PRACTICE 1: Data Structures and Abstract Data Types

Goals

- Deepen the concept of ADT (Abstract Data Type).
- Learn to choose the appropriate data structure to implement a ADT.
- Encode its primitives and use it in a main program.
- Understand the `void *` datatype from C language.

Norms

Delivered programs must:

- Be written in ANSI C, following the established programming rules.
- Compile without errors or warnings including "-ansi" and "-pedantic" flags when compiling.
- Run without problems in a command console.
- Incorporate an adequate error control. It is justifiable that a program does not admit inadequate values, but not that it behaves abnormally with these values.
- Not show memory leaks when they are used.

Workplan

- Week 1: Exercise 1 completed and most important functions of Exercise 2. Each teacher will indicate in class if a delivery has to be made and how: paper, e-mail, Moodle, etc..
- Week 2: Exercise 2.
- Week 3: Exercise 2 (Optional).

The final delivery will be made through Moodle, scrupulously following the instructions indicated in the statement of practice 0 referring to the organization and nomenclature of files and projects. Remember that the compressed file that must be delivered must be named `Px_Prog2_Gy_Pz`, where 'x' is the number of the practice, 'y' the group of practices, and 'z' the number of the pair (example of delivery of pair 5 of group 2161 : `P1_Prog2_G2161_P05.zip`).

Upload dates to Moodle for the ZIP file are the following:

- Students on Continuous Evaluation, the week of **February 24th** (each group can make the delivery until 23:55 of the night before their practice class).
- Final Evaluation students, as specified in the regulations.

PART 1: CREATION OF THE ADT NODE AND GRAPH

In this practice, we will implement a graph of nodes. To do this, we will first begin by defining the type NODE (Node), and then work on the ADT GRAPH (Graph). Note that the contents of some of the files necessary for this practice are provided at the end of this file (see all the appendices).

EXERCISE 1.

Definition of the NODE data type (Node). Implementation: selection of data structure and implementation of primitives.

In this practice, a node will be represented by an id (a long int), a name (a fixed string of characters), a label (Label type), and a given number of connections (an int).

- In order to define the data structure necessary to represent the ADT NODE according to the methodology of hidden types seen in class, the following statement must be included in node.h:

```
typedef struct _Node Node;
```

Besides, in node.c you have to include the implementation of the abstract data type, this is its data structure _Node:

```
#define NAME_L 64   /*!< Maximum node name length */

typedef struct _Node {
    char name[NAME_L]; /*!<Node name */
    long id;            /*!<Node id */
    int nConnect;      /*!<Node number of connections */
    Label label;       /*!<Node state */
} Node;
```

- To be able to interact with data of type Node, at least the public functions of its interface (whose prototypes are declared in the file node.h, see Appendix 1) will be necessary. Write the code associated with its definition in the file node.c.
- File node.c may also include the definition of those private functions that facilitate the implementation of the public functions of the interface.

Checking the correctness of the type Node definition and its primitives.

You must create a file **p1_e1.c** that defines a program (named **p1_e1**) with the following operations:

- Initialise two nodes so that the first one is a node with name "first", id 111, and label WHITE; and the second node with name "second", id 222, and label WHITE.
- Print both nodes and then print a line break.
- Check if the two nodes are equal.
- Print the id of the first node along with an explanatory phrase (see example)
- Print the name of the second node (see example below)
- Copy the first node in the second.
- Print both nodes.
- Check if the two nodes are equal
- Free both nodes.

The program must manage the memory correctly.

The output of the program should be as follows:

```
[111, first, 0, 0][222, second, 0, 0]
Are they equal? No
Id of first node: 111
Name of second node: second
[111, first, 0, 0][111, first, 0, 0]
Are they equal? Yes
```

