Universidad Zaragoza 1542

DEBUGGING

Fundamentals on Computing for Robotics, Graphics and Computer Vision

Prerequisites

To simplify the setup, this lab relies on the linux machines from the DIIS deparment. Either a local machine from the laboratories, our you can directly connect to `lab000.cps.unizar.es` machine, which has the same linux machine that the lab machines. So the single actual prerequisite is to be present in the lab or have an internet-conected computer. Please remember that your home directory is the same in all the machines of the department, so you are free to stop and continue working on a different machine.

Setup steps

Initially, you have to log into a machine directly or though ssh with `ssh lab000.cps.unizar.es` if you connect to lab000. Don't forget to add the flag −Y to have X Windows support. Supporting X Windows on your machine may require the installation of Windows Subsystem for Linux or XQuartz for Windows 10/11 or macOS, respectively.

Once you have logged in, we are going to clone the repo and build the binaries to start debugging them. So these are the required commands:

```
cd <your root directory for FoC repo>
git clone git clone https://github.com/adolfomunoz/FoC.git
cd FoC
```

Once the clonation has finished, let's continue building some test programs:

```
cd src/assignment_11_classes_materials
mkdir build-release
cd build-release
cmake -DCMAKE_BUILD_TYPE=Release ../ # generate the Makefile with cmake
make -j4 # compile the examples in parallel with 4 jobs
```

Now, inside the `build-release` directory, you should have all required binaries, and the debugging with gdb can start.

Initial Debugging

Factorial

To debug the factorial program, please type the following command:

```
gdb ./factorial
```

And you should see the following output with gdb's prompt:

```
lab000:~/.../FoC/src/assignment_11_classes_materials/build-release/ gdb ./f
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-20.el8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-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 ./factorial...(no debugging symbols found)...done.
(gdb)
```

Now, gdb is asking for your input command. The first step will be to insert a breakpoint in the main function and observe the stack and the source code.

```
(gdb) break main
Breakpoint 1 at 0x4008e0
(gdb) run
Starting program: /home/dario/04_docencia/FoC/src/assignment_11_classes_mat

Breakpoint 1, 0x00000000004008e0 in main ()
(gdb)
```

To list the actual code of the program you can use the `list` command.

```
(gdb) list
```

1. Why there is no output of the `list` command? Could you compile the code in a diffent built type to get more information?

Factorial with debug information

Please repeat the steps from the previous section.

1. What is the current output of the `list` command?

2. Create a breakpoint for the `factorial` function an then run the program until the breakpoint? What is the backtrace at this point of the execution?

3. Please delete the breakpoint for the `factorial` function and create another breakpoint in the line 9 of `factorial.cpp` source file and run the program again. *Please include the used commands in your response.*

Changing the value of variables

One powerful feature of gdb is its ability to change values in memory. For example, forcing some variables to a certain values can trigger a hard-to-find bug. Two commands can help with this task: `print` and `set`. Since gdb can change the value of a given memory position, first you have to find the address of the value you want to modity with `print` and then update the value with `set`. Imagine you want to change n in the first invocation to factorial, e.g.; invoke factorial with n equals to 10 and then let's change the value to 1:

```
# please add a breakpoint in line 5 of the code
# then, run the following commands
(gdb) p &n
$2 = (unsigned int *) 0x7fffffffd2ac
(gdb) set *((unsigned int *) 0x7fffffffd2ac) = 1
(gdb) p n
$3 = 1
(gdb) cont
Continuing.

Breakpoint 4, factorial (n=0) at /home/dario/04_docencia/FoC/src/assignment
5     if(n == 0) {
(gdb) finish
Run till exit from #0  factorial (n=0) at /home/dario/04_docencia/FoC/src/a
factorial (n=1) at /home/dario/04_docencia/FoC/src/assignment_11_classes_ma
9     return factorial(n-1) * n;
Value returned is $4 = 1
```

1. Please repeat the previous commands and describe their behaviour. Please not that the address of n may vary. gdb can also show you the assembly code at the same time you are debugging with the `layout split` command. You can return to the original layout with `tui disable`.

Conditional breakpoints

1. Please briefly describe the command required to have a condtional breakpoint in the factorial function when n is less than 5.

Segmentation faults

Segmentation faults are one the most prevalent errors working with pointers in c++. In this exercise, you need to find the segmentation faults in a program named `segmentation_fault.cpp` with gdb and write a correct version of the program in `fixed_segmentation_fault.cpp`, which is an empty file in the repo.

1. Please describe how the segmentation faults occur.

2. Please comment how to find the exacts errors with gdb.

3. Would an `std::vector` container fix the issue?

Assertions

1. Could you please add some assertions in your `max_min_heap_2d_matrix.cpp` program to detect invalid pointers and other causes of faults. Each assertion should include a justification.

Submission

Please group the all the files within a zip. In those assignments with text questions, add then at the beginning of the file as C++ comments.

Before submitting, remember to talk to your teacher and have a **small interview** describing and answering questions about the job you have done and the decissions you have made. This interview is *20%* of the total grade of the assignment, while the submitted material is evaluated for the other *80%*.

All source code files, must include a comment on top with the names and NIAs of the students involved on their development. Then, all source code files, including directory structure, must be compressed into a single `zip` file with the following naming:

- **`debugging_<nia1>_<nia2>.zip`**, where `<nia1>` and `<nia2>` are the NIAs of the involved students, if the work has been done in pairs. In this case, **only one** of the students must submit the work in Moodle.
- **`debugging_<nia>.zip`**, where `<nia>` is the NIA of the involved student if the work has been done individually.

The compressed file must not contain anything else besides the source code files. Particularly, do not submit any binary file, neither executable, library nor object. The submission of the `zip` file must be done through the corresponding task in Moodle before the stablished deadline.