

AERE 361

Computational Techniques for Aerospace Design

Laboratory Manual



Spring 2021

Prof. Matthew E. Nelson
Teaching Assistants
Hang Li

Contents

1	Lab 8 - Debugging with gdb, valgrind and Unit Testing	3
1.1	Pre-Lab	3
1.1.1	Background	3
1.1.2	Objectives	3
1.1.3	Methodology	4
1.2	Grading	4
1.2.1	Report	4
1.3	Lab Work	5
1.3.1	Getting Started	5
1.4	Pre-Compile and Compile-Time Debugging	5
1.4.1	More Warnings!	6
1.5	GDB Basics	6
1.6	Introduction to Valgrind	12
1.6.1	Running example	12
1.6.2	Prepare the program	12
1.6.3	Running your program under Memcheck	12
1.7	A Summary of Errors Valgrind can report	14
1.7.1	Illegal read / Illegal write errors	14
1.7.2	Use of uninitialized values	15
1.7.3	Illegal frees	15
1.7.4	Passing system call parameters with inadequate read/write permissions	15
1.7.5	Overlapping source and destination blocks	16
1.8	Unit Testing	16
1.8.1	μ unit Basics	17
1.9	Exercises	18
1.9.1	Exercise 1 - GNUMDebugger	18
1.9.2	Exercise 2 - Valgrind	18
1.9.3	Exercise 3 - Valgrind	18
1.9.4	Exercise 4 - More GNUMDebugger	19
1.9.5	Exercise 5 - Unit Testing	19
1.9.6	Exercise 6	19
1.10	Report	21
1.11	Wrapping Up	21

1 | Lab 8 - Debugging with gdb, valgrind and Unit Testing

Reference:

Introduction to Scientific and Technical Computing:

Chapter 5: Debugging with gdb

Chapter 10: Prototyping

C Programming Language, 2nd Edition:

Chapter 5– Pointers and Arrays

1.1 Pre-Lab

Errors in computer programs are almost always due to human error. It can be as simple as a typing mistake to a more complex problem in not understanding how computers work. For this lab, we will examine some tools such as GNUDebugger, Valgrind and performing Unit Testing with μ unit to better track down and fix these errors.

1.1.1 Background

When compiling, gcc determines memory requirements for each section of the program. When the program is run, instructions placed by the compiler request memory for your variables from the operating system. For example, when you declare a simple int variable, where is it stored? When you run the program, that variable is given an address by the operating system. We can request that address be printed out using the "&" operator, called the reference operator. We could interpret the line of code:

```
printf("Address: %u", &var);
```

as "print the address of var" where "&var" is the address of var, instead of the value.

You've seen this before when using scanf. Scnaf requires a memory address to save it's results, so we have been using the & syntax to pass it the address of the memory we had reserved. Working directly with memory addresses like this opens up several entire classes of bugs. One of the most common is the "seg-fault" that you may have seen by now. A seg-fault is caused by your program trying to read or write to memory your program isn't allowed to touch - so it is killed by the operating system.

Catching these bugs can be harder or impossible to find with simple print statements – in this lab we introduce tools and techniques for more sophisticated debugging.

1.1.2 Objectives

- Debugging and using GDB

Deliverable	Points
Exercise 1	5
Exercise 2	5
Exercise 3	10
Exercise 4	10
Exercise 5	20
Exercise 6	20
Report	50
Total Score	120

Table 1.1: Rubric for Lab 8

- Debugging with Valgrind
- Memory: `malloc` and `free`
- Unit Testing

1.1.3 Methodology

For Lab 8, students can opt to complete the lab in a couple of ways. In Lab 1, you were introduced to Repl.IT and using the online editor. This allows you to clone your repo to Repl.IT and push changes to your develop branch. Repl.IT also has a nice feature where the graders and TA can also view your Repl.IT so we can help you. We have designed the lab with Repl.IT in mind.

Students could use other sources. The HPC also has GCC and could be used. Other IDE's could be used. However, a word of caution. While it is true you can use these, you are on your own. The support that myself, my TA and graders can give you will be limited. You have been warned.

To Do:

In your lab report, under the methodology section, explain what you used to work on your programming. Don't forget the objectives, which you can copy and past from this write-up. That will conclude the Pre-Lab section of the report.

1.2 Grading

There are 6 exercises in this lab. Please see the rubric in Table 1.1. This point breakdown is also in the README.md file and you should check things off in there just like you did for Lab 1. An AutoGrader will be used to grade that you completed the exercises provided. This lab is a little different as we have some exercises that will add to your report score. The base 50 points for the report remains the same. Exercise 4 and 6 increase the report to 80 points. 40 points is then from the other exercises. This lab is more on understanding these tools and getting some of the basics down.

