# System Software

# C programming manual: lab 4

# 2017 - 2018

## *Bachelor Electronics/ICT*

*Course coördinator:  Luc Vandeurzen*
*Lab coaches:  Wouter Van der Schraelen*
*Stef Desmet*
*Tim Stas*
*Jeroen Van Aken*
*Luc Vandeurzen*

# C programming

*Lab targets: understand dynamic memory and what malloc() and free() really do, be able to program pointer manipulations, implement basic memory allocation algorithms, learn to use GDB*

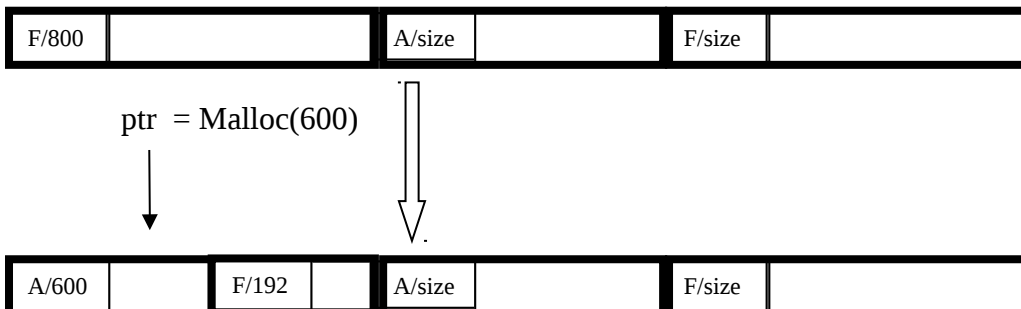**Exercise:** *dynamic memory allocator*

The best way to better understand dynamic memory in general, and function calls such as malloc() and free() in particular, is to program a simple memory allocator yourself. The first thing you need is a *memory pool*. When malloc() is called, you search a *free chunk* of contiguous memory in this pool to assign to the caller and mark this chunk as 'allocated'. When free() is called, you mark the corresponding memory chunk as 'free' such that it can be used for malloc() again.

Memory allocators often require a per memory chunk header to keep track of the size of the memory chunk, at least one flag (F = free/ A = allocated), sometimes a reference to the previous memory chunk (see exercise 2), etc. These headers are usually at the beginning of the memory chunks, so the memory allocator can easily find them when a pointer is passed to free() or realloc() by going back a couple of bytes (total size of this header). Notice that the size in these headers can also be used to compute the start of the next chunk which allows us to go through all chunks, one after the other. The following figure illustrates the use of headers (grey color) and memory chunks.

| F/size | | A/size | | F/size | |
|---|---|---|---|---|---|

*Allocating memory (malloc)*

When an application requests a chunk of bytes, the allocator searches a free chunk that is large enough to hold the requested chunk. The manner in which the allocator performs this search is determined by the *placement policy*. Some common policies are *first fit*, *next fit*, and b*est fit*. First fit returns the first free chunk that fits; best fit returns of all free chunks the one with the smallest size that fits. You may implement the simple first fit policy for this exercise. Once the allocator has located a free block that fits, it usually has to split the free chunk into two parts: the first part becomes allocated and the remaining part remains free. The figure below illustrates this (assume a header size of 8 bytes).

| F/800 | | A/size | | F/size | |
|---|---|---|---|---|---|

ptr = Malloc(600)

| A/600 | | F/192 | | A/size | | F/size | |
|---|---|---|---|---|---|---|---|

---

*Freeing memory*

Freeing memory is very easy: just mark the chunk as free again and that's it! (A better policy will be discussed in exercise 2.)

Now implement the memory allocator as discussed above. For simplicity you can use a static array of bytes (= unsigned char) as a memory pool for malloc/free. Real memory allocators get their memory from the OS using system calls such as mmap(), sbrk(), etc. but system calls will be studied later on in this course. Nevertheless, the idea remains the same. Implement the following functions:

- void ma_init();
    - o allocates array of bytes and initializes the memory allocator
- void *ma_malloc(size t size);
    - o semantics is the same as standard malloc()
- void ma_free( void *ptr );
    - o semantics is the same as standard free()

On Toledo, you find 3 template files: 'ma_alloc.h', 'ma_alloc.c' and 'main.c' you can use to get started. Be aware of the tests in 'main.c' because it also contains a 'coalescing' test. Coalescing, however, is the target of the next exercise.

