8. Debugging

In this chapter, we will cover debuggers. Debuggers are often overlooked by newer programmers in favor of “printf debugging”. Perhaps this due to the fact that debuggers seem complex and daunting. In this chapter we will try to debunk this myth.

The first thing that needs to be mentioned is that it is possible to store debugging symbols within the binary that the compiler generates. These debugging symbols help the debugger map machine instructions back to the source code. Remember that generally the compiled binary is a translation of your source code and the actual source code is lost. That is, without these debugging symbols you would have to debug your program on a machine instruction level. In order to tell the compiler to include this information, we use gcc with the -g flag. For example:

```
$ gcc -g -o test test.c
```

This will generate a binary with the name, test and this binary will contain the debugging symbols. You will notice that the binary is also larger than if it were compiled without debugging symbols.
8.1. Valgrind - Memcheck

From their website \(^1\), “Valgrind is an instrumentation framework for building dynamic analysis tools.”. That is, valgrind is actually much more than a simple debugger. However, there are tools that leverage valgrind which can be very helpful in debugging C programs. Specifically, in this course we will consider Memcheck. Memcheck is a tool for detecting memory leaks, double frees, accesses of unallocated or uninitialized memory, etc.

Installing Valgrind locally is generally as simple as searching through your Linux distribution’s application repository. To run it, simply call `valgrind` with the appropriate options and the binary you want to debug. If you are interested in using Memcheck, we call valgrind as follows (assuming our binary is called `test` and has debugging symbols included):

```
$ valgrind --tool=memcheck ./test
```

This will call and run your program. If your program requires input, you may interact with it as if you had run it natively. After your program has run, Memcheck will provide a summary of memory usage and errors. If you indeed have a memory leak, you may notice that Memcheck only provides a summary of the problem with the hint that we should “Rerun with –leak-check=full to see details of leaked memory”. This would look as follows:

```
$ valgrind --tool=memcheck --leak-check=full ./test
```

Remember, a memory leak is simply a situation in which memory is allocated and never specifically deallocated.

If you have a memory leak in your program, you may see a report that looks something like this:

```
5 bytes in 1 blocks are definitely lost in loss record 1 of 1
at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x40054C: allocate_something (test.c:11)
by 0x40056C: main (test.c:18)
```

The first line tells us that a single block of 5 bytes (i.e., some data structure of 5 bytes) was allocated and no pointer was maintained. The indented portion after the first line is a call trace or stack trace. This provides a sequence of functions that were called to get to the allocation within our code, starting at the bottom. We see (starting at the bottom), that the first function is the `main` function. You will notice the text in parenthesis after the name of the function. In this case, it is “test.c:18”. This indicates to us that at this location, the program called the next function in our trace. In this case, it is the `allocate_something` function. Here we see that this function calls malloc on line 11 of `test.c` from the text in parenthesis and at the call to malloc the trace stops. This indicates to us that the offending allocation happens on line 11 of `test.c`, which in fact is the case. With this information, we can now try to determine the situation that lead to this allocation never being freed and correct it.

\(^1\) http://valgrind.org/
8.1.1. Memcheck Error Overview

In addition to the memory leaks outlined above, Memcheck is able to detect other memory errors. If Memcheck finds other errors, it will print out a message along with a call trace similar to the example we saw above with respect to memory leaks. This section briefly describes each of these error types that Memcheck is capable of detecting. The following information is taken from the official Valgrind documentation. For a more detailed explanation of each error, please see this reference.

Invalid reads/writes This error indicates that your program is reading from or writing to memory which it shouldn’t.

Use of uninitialized values This error indicates that your program is using a variable which has not be initialized. That is, the variable has never been assigned a value.

Use of invalid values in system calls You will probably not see this error all to often unless you are using system calls directly. This error indicates that some value passed to a system call is uninitialized or unaddressable.

Illegal frees This error means your program is freeing an area of memory that cannot or should not be freed. This generally comes in the form of a double free. A double free simply means that your program tries to free the same area of memory twice.

Inappropriate allocation function As we are exclusively using malloc and free for now, this error should never occur. There are in fact more way to allocate memory in C++. This error means that your program is trying to “mix and match” allocation and deallocation functions. That is, if I use a C++ specific allocation mechanism, I cannot use free to deallocate it, but must rather use a specific deallocation function.

Overlapping source and destination This error indicates that you are calling a function such as strncpy, which requires a source and destination buffer, with overlapping source and destination buffers. This can result in strange results and should never happen.

Memory leak detection This error was explained in a bit further detail in the previous subsection. It indicates that memory has been allocated and was never deallocated (freed).

8.1.2. Reference

For a look at the complete Valgrid documentation, see [http://valgrind.org/docs/manual/](http://valgrind.org/docs/manual/)

8.2. GDB

GDB stands for “GNU Debugger” and is the de facto standard debugger for all GNU operating systems. GDB is quite powerful and has many features. We will concentrate on the most basic features required for the debugging of simple programs.

To get started with GDB, simply run gdb with your binary as the first argument. Make sure that your binary was compiled to include debugging symbols. The call should look like this (assuming your binary is called test):

$ gdb ./test

This will start gdb and give you a command line environment. Notice that your program has not yet started execution at this point.

Also, you can exit GDB at any time with the quit or q command on the GDB command line.

8.2.1. Executing your Program

To execute your program once you have started GDB, use the run command. If you want to execute your program with some parameters, you may include these after the run command. For example, to start your program execution with the command line parameters abc and 123, simply run:

(gdb) run abc 123

This will run your program with the command line arguments abc 123.

