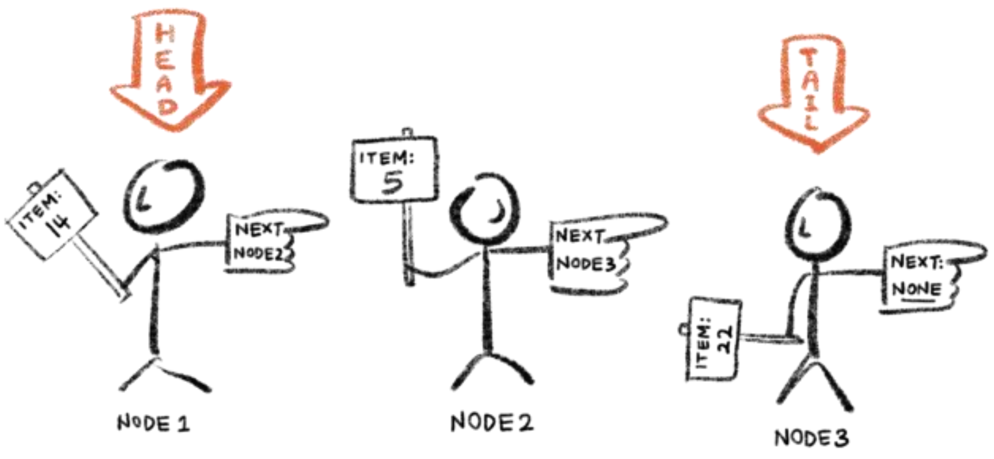
GROUP T LEUVEN CAMPUS

Operating Systems

Programming with C: lab 3

Project Milestone 1: linked lists



Bachelor Electronics/ICT

Course coordinator: Bert Lagaisse

Lab coaches:

- Ludo Bruynseels
- Toon Dehaene

Last update: juli 12, 2024

Introduction

Lab target 1: Understanding dynamic data structures and implementing a double-linked pointer list where the elements are generic pointers (void *) with generic pointer elements and callback functions (= pointers to functions).

Lab target 2: Submit milestone 1 (the generic double-linked list) on Toledo.

Dynamic data structures such as linked lists are essential for operating systems and software development. These data structures change at runtime in terms of size, and we often don't have any information on maximum size upfront.

In this series of exercises we will build a generic double-linked list that takes generic pointers (void *) as elements. Also pointers to functions are needed to properly handle the copy/delete of list elements. (you will certainly miss all the goodies of object oriented programming!).

The implementation of this double-linked pointer list will be reused in future lab sessions, hence it is an important first milestone in your final project for this course.

The *first* step is to build a double-linked list where the elements are basic types like int or char. Once you have this working, the elements in the *next* exercise will be pointers. In our case: strings. (char *). The *third* and final step is modify your code such that it can handle any element. The elements are now generic pointers and to manipulate the elements we provide pointers to functions (aka callback). In the final version the list itself is completely ignorant about the elements it holds.

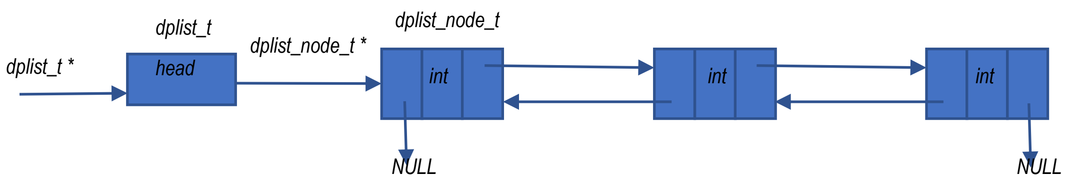
You don't need to get started from scratch. In the zip with the startcode, you find the necessary files to get started for each exercise. In **dplist.h** you find a basic set of operators that you should implement. We also provide a **dplist.c** to get you started with the implementation of the operations on a double-linked list (insert, remove, create, delete). We also get you started with an initial Makefile and test program. Feel free to extend it.

1. Exercise 1: a linked list 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, …). Some code is given to you to ease the startup of project.