Appendix 1

Contents of file `node.h`.

```
/**
 * @file  node.h
 * @author Profesores Prog2
 * @date 29 January 2020
 * @version 1.0
 * @brief Library to manage TAD Node
 *
 * @details
 *
 * @see
 */
```

```
#ifndef NODE_H_
#define NODE_H_

#include "types.h"

/**
 * @brief Label to charactrize the node state (to be used in P2)
 *
 **/
typedef enum {
    WHITE, /*!< node not visited */
    BLACK,  /*!< nodo visited */
    ERROR_NODE   /*!< not valid node */
} Label;

/**
 * @brief Data structure to implement tha ADT node. To be defined in node.c
 *
 **/
typedef struct _Node Node;


/**
 * @brief Constructor. Initialize a node, reserving memory.
 *
 * This function initiates the node structure fields to "", -1, 0 and WHITE
 * respectively
 * @code
 * Node *n;
 * n = node_init ();
 * @endcode
 * @return Return the initialized node if
  * it was done correctly, otherwise return NULL and print the corresponding
  * string to the error in stderror
*/
Node * node_init ();

/**
 * @brief Destructor. Free the dynamic memory reserved for a node
 * @param n Node to free
 */
void node_free (void * n);
```

```
/**
 * @brief  Gets the node id
 * @param n Node address
 * @return  Returns the id of a given node, or -1 in case of error
 *  */
long node_getId (const Node * n);

/**
 * @brief Gets the node name
 * @param n Node address
 * @return Returns a pointer to the name of the node, or NULL in case of error
 */
const char* node_getName (Node * n);

/**
 * @brief Gets the number of connections of a given node.
 * @param n Node address
 * @return Returns the number of connections of a given node, or -1 in case
 * of error
 */
int node_getConnect (const Node * n);

/**
 * @brief Gets the label of a given node.
 * @param Node address
 * @return Returns the label of a given node, or -1 in case of error
 */
Label node_getLabel (const Node*n);


/**
 * @brief Modifies the label of a given node
 * @param n Node address
 * @param id New node label
 * @return Returns OK or ERROR in case of error
 */
Status node_setLabel (Node *n, Label l);

/**
 * @brief Modifies the id of a given node
 * @param n Node address
 * @param id New node id
 * @return Returns OK or ERROR in case of error
 */
```

```
Status node_setId (Node * n, const long id);

/**
 * @brief Modifies the name of a given node
 * @param n Node address
 * @param id New node name
 * @return  Returns OK or ERROR in case of error
 */
Status node_setName (Node *n, const char *name);

/**
 * @brief Modifies the number of connections of a given node
 * @param n Node address
 * @param id Number of connections for the node
 * @return  Returns OK or ERROR in case of error
 */
Status node_setNConnect (Node *n, const int cn);


/**
 * @brief Compares two nodes by the id and then the name.
 * @param n1,n2 Nodes to compare.
 * @return return an integer less than, equal to, or greater than zero
 * if n1 name is found, respectively, to be less than, to match, or be
 * greater than n2 name.
 */
int node_cmp (const void *n1, const void *n2);

/**
 * @brief Reserves memory for a node where it copies the data from the node src.
 * @code
 * Node *trg, *src;
 * src = node_init ();
 * trg = node_copy(src);
 * // .... aditional code ...
 * // free nodes
 * node_free (src);
 * node_free (trg);
 * @endcode
 * @param  Original node
 * @return Returns the address of the copied node if everything went well,
 * or NULL otherwise
 */
void * node_copy (const void *src);

/**
```

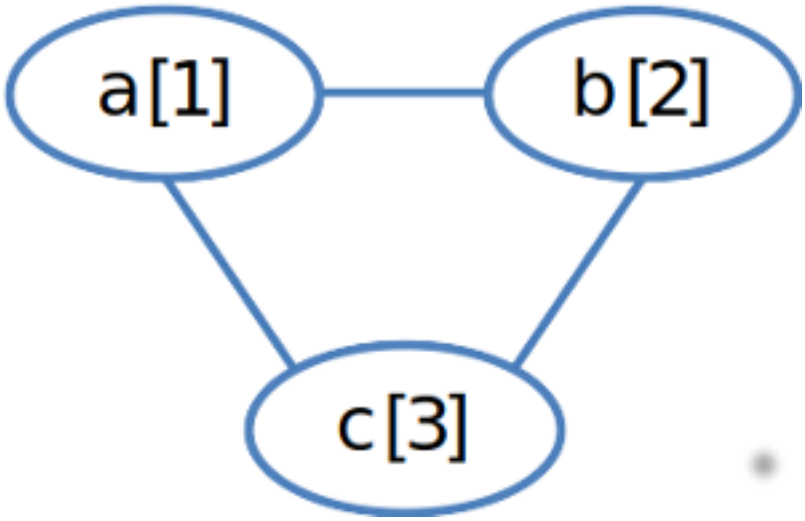
```
 * @brief Prints in pf the data of a node
 *
 * The format will be: [id, name, label, nConnect]
 * @code
 * Node *n;
 * n = node_init ();
 * node_print (stdout, n);
 * @endcode
 * @param pf File descriptor
 * @param n Node to be printted
 * @return Returns the number of characters that have been written successfully.
  * Checks if there have been errors in the output flow, in that case prints
  * an error message in stderror and returns -1.
 */
int node_print (FILE *pf, const void *n);

#endif /* NODE_H_ */
```

EXERCISE 2.