You will notice that your program will simply run through as if it had been executed on the command line. The exception to this is if your program ends in an unhandled exception (e.g., a segmentation fault). In this case you will be able to inspect the contents of variables and registers immediately after the exception. This can be helpful if you are trying to debug segmentation faults, for example. Additionally, you may press CTRL+C at any time while your program is running it to pause execution. This will pause execution, but you cannot be sure exactly where your program will pause. Finally, for more fine grained control, your program will pause execution if it hits a breakpoint that you set at an earlier time point. Breakpoints and their use will be discussed in a further subsection.

There are some additional mechanisms by which GDB can pause the execution of your program, but we will not cover them in this primer.

8.2.2. Listing Source Code

At any time, you can have DBG list the source code around your current position with the list command. If you would like to print the source code for a specific location, you can specify a filename and line number in the form of filename:line_number. For example,

(gdb) list test.c:7

will print lines of source code around line 7 of test.c (assuming test.c exists), while a simple
(gdb) list
will print lines of source code around your current location.

8.2.3. Breakpoints

As mentioned in a previous subsection, you may set breakpoints in your code. A breakpoint is simply a point in the code where you would like the execution to pause so that you can further inspect the state of your program. This might include looking at the values of variables or registers, for example.

Setting Breakpoints

To set a breakpoint use the `breakpoint` command (or `b` for shorthand).

The `breakpoint` command requires some additional input. GDB must know where you want to set your breakpoint. There are two variations that we will cover in this lecture.

First you can set a breakpoint at the beginning of any function by passing the function name to the `breakpoint` command. For example,

(gdb) breakpoint func1

will set a breakpoint at the start of the `func1` function (given this function exists).

Additionally, you can set a breakpoint by passing the filename and line number to GDB. To do this, you simply pass this information to the `breakpoint` command in the form of `filename:line_number`. For example,

(gdb) breakpoint main.c:12

will set a breakpoint on line 12 of the `main.c` file (given this file exists).

Once you have set your breakpoints, you can run your program as described in a previous subsection and GDB will automatically pause your program when a breakpoint is hit.

Listing Breakpoints

Of course, you can set multiple breakpoints at once. To list all your breakpoints, use:

(gdb) info break

This will list all set breakpoints.

Deleting Breakpoints

To delete a breakpoint use the `delete` command. By itself, this will delete all breakpoints. If you pass it an argument, you can specify which breakpoint to delete. You must pass `delete` the breakpoint number of the breakpoint you wish to delete as given by the `info break` command. For example, if I wish to delete breakpoint 2 as specified by `info break`, you would execute:

(gdb) delete 2

or to delete all breakpoints, you would execute:

(gdb) delete
8.2.4. Continuing Execution

Once your program execution is paused (usually through the use of a breakpoint), you will eventually want to continue execution. There are several ways to go about this.

**Continue**

In the simplest case, you may simply want to continue execution until the next breakpoint is hit or your program ends. In this case, you can simply execute the command `continue` (or `c` for shorthand). That is, to simply continue execution, execute:

```
(gdb) continue
```

**Finish**

Another simple case is the situation in which you continue execution until the end of the current function. This can be accomplished with the `finish` command. Its use is straightforward, simply execute:

```
(gdb) finish
```

**Single-stepping**

In addition to simply continuing execution of your program you also have the ability to walk through your program by executing one line at a time. This is called single-stepping. There are two commands that can perform this for you, namely `step` (or `s` for shorthand) and `next` (or `n` for shorthand). They differ in one very important respect: `step` will single-step into functions, while `next` will single-step over functions. That is, `next` will execute a function as if it is a single line and `step` will take you into the function that is being called.

These are very simple commands that do not require any parameters for the purposes of this primer and can be used as follows:

```
(gdb) step
and
(gdb) next
```

8.2.5. Inspecting State

Any time your system is paused you can also inspect the current state of the machine. This includes inspecting the values of variables and registers.

**Show Arguments**

At any time while the system is paused, you can inspect the arguments of the current function. This is accomplished with the `info args` command. That is,

```
(gdb) info args
```

will print all arguments of the current function along with their current values.
Show Local Variables

In addition to the arguments of a function, GDB will allow you to inspect the local variables of the current function. This can be achieved with the `info locals` command. For example,

\[(gdb) \text{info locals}\]

will print all local variables and their current variables within the current function.

Show Arbitrary Variables

Finally, it also possible to print the value of arbitrary variables with the `print` command. This command must be passed the name of a variable as an argument. Of course, any printed variable must currently be in scope. This command may be useful if you are interested in a global variable or a variable that is dynamically allocated. For example,

\[(gdb) \text{print x}\]

will print the value of a variable named \(x\) (given a variable of this name exists in the current scope).

8.2.6. References

As I mentioned at the beginning of this section, GDB has many functions above what I have introduced here. There are many additional features that you may find useful as you get comfortable with using debuggers. For the complete online documentation, see [http://sourceware.org/gdb/current/onlinedocs/gdb/](http://sourceware.org/gdb/current/onlinedocs/gdb/).

Additionally, there are several nice GDB “cheatsheets” out there with a reference to the most commonly used commands. Two that might find useful can be found at [http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf](http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf) and [http://www.cs.berkeley.edu/~mavam/teaching/cs161-sp11/gdb-refcard.pdf](http://www.cs.berkeley.edu/~mavam/teaching/cs161-sp11/gdb-refcard.pdf).