This exercise helps you to understand that:

- using (dangling pointer) or overwriting (out of boundary) unallocated memory is very dangerous;
- after a number of malloc/free combinations, the memory becomes fragmented. It's even possible that malloc() fails because it seems there isn't enough free memory while the aggregated sizes of all free chunks is more than enough;
- malloc/free are so called "non-deterministic" functions, in time as well as in space. A call to malloc() doesn't return in a constant, predictable time as it depends heavily on how long the allocator has to search for a free chunk. This might be a problem for real-time systems. The caller of malloc() has also no control over the final location in memory that malloc() will return: a call to malloc() immediately followed by a free(), followed by the initial malloc() again might result in different memory locations.

**Exercise:** *GDB and DDD*
GDB is the GNU debugger with a command line interface. It's a very powerful tool that you can use to step by step execute a program, set breakpoints, inspect and even change variables, display the registers, do stack manipulations, get memory dumps and much more. But all this comes with a price: it takes some time to learn this tool and the text-based interface doesn't simplify this. The official documentation can be found on http://www.gnu.org/software/gdb/documentation.
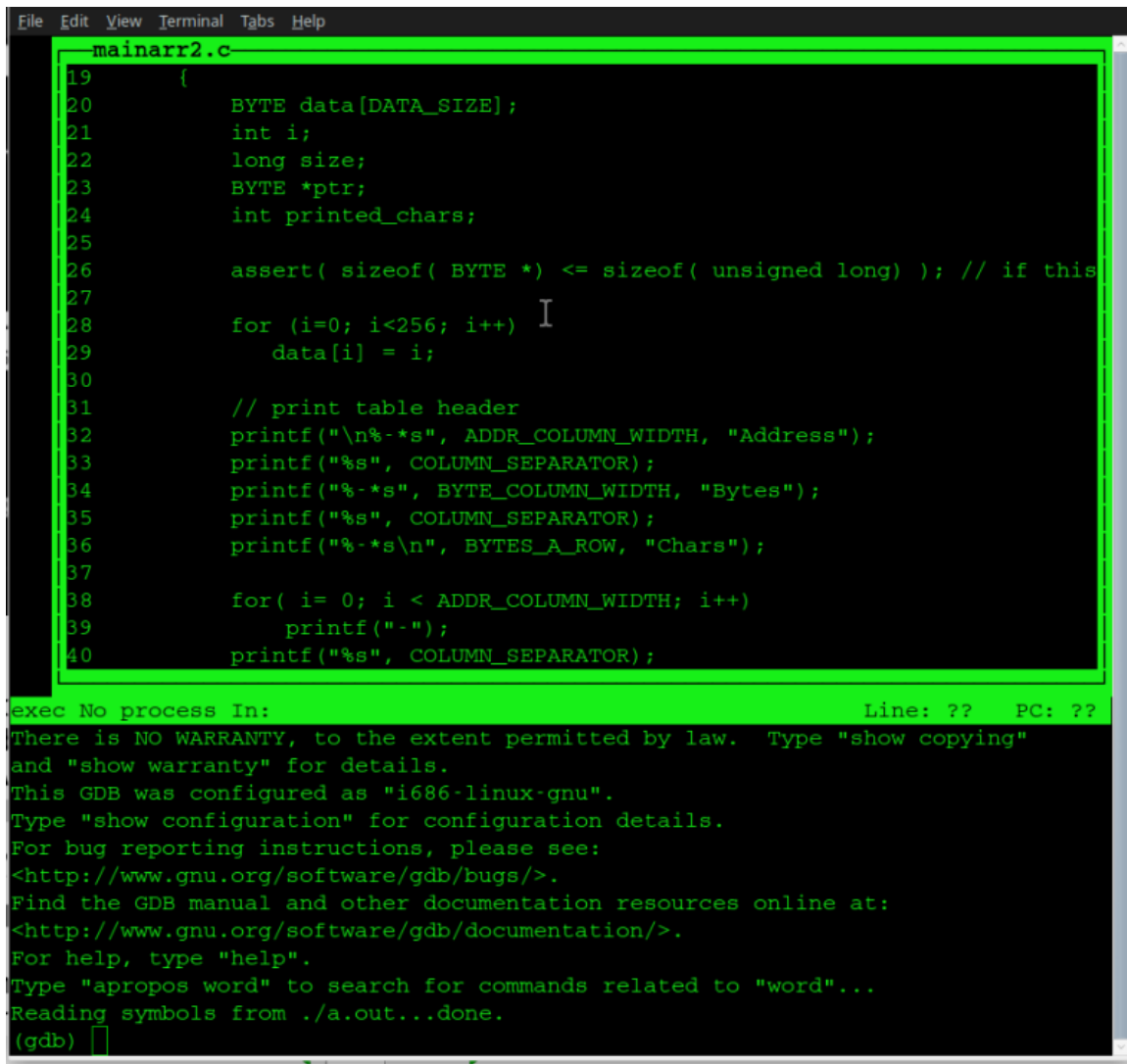To ease the use of GDB we will work with a graphical front-end of GDB. Several GUI front-ends exist and we will discuss briefly two options. The first option, is the ncursus-GUI of GDB itself. The second option, is DDD. If DDD is not pre-installed, you have to install it yourself. In both cases, you can only debug your program if debug information was added. This means that while compiling your code, you have to indicate that debug information should be added. This is done with the '-g' option of 'gcc', e.g. 'gcc -g …'.