In this part of the practice the Abstract Data Type (ADT) GRAPH will be defined as a set of homogeneous elements (of the same type) and a set of connections (edges) that define a binary relation between the elements.

Definition of abstract data type Graph and implementation: selection of the data structure and implementation of primitives.

You need to define a data structure to represent the ADT GRAPH. Assume that the data to be stored in the graph are of Node type and that its maximum capacity is 4096 elements. The information about the connections will be stored in an adjacency matrix (a matrix of 0's and 1's indicating if the node corresponding to the row is connected to the node corresponding to that column). That is, the following graph would have the adjacency matrix shown below:



	Node A	Node B	Node C
Node A	0	1	1
Node B	0	0	1
Node C	0	0	0

To implement the ADTs related to the graph:

- Declare in the file **graph.h** the interface of the new Graph data type. This must include, at least, the functions indicated in Appendix 2.
- Define in **graph.c** the data structure _Graph:

```
typedef struct _Graph {
    Node *nodes[MAX_NODES]; /*!<Array with the graph nodes */
    Bool connections[MAX_NODES][MAX_NODES];  /*!<Adjacency matrix  */
    int num_nodes;          /*!<Total number of nodes in the graph  */
    int num_edges;          /*!<Total number of connection in the graph  */
} Graph;
```

- Implement in **graph.c** those functions declared in the interface defined in file graph.h.
- Include in graph.c the private functions that you consider useful. The following two private functions are provided (see their implementation in Appendix 2):

```
int find_node_index (const Graph * g, int nId1)
int* graph_getConnectionsIndex (const Graph * g, int index)
```

Checking the correctness of the definition of the type Graph and its primitives.

Define a program in a file named **p1_e2.c** whose executable is called **p1_e2**, and that performs the following operations:

- Initialise two nodes. The first one with name "first", id 111, and label WHITE and the second one with name "second", id 222 and label WHITE.
- Initialise a graph.
- Insert node 1 and verify if the insertion was successful.
- Insert node 2 and verify if the insertion was successful.
- Insert a connection between node 2 and node 1.
- Check if node 1 is connected to node 2 (see message below).
- Check if node 2 is connected to node 1 (see message below).
- Insert node 2 and verify the result.
- Print the graph.
- Free the resources by destroying the nodes and the graph.

Output:

```
Inserting node 1... result...: 1
Inserting node 2... result...: 1
Inserting edge: node 2  ---> node 1
Connected node 1 and node 2? No
Connected node 2 and node 1? Yes
Inserting node 2... result...: 0
Graph
[first, 111, 0, 0]
[second, 222, 0, 1]111
```

Checking a graph behaviour through files.

Implement the following function from the Graph interface:

```
Status graph_readFromFile (FILE *fin, Graph *g);
```

that allows to load a graph from information read in a text file while fulfilling a given format.

Create a main program **p1_e3.c** that as an input parameter (remember the parameters `argc` and `argv` of the main function) receives the name of a file. This file should be read and printed on the screen using the ADT Graph interface developed in the previous exercises.

The execution of this program should run without problems, including, proper memory management (i.e., `valgrind` should not show memory leaks when running).

An example of an input data file is the following, where the first line indicates the number of nodes to be entered in the graph and in subsequent lines the information corresponding to each node (id, name, label); after these lines, pairs of integers will appear in separate lines indicating which nodes are connected:

```
3
1 a 0
2 b 0
3 c 0
1 2
1 3
2 3
```

In this way, this file and the graph of the figure in the previous section are equivalent.

