is %u\n", depth);
}

static void test_values(tree_t tree)
{
        double max = btree_get_max_value(tree);
        double min = btree_get_min_value(tree);

        printf("The tree's values range from %f to %f\n", min, max);
}

int main(void)
{
        tree_t tree = NULL;

        populate_tree(&tree);
        test_size(tree);
        test_values(tree);
        return (0);
}
```

Terminal

```
~/B-CPP-300> ./a.out
The tree's size is 5
The tree's depth is 3
The tree's values range from 5.000000 to 100.000000
```

Exercise 2 - Generic List

Turn in: generic_list.c
Notes: The generic_list.h file is provided. You must use it without modifying it.

The purpose of this exercise is to create a generic list.
The difference between this and the Simple List exercise is that a node is defined like this:

```
typedef struct node
{
        void *value;
        struct node *next;
} node_t;

typedef node_t * list_t;
```

The functions you have to implement are similar, with some minor differences in their prototypes:

```
unsigned int list_get_size(list_t list);
bool_t list_is_empty(list_t list);
bool_t list_add_elem_at_front(list_t *front_ptr, void *elem);
bool_t list_add_elem_at_back(list_t *front_ptr, void *elem);
bool_t list_add_elem_at_position(list_t *front_ptr, void *elem, unsigned int pos);
bool_t list_del_elem_at_front(list_t *front_ptr);
bool_t list_del_elem_at_back(list_t *front_ptr);
bool_t list_del_elem_at_position(list_t *front_ptr, unsigned int position);
void list_clear(list_t *front); // Releases all nodes in the list and
                                // makes 'front_ptr' point to an empty list
void *list_get_elem_at_front(list_t list);
void *list_get_elem_at_back(list_t list);
void *list_get_elem_at_position(list_t list, unsigned int position);
```

Only two functions truly differ:

```
typedef void (*value_displayer_t)(void *value);
void list_dump(list_t list, value_displayer_t val_disp);
```

list_dump now takes a value_displayer_t function pointer as its second parameter.
Using the function pointed to by val_disp, it is now possible to display the value of each node, followed by a newline.

```
typedef int (*value_comparator_t)(void *first, void *second);
node_t *list_get_first_node_with_value(list_t list, void *value,
                                       value_comparator_t val_comp);
```

list_get_first_node_with_value now takes a value_comparator_t function pointer as its second parameter, which lets you compare two values of the list.
The comparison function returns a positive value if first is greater than second, a negative value if second is greater than first, and 0 if first and second are equal.

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

```c
static void int_displayer(void *data)
{
        int value = *((int *)data);

        printf("%d\n", value);
}

static int int_compare(void *first, void *second)
{
         int val1 = *((int *)first);
         int val2 = *((int *)second);

         return (val1 - val2);
}

static void test_size(list_t list_head)
{
         printf("There are %u elements in the list\n", list_get_size(list_head));
         list_dump(list_head, &int_displayer);
}

static void test_del(list_t *list_head)
{
         list_del_elem_at_back(list_head);
         printf("There are %u elements in the list\n", list_get_size(*list_head));
         list_dump(*list_head, &int_displayer);
}

int main(void)
{
         list_t list_head = NULL;
         int i = 5;
         int j = 42;
         int k = 3;

         list_add_elem_at_back(&list_head, &i);
         list_add_elem_at_back(&list_head, &j);
         list_add_elem_at_back(&list_head, &k);
         test_size(list_head);
         test_del(&list_head);
         return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
There are 3 elements in the list
5
42
3
There are 2 elements in the list
5
42
```

EXERCISE 3 - STACK

Turn in: stack.c, generic_list.c
Notes: The stack.h and generic_list.h files are provided. You must use them without modifying them.

A code built around another code is called a wrapper.
The purpose of this exercise is to create a stack based on the previously created generic list.

Reuse the generic_list.c file from the previous exercises without modifying it.

As you may have guessed, we will consider a stack as a list which has smart feature limitations. Therefore:

```
typedef list_t stack_t;
```

Implement the following functions:

```
01. unsigned int stack_get_size(stack_t stack);
```

returns the number of elements in the stack.

