C programming basics

—

MOSIG M1

https://systemes.pages.ensimag.fr/www-c/

Grégory Mounié (CC-BY-SA)*

2022-2023

The goal of this practical is to manipulate the basics of C, especially around the memory model. Never hesitate to ask questions if you are stuck.

Preamble

If you are totally new to C programming, you must change that fact very quickly, as most lectures will use C as support langage, including the "Operating System" lecture of the first semester. You should probably start with another training first. Many options are possibles.

The book is *Modern C* from Jens Gustedt is organized by levels, from 0, "early beginner", to 3 "expert" (PDF avaliable here https://hal.inria.fr/hal-02383654, source of the document https://gitlab.inria.fr/gustedt/modern-c)

We advise you also to read the C wikibook (https://en.wikibooks.org/wiki/C_Programming/) and do simultaneously the C track of `Exercism.io` (https://exercism.io/tracks/c).

For fluent French reader, you should read the excellent, and broadly used, « Introduction au langage C », from Bernard Cassagne (retired research engineer of Grenoble). The most up-to-date electronic version is at http://matthieu-moy.fr/cours/poly-c/

In any case, feel free to use the C programming wikibook or any suitable resources (paper books, internet, etc.) to help you.

*parts are the translation of Matthieu Moy slides

1 Introduction

The following command get the code skeletons and associated tests.

```
$ git clone https://github.com/gmounie/ensimag-rappeldec.git
```

The directory `ensimag-rappeldec/` contains all the materials.
The code edition will be done in the sub-directory `ensimag-rappeldec/src/`.
Add your login to `ensimag-rappeldec/CMakeList.txt` file before any compilation.
Beware of some IDE (Atom or VS Code). It may try to compile the code, but, it will fail and pollute the directories.
The workflow is to do all the compilation in the sub-directory `build/`.
To prepare the compilation process, you need to do once :

```
$ cd ensimag-rappeldec/build
$ cmake .. # Yes, with the two points !
```

Still in the `build/` directory, to compile and run the tests, do the following commands :

```
$pwd
<your path>/ensimag-rappeldec/build
$ make
$ make test
$ make check
```

Remember to edit first `ensimag-rappeldec/CMakeList.txt`

2 Tutorial on common memory bug in C (30 min)

The tutorial part is quite guided. This is not the case of following exercises. (sec. 3, page 9).

Pointers express the programmer's view on the memory at a particular address. This section proposes to manipulate several debugging tools on small, yet tricky, examples.

When programming, especially in C, you probably pass quite some time to debug your creation. For simple small exercises, « printf debugging », by adding `printf` everywhere in the code, is quite common, and mostly sufficient. Nevertheless, some previous teacher probably shows you basic functions of a debugger and other tools, and tried to convince you to use them.

We will do the same, again, with advanced function of GDB.

You may fully succeed your diploma without using debugger. Unless you do some really technical projects (OS project for example), where you will learn to use a debugger, or fail the project.

But it is a pity to spend hours of your valuable study time in editing and compiling to add some `printf`, instead of using a tool that will do the same, and better, much faster.

2.1 Assertions, GDB, function call stack

The code of `src/bug_assert.c` check a boolean predicate, always false in the example. Note that `false` is equal to `0`

Run the program `./bug_assert`, in the `build/` repository. Note the error message of the assert function.

```
you@ensipc$ ./bug_assert
bug_assert: /home/gregory/ensimag-rappeldec/src/bugs/bug_assert.c:20:
↪  main: Assertion `0' failed.
Abandon
```

In a terminal, run `gdb` with the program as argument. Display graphically the code. Add a breakpoint to the main function. Run the program. Continue after the breakpoint. After the stop at assert, display the call stack and go up in it up to the main function at the `assert(false);`.

```
you@ensipc$ gdb ./bug_assert
[...]
(gdb) layout src
(gdb) break main
(gdb) run
(gdb) cont
(gdb) where
(gdb) up
(gdb) up
(gdb) up
(gdb) up
(gdb) quit # confirm
```

2.2 Double free : C library check and Valgrind detection

The code `src/bug_doublefree.c` frees two times the array : once just before the end of the function `fibon()`, once just after.

