



F. Wermelinger
Office: Pierce Hall 211

Lab 6

Debugging with gdb and valgrind

Issued: April 17, 2023 Due: April 28, 2023 11:59pm

In this lab we are looking into the basics of debugging code using the GNU gdb¹ debugger as well as the valgrind² tool to locate memory bugs.

In the following you will report your findings in the file `answers.md`. Please add appropriate headings such that your answers can be related to the appropriate questions. For example, add a heading `# Task 1 a.)` for answers to task 1 a.).

Submission instructions: please see the lab submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#lab-submission>

Task 1: Debugging

Debugging is the process of finding anomalous behavior in a program by a tool that allows to step through the executable by lines of source code or even by the granularity of individual instructions.

A frequent debugging technique used is to add print statements in code followed by cycles of recompiling and running the code. This workflow is time consuming and inefficient especially when the process is iterative. In this task we are going to work with the GNU gdb debugger that allows you to interact with the program dynamically. For Macintosh users the debugger tool is called lldb which is part of the LLVM software tools.

You can find the gdb online documentation at <https://sourceware.org/gdb/current/onlinedocs/gdb/>.

- a) Before you can use the debugger you need to compile your code to include *debug symbols* and prevent the compiler from performing code motion (optimizations) that may obfuscate the source code and generated machine code. You can tell the compiler to include debug symbols in your code by passing the `-g` flag³ and prevent it from

¹<https://sourceware.org/gdb/>

²<https://valgrind.org/>

³It is good practice to always include this flag. It does not hurt performance and the included debug symbols can always be stripped away later.

optimizing the code by passing `-O0` (default if omitted). Inside the directory `code/t1` you are given three programs `hello.c`, `hello.cpp` and `hello.f90` written in C, C++ and Fortran90, respectively.

Write a small Makefile that compiles the three programs to `hello_c`, `hello_cpp` and `hello_f90`, respectively. The programs must include debug symbols and should not be optimized. You can use the `gfortran` compiler to compile Fortran code. Report the size of the three executables in the file `answers.md`.

Be sure to load

```
$ module load gcc/12.1.0-fasrc01
```

as we will work with this compiler in the lab.

Hint: You can use the command `wc -c file` to print the number of bytes in file.

- b) As explained above, the `-g` flag includes the symbol table in the executable. You can list the symbols in an executable with the `nm` command (see `man nm` for more information).

Investigate the symbol names for the `Hello()` functions in the three programs of the previous part. Document your observations in the file `answers.md`. Why does C++ have such weird looking symbols? Are `hello()` and `Hello()` the same functions in C? What about Fortran?

Hint: You can pipe the output of `nm` through `grep -i hello` to find the symbol more easily.

Hint: See also https://en.wikipedia.org/wiki/Name_mangling.

- c) You can always strip the symbol table from an executable. This might be useful when you ship code to a customer where the symbol table might not be needed. Stripping the symbol table will further reduce the size of the executable.

Use the `strip` command to strip off the symbol table from the three executables you have created above. See `man strip` for further information. Report the size of the stripped executables in the file `answers.md`. What happens if you try to list the symbol table using the `nm` tool?

- d) There are three ways to invoke `gdb`:
1. Wrapping it around an executable
 2. Attaching it to an already running program
 3. Loading a *core dump* with it

Here we are going to focus on the first option. The following introduces the basic `gdb` commands you need to be familiar with. Many other debuggers make use of the same command names. Before you start, make sure the `gdb` version is

```
$ gdb --version
GNU gdb (GDB) 11.2
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

If your version is not that recent, please execute the following commands and try again

```
$ $SHARED_DATA/local/install.sh
$ source $HOME/.bashrc
```

You can invoke gdb by executing, for example

```
$ gdb hello_c
For help, type "help".
Type "apropos word" to search for commands related to "word"...

