TP3: Debugging Tools GDB and Valgrind

The objective of this lab is to discover and apply the principles of debugging using GDB and Valgrind in order to find the possible algorithmic errors of the code, to spot infinite loops and solve segmentation faults. Do not hesitate to consult the "*GDB Quick Reference Sheet*".
The directory **/share/l3info/CUnix/tp3** has the files to be used in this lab. Copy this directory to your personal space.

1 General principles

The debugger is a software that helps the developer to analyze the bugs of a program. To do so, it allows to execute the program step by step, to insert breakpoints at conditions and program limes, to display the values of the variables at each time, etc.
GDB allows to do the following :

— start the program execution ;

— stop the program based on specific conditions ;

— examine what has happened when the program is stopped ;

— execute the program step by step, at instruction level or at machine level ;

— change something in the program and test to see the potential changes over the bug.

In order to be able to make the link between the source code and the machine code, gdb requires *debugging information*. This information is produced by the compiler when the flag `-g` is used. A certain number of options allows to controller the type of information provided by the compiler (see `man gcc`). More precisely, `-ggdb` allows to exploit the extension of gdb, and `-g3` to make visible the macros (among other things).

2 Debug

The program of the source code `bug1.c` takes as an argument an integer n from the command line and creates a table of size n. It initializes each element i with the value i^2 and then it deallocates the table. But it doesn't work. Why ?

1. Compile bug1.c with the command `make bug1`.
2. Start gdb and load the binary `bug1` (command `file`).
3. Start the execution with 3 as an argument (`run 3`) and report the line in which the error occurred
4. Display the stack of the function calls connected to the error (command `backtrace`).
5. Type `help breakpoints` to receive the list of the commands that allows you to use the breakpoints. Put a break point at the start of the function main, another at the function traitement. Remove the one from the function main.
6. Re-lance the program.

7. When you are stopped in the function traitement, put in surveillance the variable i (command watch). Use the command `watch i` to display i everytime it is modified.
8. Type `cont` to continue, and watch the execution until you find the problem.

3 Algorithmic Bugs

Look at the file `bug2.c`. The program searches for an element in the ordered table (`tab`) of size `nb` using dichotomy. Using the techniques that we have seen in the previous example to identify and correct the two existing bugs.

To watch the content of local variables, we can use the command `display`, and continue the execution step by step using the command `continue`. In this case, `watch` does not provide too much help : the value of the variables do not change after some time

To add a breakpoint to the function `recherche` at the fourth call, we can use either `ignore`, or `cond` (inside gdb, type `help ignore` and `help cond` for the syntax).

4 Other bugs

1. Find and correct the bug in `boucle.c`. Use the command `info locals` to understand the source of the problem
2. Find and correct the bug in `perror.c`.
3. Find and correct the bug in `prime.c`.

5 Bugs in the lists

The chained lists are a type of data structure used by the programs. But they are also the source of a wide range of bugs.

Use the `ddd`[1] (the graphic interface of gdb) to find and correct the bugs in `liste1.c` and in `liste2.c`. In case the code is not correctly loaded, select the open source dialog from the file menu. We will use the function of visualizing data structures, such as lists, to watch their evolution during the program execution. To do so, you display the pointer thelist and see the value it points to by double clicking its value.

It is possible that a bug does not manifest itself always. Depending on the used machine, the `liste2` may work correctly by chance. The tool valgrind allows to test these cases. Type `valgrind ./liste2`. What are the problems ? Use `--leak-check=full` to see details of leaked memory and `--track-origins=yes` to see where the uninitialized values come from.

6 More Valgrind

Use `valgrind --tool=memcheck` to report the missing initialisation of the pointer of `boom.c`

1. see the document ddd.pdf.