Run the program `./bug_doublefree`. The C library detects the problem while executing the second free fonction.

```
you@ensipc$ ./bug_doublefree
free(): double free detected in tcache 2
Abandon
```

Run the program with `valgrind`. It detects the bug and give also the code line numbers of the allocation and the two free. At the end, Valgrind gives also a summary of the execution (1 alloc, 2 free).

```
you@ensipc$ valgrind ./bug_doublefree
[...]
==31719== Invalid free() / delete / delete[] / realloc()
```

```
==31719==    at 0x484217B: free (vg_replace_malloc.c:872)
==31719==    by 0x109291: main (bug_doublefree.c:24)
==31719==  Address 0x4a8f040 is 0 bytes inside a block of size 400
↪  free'd
==31719==    at 0x484217B: free (vg_replace_malloc.c:872)
==31719==    by 0x1091EF: fibon (bug_doublefree.c:13)
==31719==    by 0x109285: main (bug_doublefree.c:22)
==31719==  Block was alloc'd at
==31719==    at 0x483F7B5: malloc (vg_replace_malloc.c:381)
==31719==    by 0x10923F: main (bug_doublefree.c:19)
==31719==
==31719== HEAP SUMMARY:
==31719==     in use at exit: 0 bytes in 0 blocks
==31719==   total heap usage: 1 allocs, 2 frees, 400 bytes allocated
==31719==
==31719== All heap blocks were freed -- no leaks are possible
==31719==
==31719== For lists of detected and suppressed errors, rerun with: -s
==31719== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
↪  0)
```

2.3 Abusive free ! Free without malloc

The `free()` function is used to free memory allocated in the heap with `malloc`, `calloc` or `realloc`.

The code `src/bug_stackallocthenfree.c` frees an array allocated in the stack. This is non-sense from memory allocator point of view.

Run the program `./bug_stackallocthenfree` without Valgrind then with Valgrind. Valgrind precises that the variable was allocated in the frame stack of main, thus is probably a local variable of main.

```
you@ensipc$ ./bug_stackallocthenfree
munmap_chunk(): invalid pointer
you@ensipc$ valgrind ./bug_stackallocthenfree
[...]
==11348== Invalid free() / delete / delete[] / realloc()
==11348==    at 0x484217B: free (vg_replace_malloc.c:872)
==11348==    by 0x1091DF: fibon (bug_stackallocthenfree.c:13)
==11348==    by 0x1092A6: main (bug_stackallocthenfree.c:22)
==11348==  Address 0x1fff0002e0 is on thread 1's stack
==11348==  in frame #2, created by main (bug_stackallocthenfree.c:16)
[...]
```

To confirm exactly the variable name in the stack, we need to ask gdb. Thus, the solution is to connect valgrind and gdb and display all variables name (and values) then the addresses of the stack variables of main. Note that the faulty address is `0x1fff0002e0`.

```
you@ensipc$ valgrind --vgdb=full --vgdb-error=0 ./bug_stackallocthenfree
↪ &
[...] # copy gdb command: "target remote | vgdb --pid=..."
you@ensipc$ gdb ./bug_stackallocthenfree
[...]
(gdb) target remote | /usr/bin/vgdb --pid=14526 # paste
(gdb) cont # valgrind stop at bug free(0x1fff0002e0)
(gdb) where # call stack
(gdb) up 2 # 2 frames up to go to main
(gdb) info locals
(gdb) print &p
$1 = (unsigned int (*)[100]) 0x1fff0002e0
(gdb) quit # confirm
```

The problem occurs with the p variable of the main.

2.4 Stack overflow : gdb and the call stack

The program `src/bug_stackoverflow.c` uses a circular linked list and calls a recursive function on it.

The stack is a memory block of few MiB where the recursive functions store their arguments and local variables. The command `ulimit -a` gives the list of limits, including stack size.