Reading symbols from hello_c...
(No debugging symbols found in hello_c)
+(gdb)
```

(some output has been omitted for brevity). Note that there are no debugging symbols found. You can verify that by executing the list command

```
+(gdb) list
No symbol table is loaded.  Use the "file" command.
```

To fix this you must recompile the executables (because you have stripped them in the previous part). You can exit gdb by pressing Ctrl-d on your keyboard or with

```
+(gdb) quit
```

After you recompiled the executable and started gdb again, you can list the current source with

```
+(gdb) list
1      #include <stdio.h>
2
3      void Hello(void) { printf("Hello World!\n"); }
4
5      int main(void)
6      {
7          Hello();
```

```
8         return 0;
9     }
```

At this stage, the program is not running. If you want to start debugging your program you type

```
+(gdb) run
Starting program: /n/academic_homes/g_100601/u_399544_g_100601/hello_c
Hello World!
[Inferior 1 (process 6340) exited normally]
```

You can get help in gdb by typing `help` at the prompt. To get help for a specific command type, for example, `help run`.

To start stepping through your program you first need to define a *breakpoint*. See `help break` for more information. You can specify a breakpoint at the main entry point with

```
+(gdb) b main
Breakpoint 1 at 0x400517: file hello.c, line 7.
```

Note that as long as the command is not ambiguous with other commands, you can abbreviate it to save typing effort. When you now run your program again, the debugger will stop at the specified break point

```
+(gdb) run
Breakpoint 1, main () at hello.c:7
7         Hello();
```

Once arrived at a breakpoint you can investigate the program state. To continue running type `continue`. Note that you can get information about your current breakpoints with `info break`. To delete breakpoints use the `delete` command (see `help delete`).

Instead of typing `continue`, you can also step through the program starting from a breakpoint with the command `next` (stepping over function calls) or `step` (stepping into function calls). If you want to step at the instruction level use `nexti` or `stepi`, respectively. When you are inside a function and want to finish it you can type `finish` to execute until the selected *stack frame* returns. The `until` command is useful to step over a loop using only one command. While `next` will step through all iterations, `until` will automatically continue at jump instructions and stop once the loop has finished. You can also specify a line number argument to continue until the specified location. The `until` command will otherwise behave similarly to `finish`⁴.

Finally, if your program takes arguments, you can either specify a default set of arguments when you run gdb or pass the arguments to the `run` command. To run a gdb session where an executable takes arguments do

⁴See <https://sourceware.org/gdb/current/onlinedocs/gdb/Continuing-and-Stepping.html#Continuing-and-Stepping> for the full documentation.

```
$ gdb --args hello_args 2
+(gdb) show args
Argument list to give program being debugged when it is started is "2".
```

You can show the currently set arguments with the `show args` command and you can set them to something else with the `set args` command. If you just run `set args` the current arguments will be removed. The following

```
$ gdb --args hello_args 2
+(gdb) run
```

is equivalent to

```
$ gdb hello_args
+(gdb) run 2
```

Add a new target in your Makefile for the `hello_args.cpp` program and run it with `gdb` using different arguments. Set a breakpoint and get familiar with the above commands.

If you run the program without any arguments you will get the following runtime error

```
$ ./hello_args
Segmentation fault (core dumped)
```

A segmentation fault indicates that you access a memory address you are not allowed to. When you run the program in `gdb` (without arguments) to reproduce the segmentation fault, it will crash and `gdb` will stop at the point of incidence. At this point you can print the *backtrace* to list all the stack frames in the current execution path. This is extremely valuable information as you can precisely analyze the current values of variables to figure out what happened. To print the backtrace you can execute `backtrace`, `bt` or `where` in `gdb`. You can get your current frame location by typing `frame`. Right after the crash you should be on frame 0 which is the bottom of the frame stack (*recall that the stack grows **downwards***). You can move *up* the stack with the `up` command and *down* the stack with the `down` command. If you want to move to a particular frame you can pass the frame number to the `frame` command, see `help frame`. Variables are only accessible in the stack frame where they are defined (automatic storage) and you need to change to a particular frame in order to access them. *When you are in the debugger, all you do is examining the stack and its associated code.* Refer to slide 31 in lecture 2 to see where you are located in the virtual memory space of your process.

In each stack frame you can print variable values (therefore, adding explicit print statements in your code is not necessary) and even set them to a different value if you like. To print the value of a variable `var` use the command `print var` or just `p var`. There are different format options you can choose from. See the `examine` command `help x`

for these format options. You can set the value of a variable using the `set` command. This command allows to set many values in `gdb` and you should be explicit about what you want to set by executing `set variable var=<value>`. If there is no ambiguity with other subcommands you could omit the `variable` subcommand. See `help set` for more documentation. Apart from variables in your code you can also investigate and set the state of registers. To print the current state of the registers you can invoke `info registers` or you can print individual registers, for example `print $rax`. You can set the value of registers using the `set` command. Note that register names begin with “\$”. You can try this out by setting the value of the *instruction pointer* to zero (this register points to the address of the current instruction), which will cause your program to crash when you want to execute the next instruction:

