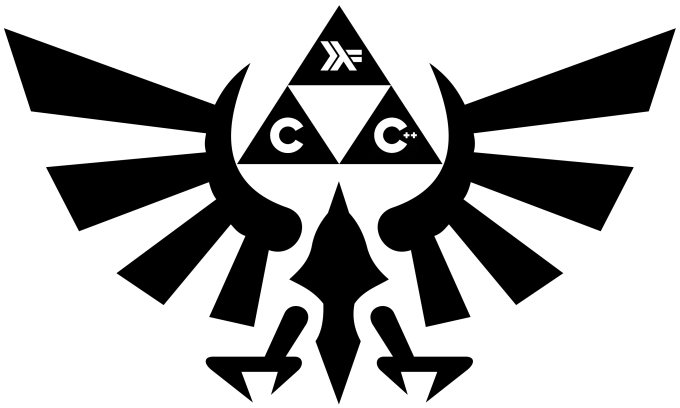
B3 - Paradigms Seminar

B-PDG-300

Day 04 PM

Marvin's List



2.3

Day 04 PM

language: C

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

All your exercises will be compiled with the `-std=gnu17 -Wall -Wextra` **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 to turn-in are to be at the root of the directory. So you **don't** have to put everything in an `exXX` folder.

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 - Integer List (the beginning)

Turn in: all the .c files required
Notes: The `int_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 of integer.
We will consider a list as the following:

```
typedef struct int_list_s {
    int value;
    struct int_list_s *next;
} int_list_t;
```

An empty list is represented by a `NULL` pointer.

Read carefully the first 4 exercises, they are related and you can factorise your code and write some of your functions recursively… 'member functional programming ?

Implement the following functions:

```
bool int_list_add_elem_at_back(int_list_t **front_ptr, int 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.

```
void int_list_dump(int_list_t *list);
```

Displays every element in the list, separated by new-line characters. Use the default display of `printf` (`%d`).

```
unsigned int int_list_get_size(int_list_t *list);
```

Returns the number of elements in the list.

```
bool int_list_is_empty(int_list_t *list);
```

Returns `true` if the list is empty, `false` otherwise.

```
void int_list_clear(int_list_t **front_ptr);
```

Deletes every node of the list; makes `front_ptr` point to an empty list.

Did you notice the type `bool` ?

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

```
int main(void)
{
    int_list_t *list_head = NULL;

    if (int_list_is_empty(list_head))
        printf("The list is empty\n");
    else
        printf("The list is not empty\n");
    int_list_add_elem_at_back(&list_head, 1);
    int_list_add_elem_at_back(&list_head, 2);
    printf("There are %u elements in the list\n", int_list_get_size(list_head));
    int_list_dump(list_head);
    int_list_clear(&list_head);
    printf("There are %u elements in the list\n", int_list_get_size(list_head));
    return 0;
}
```

Terminal

```
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
The list is empty
There are 2 elements in the list
1
2
There are 0 elements in the list
```

Exercise 1 - Integer List (not the beginning anymore)

Turn in: all the .c files required
Notes: The `int_list.h` file is provided. You must use it without modifying it.

You will continue the work you did in the previous exercise by creating a set of functions to add elements to the integer list.

Implement the following functions:

```
bool int_list_add_elem_at_front(int_list_t **front_ptr, int 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.

```
bool int_list_add_elem_at_position(int_list_t **front_ptr, int 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 `int_list_add_elem_at_front`.

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

```
int main(void)
{
    int_list_t *list_head = NULL;

    int_list_add_elem_at_front(&list_head, 1);
    int_list_add_elem_at_front(&list_head, 2);
    int_list_add_elem_at_position(&list_head, 3, 1);
    printf("There are %u elements in the list\n", int_list_get_size(list_head));
    int_list_dump(list_head);
    int_list_clear(&list_head);
    return 0;
}
```

```
Terminal
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
There are 3 elements in the list
2
3
1
```

Exercise 2 - Integer List (almost the end)

Turn in: all the .c files required
Notes: The `int_list.h` file is provided. You must use it without modifying it.

Now you must create functions to access elements from the list.

Implement the following functions:

```
int int_list_get_elem_at_front(int_list_t *list);
```

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

```
int int_list_get_elem_at_back(int_list_t *list);
```

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

```
int int_list_get_elem_at_position(int_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 `int_list_get_elem_at_front`.

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

```
int main(void)
{
    int_list_t *list_head = NULL;

    int_list_add_elem_at_front(&list_head, 1);
    int_list_add_elem_at_front(&list_head, 2);
    int_list_add_elem_at_position(&list_head, 3, 1);
    printf("There are %u elements in the list\n", int_list_get_size(list_head));
    printf("The first element is %d\n", int_list_get_elem_at_front(list_head));
    printf("The last element is %d\n", int_list_get_elem_at_back(list_head));
    int_list_clear(&list_head);
    return 0;
}
```

```
Terminal
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
There are 3 elements in the list
The first element is 2
The last element is 1
```

Exercise 3 - Integer List (the end)

Turn in: all the .c files required
Notes: The `int_list.h` file is provided. You must use it without modifying it.

Finally you must create functions to remove elements from the list.

