**COMP 4736 Lab 02: Introduction to Debugging Tools**

In this lab, we will learn to use simple debugging tools like gdb and valgrind. You are provided with a few buggy programs with this lab: pointers.cpp, fibonacci.cpp, and memory\_bugs.c. You are expected to debug them and demonstrate your understanding of debugging tools during your evaluation.

**Part A: Debugging with GDB**

A debugger is a program that runs other programs, with the user being allowed to exercise control over how these programs run. Users can insert breaks in the program, examine variables and so on, during program execution. GNU Debugger, which is also called gdb, is the most popular debugger for UNIX systems to debug C/C++ programs. GDB can only be used to find runtime or logical errors. Please note that compile time errors are detected by the compiler and not a debugger. In this part of the assignment you will use GDB to debug simple logical errors and a segmentation fault runtime error.

**GDB and G++ Installation**

The first step to using GDB is to install it on your computers. You will also need to install G++ if it is not installed yet.

**(For Windows WSL)**

You may need to install the GDB 13.2 if you run into problems when inserting breakpoints. To install this latest version, you will need to compile it yourself.

First, install GMP (libgmp-dev).

sudo apt-get install libgmp-dev

Then navigate to your desired directory, and download the GDB source code.

cd [your\_directory]

wget " https://ftp.gnu.org/gnu/gdb/gdb-13.2.tar.gz"

Extract the package.

tar -xvzf gdb-13.2.tar.gz

Configure and compile it. (This step can take a while.)

cd gdb-13.2

./configure

make

Finally, install GDB.

sudo make install

To verify GDB is installed properly.

gdb –version

**(For Other VMs)**

You can simply install GDB 12.1 via apt-get and try first. If you run into problems when inserting breakpoints, uninstall GDB 12.1 and follow the steps for Windows WSL to install GDB 13.2.

To uninstall GDB (if needed).

sudo apt-get remove gdb

To install GDB via apt-get.

sudo apt-get install gdb

G++ can be installed via apt-get.

sudo apt-get install g++

**How to Use GDB**

Before using GDB you will have to compile your program using gcc or g++ to generate an executable. The -g flag is particularly important as it generates debugging information that gdb can use.

gcc -g c\_source\_name -o executable\_name -Wall

or

g++ -g cpp\_source\_name -o executable\_name -Wall

GDB is invoked with the shell command gdb. Once started, it reads commands from the terminal until you tell it to exit with the GDB command quit. You can get online help from gdb itself by using the command help. You can run gdb with no arguments or options; but the most common way to start GDB is with commandline arguments, specifying an executable program as shown below:

gdb executable\_name

The command above will attach gdb to your executable and open the gdb shell in which you can write commands to debug your program.

**GDB Commands**

Here are some of the frequently used GDB commands. The information below is sufficient to attempt the exercises in this lab. For more information, please refer to the man page of GDB.

<https://www.tutorialspoint.com/gnu_debugger/installing_gdb.htm> <https://www.gdbtutorial.com/tutorial/how-install-gdb>

**break**, **b**

Set a breakpoint at specified location.

**break** [file:]function

Set a breakpoint at function (in file).

**break** [file:]linenumber

Set a breakpoint at a line number (in file).

**run**, **r** [arglist]

Start your program (with arglist, if specified).

**backtrace**, **bt**

Backtrace: display the program stack.

**print**, **p** expr

Display the value of an expression.

**continue**, **c**

Continue running your program (after stopping, e.g., at a breakpoint).

**next**, **n**

Execute next program line (after stopping), with stepping over any function calls in the line.

**step**, **s**

Execute next program line (after stopping), with stepping into any function calls in the line.

**list**, **l**

List specific function or line.

**list** [file:]function

List around beginning of that function (in file).

**list** [file:]linenumber

List around that line (in file).

**help**, **h** [name]

Show information about the GDB command name, or general information about using GDB.

**quit**, **q**

Exit from GDB.

**Part B: Memory Check with Valgrind**

Valgrind is a memory mismanagement detector. It helps you identify memory leaks, deallocation errors, use of uninitialized memory, and various other such memory usage errors. In fact, Valgrind is a wrapper around a collection of tools that do many other things, e.g., cache profiling. However, here we focus on the default tool, memcheck.

**How to Use Valgrind**

The first step to using GDB is to install it on your computers. You will also need to install G++ if it is not installed yet.

sudo apt-get install valgrind

Make sure you compile your program with debug information (i.e., with -g option).

You can now run Valgrind against your executable to figure out memory related bugs in your program using the following command (see the man page to learn more about the various options.)

valgrind −−tool=memcheck −−leak-check=yes −−show-reachable=yes −−num-callers=20 executable\_name

**Questions:**

1. Debug the pointers.cpp program given to you using GDB. The program contains some pointer related operations. A bug has been deliberately introduced in the code so that it generates a segmentation fault. Your task is to use GDB to find the line number of the wrong statement. Please make use of the GDB commands provided above in order to find the bug.

A screenshot of a computer screen

Description automatically generated

Error at Line 13.

1. Debug the fibonacci.cpp program given to you using GDB. This program is supposed to print fibonacci numbers until a certain value of n. However, there is a logical error introduced in the code which causes it to print wrong output. Your task is to use GDB to debug the program. You must insert suitable breakpoints, pause program execution, print intermediate values of variables from GDB, and monitor the execution step by step in order to find the logical error. Even if you can identify the error without stepping through the code, you must be able to demonstrate the process of debugging using GDB during your evaluation.

A black background with white text

Description automatically generatedA computer screen with white text

Description automatically generated

I believe these 2 lines are incorrect. Assuming we are trying to print the first 10 Fibonacci sequence, these 2 lines show a logical error. The line 16 should assign “second\_last” to “last” variable and on line 17, “next” should be assigned to second\_last variable. This will actually calculate the Fibonacci sequence. From the current code, it prints the incorrect sequence as shown below.

A screen shot of a computer

Description automatically generated

1. A program memory\_bugs.c is provided to you. This program is riddled with memory bugs. You may be able to find some bugs just by looking at the code also. Valgrind can help you find these bugs automatically. Your job is to compile the program using the command provided above and use Valgrind to find the possible issues present in the program. You should first understand the different issues Valgrind can detect, and then use the command given above to find the issues present in the program. You might be asked to provide possible reasons and fixes for those issues during your evaluation.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer screen

Description automatically generated

For the first error mentioned in the first paragraph, the error occurs when attempting to print the array but its only providing 10 bytes in the write argument which should instead be 4 bytes for int multiplied by 10 for the total number of elements in the array.

For the second error, there are actually multiple issues with this. Firstly, we originally allocated 30 bytes to the variable p but then we allocate once again on line 22 for 12 bytes which then we lose access to those original 30 bytes causing a memory leak. On the next we then free the 12 bytes which is fine but then we attempt to assign a char “A” to p variable when we just freed the memory so that’s another error.

Leading on from the previous error, the next error is related to that as we try to print the p variable right after assign a char value of “A” but since we freed after allocating the memory this would be an error to read the variable.

Lastly, we allocate 50 bytes to variable q which never gets freed anywhere in the code so that is another memory leak issue. In total we lost 80 bytes of memory since we did not free them before it which is shown in the heap summary results as shown in the screenshot.

**— End of Lab 02 —**