The GUI of GDB can be started as:

        gdb -tui ./a.out

assuming that 'a.out' is the program you want to debug and that 'a.out' was compiled with the '-g' option on. Once this command executed, you see a screen like the one below.
The window layout consists of a 'source code' window (upper part) and a 'command' window (lower part) frame. By default, the window focus is on the source window frame. With the cursor up/down/left/right you can move through the source code. You can always switch the focus back and forth with the command 'focus src' and 'focus cmd'. Is there any other window layout possible? Yes, you can view the assembly code with the command 'layout asm' or even view source and assembly code with 'layout split'. To find out what 'focus' or 'layout' options you have, use the help function: 'help focus' or 'help layout'. Once you understand this, your are ready to run the program, set breakpoints, and the like. There are many tutorials on GDB. To get started, we suggest reading 'Beej's Quick guide to GDB' (https://beej.us/guide/bggdb).

```
File  Edit  View  Terminal  Tabs  Help
  ─mainarr2.c─
19      {
20          BYTE data[DATA_SIZE];
21          int i;
22          long size;
23          BYTE *ptr;
24          int printed_chars;
25
26          assert( sizeof( BYTE *) <= sizeof( unsigned long) ); // if this
27
28          for (i=0; i<256; i++)
29              data[i] = i;
30
31          // print table header
32          printf("\n%-*s", ADDR_COLUMN_WIDTH, "Address");
33          printf("%s", COLUMN_SEPARATOR);
34          printf("%-*s", BYTE_COLUMN_WIDTH, "Bytes");
35          printf("%s", COLUMN_SEPARATOR);
36          printf("%-*s\n", BYTES_A_ROW, "Chars");
37
38          for( i= 0; i < ADDR_COLUMN_WIDTH; i++)
39              printf("-");
40          printf("%s", COLUMN_SEPARATOR);
exec No process In:                                    Line: ??    PC: ??
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...done.
(gdb)
```

If you don't like the built-in GUI of GDB, don't panic yet. Start DDD on the command line as 'ddd ./a.out &' or start if from the 'start menu' like any other GUI-based program. You should get a screen like the one below. You recognize the middle and lower window frames. The upper window frame is the 'data' frame. Here, variables and data structures can be graphically visualized. It's this feature that makes DDD more interesting than a number of other, even better looking, graphical debuggers