Appendix 2

Private functions for ADT Graph.

```
// It returns the index of the node with id nId1
int find_node_index(const Graph * g, long nId1) {
    int i;

    if (!g) return -1;

    for(i=0; i < g->num_nodes; i++) {
        if (node_getId (g->nodes[i]) ==  nId1) return i;
    }

    // ID not found
   return -1;
}

// It returns an array with the indices of the nodes connected to the node
// whose index is index
// It also allocates memory for the array.
int* graph_getConnectionsIndex(const Graph * g, int index) {
    int *array = NULL, i, j = 0, size;

    if (!g) return NULL;
    if (index < 0 || index >g->num_nodes) return NULL;

    // get memory for the array
    size = node_getConnect (g->nodes[index]);
```

```
    array = (int *) malloc(sizeof (int) * size);
    if (!array) {
        // print error message
        fprintf (stderr, "%s\n", strerror(errno));
        return NULL;
    }

    // assign values to the array with the indices of the connected nodes
    for(i = 0; i< g->num_nodes; i++) {
        if (g->connections[index][i] == TRUE) {
            array[j++] = i;
        }
    }

    return array;
}
```

Contents of file graph.h.

```
/**
 * @file graph.h
 * @author Profesores Prog2
 * @date Created on 29 January 2020, 15:03
 * @version 1.0
 * @brief Library to manage TAD Graph
 *
 * @see
 */

#ifndef GRAPH_H
#define GRAPH_H

#include "node.h"

typedef struct _Graph Graph;

/**
 * @brief Initializes a graph, reserving memory.
 * @return Returns the graph address if it has done it correctly, otherwise
 *  it returns NULL and prints the string associated with error in stderror
 **/
Graph * graph_init ();
```

```
/**
 * @brief Frees the dynamic memory reserved for the graph
 **/
void graph_free (Graph *g);


/**
 * @brief Adds a node to the graph (reserving new memory for that node) only
 * when there was no other node with the same id in the graph. It updates
 * the necessary graph's attributes.
 * @param g Pointer to the graph where the new node to be inserted
 * @param n Node to be inserted
 * @return Returns OK or ERROR.
 **/
Status graph_insertNode (Graph *g, const Node *n);

/**
 * @brief Adds an edge from the node with id "nId1" to the node "nId2" in a
 * directed graph.
 * @param Pointer to the graph where the new edge to be inserted
 * @param nId1, nId2  Nodes ids
 * @return Returns OK or ERROR.
 **/
Status graph_insertEdge (Graph *g, const long nId1, const long nId2);

/**
 * @brief  Returns a copy of the node of id "nId" storaged in the graph
 * @param Pointer to the graph
 * @param nId1 Node id
 * @return Returns a ponter to the copy or NULL if the node does not exit
 *  in the graph.
 **/
Node *graph_getNode (const Graph *g, long nId);

/**
 * @brief Actualize the graph node with the same id
 * @param g Pointer to the graph
 * @param n
 * @return Returns OK or ERROR.
 **/
Status graph_setNode (Graph *g, const Node *n);

/**
 * @brief Returns the address of an array with the ids of all nodes in the graph.
 * Reserves memory for the array.
 * @param g Pointer to the graph
```

```
 * @return Returns the address of an array with the ids of all nodes in
 *  the graph or NULL if there are any error.
 **/
long * graph_getNodesId (const Graph *g);

/**
 * @brief Returns the total number of nodes in the graph.
 * @param g Pointer to the graph
 * @return Returns the number of nodes in the graph. -1 if there have been errors
 **/
int graph_getNumberOfNodes (const Graph *g);

/**
 * @brief Returns the total number of edges in the graph.
 * @param g Pointer to the graph
 * @return Returns the number of nodes in the graph. -1 if there have been
 * errors
 **/
int graph_getNumberOfEdges (const Graph *g);

/**
 * @brief Determines if two nodes are connected.
 * @param nId1, nId2 Nodes id
 * @return Returns TRUE or FALSE
 **/
Bool graph_areConnected (const Graph *g, const long nId1, const long nId2);

/**
 * @brief Get the total number of connections from the node with a given id.
 * @param fromId Node id
 * @param g Graph
 * @return Returns the total number of connections or -1 if there are any error
 **/
int graph_getNumberOfConnectionsFrom (const Graph *g, const long fromId);

/**
 * @brief Returns an array with the id of the nodes connected to a given node.
 * Allocates memory for the array.
 * @param g The graph
 * @param fromId The id of the given node
 * @return Returns the number of connections from the id node fromId or -1 if
 * there are any error
 */
long* graph_getConnectionsFrom (const Graph *g, const long fromId);

/**
```

```
 * @brief Prints in the file pf the data of a graph, returning the number of
 * printed characters.
 *
 * The format to be followed is: print a line by node with the information
 * associated with the node and the id of their connections:
 * @code
 * [1, a, 0, 2]  2 3
 * [2, b, 0, 2] 1 3
 * [3, c, 0, 2]] 1 2
 * @endcode
 * @param pf File descriptor
 * @param g Pointer to the graph
 * @return Return the number of characters printed.
 * If there have been errors in the Output flow
 * prints an error message in stderror and returns the value -1.
 */
int graph_print (FILE *pf, const Graph *g);


/**
 * @brief Read from the stream pointer fin the graph information.
 *
 * This function is optative. See execirse 3.
 * @param pf Stream pointer
 * @param g Pointer to the graph
 * @return OK or ERROR
 */
Status graph_readFromFile (FILE *fin, Graph *g);

#endif /* GRAPH_H */
```