```
+(gdb) print $rip
$14 = (void (*)(void)) 0x400731 <main(int, char**)+15>
+(gdb) set $rip = 0
+(gdb) print $rip
$15 = (void (*)(void)) 0x0
+(gdb) stepi
Program received signal SIGSEGV, Segmentation fault.
0x0000000000000000 in ?? ()
```

Run the command `gdb hello_args` and determine where the segmentation fault happens by examining the stack frames using the `backtrace` command. Find out the name of the function where the segmentation fault occurs and report it in your [answers.md](#) file. Did you call this function in your main program? What is the name of the function in your main program that causes the crash? Where does the memory violation happen in your main program?

- e) In this part you are going to debug an application. You are given an executable located in [\\$SHARED_DATA/lab6/echo](#) that you can copy into your local directory. When you run it, the program waits for you to input a string which it then echoes to `stdout`. The program works for example with

```
$ ./echo
Hello CS205
Hello CS205
```

but something goes wrong for longer strings

```
$ ./echo
Hello CS205 students
Hello CS205 students
Segmentation fault (core dumped)
```

Start a debugging session with `gdb echo` and try to figure out what goes wrong. You can list all of the source code with `list 1,29`. It might be helpful to first check what

is happening in the main function when you execute the program as shown above. You can inspect the source code of this function with `list main`. You will see there is only one function call to `echo()` in there, so the bad stuff must happen in this function. Before you focus on that, have a look at the assembly code of the main function with

```
1 +(gdb) disassemble main
2 Dump of assembler code for function main:
3     0x0000000000400606 <+0>:      sub     rsp,0x8
4     0x000000000040060a <+4>:      mov     eax,0x0
5     0x000000000040060f <+9>:      call   0x4005e9 <echo>
6     0x0000000000400614 <+14>:     mov     eax,0x0
7     0x0000000000400619 <+19>:     add     rsp,0x8
8     0x000000000040061d <+23>:     ret
9 End of assembler dump.
```

This assembly uses the Intel syntax. You will likely see the AT&T syntax when you reproduce this assembly code. You can set the assembly syntax in `gdb` to Intel with the command `set disassembly-flavor intel`. These are the main events that happen in this code:

1. At address `0x400606` the current stack pointer is *decremented* by 8 bytes. This creates the *stack frame* for the main function. *The stack grows downwards*.
2. At address `0x40060f` the `echo` function is called. The first instruction for that function is located at address `0x4005e9` (you can verify this with the assembly code of `echo` below). The `call` instruction will do two things:
 - (a) *Push the return address onto the stack*. The return address is the jump point where the instruction pointer must be set to when the `echo` function returns. The return address is the one of the instruction immediately following the `call` instruction, i. e., `0x400614`.
 - (b) Set the instruction pointer to address `0x4005e9`.The calling convention expects that every function called with the `call` instruction executes the `ret` (return) instruction as its last instruction.
3. At address `0x400619`, 8 bytes are added again to the stack pointer, effectively *releasing* the stack frame for the main function.
4. At address `0x40061d`, since the main function itself was executed by a `call` instruction, it must return with a `ret` instruction. This instruction performs the *inverse* of the `call` instruction:
 - (a) *Pop the top 8 bytes from the stack* (addresses are 64-bit).
 - (b) Set the instruction pointer to the popped value.The return *value* of the main function is passed in the `eax` register according to convention. The value in this register is `0` because the source code executes the `return 0;` statement at the end.

A visualization for the state of the stack when inside the `echo` function is shown in figure 1.

It might be helpful to *disassemble* the echo function and inspect the assembly code since there seems to be something wrong in there. You can do that in gdb with the following command:

```
1 +(gdb) disassemble echo
2 Dump of assembler code for function echo:
3    0x00000000004005e9 <+0>:    sub    rsp,0x18
4    0x00000000004005ed <+4>:    lea    rdi,[rsp+0x8]
5    0x00000000004005f2 <+9>:    call  0x4005a2 <gets>
6    0x00000000004005f7 <+14>:   lea    rdi,[rsp+0x8]
7    0x00000000004005fc <+19>:   call  0x400490 <puts@plt>
8    0x0000000000400601 <+24>:   add    rsp,0x18
9    0x0000000000400605 <+28>:   ret
10 End of assembler dump.
```

This assembly uses again the Intel syntax. In line 3 the *stack pointer* in register `rsp` (this register is always used for the stack pointer) is shifted *down* by 24 bytes (recall the stack grows *downward* therefore the `0x18` bytes are *subtracted* from the current stack pointer address). This allocates 24 bytes on the stack for the echo function. The `rdi` register is used for function arguments. In line 4 `rdi` is assigned the address that is computed from the current value of `rsp` plus 8 bytes and in the next line 5 the function `gets` is called. The argument to that function is the `buf` array (see `list echo`) and the compiler uses the first 8 bytes on the stack for this array. In line 6 the `rdi` register is again set to the address `rsp + 0x8` before the `puts` function is called. See figure 1 for a visualization of the situation on the stack. You can set breakpoints at a specific instruction address using the `*` operator before the location (see `help break`). For example, to stop at the first instruction in the echo function, you can use the address of that instruction shown above (note that address locations must be prefixed with `"*"` in gdb):

```
+(gdb) break *0x4005e9
Breakpoint 1 at 0x4005e9: file echo.c, line 19.
```

When you now run your program in gdb, it will stop at this instruction. To examine memory at a given location you can use the `x` command (examine). This command

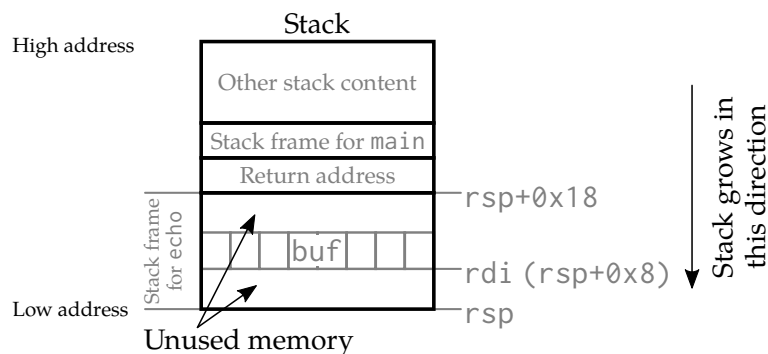


FIGURE 1: Stack layout when the echo function is called.

is useful to examine the stack. For example, if you set the breakpoint above and you want to examine the stack at the beginning of the echo function *before* the stack pointer is set to the address shown in figure 1, you can do the following:

```
+(gdb) x/24bx $rsp-0x18
```

This will examine 24 bytes (b) formatted in hexadecimal notation (x) starting at the address that is 24 bytes *below* the current stack pointer `rsp`. Note that when you examine multiple bytes like this in gdb, these bytes are displayed in *increasing* order (i. e., in the opposite direction in which the stack pointer moves). The 24 bytes you have just examined are illustrated in the "stack frame for echo" section in figure 1. If you now *step one instruction*, the stack pointer will be decremented by 24 bytes (and hence creates the stack frame for the echo function you are about to execute), see line 3 in the assembly above. If you now examine the first 24 bytes on the stack again, like this

```
+(gdb) x/24bx $rsp
```

you should see the same 24 bytes again. The stack pointer `rsp` is now pointing to the top⁵ of the echo function and ready to execute the function. As you debug the echo function, you might want to examine strings or characters on the stack. You can do this by simply changing the format with

```
+(gdb) x/24bs $rsp
```

or

```
+(gdb) x/24bc $rsp
```

respectively. See `help x` for more information.

Find the reason for the segmentation fault observed above. Please explain why the code crashes and what is dangerous in this code. Append your answers in [answers.md](#). If you set the breakpoint as described above, you can simply start over by typing `run` in gdb. This will bring you back to the start of the echo function and you can then stepping through instructions with the `next i` command. For the `call` instruction in line 5 above, you will need to type in a string and hit enter when done. When the instruction finished, you can examine the stack to see what happened.

Hint: Use `list echo` or `list gets` to list the source code of these two functions.

Hint: The `gets` function reads characters from `stdin` and saves them in `buf`.

Hint: Try to figure out at how many characters the code generates a segmentation fault. Compare the number of characters you count with the observations you get by looking at the code. Does the number you count make sense?

⁵Remember that the stack grows *downward*, you could also say "bottom" in this context. It does not really matter from which angle you look at it, as long as you understand how the stack works.

- f) In this last part you are going to debug another application that does some floating point computation. You are given the code [code/t1/float.c](#). Compile the code with debugging symbols. When you run this code the result is