Once again, we don't wish to write a tutorial on DDD but refer to existing one's like for instance an article in DrDobs (see drbobs.pdf on Toledo) or http://www.tldp.org/LDP/LG/issue73/mauerer.html. You can also watch some videos on YouTube. The full manual can be found on the DDD website (http://www.gnu.org/software/ddd/).
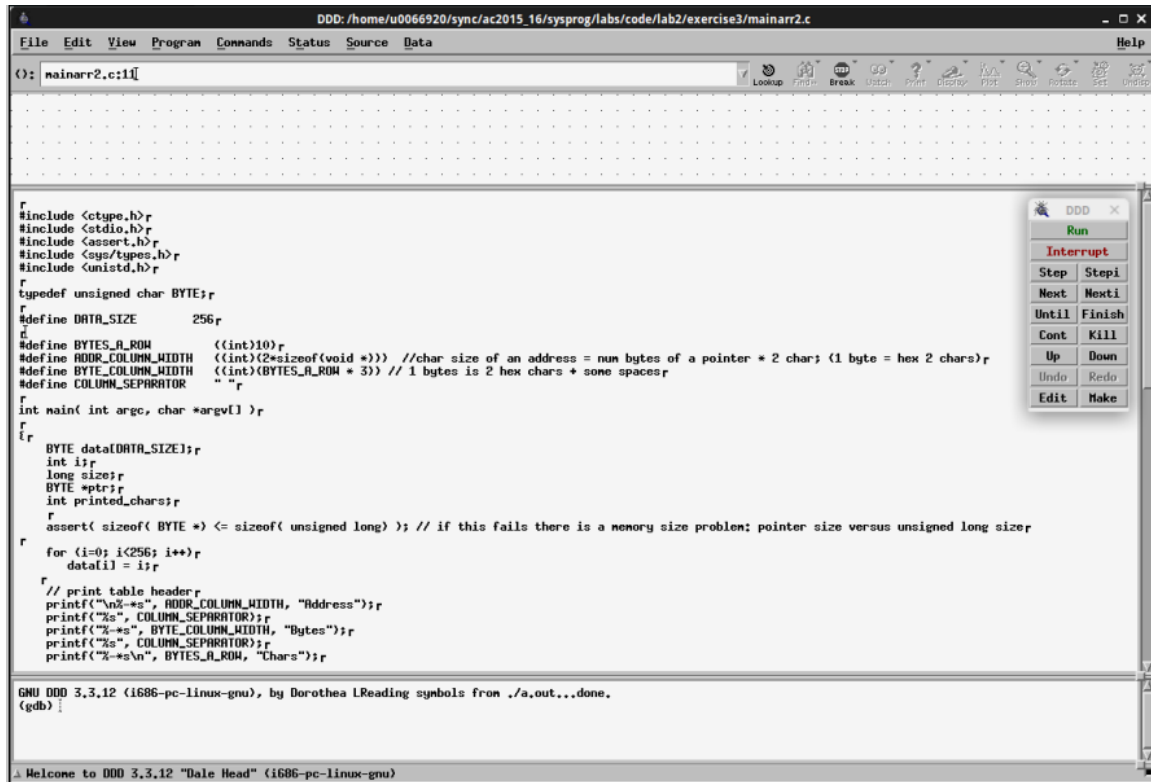
Still not happy with the a bit out-dated GUI of DDD? Ok, then you could try one of the following GDB front-ends:

- 'Kdbg' : you can install 'kdbg' from the 'synaptic package manager' or 'Ubuntu software center'; be aware there could be a small issue running this program on Ubuntu. Fortunately, there exist a simple solution described in https://bugs.launchpad.net/ubuntu/+source/kdbg/+bug/1311893 that actually explains you need to delete a mng-file as follows: sudo rm /usr/share/kde4/apps/kdbg/icons/hicolor/22x22/actions/pulse.mng;
- 'Gede' : you can download 'gede' from http://gede.acidron.com/ and following the installation instructions.