```
02. bool_t stack_is_empty(stack_t stack);
```

returns TRUE if the stack is empty, FALSE otherwise.

```
03. bool_t stack_push(stack_t *stack_ptr, void *elem);
```

pushes elem to the top of the stack; returns FALSE if the new element could not be pushed, TRUE otherwise.

```
04. bool_t stack_pop(stack_t *stack_ptr);
```

pops the top element off the stack; returns FALSE if the stack is empty, TRUE otherwise.

```
05. void *stack_top(stack_t stack);
```

returns the value of the element on top of the stack.

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

```c
int main(void)
{
        stack_t stack = NULL;
        int i = 5;
        int j = 4;
        int *data = NULL;

        stack_push(&stack, &i);
        stack_push(&stack, &j);
        data = (int *)stack_top(stack);
        printf("%d\n", *data);
        return (0);
}
```

Terminal

```
~/B-CPP-300> ./a.out
4
```

Exercise 4 - Queue

Turn in: `queue.c`, `generic_list.c`
Notes: The `queue.h` and `generic_list.h` files are provided. You must use them without modifying them.

The purpose of this exercise is to create a queue based on the previously created generic list.

> Reuse the `generic_list.c` file from the previous exercicses without modifying it.

As you may have guessed again, we will consider a queue as a list with some smart feature limitations. Therefore:

```c
typedef list_t queue_t;
```

Implement the following functions:

```c
01. unsigned int queue_get_size(queue_t queue);
```

returns the number of elements in the queue.

```c
02. bool_t queue_is_empty(queue_t queue);
```

returns `TRUE` if the queue is empty, `FALSE` otherwise.

```c
03. bool_t queue_push(queue_t *queue_ptr, void *elem);
```

pushes `elem` into the queue; returns `FALSE` if the new element cannot be pushed, `TRUE` otherwise.

```c
04. bool_t queue_pop(queue_t *queue_ptr);
```

pops the next element from the queue; returns `FALSE` if the queue is empty, `TRUE` otherwise.

```c
05. void *queue_front(queue_t queue);
```

returns the value of the next element in the queue.

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

```
int main(void)
{
        queue_t queue = NULL;
        int i = 5;
        int j = 4;
        int *data = NULL;

        queue_push(&queue, &i);
        queue_push(&queue, &j);
        data = (int *)queue_front(queue);
        printf("%d\n", *data);
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
5
```

Exercise 5 - Map

** Turn in: map.c, generic_list.c
Notes**: The map.h and generic_list.h files are provided. You must use them without modifying them.

The purpose of this exercise is to create a map (which you may know as an associative array) based on the previously create generic list.

Reuse the generic_list.c file from the previous exercises without modifying it.

Once again, you may have guessed it: we will consider a map as a list with some smart feature limitations. Therefore:

```c
typedef list_t map_t;
```

The remaining question you may have is: "What is a map a list of?".
Well, here's the answer:

```c
typedef struct pair
{
        void *key;
        void *value;
} pair_t;
```

Think about it...

Implement the following functions:

```c
01. unsigned int map_get_size(map_t map);
```

returns the number of elements in the map.

