Gnu Debugger in action

In this short hands-on you we will briefly see how to use and debug simple C/Fortan codes by means of Gnu Debugger.

short intro : What can I do with a debugger ?

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.

GNU Debugger, which is also called **gdb,** is the most popular debugger for UNIX systems to debug C, Fortran and C++ programs.

GDB uses a simple command line interface. We will learn how to use the basic command of such CLI

How does GDB Debug?

GDB allows you to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple command line interface.

Let us starting checking if gdb is installed and which version do we have:

```
>>gdb -v
GNU gdb (GDB) 8.2
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Let us now generate an appropriate executable with the debug symbol table

```
gcc -g hello.c -o hello.x
```

and then run the executable under the gdb

```
gdb hello.x
```

check the stack

Examining the stack is often of vital importance. With GDB you can have a quick and detailed inspection of all the stack frames.

```
backtrace[args]
n
-n
full
where, info stack
print the backtrace of the whole stack
print only the n innermost frames
print only the n outermost frames
print local variables value, also
additional aliases
```

check memory

watch points

You can set watchpoints (aka ‚keep an eye on this and that') instead of breakpoints, to stop the execution

Attach to a process

GDB built-in tui

ex02.c

```
17      {
18          int i, j;
19          const int  n=1000, m=1000;
20          double  a[n][m];
21          double  t1,t2;
22
23          t1=mysecond();
24          for ( i=0; i<1001; ++i) {
25              for ( j=0; j<1000; ++j) {
26                 a[i][j] = i+j;
27              }
28          }
29          t2=mysecond();
30
31          printf("time used %g\n", t2-t1);
32          return 0;
33      }
```

```
exec No process In:
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /u/exact/exact/Basic_debugging/ex02-c...done.
(gdb)
```

example: ex01.f / ex01.f90 / ex01.c.

- Task: run the program under the control of a debugger, set/delete break points, watches, inspect data.

Check the stack

Examining the stack is often of vital importance. With GDB you can have a quick and detailed inspection of all the stack frames.

```
backtrace[args]
n
-n
full
where, info stack
print the backtrace of the whole stack
print only the n innermost frames
print only the n outermost frames
print local variables value, also
additional aliases
```

check memory

watch points

You can set watchpoints (aka ‚keep an eye on this and that') instead of breakpoints, to stop the execution

Attach to a process

example 2: - ex02.f / ex02.f90 / ex02.c

This example introduces an "out-of-bounds" bug in one of the loops, by increasing the loop length without changing the size of the allocated memory block. Task: find cause for segmentation fault using the debugger: either through running in debugger or by generating a core dump and inspecting it with the debugger.

Post Mortem approach

Let us first check your limit to see if a core file can be dumped:

```
ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 127797
max locked memory       (kbytes, -l) 4096000
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) unlimited
cpu time               (seconds, -t) 900
max user processes              (-u) 80
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

and we set the core file size to unlimited this way:

```
ulimit -c unlimited
```

We can now execute the code normally, getting as expected a Segmentation fault

```
>./ex02-c
Segmentation fault (core dumped)
```

We can at this point prooced with a post-mortem analysis specifying the name of the executable plus the core file generated, in our case:

```
gdb ex02-c  core.14490
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-114.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /u/exact/exact/Basic_debugging/ex02-c...done.
[New LWP 14490]
Core was generated by `./ex02-c'.
Program terminated with signal 11, Segmentation fault.
#0  0x0000000000400746 in main (argc=1083140096, argv=0x408f600000000000) at ex02.c:26
26              a[i][j] = i+j;
```

The debugger shows the line where it got the segmentatation fault.

Interactive approach

In this case we run the code within the debugger from the beginning:

```
gdb ex02-c
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-114.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /u/exact/exact/Basic_debugging/ex02-c...done.
(gdb)
```

and once there we can run the code:

```
(gdb)run
Starting program: /u/exact/exact/Basic_debugging/ex02-c

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400746 in main (argc=1083140096, argv=0x408f600000000000) at ex02.c:26
26              a[i][j] = i+j;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-260.el7_6.4.x86_64
(gdb)
```

We can now close the execution and fix the program.

example 3: ex03.f / ex03.f90

This only fortran example have a similar, but more subtle version of the same bu inserted in example 2. In this case we do not get a segmentation fault, but rather a faulty result, due to the array "b" being located in memory just behind "a" and thus out-of-bounds accesses of "a" will write to "b". Note how the result of "b" is correct in the second part, and that it will get corrupted after we checked it.

Task: run and notice the inconsistency. Then compile with bounds checking enabled and locate the bug.

example ex04.c / ex05.c / ex06.c:

These are examples of memory leaks. We first allocate the array and then in ex04.c nothing is free()d. ex05.c corrects for the individual array storage, but misses the list of arrays allocation. ex06 is corrected. Task: run with valgrind and see the memory leak(s).