Every recursive call consumes a part of the stack for the return address of the call, the arguments and the local variables. The part is given back only at the end of the function call. Hence, it is not possible in C to infinitely calls a recursive function.

Note that OCaml, LISP and friends, or Haskell may remove terminal recursion, thus allows infinite calls. The Go language is able to grow its stack up to few GiB. Go language does not solve the problem but make it less likely.

Run the program `./bug_stackoverflow`. The system detects the illegal usage of the memory (SEGFAULT). But this illegal usage looks very much like many illegal usage any C programmer has already done.

```
you@ensipc$ /bug_stackoverflow
Erreur de segmentation (core dumped)
```

Run the program with valgrind. It detects the same problem but it indicates that the problem is a stack overflow and not something else.

```
you@ensipc$ valgrind ./bug_stackoverflow
[...]
==8672== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==8672==
==8672== Process terminating with default action of signal 11 (SIGSEGV)
==8672==  Access not within mapped region at address 0x1FFE801FF8
==8672== Stack overflow in thread #1: can't grow stack to 0x1ffe801000
==8672==    at 0x109141: functional_list_length
↪ (bug_stackoverflow.c:22)
```

```
[...]
```

To debug, many times, the programmers add printf to the code. This method requires knowing in advance what do you want to observe, to edit the code, to recompile and to loop many times.

It is a bit tedious.

Run the program in gdb. Draw the listing of the code at the fault. Add a printf each time you pass at line 10 (or the relevant line number). Restart the program.

Note that you can stop the execution of the program at any time with «Control-C» inside gdb.

```
you@ensipc$ gdb ./bug_stackoverflow
[...]
(gdb) run
(gdb) list
(gdb) where # then q to quit, otherwise, view roughtly 206000 frames
(gdb) dprintf 10,"h: %x, h->next: %x\n", h, h->next
(gdb) run # then confirm the restart from beginning
          # then Control-C to stop the execution
```

2.5 Catching bugs on the fly and changing values of variables

Warning : the buggy code is looping, consuming energy. You limit its execution to 300s maximum.

The code `src/bug_loop.c` is similar to section 2.4, page 5, but without stack overflow.

It also display the command to catch it on the fly with GDB. To do that you just need the process PID, available with the `ps` command.

Run the program, limiting its maximum execution time. Run the gdb catching command. Display the code and proceed up to line 14. Change the value of h. Continue up to the end of function, then up to the end of the program.

```
you@ensipc$ ( ulimit -t 300; ./bug_loop ) &
My PID is: 12419. Catch me with 'gdb ./bug_loop 12419' !
you@ensipc$ ps
[...]
 12419 pts/3    00:00:01 bug_loop
[...]
you@ensipc$ gdb ./bug_loop 12419
[...]
(gdb) list
(gdb) next # variable number (0-3) of next to arrive at l++
(gdb) next # ... (2 next here)
(gdb) set variable h = 0 # change h
(gdb) finish # go to the end of the fonction
(gdb) next
(gdb) next
```

```
(gdb) next # end of procesus processus
(gdb) quit # no confirmation here as processus is dead
```

2.6 Examining a specific location in memory : several examples

The bugs of this part are quite similar. Nevertheless, the behavior of the programs are quite different as the solutions to find them.

The baseline program is identical to the programs of section 2.2, page 3 and section 2.3, page 4.

The bug consist in slightly overfilling the initial allocation.

The code `src/bug_overreadheap_tiny.c` read a value just before the allocation, triggering immediately a SEGFAULT. Thus, it is quite easy with gdb and valgrind to find the line with the bug (line 27). You will use the e(X)amine command to display in decimal, the 20th first entries (`/20d`) of the array `p`.

```
you@ensipc$ ./bug_overreadheap_tiny
Erreur de segmentation (core dumped)
you@ensipc$ gdb ./bug_overreadheap_tiny
(gdb) layout src
(gdb) run
(gdb) x /20d p
(gdb) print p[i-2]
Cannot access memory at address 0x55595555929c
(gdb) quit # confirm
you@ensipc$ valgrind ./bug_overreadheap_tiny
[...]
==19345== Invalid read of size 4
==19345==    at 0x1091BC: fibon (bug_overreadheap_tiny.c:27)
==19345==    by 0x109276: main (bug_overreadheap_tiny.c:35)
==19345==  Address 0x404a8003c is not stack'd, malloc'd or (recently)
 ↪  free'd
==19345==
[...]
```

