Reminding C: back to basis (pointers etc.)

Sebastien Varrette, Frederic Pinel and Pascal Bouvry
<Firstname.Lastname@uni.lu>

Version 1.0

Exercise 1 *Helloworld, Makefile and modular programming*

1. Write the C helloworld in the file `helloworld.c`.
   a) Remind the purpose of the `gcc` options `-g`, `-O`, `-I`, `-L` and `-Werror`.
   b) Compile your program with `gcc` (command-line) in a way that raise the maxima warnings *i.e.* with

```
gcc -Wall -pedantic --std=c99 helloworld.c -o helloworld
```

2. create a header file `hello.h` that declare the function `void print_hello()`. Pay attention to the declaration of a pre-processor macro to avoid the "*multiple file inclusion*" error.
3. implement this function in `hello.c` and adapt `helloworld.c` and your `Makefile` accordingly.

Exercise 2 *GNU Debugger gdb*

Consider the following program:

```
#include <stdio.h>

/**
 * Factorial computation (buggy version)
 */
long fact(int n) {
   long f = 1;
   while(n) {
      f *= n;
      n--;
   }
   return f;
}

int main()
{
   printf("fact(4) = %ld\n",fact(4));
   printf("fact(10) = %ld\n",fact(10));
   printf("fact(-1) = %ld\n",fact(-1));
   return 0;
}
```

1. Compile this program to be able to run `gdb` on it.
2. Run `gdb` on the program.
   a) run the program
   b) interrupt the program with CTRL-C

c) use `list` (or `l`) to show the current lines of code
d) use `backtrace` (or `bt`) to get the execution context
e) put a breakpoint on the `fact` function and re-run the program
f) use `cont` (or `c`) to continue the execution (do it twice)
g) use `next` (or `n`) to execute the next instruction
h) use `print` (or `p`) to print the value of `n`
i) use `watch` (or `w`) to add a watchpoint on `n`
j) continue the execution and deduce the bug of the program

Exercise 3 *Ordered Doubly-linked lists*

A doubly-linked list is a list structure with a reference to the next and previous elements. The variable `head` points to the first element of the list. By convention, `head`'s previous element is `NULL`.
If the list is empty, then `head == NULL`. Similarly, the variable `tail` points to the last element of the list. If the list is empty, then `tail == NULL`. `tail`'s next element is also `NULL`. We are interested in a doubly-linked list of integers, sorted in ascending order (smallest integer first). An example of such a list is presented in figure 1.

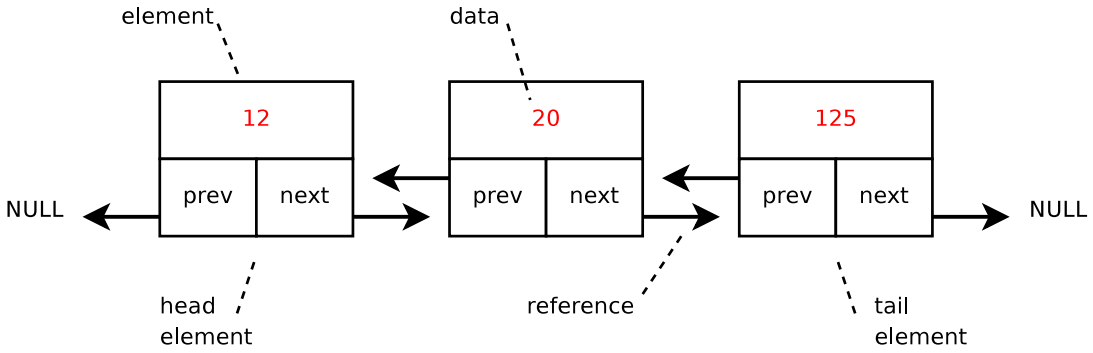

Figure 1: Structure of a sorted doubly-linked list

Each element of the list is represented by a triplet `<data,next,prev>`.
Insertions and deletions in the sorted doubly-linked list must preserve the order of the list. The order is the increasing value of the `data` field. The objective of this exercise is to write a program which handles doubly-linked lists. The program should be organized in such a way that it can be reused by other programs, which need doubly-linked lists.