```c
02. bool_t map_is_empty(map_t map);
```

returns TRUE if the map is empty, FALSE otherwise.

Here comes the tricky part.

Because our map is generic, the key may contain any data type.
To be able to compare these data and know whether two keys are equal (among other things), we need a key comparator:

```
03. typedef int (*key_comparator_t)(void *first_key, void *second_key);
```

returns 0 if the keys are equal, a positive number if `first_key` is greater than `second_key`, and a negative number if `second_key` is greater than `first_key`.

If you remember correctly, our generic list uses the same function pointer system to find a node with a particular value.

The question now is "How can we make the function called by our list call the key comparison function when we cannot add new parameters?".
There are two solutions to this problem:

- a global variable,
- a wrapper around a global variable ;)

Because you love nice and maintainable code, you will obviously choose the second solution. Good.
Implement the following functions:

```
04. key_comparator_t key_cmp_container(bool_t store, key_comparator_t new_key_cmp);
```

holds a static `key_comparator_t`.
If `store` is set to `TRUE`, the value of the static variable must be set to `new_key_cmp`.
The function always returns the value of the static variable.
This simulates the behavior of a global variable: if you want to set its value, call this function with `TRUE` as its first parameter and the value as its second. If you want to access the value, call this function with `FALSE` as its argument and `NULL` as its second.

```
05. int pair_comparator(void *first, void *second);
```

compares the keys in each pair (the two parameters being values from the list which point to `pair_t`s).
Returns 0 if the keys are equal, a positive value if the key of `first` is greater than that of `second`, and a negative value if the key of `second` is greater than that of `first`.

Before we go back to our map, add a basic function to the generic list.

Implement this function in `generic_list.c`.

```
06. bool_t list_del_node(list_t *front_ptr, node_t *node_ptr);
```

deletes `node_ptr` from the list; returns `FALSE` if the node is not in the list, `TRUE` otherwise.
Now back to the map (in `map.c`):

```
07. bool_t map_add_elem(map_t *map_ptr, void *key, void *value);
```

adds `value` at the `key` index of the map.
If a value already exists at the `key` index, it is replaced by `value`.
`key_cmp` is to be called to compare the keys of the map.
Returns `FALSE` if the element could not be added, `TRUE` otherwise.

```
08. bool_t map_del_elem(map_t *map_ptr, void *key, key_comparator_t key_cmp);
```

deletes the value at the `key` index.
`key_cmp` is to be called to compare the keys of the map.
Returns `FALSE` if there is no value at the `key` index, `TRUE` otherwise.

```
09. void *map_get_elem(map_t map, void *key, key_comparator_t key_cmp);
```

returns the value held at the `key` index of the map.
If there is no value at the `key` index, returns `NULL`.
`key_cmp` is to be called to compare the keys of the map.

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

```c
int int_comparator(void *first, void *second)
{
        int val1 = *(int *)first;
        int val2 = *(int *)second;
        return (val1 - val2);
}

int main(void)
{
        map_t map = NULL;
        int first_key = 1;
        int second_key = 2;
        int third_key = 3;
        char *first_value = "first";
        char *first_value_rw = "first_rw";
        char *second_value = "second";
        char *third_value = "third";
        char **data = NULL;

        map_add_elem(&map, &first_key, &first_value, &int_comparator);
        map_add_elem(&map, &first_key, &first_value_rw, &int_comparator);
        map_add_elem(&map, &second_key, &second_value, &int_comparator);
        map_add_elem(&map, &third_key, &third_value, &int_comparator);
        data = (char **)map_get_elem(map, &second_key, &int_comparator);
        printf("The key [%d] maps to value [%s]\n", second_key, *data);
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
The key [2] maps to value [second]
```

EXERCISE 6 - TREE TRAVERSAL

Turn in: tree_traversal.c, stack.c, queue.c, generic_list.c
Notes: The tree_traversal.h, stack.h, queue.h and generic_list.h files are provided. You have to use them without modifying them.

The purpose of this exercise is to iterate over a tree in a generic way, using containers.
Here is how we'll define a tree:

```
typedef struct tree_node
{
        void *data;
        struct tree_node *parent;
        list_t children;
} tree_node_t;

typedef tree_node_t *tree_t;
```

- data is the data contained in the node,
- parent is a pointer to the parent node,
- children is a generic list of child nodes.

An empty tree is represented by a NULL pointer.

Implement the following functions:

```
01. bool_t tree_is_empty(tree_t tree);
```

returns TRUE if the tree is empty, FALSE otherwise.