PART 2: QUESTIONS ABOUT THE PRACTICE

Answer the following questions

1. Provide a script that includes the necessary commands to compile, link, and create an executable file with the gcc compiler, indicating what action is performed in each of the sentences in the script. Note that a Makefile file is NOT being requested.
2. Briefly justify if the following implementations of these functions are correct and if they are not, justify the reason why (suppose that the rest of the functions have been declared and implemented as in the practice):

   2.a)

```
int main() {
    Node *n1;

    n1 = (Node*) malloc(sizeof(Node));
    if (!n1) return EXIT_FAILURE;

    /* Set fields */
    node_setId (n1, -1);
    node_setName (n1, "");
    node_setConnect (n1, 0);

    node_free (n1);
    return EXIT_SUCCESS;
}
```

2.b)

```
// In the file node.h
Status node_init (Node *n);

// In the file node.c
Status node_init (Node *n) {

    n = (Node *) malloc(sizeof(Node));
    if (!n) return ERROR;

    /* init fields */
    node_setId (n, -1);
    node_setName (n, "");
    node_setConnect (n, 0);
    node_setLabel (n, BLANCO);
```

```
    return OK;
}

// In main.c
int main() {
    Node *n1;

    if (node_ini (n) == ERROR)
              return EXIT_FAILURE;

    node_free (n1);

    return EXIT_SUCCESS;
}
```

2.c)

```
// In the file node.h
Status node_init (Node **n);

// In the file node.c
Status node_init (Node **n) {

    *n = (Node *) malloc(sizeof(Node));
    if (*n == NULL) return NULL;

    /* inicializa campos */
    node_setId (*n, -1);
    node_setName (*n, "");
    node_setConnect (*n, 0);

    return OK;
}

// In main.c
int main() {
    Node *n1;

    if (node_ini (&n) == ERROR)
              return EXIT_FAILURE;

    node_free (n1);
    return EXIT_SUCCESS;
}
```

3. Would it be possible to implement the copy function for Node using the following prototype?:

`STATUS node_copy (Node nDest, const Node nOrigin);`

Why?

4. Is the pointer `Node *` essential in the prototype of the function `int node_print (FILE * pf, const Node * n);` or it could be `int node_print (FILE * pf, const Node p);`?

If the answer is yes: Why?

If the answer is no: Why is it used, then?

5. What changes should be made in the function of copying nodes if we want it to receive a node as an argument where the information should be copied? That is, how should it be implemented if instead of `Node * node_copy (const Node * nOrigin)`, it would have been defined as `STATUS node_copy (const Node * nSource, Node * nDest)`? Would the following be valid: `STATUS node_copy (const Node * nSource, Node ** nDest)`? Discuss the differences.
6. Why should the private functions for ADT Graph not be public functions? Justify the answer.
7. Could the return type of function `node_copy` be anything different to `void *`? Justify the answer.

Appendix 3

Contents of file `types.h`.

```
/**
 * @file types.h
 * @author Profesores Programación 2
 * @date 2 February 2020
 * @brief ADT Boolean and Status
 *
 * @details Here typically goes a more extensive explanation of what the header
 * defines. Doxygens tags are words preceeded by @.
 *
 * @see
 */

#ifndef TYPES_H_
#define TYPES_H_

/**
 * @brief ADT Boolean
```

```
 */
typedef enum {
    FALSE=0, /*!< False value */
    TRUE=1  /*!< True value  */
} Bool;

/**
 * @brief ADT Status
 */
typedef enum {
    ERROR=0, /*!< To codify an ERROR output */
    OK=1     /*!< OK output */
} Status;

#endif /* TYPES_H_ */
```