The code `src/bug_overwriteheap_tiny.c` writes a value just after the allocation (4 Bytes integer, 0 byte after the end). Nothing happen with the simple execution. Yet the bug is there and Valgrind is able to detect it.

```
you@ensipc$ ./bug_overwriteheap_tiny
you@ensipc$ valgrind ./bug_overwriteheap_tiny
[...]
==19897== Invalid write of size 4
==19897==    at 0x1091D2: fibon (bug_overwriteheap_tiny.c:27)
==19897==    by 0x109276: main (bug_overwriteheap_tiny.c:36)
==19897==  Address 0x4a89c80 is 0 bytes after a block of size 40,000
 ↪  alloc'd
==19897==    at 0x483F7B5: malloc (vg_replace_malloc.c:381)
```

```
==19897==    by 0x109230: main (bug_overwriteheap_tiny.c:33)
==19897==
==19897==
[...]
```

In the code `src/bug_overwriteheap_large.c` the allocation is too small. Hence, many writing overwrite it. Nevertheless, the detection is done only at the free. Valgrind is able to pinpoint the problem, but we may also ask gdb to check any modification just after the allocation. This technique is especially useful when the programmer know that a data structure was modified but do not when and by whom. It works also with read only, or both reading or writing.

Start gdb with the program. Put a breakpoint at malloc, line 33. Run the program. Do the malloc. Put a watchpoint. Note the direct use of the address, as gdb know the type of p. Thus `(p+SIZE)` shift the address 4 times more than `(0x5555555592a0 + SIZE)`. This mismatch of pointer arithmetic is exactly the bug here.

We do not use «layout src» to ease the copy and paste of the pointer value.

```
you@ensipc$ ./bug_overwriteheap_large
double free or corruption (!prev)
you@ensipc$ gdb ./bug_overwriteheap_large
(gdb) break 33
(gdb) run
(gdb) next
(gdb) print p
$1 = (unsigned int*) 0x5555555592a0
(gdb) watch *(int *)(0x5555555592a0 + SIZE)
Hardware watchpoint 2: *(int *)(0x5555555592a0 + SIZE)
(gdb) cont
(gdb) print i # fail at 1/4 of the expected length
$2 = 2500
(gdb) print size
$3 = 10000
(gdb) quit # confirm
```

2.7 Being between Scylla (not using malloc/free) and Charybdis (using malloc/free)

Into Scylla, using the stack, he fell, wishing to avoid Charybdis of using the heap (malloc/free).

The problem of using the stack, is that it is possible to write it everywhere, at all time. No tool may help you. The compiler will be happy, the C library is not used thus detect nothing, and Valgrind will let you happily overwrite your variables by strange means.

The code `src/bug_allocinstack.c` is similar to a common scheme in Python. But, Python always uses the heap to malloc every variable content, even if the programmer does not write «malloc». Not the reverse. The C code is a complete misinterpretation of that point.

```
you@ensipc$ ./bug_allocinstack
Erreur de segmentation (core dumped)
you@ensipc$ valgrind ./bug_allocinstack
[...] # no acurate indication
==21326== Invalid write of size 8
==21326==    at 0x1091A4: main (bug_allocinstack.c:28)
==21326==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

To be complete, the previous bug may be also track. Compiler may warn you in simple cases like this code. In the buggy code, the compiler is tricked by the pointer arithmetic operation. The watch point technic of section 2.6, page 7 is also useable. Finally, some authors advice to use «register» to all local variables to let the compiler warns you in case of memory stack aliasing in a pointer.

2.8 Static analysis of your compiler

With modern compiler (gcc-10+), the compiler may spend time to analyze your code at compile time. You may add the option `-fanalyzer` in the FLAGS at the start of `CMakeLists.txt`.

E.g. you will get a detailed warning for `bug_doublefree` code.