```
02. typedef void (*dump_func_t)(void *data);
void tree_node_dump(tree_node_t *tree_node_t, dump_func_t dump_func);
```

displays the content of a node.
The first argument is a pointer to a node, and the second is a function pointer to a display function.

```
03. bool_t init_tree(tree_t *tree_ptr, void *data);
```

initializes tree_ptr by creating a root node holding data.
Returns FALSE if the root node could not be allocated, TRUE otherwise.

```
04. tree_node_t *tree_add_child(tree_node_t *tree_node, void *data);
```

adds a child node holding data to tree_node.
Returns a pointer to the child node, or NULL if the child node could not be added.

```
05. bool_t tree_destroy(tree_t *tree_ptr);
```

deletes tree_ptr, including all its children.
Resets tree_ptr to an empty tree.
Returns FALSE if it fails, TRUE otherwise.

To code the ultimate function, we need to define a generic container:

```c
typedef struct container
{
        void *container;
        push_func_t push_func;
        pop_func_t pop_func;
} container_t;

typedef bool_t (*push_func_t)(void *container, void *data);
typedef void *(*pop_func_t)(void *container);
```

`container_t` is a generic container.
The `container` field holds the adress of the actual container. `push_func` is a function pointer that inserts an element in the container.
`pop_func` is a function pointer that extracts an element from the container.

Here is the ultimate function you must implement:

```c
06. void tree_traversal(tree_t tree, container_t *container, dump_func_t dump_func);
```

iterates over `tree` and displays its content using `container` and `dump_func`.

To do this, each node of the tree has to insert its child nodes in the container, display itself, and start over with the next node, extrated from the container.

Output must go from left to right with a FIFO container and from right to left with a LIFO container, naturally.

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

```
void dump_int(void *data)
{
        printf("%d\n", *(int *)data);
}

bool_t generic_push_stack(void *container, void *data)
{
        return stack_push((stack_t *)container, data);
}

void *generic_pop_stack(void *container)
{
        void *data = stack_top(*(stack_t *)container);

        stack_pop((stack_t *)container);
        return data;
}

bool_t generic_push_queue(void *container, void *data)
{
        return queue_push((queue_t *)container, data);
}

void *generic_pop_queue(void *container)
{
        void *data = queue_front(*(queue_t *)container);

        queue_pop((queue_t *)container);
        return data;
}

static void test_depth(tree_t tree)
{
        container_t container;
        stack_t stack = NULL;

        printf("Depth walk:\n");
        container.container = &stack;
        container.push_func = &generic_push_stack;
        container.pop_func = &generic_pop_stack;
        tree_traversal(tree, &container, &dump_int);
}

static void test_width(tree_t tree)
{
        container_t container;
        queue_t queue = NULL;

        printf("Width walk:\n");
        container.container = &queue;
        container.push_func = &generic_push_queue;
        container.pop_func = &generic_pop_queue;
        tree_traversal(tree, &container, &dump_int);
}
```

```c
static void fill_tree(tree_t tree)
{
        int val_a = 1;
        int val_aa = 11;
        int val_b = 2;
        int val_c = 3;
        int val_ca = 31;
        int val_cb = 32;
        int val_cc = 33;
        tree_node_t node = NULL;

        node = tree_add_child(tree, &val_a);
        tree_add_child(node, &val_aa);
        tree_add_child(tree, &val_b);
        node = tree_add_child(tree, &val_c);
        tree_add_child(node, &val_ca);
        tree_add_child(node, &val_cb);
        tree_add_child(node, &val_cc);
}

int main(void)
{
        int val_0 = 0;
        tree_t tree = NULL;

        init_tree(&tree, &val_0);
        fill_tree(tree);
        test_depth(tree);
        test_width(tree);
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out
Depth walk:
0
3
33
32
31
2
1
11
Width walk:
0
1
2
3
11
31
32
33
```