System Programming

C programming manual: lab 5

2016 - 2017

Bachelor Electronics/ICT

Course coördinator: *Luc Vandeurzen*
Lab coaches: *Jeroen Van Aken*
Stef Desmet
Tim stas
Luc Vandeurzen

Last update: March 14, 2017

C programming

Lab targets: understanding dynamic data structures and implementing a double-linked pointer list with void pointer elements and callback functions

Once in your life you should have programmed a single and double linked pointer list before you can call yourself a software programmer, or, even better, a 'distinguished' C programmer! Well, this is the time to take that hurdle …

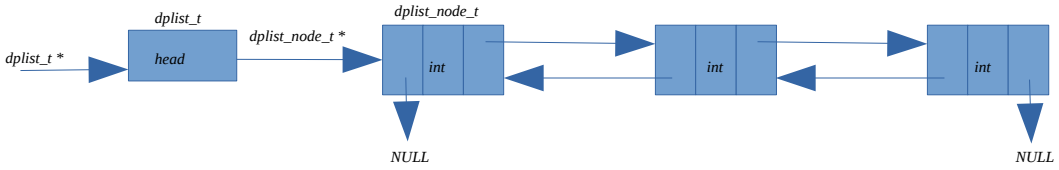
The following exercises guide you in a few steps towards a full implementation of a double-linked pointer list that can handle any element type using void pointers and callback functions for element-specific operations. The implementation of this double-linked pointer list will be archived in a library (see future exercises) such that it can be re-used in other applications.

Exercise 1: *pointer list drawings*

Before starting to implement a pointer list, it might be a good idea to first study some existing code and draw the pointer operations. Use the template code of exercise 1 and 2 for this. In the ‘dplist.c’ file you will notice some comment lines “// pointer drawing breakpoint”. Start executing step-by-step at least the first four statements of the main-function in ‘main.c’ and each time you meet a ‘pointer drawing breakpoint’, you update your pointer list drawing to see what the code is really doing.

Exercise 2: *pointer list implementations for basic element types*

The first step to take is an implementation of a double-linked pointer list able to store elements of a basic type (int, float, char, …). The type of elements stored in the pointer list should not really matter and, therefore, the type **element_t** is introduced. Currently, element_t is defined as an int, but once your implementation is working, also try, for instance, a float. A list consists basically out of list nodes. List nodes contain an element stored in the node, and *next* and *previous* references to, respectively, the next and previous list nodes. List nodes are define by the data type **dplist_node_t.** Graphically, the double-linked pointer list looks as follows:



You don't need to get started from scratch. On Toledo you find the template files main.c, dplist.c, and dplist.h. In dplist.h you find a very basic list of operators that you are supposed to implement. In exercise 4, you will extend this set of operators to a more comfortable one. The file dplist.c, contains a few macros for error handling and

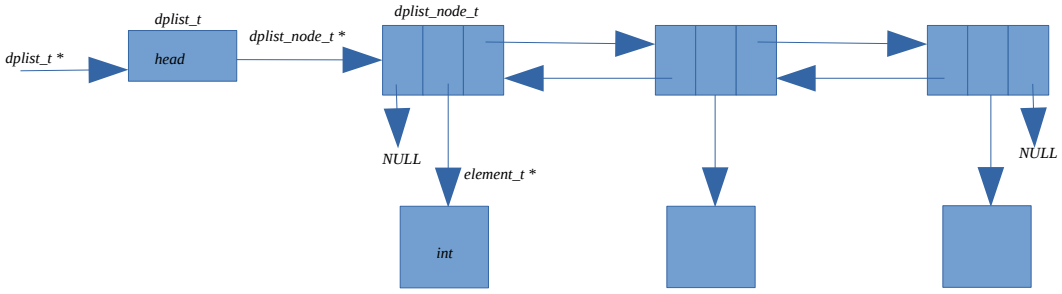
debugging, the real type definitions of the pointer list, and a sample implementation of some of the list operators. This is sample code to get you on track – you are free to change, improve or even completely substitute it with your own code.

Once you have implemented a list operator, test it! Don't wait too long with testing because the complexity and the amount of errors might overwhelm you too much.

Once all operators are implemented and tested, use Valgrind to check for any memory leaks.

Exercise 3: *issues of the pointer list implementation*

The implementation of the previous exercise has a number of drawbacks that become visible as soon as the element type is set to a more complex type then int or float. For example, set the type to int pointers (int *) in dplist.h. Graphically, the new double-linked pointer list looks as follows:



Now use the test code in ‘main.c’ of exercise 2 to discover some issues with the current implementation.