Implement the following functions:

```
bool int_list_del_elem_at_front(int_list_t **front_ptr);
```

Deletes the first node of the list; returns `false` if the list is empty, `true` otherwise.

```
bool int_list_del_elem_at_back(int_list_t **front_ptr);
```

Deletes the last node of the list; returns `false` if the list is empty, `true` otherwise.

```
bool int_list_del_elem_at_position(int_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 `int_list_del_elem_at_front`.

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

```c
int main(void)
{
    int_list_t *list_head = NULL;

    int_list_add_elem_at_back(&list_head, 1);
    int_list_add_elem_at_back(&list_head, 2);
    int_list_add_elem_at_back(&list_head, 3);
    printf("There are %u elements in the list\n", int_list_get_size(list_head));
    int_list_del_elem_at_front(&list_head);
    int_list_dump(list_head);
    int_list_clear(&list_head);
    return 0;
}
```

```
Terminal
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
There are 3 elements in the list
2
3
```

Exercise 4 - Generic List

Turn in: all the .c files required
Notes: The `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 `Integer List` exercise is that a node is defined like this:

```
typedef struct list_s {
    void *value;
    struct node_s *next;
} 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 list_is_empty(list_t *list);
bool list_add_elem_at_front(list_t **front_ptr, void *elem);
bool list_add_elem_at_back(list_t **front_ptr, void *elem);
bool list_add_elem_at_position(list_t **front_ptr, void *elem, unsigned int pos);
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);
bool list_del_elem_at_front(list_t **front_ptr);
bool list_del_elem_at_back(list_t **front_ptr);
bool list_del_elem_at_position(list_t **front_ptr, unsigned int position);
void list_clear(list_t **front);
```

Only one functions truly differ:

```
typedef void (*value_displayer_t)(const 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.

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

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

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

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);
     list_clear(&list_head);
     return 0;
}
```

Terminal

```
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
There are 3 elements in the list
5
42
3
There are 2 elements in the list
5
42
```

EXERCISE 5 - STACK

Turn in: all the .c files required
Notes: The `stack.h` and `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 your functions from the previous exercises.

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 stack_is_empty(stack_t *stack);
```

Returns `true` if the stack is empty, `false` otherwise.

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

Rushes `elem` to the top of the stack; returns `false` if the new element could not be pushed, `true` otherwise.

```
04. bool 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_clear(stack_t **stack_ptr);
```

Remove every elements of the stack; makes `stack_ptr` point to an empty stack.

```
06. 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);
    stack_clear(&stack);
    return (0);
}
```

Terminal

```
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
4
```

Exercise 6 - Queue

Turn in: all the .c files required
Notes: The `queue.h` and `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 your functions from the previous exercises.

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

```
typedef list_t queue_t;
```

Implement the following functions:

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

Returns the number of elements in the queue.

```
02. bool queue_is_empty(queue_t *queue);
```

Returns `true` if the queue is empty, `false` otherwise.

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

Pushes `elem` into the queue; returns `false` if the new element cannot be pushed, `true` otherwise.

```
04. bool queue_pop(queue_t **queue_ptr);
```

Pops the next element from the queue; returns `false` if the queue is empty, `true` otherwise.

```
05. void queue_clear(queue_t **queue_ptr);
```

Remove every elements of the queue; makes `queue_ptr` point to an empty queue.

```
06. 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:

```c
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);
    queue_clear(&queue);
    return (0);
}
```

Terminal

```
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
5
```

Exercise 7 - Map

Turn in: all the .c files required
Notes: The `map.h` and `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 your functions from the previous exercises.

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

```
typedef list_t map_t;
```

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

```
typedef struct pair_s {
    void *key;
    void *value;
} pair_t;
```

Think about it…

Implement the following functions:

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

Returns the number of elements in the map.

```
02. bool 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:

```
typedef int (*key_comparator_t)(const void *first_key, const 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`.

```
03. bool map_add_elem(map_t **map_ptr, void *key, void *value, key_comparator_t
    key_cmp);
```

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.

```
04. 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.

```
05. bool 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.

```
06. void map_clear(map_t **map_ptr);
```

Deletes every pair of the map; makes `map_ptr` point to an empty map.

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

```c
int int_comparator(const void *first, const void *second)
{
    int val1 = *(const int *)first;
    int val2 = *(const 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);
    map_clear(&map);
    return (0);
}
```

```
Terminal
~/B-PDG-300> gcc -Wall -Wextra -std=gnu17 *.c
~/B-PDG-300> ./a.out
The key [2] maps to value [second]
```

Exercise 8 - Tree Traversal

Turn in: all the .c files required
Notes: The `tree.h`, `stack.h`, `queue.h` and `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_s {
    void *data;
    list_t *children;
} tree_t;
```

- `data` is the data contained in the node,
- `children` is a generic list of child nodes (`tree_t`).

An empty tree is represented by a `NULL` pointer.

Implement the following functions:

```
01. bool 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_t *node, 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 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_t *tree_add_child(tree_t *node, void *data);
```

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

```
05. bool 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:

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

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

`container_t` is a generic container. The `container` field holds the address 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:

```
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, extracted 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 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 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
int main(void)
{
    int val_0 = 0;
    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_t *tree = NULL;
    init_tree(&tree, &val_0);

    tree_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);

    test_depth(tree);
    test_width(tree);
    tree_destroy(&tree);

    return 0;
}
```

Terminal

```
~/B-PDG-300> gcc -Wall -Wextra -Werror *.c
~/B-PDG-300> ./a.out
Depth walk:
0
3
33
32
31
2
1
11
Width walk:
0
1
2
3
11
31
32
33
```