**Exercise:** *dynamic memory allocator: coalescing freed memory*

*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 DE-SCRIBED ON syssoft.groept.be ARE SATISFIED!*

When the memory allocator frees an allocated chunk, there might be other free chunks that are adjacent to the newly freed chunk. Such adjacent free chunks can cause a false fragmentation problem: an initially large free memory chunk finally ends up into small, unusable free chunks!
To avoid this problem, free adjacent chunks must be merged together – this is called *coalescing*.
For example, given the situation depicted below. If free(ptr) is now called, the memory allocator should merge the entire memory to one large free chunk of memory.

ptr

| F/size | | A/size | | F/size | |

free(ptr)

| F/size | |
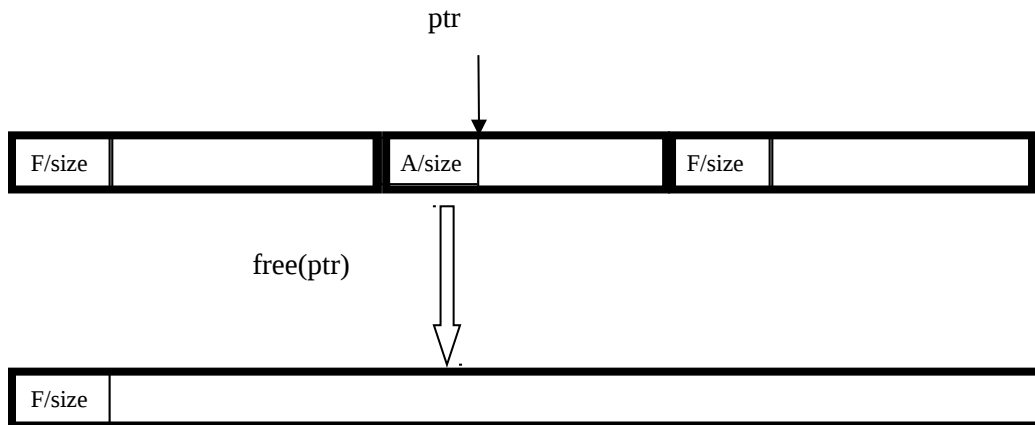
If a chunk is freed, you first check if the next chunk in memory - which is easy to find using the size in the header of the freed chunk - is also a free chunk. If that is the case, then coalescing is straightforward and efficient: just remove the header of the next chunk and update the header of the freed chunk with the new size. But what if the previous chunk is free? The real problem is to find the header of this chunk. There is no direct link to the header of the previous chunk. Of course, you could start searching from the beginning until you find this header, but that's not very efficient. A very simple solution to this problem was proposed by *Knuth*. The idea is to add not only a header but also a *footer* to every memory chunk. Which information this footer must contain? Again, we go for a very simple answer: just copy the entire header information into the footer. This is illustrated in the following picture:

Header          Footer

| A/200 | | A/200 | F/800 | | F/800 |

Allocated chunk          Free chunk
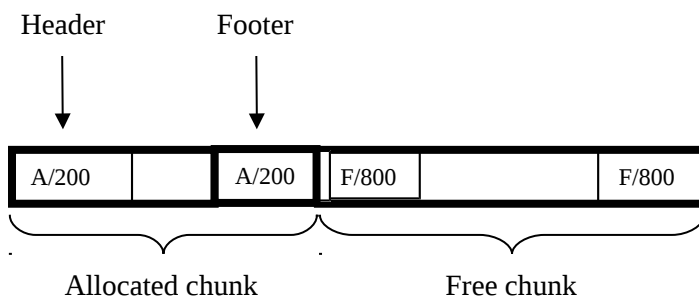
**A final note**
In this lab we explored a very simplistic implementation of a memory allocator. Taking into account the lab targets and your limited knowledge on C and system calls at this moment, this approach is justified. But you must realize that many more and much better algorithms and implementations exist, as indicated by the following examples:

- liballoc (https://github.com/blanham/liballoc) - a memory allocator for hobbyist operating                                                                                  systems.

- dlmalloc (http://g.oswego.edu/dl/html/malloc.html) - Doug Lea's Memory Allocator. A good all purpose memory allocator that is widely used and ported.

- TCMalloc (http://goog-perftools.sourceforge.net/doc/tcmalloc.html) Thread-Caching        Malloc.        An        experimental        scalable        allocator.

- nedmalloc (http://www.nedprod.com/programs/portable/nedmalloc/) A very fast and very scalable allocator - popular in multi-threaded video games.

- ptmalloc (http://www.malloc.de/en/) A widely used memory allocator included with glibc that scales reasonably while being space efficient.