```
$ ./float
Result = -nan
```

NaN means “not a number” and is defined in the IEEE 754 standard.⁶ NaN are never equal to anything, not even themselves. You can use this property to figure out when a floating point number turns NaN as the condition `x != x` will evaluate to true. In gdb you can define *conditional* breakpoints which are very useful to let the code run and only stop when the condition is met. The point where your simulation starts to blow up is usually when a variable turns NaN and it is useful to figure out when that happens such that you can investigate the states of other variables and track down the root cause. For example, you can define a conditional breakpoint in gdb with

```
+(gdb) break float.c:8 if x != x
Breakpoint 1 at 0x1178: file float.c, line 8.
```

The breakpoint set at line 8 in file `float.c` and will activate if the condition `x != x` for variable `x` evaluates to true.

Another useful technique for debugging is to define a *watchpoint* (see `help watch`). A watchpoint is different from a breakpoint in that it will watch a memory address and stop execution only when the value at that memory address changes. You could define a watchpoint (we do not need it for this task) with

```
+(gdb) watch x
Watchpoint 2: x
```

Use `info break` to see all defined breakpoints (including watchpoints) and how often you have hit them. If you are only interested in watchpoints use `info watch`.

Find the iteration index `i` at which the relevant variable in the [code/t1/float](#) executable becomes NaN. Please report your finding in the file [answers.md](#).

Task 2: Debugging Memory

So far we have seen two problems with pointers in this lab:

1. the pointer points to an invalid memory address which happened in the `hello_args.cpp` program when no argument was passed (`argv[1]` points to `0x0`)
2. accessing memory out-of-bounds as was the case in the `echo` executable

A third memory violation often remains undetected and your program runs normal until some point. Such memory errors are hard to detect and we need another tool to catch them. Consider for example the following code

⁶https://en.wikipedia.org/wiki/IEEE_754

```
while (true) {  
    double *ptr = new double[1000]; // allocates 8kB  
}
```

After every iteration, the address the pointer `ptr` holds is overwritten and the 8 kB memory chunk is lost in the void, i.e., the memory *remains allocated* but the reference to it is lost. This will flood your RAM and eventually crash your whole system, not only the program. Such a bug is called a *memory leak*. Finding such bugs in the debugger is hard and inefficient.

The tool for this job is called `valgrind`.⁷ An alternative would be to use the `-fsanitize=address` flag with GCC or LLVM compilers. We are not going to discuss them here. See `man gcc` for example. In order to use `valgrind` on the cluster you need to load

```
$ module load valgrind/3.13.0-fasrc01
```

Note that if you use the GCC 12.1.0 compiler to build your code, `valgrind` may complain about an error. You can use GCC 10.2.0 instead to avoid these errors.

- a) Running `valgrind` is really easy and convenient as you do not need to create a specially instrumented executable as it would be the case with the `-fsanitize` option to `gcc` or `clang`.

You are given the code `code/t2/mem_leak.cpp` which you can compile with the provided Makefile. When you run a memory leak check on the executable you will find that there is a problem. You can run `valgrind` like so

```
$ valgrind ./mem_leak
```

or with a full leak check

```
$ valgrind --leack-check=full ./mem_leak
```

Have a look at `man valgrind` too. In the heap summary you can see that the total number of memory allocations does not match the total number of frees, resulting in memory leaks. Using this code in production would be bad.

Find the cause of the memory leak in the `code/t2/mem_leak.cpp` code and fix it. When the bug is isolated, the leak check should return clean summary.

A correct program should result in the following leak check result:

```
$ valgrind ./mem_leak  
==2525== Memcheck, a memory error detector  
==2525== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==2525== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info  
==2525== Command: ./mem_leak  
==2525==
```

⁷<https://valgrind.org/>

```
Vector = [0 1 2]
Matrix = [0 1 2 3 4 5 6 7 8]
==2525==
==2525== HEAP SUMMARY:
==2525==      in use at exit: 0 bytes in 0 blocks
==2525==    total heap usage: 5 allocs, 5 frees, 72,836 bytes allocated
==2525==
==2525== All heap blocks were freed -- no leaks are possible
==2525==
==2525== For counts of detected and suppressed errors, rerun with: -v
==2525== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Hint: The bug is a typical C++ issue related to polymorphism