```
you@ensipc$ emacs ../CMakeLists.txt # ajouter -fanalyzer dans les flags
you@ensipc$ rm -i ./bug_doublefree
you@ensipc$ make
[...] # read and decrypt the long warning
```

2.9 WARNING(Valgrind becomes unuseable) : asking the compiler to add code to check allocations

You will not be able to use valgrind anymore, but you mays ask your compiler to add automatic check code with the option `-fsanitize=address` in the FLAGS at the start of `CMakeLists.txt`.

3 Linked list

The first exercise is the implementation of a linked list, in the directory `src/`, in the file `listechainee.c` (linked list in French).

You have to implement the following functions :

```
/* print each element of the list in the standard output */
void affichage_liste(struct elem *liste);

/* Create a linked list by creating elements and taking the value of
 ↪  each
 * new element in the array valeurs. Returns the first element address.
 ↪  */
```

```
struct elem *creation_liste(long unsigned int *valeurs, size_t
↪  nb_elems);

/* Free the linked list */
void destruction_liste(struct elem *liste);

/* Invert, in place, the linked list */
void inversion_liste(struct elem **liste);
```

3.1 Use a debugger !

The most fundamental point of a debugger is to be able to check, and modify, the state of the execution : current instruction, call stack, variable value, arbitrary memory value.

After programming `affichage_liste`, add your debugging function to print all the values of the local variables given as argument and the content of the lists.

Example :

```
void debug_inversion(struct elem *h1, struct elem *h2) {
    printf("head1= %p, head2= %p\n", h1, h2);
    affichage_liste(h1);
    affichage_liste(h2);
}

void inversion_liste(struct elem **liste) {
   struct elem *one_list_head= *liste;
   struct elem *another_list_head;
...
   Inversion loop is here, e.g. at line 90
...
}
```

Then, start GDB on the program, define a breakpoint, and a command at the breakpoint calling `debug_inversion` with the pointer arguments. Replace the arguments in the command by the right variables names of your code.

Every time, your execution hits the breakpoint, the `debug_function` will be called.

```
$ gdb listechainee
(gdb) break listechainee.c:90
....
(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>call debug_inversion(one_list_head, another_list_head)
>end
(gdb) run
...
(gdb) cont
```

```
...
(gdb) cont 10 # pass through without stopping 9 times (stop at 10)
```

3.2 Few words about the linked list inversion

Please contact your teacher in case of algorithmic problem !

One way consists in filling a temporary second linked list with a head insertion of each head element of original list.

In other words, as long as the list is not empty, first, remove the first element of the original list, second, insert it as the first element in the temporary list. In the end, update the original link pointer with the new list head of the temporary list.

Note that the deletion and insertion operations on the two lists may be combined in order to remove duplicate changes of the pointers.

3.2.1 C is not OCaml, Haskell or LISP

Note that any functional, recursive, implementation of your solution is wrong. Why ?

4 Binary operators

In the `ensimag-rappeldec/src` directory, complete the file `binaires.c`. The function `unsigned char crand48()` should use a global static variable X. It applies the following function on it :

$$X_{n+1} = (aX_n + c) mod (m)$$

with $m = 2^{48}$, `a = 0x5DEECE66D` and `c = 0xB`.

The function must return the byte corresponding to bits 32 to 39, starting with bit 0 for the least significant bit.

You should use arithmetic operators `+`, `*`, `%` and binary operators `<<`, `>>`, `&`, `|`, `~`, `^` (left shift, right shift, binary AND, binary OR, binary NOT, binary XOR).

5 Hand-made, simple, garbage collector

Programming a general-purpose garbage collector is quite difficult. Think about the performance of large Java programs. Nevertheless, if all the objects are identical, in constant number, and with well delimited references, a simple, stop-the-world, GC is doable in few lines of code.

You implement three functions : an initialization function, an allocation function and a garbage collection function.

5.1 Allocation and garbage collector

The elements of the linked list have the following structure :

```c
struct elem {
  int val;
  struct elem *next;
};
```

A single block of elements, an array, of elements is allocated once.

To follow the usage of each element, an array of same number of boolean is used through 3 given functions : read a single boolean value, write a single boolean value, reset to `false` all the boolean values.