1.2.1 Report

In the report folder you will find a starter "main.tex" file. Like the last lab, this template is very barren as you are expected to fill it out. At this point, you are now expected to read the lab writeup and when asked, fill out the appropriate section.

Be Careful:

Make sure your report compiles. It should not only compile in Overleaf but needs to compile using our LaTeX builder script in GitHub. We will always grade what the GitHub action generates. This happens during a pull request and can be triggered manually.

1.3 Lab Work

1.3.1 Getting Started

To start with, go to the link found in Canvas for this lab. Click on it, and then clone the repo. You should have a an icon for Repl.IT in your Readme.MD file. You can click on that to clone this to Repl.IT and start working. This is the same process you did for Lab 1.

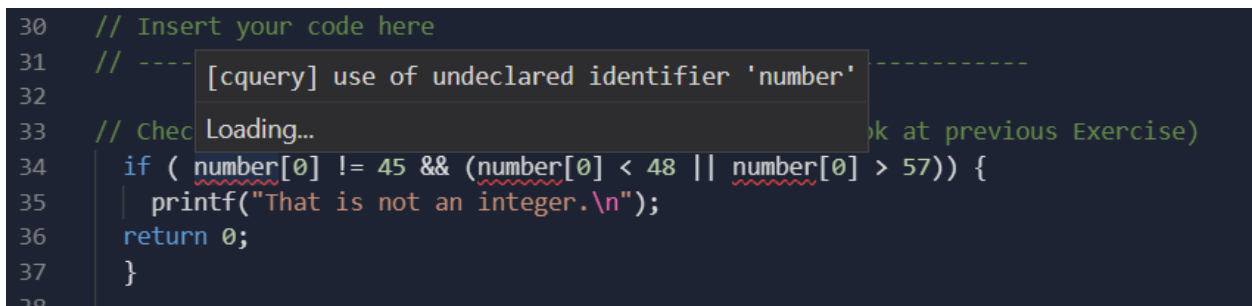
Be Careful:

Don't forget that your first step is to create a new branch called “develop”. You will do all of your programming work in this branch.

If you decide to not use Repl.IT, then you can use the normal *git* commands to pull the repo, create a branch, and do all of your commits. Any computer running GCC and has the Git CLI tools installed should work. But, as we stated, you are own your own with this.

1.4 Pre-Compile and Compile-Time Debugging

In the “before times”, programmers had to utilize various tools to check and debug their code. Today, we have more powerful IDEs (Integrated Development Environments) that can help catch some errors and reduce some errors as well. Modern IDEs can help us reduce errors through auto-completes and can often identify possible warnings and errors even before we hit compile. This is often even done in real-time, as we are working on our code. Repl.IT, for example, supports this and you have probably already seen this in your previous labs. If the IDE detects an error, it will often underline where it thinks the mistake is as shown in Figure 1.1.



```
30 // Insert your code here
31 // ----
32
33 // Check (Loading...) (look at previous Exercise)
34 if ( number[0] != 45 && (number[0] < 48 || number[0] > 57)) {
35 | printf("That is not an integer.\n");
36 return 0;
37 }
38 ;
```

Figure 1.1:

This is not always helpful however. What do you think the error in Figure 1.1 is saying? In this case, it is that “number” is not defined as I removed the variable type in front of it. Sometimes these error messages are not always helpful, but they all have good intentions and in almost all cases it means that your program will not compile or run correctly. Therefore, you need to address any error messages. Otherwise your program will not compile and will not run.

Another type of message you may see in an IDE is a warning. In Repl.IT these are usually green, in other IDEs this might be orange or a different color. Warnings are just that, your code will compile, but it may not behave as expected. In almost all cases, you should pay attention to warnings and fix them. This is why we started using the `-Wall` flag to show these warnings.

While IDEs like Repl.IT, Visual Code, Eclipse and others are useful, sometimes you are not in an environment where you can use them. This might be editing some code on the HPC for example. In addition, these tools are not perfect. They may make assumptions that are not correct. In these cases, you must rely on the compiler (in most cases for us, GCC) to output any errors or warning. Then we need to look at these errors and warnings and address them.

Here are some good rules to follow for debugging after a compilation attempt:

1. Read the last line of the error first.
2. Read up the error message until you see the first error (which is often on the last line).
3. **Only try to fix one error at a time.** All of the other errors may stem from just one! So the best strategy is to only fix one error at a time, then re-compile to see if any of the others are still there.

1.4.1 More Warnings!

When debugging, the compiler can help by adding more warnings and errors to help you spot code that might be causing undesired behavior. So far, we have not used additional compiler flags that may give us more information on what is wrong. The following flags, when added to your compile command, will make it emit a handful of additional warnings. Remember that a warning doesn't mean something is bad, just suspicious and you should take a deeper look at the code to ensure it really is doing what you intended. To compile with additional warnings, try using:

```
-std=c99 -pedantic -Wall
```

A few things we will point here. First `-Wall` gives us all warnings (or as some like to say a wall of text). This will output additional information including some warnings that might normally be suppressed. Again, warnings do not mean the code will not compile, it will. But, the compiler is warning you that either it may not be behaving as expected or does not meet certain code standards. Speaking of code standards, let's talk about those next.

