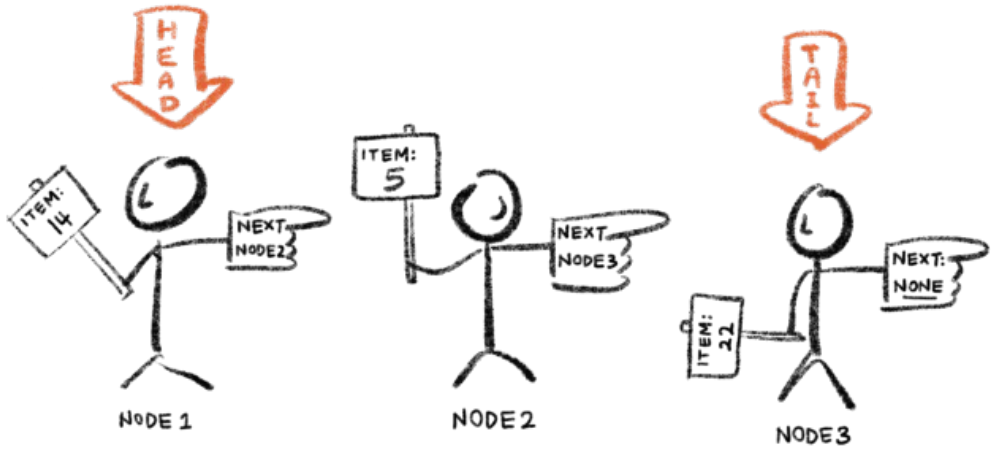
KU LEUVEN

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:

- Jeroen Van Aken
- Yuri Cauwerts
- Ludo Bruynseels

Last update: October 24, 2022

Programming with C: Lab 3

Lab target 1: Understanding dynamic data structures and implementing a double-linked pointer list with void pointer elements and callback 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.
The following exercises guide you in a few steps towards an implementation of a generic 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 reused in future lab sessions, hence it is an important first milestone in your final project for this course.
First we learn and practice these key data structures in software systems with two exercises where we create a double-linked list for a basic element type and for a pointer type. The final part of this lab is the development of a generic double-linked list using void pointers, which you **need to submit as the first milestone** for your project of this course.

We provide skeleton code for the two exercises and the project's first milestone in your Git repo in the branch `clab3lists` (see folders ex1, ex2, and milestone1). The skeleton code illustrates the implementation of some operations in the list as well as some possible unit tests. **The code will compile, but not all unit tests will succeed** as you still need to implement some crucial operations of the list. The make files and our example code **require the installation of some development tools** via "sudo apt install":

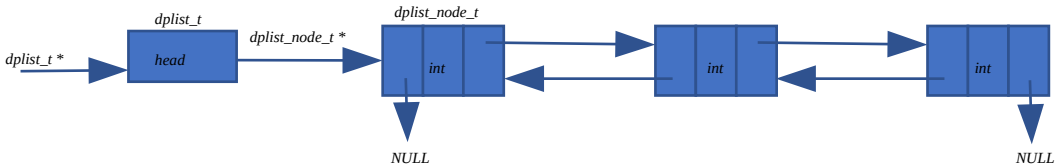
cppcheck: this tool will check your code against some coding rules

pkg-config: this tool is needed to link all your required libraries during compilation

check: this tool and library is used for the unit tests.

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, …). 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 a *char*, 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 defined by the data type **dplist_node_t.** Graphically, the double-linked pointer list looks as follows:

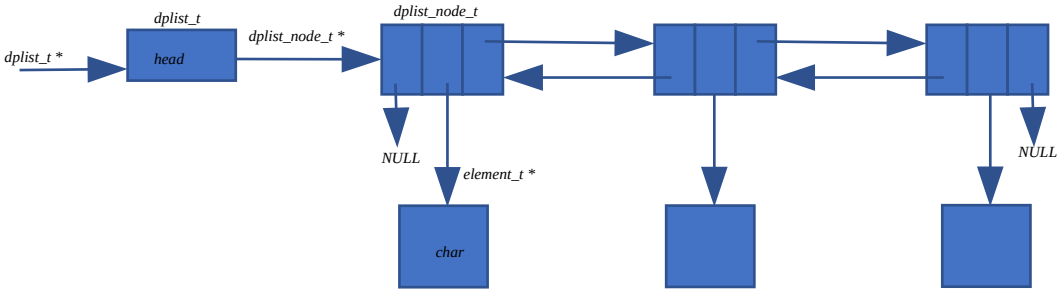


A set of error codes is defined and a global variable 'dplist_errno' is used for error messaging. Every list operator will reset dplist_errno to 0 at the start of the function implementing the operator. If any error happens during the execution of the function, it will set dplist_errno to a meaningful error code. It is the responsibility of the caller of the list operator to check the value of dplist_errno.
You don't need to get started from scratch. In your Git repo you find the files **dplist_test.c, dplist.c, and dplist.h**. In **dplist.h** you find a basic set of operators that you should implement. 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. The file **dplist_test.c** illustrates an example of how to use the unit testing library "check".

Hint 1: The unit testing macro's you define might confuse the debugger to halt on breakpoints due to forking of unit tests in the check framework. In your *homedir* of Linux (e.g. /home/users/osc), add a text file `.gdbinit` containing a gdb setting to not fork the unit tests: `set environment CK_FORK=no`
Remote development in your IDE will automatically use this setting for gdb too.
Hint 2: 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.
Hint 3: Once all operators are implemented and tested, use Valgrind to check for any memory leaks.
Hint 4: Feel free to change the makefile with your own build targets (e.g. *all,* as used by default for clion), add debug info to the build targets when you need it (gcc -g), or build and debug the code with your IDE (see homework 2 for instructions).

Exercise 2: Towards a linked list for pointer types

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 than int or float. For example, set the type to int pointers (char *) in dplist.h. Graphically, the new double-linked pointer list looks as follows:



Now use the dplist and unit-test code you wrote in exercise 1 and convert it so the data can hold a char * instead of a char. Make pointer list **drawings** for yourself.

Project Milestone 1: a generic double-linked list.

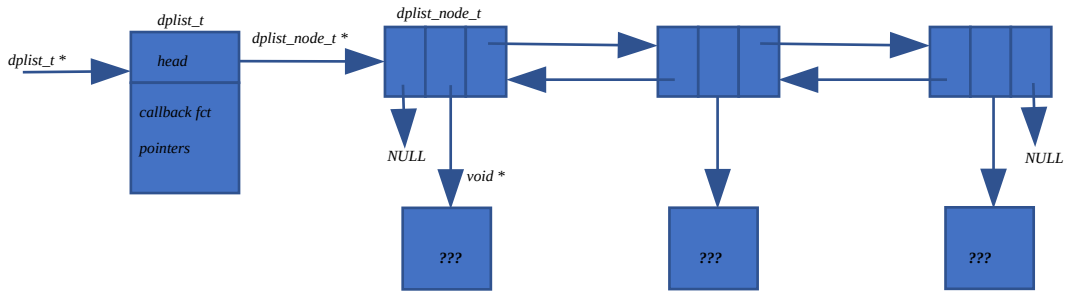
Reminder: Upload your solution for this milestone on Toledo as milestone1.zip!

Use the zip build target in the Makefile to create milestone1.zip ! It will package all your c and h files in a zip.

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.