- **dplist.h,** including**:**
    - Header guard.
    - Declarations for all the functions you need to implement.
    - Description of what the functions should do.
- **dplist.c**, including some implementations to demonstrate how it works.
    - void dpl_create()
    - dplist_t *dpl_insert_at_index(dplist_t *list, element_t element, int index)
- **program.c** - the main program file.
    - From there on, you set up a test harness to put your list under test. The ck_assert_msg helps in validating your assumptions.
- **Makefile**

The list stores elements of type element_t. In your first version, element_t is a char (defined in dplist.h). A list is a chain of list nodes. List nodes contain an element stored in the node, and references to the next and previous nodes. Nodes are of type **dplist_node_t.** Graphically, the double-linked pointer list looks as follows:

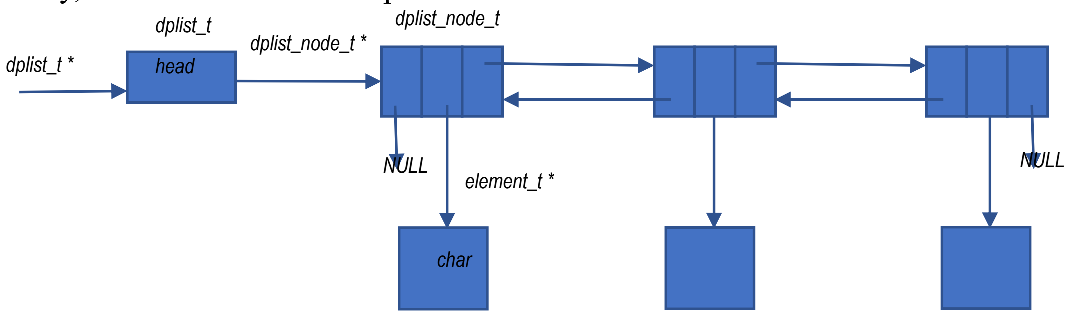


Tips:

1. Work in small steps and test every step.
2. By resolving every step your insight will deepen.
3. If you do too much at once, the number of errors might explode, or every error might hide 2 other errors. You might get discouraged and gain no insight at all.
4. Run valgrind very often. It might find errors that you don't even think about.
5. Test often, after every step. Verify your assumptions using the debugger.

2. Exercise 2: Towards a linked list for pointer types

This exercise is similar to exercise 1 but the elements are now strings (char *). Graphically, the new double-linked pointer list looks as follows:



Now use the dplist.c and test code you wrote in exercise 1 and convert it so the data can hold a char * instead of a char. We provided the dplist.h and Makefile for you.

3. Project Milestone 1: a generic double-linked list.

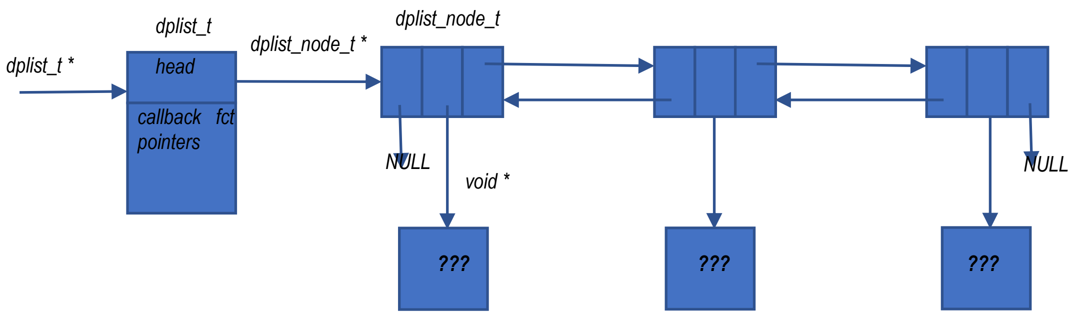
Reminder: Upload your solution for this milestone on Toledo as milestone1.zip!

Use the zip build target in the Makefile to create milestone1.zip ! It will package all your c and h files in a zip. Only use this method to create the zip, or our automated tests on your online submission on Toledo will fail.

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 be 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 a list the choice 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);
```

1. 'free_element' is true: remove the list node containing the element and use the callback to free the memory of the removed element;
2. '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().

Use the versions of dplist.h/.c in `milestone1` to implement the solution. Do not change the name of the files or any of the functions of declarations. If you do, it becomes impossible to grade your work!