Both the `-std=c99` and `-pedantic` tell the compiler to comply with certain standards. `-pedantic` tells the compiler to follow all ANSI C standards. Where `-std=c99` means to follow the C99 standard (which is the standard adopted in 1999). Other standards can of course be used. C11 is another standard adopted in 2011. Despite being old, C99 is often a standard still used today. But different standards have various pros and cons. In many cases, this is sometimes not an option you get to make. The company you work for will probably have a standard they use or they are required to use due to client requirements or even state or federal laws.

1.5 GDB Basics

This section introduces the basic and most commonly used GDB commands via small examples. A good GDB tutorial can be found here <https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>. You may find more useful information using Google.

Running example. We use the following example to explain how to use GDB in a common way. The C program listed below is one kind of implementation to compute the factorial number for a given integer number. (Recall that for an integer n , the corresponding factorial number is $n! = n * (n - 1) * \dots * 1$.) Listing 1.1 shows this code and this is also in your lab repo as `fac_1.c`:

Listing 1.1: `fac_1.c` code

```
1 || /*
```

```

2  * Compute the factorial number for a given integer
3  */
4
5  #include<stdio.h>
6
7  long factorial(int n);
8
9  int main()
10 {
11     int n = 0;
12     printf ("Please input a integer:\n");
13     scanf ("%d", n);
14     long val=factorial(n);
15     printf ("%ld\n", val);
16     return 0;
17 }
18
19 long factorial(int n)
20 {
21     long result = 1;
22     while(n--)
23     {
24         result *= n;
25     }
26     return result;
27 }

```

Unfortunately, the program above returns an incorrect result. Compile the following code, and run it with an input, e.g. 1, 2 or 3. You will receive the error information as follows:

```

[~]$ ./a.out
Please input a integer:
1
Segmentation fault (core dumped)
[~]$ ./a.out
Please input a integer:
2
Segmentation fault (core dumped)
[~]$ ./a.out
Please input a integer:
3
Segmentation fault (core dumped)

```

Examining this error, a first instinct may be to check the code line by line to locate the statements which cause the error. This is a valid approach, and sometimes the best, for simple and VERY SMALL programs. In fact, if you already have fair experience programming, you can easily find the bug in the code above. However, this is not a scalable way to fix the bugs in your program. Imagine trying to fix the bug for a program with more than one thousand lines. Are you still expecting to check the code line by line to locate the bug? This is why debugging tools such as GDB are so important for programming. Debugging is not a crutch: It is one of the most important steps in software development.

Enable GDB debugging for your program. When compiling the program with GCC, use the “-g” flag. In the example above (assume the C file name is “fac_1.c”), compiling for debugging GDB looks like:

```
gcc fac_1.c -g
```

Right off the bat, you will probably see that something is wrong. Actually, you will probably even notice something wrong by looking at the code in Repl.It as well. Both Repl.It and GCC is already letting us know what the issue is. But, for the purpose of this exercise, let’s continue in using GDB so we can get some practice.

Start Debugging. Type the command “gdb ./a.out” in your terminal, and a new prompt shows information similar to the following:

```
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
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.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-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".
(gdb)
```

Now you can start to debug your “a.out” program. Notably, if you need to provide command line arguments to the “a.out” program, use: “gdb -args ./a.out [FLAGS]”.

Set/delete the breakpoints. Before running the program “a.out” in the debugging mode, we want to set a “breakpoint” which will cause our program to pause. Without this, our program will run as normal without the opportunity to debug. Recall that the “main” function is the entry to the program, so we can set the first breakpoint as follows.

```
(gdb) break main
Breakpoint 1 at 0x4005f5: file fac_1.c, line 15.
```

Another two options to set up the breakpoints in GDB are via line number or file name:line number. Examples are as follows:

```
(gdb) break 16
Breakpoint 2 at 0x4005fc: file fac_1.c, line 16.
(gdb) break fab.c:17
Breakpoint 3 at 0x400606: file fac_1.c, line 17.
```

If there is more than one source file in your program, the last one should be the best option. To delete a breakpoint in GDB, the “delete” command is used. For example,

```
(gdb) info break
Num      Type           Disp Enb Address            What
1        breakpoint      keep y   0x0000000004005f5 in main at fab.c:7
2        breakpoint      keep y   0x0000000004005fc in main at fab.c:8
3        breakpoint      keep y   0x000000000400606 in main at fab.c:9
(gdb) delete 2
(gdb) delete 3
(gdb) info break
Num      Type           Disp Enb Address            What
1        breakpoint      keep y   0x0000000004005f5 in main at fab.c:7
```

The “info break” command is used to print out all breakpoints set in GDB, and the “delete N” command will delete a breakpoint N.