Make pointer list **drawings** that explain all these issues.

Finally, assume we wish to use two lists in your program: one list to store pointers to integers, another one to store pointers to structures containing data from sensor nodes. How could you do that with the current implementation?

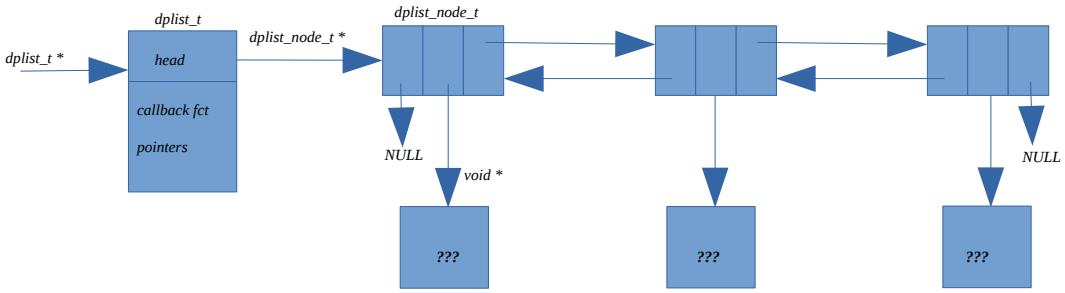
Exercise 4: *pointer list implementations with void pointer and callbacks*

First of all, the data type of the elements stored in the list should not matter at all. If you wish to use multiple lists with different element types in the same program, that should be possible. A typical solution for this problem is using void pointers as element data type in the implementation of the list. It's the responsibility of the user of the list to keep track of the data type of the stored elements, and if needed, typecast returned elements by list operators back to their real data type.

Secondly, the implementation of the pointer list requires the comparison of two elements (e.g. dpl_get_index_of_element()) or the release of memory if an element is using

dynamic memory (e.g. dpl_free()). But how can the pointer list implementation do these operations if it doesn't know the real data type of an element? It can't! Hence, it will need the help of the user (caller) of the list to do these operations on elements. And that help can be implemented with callback functions that correctly implement the operations like copying, comparing and freeing of elements. These callback functions should be implemented by the user (caller) of the pointer list. When a new list is created, the callback functions are provided as arguments and the dplist_create() operator will store the function pointers in the list data structure such that they can called by other list operators when needed. Look at the new template file of dplist.h/.c for some example code.

Putting everything together, the graphical representation of the double-linked pointer list will finally look like:



Thirdly, we like to point out the following subtle memory freeing problem. Assume that a list element uses dynamic memory. An interesting situation occurs when implementing the dpl_remove_at_index() operator. What should you do then with the element contained in the list_node that will be removed? Use the callback function to free the element or not? If list doesn't free it and the user of the list didn't maintain a reference to the element, then the memory is lost and can't be freed anymore. If list does free the element and the user of list did maintain a reference to the element, then this reference is pointing to invalid memory. To give the user of list the choice in what must be done in this case, a boolean parameter is used:

dplist_t * dpl_remove_at_index(dplist_t *list, int index, bool free_element);

- 'free_element' is true: remove the list node containing the element and use the callback to free the memory of the removed element;

- 'free_element' is false: remove the list node containing the element without freeing the memory of the removed element;

A similar problem occurs when a new element must be inserted in the list. What exactly should be inserted into the list: a pointer reference to the element or a 'deep copy' of the

element? Again, this problem is solved by introducing an extra boolean argument in the function dpl_insert_at_index(). And finally, also dpl_free() will offer the caller a choice between freeing the elements or not.

Use the update versions of dplist.h/.c to implement the solution of this exercise.

Exercise 5: *pointer list implementations with the extra operators*

*THE SOLUTION OF THIS EXERCISE NEEDS TO BE UPLOADED AS A **ZIP FILE** ON syssoft.groept.be BEFORE THE NEXT LAB.*

YOUR SOLUTION IS ONLY ACCEPTED IF THE CRITERIA FOR THIS EXERCISE AS DESCRIBED ON syssoft.groept.be ARE SATISFIED!

Implement the full set of list operators (the 'extra' operators) as defined in dplist.h/.c. Once all operators are implemented and tested, use Valgrind to check for any memory leaks.