```c
bool bt1k_get(long unsigned int n); // get value of bit n
void bt1k_set(long unsigned int n, bool val); // set value of bit n
void bt1k_reset(); // reset all bit to false
```

5.2 Forbidden functions for this exercise

The goal of this exercise is to manage the memory yourself, thus you must not use dynamic allocation functions to manipulate the `struct elem`. Thus, your code must not call `malloc()`, `free()`, `realloc()`, etc.

5.3 Your job

Implement in `src/elempool.c` two functions

```c
struct elem *alloc_elem();
void gc_elems(struct elem **heads, int nbheads);
```

Theses functions change the elements of `static unsigned char *memoire_elem_pool` in file `src/elempool.c` (and the array of boolean). The 1000 elements are allocated by the function `void init_elems()`. The initialization function is called at the beginning of every test function.

The output of `struct elem *alloc_elem(void)` function is :

0 or NULL : if all elements are already used,

address : the address of the `struct elem` to use.

The vector of boolean is used to keep track of the free elements. To find a free element, a simple linear check of every entry is sufficient for this exercise.

The garbage collection function is `void gc_elems(struct elem **heads, int nbheads)`.

It checks all the linked lists given in argument to find all the elements still in use in the lists. **A element is free if it is not part of one of the lists !**

At every call of `gc_elems()`, the vector of boolean has to be reset to false for all elements. Then the function will set to true all elements found in the lists.

6 Hello World !

In directory `src/`, complete the C file `hello.c`.

Your program should display

$$\forall\ p \in world,\ \ hello\ p$$

including the various utf8 characters. You should learn how to enter any UFT-8 symbols in your favorite editor (eg. `Ctrl-x 8 RET` in Emacs), or in your desktop environment.

7 Floating point number

In the directory `src/`, complete the C file `flottants.c`

7.1 Easy exercise

Compute the operation $0.1 + 0.2 - 0.3$ with the three different floating point number size (`float` , `double` , `long double`).

Initialization constants should use the right suffix (eg. `0.1` , `0.1f` or `0.1L`)

The three computed results will be print one per line in scientific notation (`[-]d.ddddddedd`, with the power of 10). The three results are given with respect of the increasing byte size of the data types.

Hint : you should create temporary variables.

7.2 Option : Not so easy exercise

Without the temporary variables, it is not so easy.

8 Function and function pointer

In the directory `src/`, complete the file `fqsort.c` to sort the array of complex numbers. The numbers must be sorted with respect of their complex argument (in polar coordinate, the angle θ, as in $z = re^{i\theta}$).

You must use the standard function `qsort`.

9 Other exercises if you finish early

9.1 Garbage collector

Copy and modify the linked list exercise with malloc, to use instead the GNU Boehm garbage collector. May be looping the tests will be necessary to check for memory leaks.

9.2 uthash/utlist

Copy and modify the linked list exercise with malloc, to use instead uthash/utlist list package. You may need to download uthash.h/utlist.h (http://troydhanson.github.io/uthash/)

9.3 IOCCC

Compile and play with some of the ioccc contests. For example, OMLET : Machine learning https://www.ioccc.org/2019/mills/.

9.4 Netcat

Implement a tool similar to netcat (server and client part) with

easy : TCP sockets (see Libc Info documentation for example) ;

easy : using simultaneously several buffers with readv/writeV (iovec) ;

medium : IPv6 support for your TCP sockets ;

medium : Enable ZeroMQ, or Nanomsg, sockets ;

hard : local DBUS communication ;

hard : Enable SCTP socket communications.