Part I — Preliminary

1. Organize your working directory (setup the `Makefile` etc.)

Part II — Doubly-linked lists

First, we will deal with unordered doubly-linked lists.

1. Define the data structure `elm`, required for the implementation of a doubly-linked list. By using the keyword `typedef`, define the type `pelm`: pointer-to-an-element. A list will be defined by a couple of `pelm`, pointing to the head and tail of the list. Define the data structure `list` for such a representation.
2. Write the function `void print_list(list *l);`
   which displays the contents of the list `l`.

3. Write the function `int len_list(list *l);`
   which returns the number of elements in the list `l`.
4. Write the function `void add_head(list *l, int d);`
   which inserts a new element at the beginning of the list (it becomes the new `head`). The list is not ordered. The data part of the new element is initialized with integer `d`.
5. Write the function `void add_tail(list *l, int d);`
   which inserts a new element at the end of the list (it becomes the new `tail`). The list is not ordered. The data part of the new element is initialized with the integer `d`.
6. Write the function `void del_list(list *l)`
   which frees the memory allocated to store the elements of a list.
   Provide a recursive version of the function `void del_list_r(list *l)`.

Part III — Ordered doubly-linked lists

We now turn to sorted doubly-linked lists.

1. To insert an integer in the list, we must locate the appropriate place in the list where to insert it. If the integer is already present in the list, we do not add it.
   Write the function `int insert(list *l, int d);`
   which inserts a new element at the correct place. This means: preserving the order of the list. The list `l` is considered sorted when entering the function. The data part of the new element is initialized with `d`. This function returns `0` if the insert is successful, `1` otherwise. An unsuccessful operation can occur if the integer is already present in the list. Which other error conditions exist?
2. Deleting an element by value, means searching for an element in the list, whose value matches, and if present, removing it. This involves modifying the references to the previous and next item so as to maintain the list.
   Write the function `int del_data(list *l, int d);`
   which deletes the element with data equal to `d`, if present. The function returns `0` if the delete is successful, `1` otherwise. What are the possible errors?

To help you, here is an *example* program.

```
void print_info_on(struct list * l)
{
   print_list (l);
   printf ("length: %d\n", len_list (l));
}

int main()
{
      struct list l = {NULL, NULL};

      printf("=== add_tail test ===\n");
      for (int i = 0; i < 10; i++)
         add_tail (&l, 2*i);
      print_info_on(&l);
      printf("=== add_head test ===\n");
      for (int i = 0; i < 10; i++)
         add_head (&l, i);
      print_info_on(&l);
      printf("=== del_list_r test ===\n");
      del_list_r (&l);
      print_info_on (&l);
```

```
      printf("=== insert test ===\n");
      for (int i = 0; i < 10; i++)
         insert (&l, 2*i);
      print_info_on(&l);
      for (int i = 0; i < 10; i++)
            insert (&l, i);
      print_info_on (&l);
      printf("=== del_data test ===\n");
      for (int i = 5; i < 16; i++)
            del_data (&l, i);
      for (int i = 5; i < 16; i++)
            del_data (&l, i);
      print_info_on (&l);
      printf("=== final del_list_r test ===\n");
      del_list_r (&l);
      print_info_on (&l);
      return 0;
}
```

The standard output for the above program is:

```
=== add_tail test ===
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
length: 10
=== add_head test ===
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
length: 20
=== del_list_r test ===
()
length: 0
=== insert test ===
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
length: 10
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18)
length: 15
=== del_data test ===
(0, 1, 2, 3, 4, 16, 18)
length: 7
=== final del_list_r test ===
()
length: 0
```