Run the program. The “run” command is used to run the program to be debugged in GDB. And it will stop when one of the breakpoints is hit. If there are no breakpoints set up, or none of the breakpoints are met during the execution of the program, the program is executed as if it’s in bash.


```
(gdb) run
Starting program: /home/****/./a.out
```

```
Breakpoint 1, main () at fac_1.c:15
15      int n = 0;
```

Run the program line by line. Once the program is executed by “run” and stopped by a breakpoint, the “next” command is used to execute the program line by line. For example,

```
(gdb) next
16      printf ("Please input a integer:\n");
(gdb) next
Please input a integer:
17      scanf ("%d", n);
```

There is another command “step” which accomplishes similar functionality with “next”. It is suggested to google the difference between “next” and “step”.

Identify the first error. By executing one more “next” in what’s shown above, the first error of the program can be caught. To do this, type in next again and it will for you to enter a number, enter “1” and then you should get the error message below.

```
(gdb) next
16      printf ("Please input a integer:\n");
(gdb) next
Please input a integer:
17      scanf ("%d", n);
(gdb) next
1
```

```
Program received signal SIGSEGV, Segmentation fault.
0x00007f5b628939a2 in _IO_vfscanf_internal (s=<optimized out>, format=<optimized out>,
    argptr=argptr@entry=0x7ffe7eb20f50, errp=errp@entry=0x0) at vfscanf.c:1898
1898      vfscanf.c: No such file or directory.
```

So we learn that it is the “scanf” function that causes the error. It seems that the function is not used correctly. After fixing this problem, we have the revised codes as shown in Listing 1.2. Go ahead and fix your code using Listing 1.2 as guide.

Listing 1.2: Fixed fac_1.c code

```
1  /*
2   * Compute the factorial number for a given integer
3   */
4
5  #include<stdio.h>
6
7  long factorial(int n);
8
9  int main()
10 {
11     int n = 0;
12     printf ("Please input a integer:\n");
13     scanf ("%d", &n);
14     long val=factorial(n);
15     printf ("%ld\n", val);
16     return 0;
17 }
18
19 long factorial(int n)
20 {
21     long result = 1;
```

```

22 |     while(n--)
23 |     {
24 |         result *= n;
25 |     }
26 |     return result;
27 | }

```

Now re-compile the code again and test the program with the input 1, 2 or 3 again. Remember to use the `-g` flag when compiling. This gives extra information for GDB to use.

```

[ ~]$ ./a.out
Please input a integer:
1
0
[~]$ ./a.out
Please input a integer:
2
0
[~]$ ./a.out
Please input a integer:
3
0

```

This is better, but we are still not out of the woods yet. While we do not have a SIG FAULT anymore, the values are incorrect. These errors are much more difficult to fix. The compiler would now be happy and you will notice that Repl.IT is also happy as well. But we clearly do not have the right values. However, GDB can still help us, so let's us GDB to find this new error.

Watch a variable in the program. In the source code, the factorial number is computed by the function “factorial”, in which the value of n relates to the final result. As a result, we may want to see how the values of n update in the function. First, let's delete the breakpoints we have. We can use `delete 1` for example to delete the first breakpoint. Also recall we can use `info break` to list out the breaks we have. Now we set the breakpoint to the beginning of the function “factorial” then ask GDB to print the value of the variable “ n ”:

```

(gdb) break factorial
Breakpoint 1 at 0x40064e: file fac_1.c, line 25.
(gdb) run
Starting program: /home/*****/./a.out
Please input a integer:
1

Breakpoint 1, factorial (n=1) at fab.c:17
21     long result = 1;
(gdb) print n
$1 = 1

```

Another way is to use the “watch” command to monitor the value changes of n . With `watch`, we can now use continue with the program and it will update n as we continue with our code. You can use the commands below to walk through this.

```

(gdb) watch n
Hardware watchpoint 2: n
(gdb) continue
Continuing.
Hardware watchpoint 2: n

Old value = 1

```

```

New value = 0
0x0000000000400672 in factorial (n=0) at fab.c:18
18     while(n--)
(gdb) continue
Continuing.
Hardware watchpoint 2: n

Old value = 0
New value = -1

```

In the above example, we first set up a watch on the variable n , whose id is 2. Then the “continue” command is used to execute the program until the value of the watched variable n changes: from 1 to 0. Continue again and the value of n changes again from 0 to -1. But recall that the computation of a factorial number, n must be greater than 0. As a result, we find that the problem is caused by line 18, where `while(n--)` allows the value of n to be 0 inside the while loop. Now we revise the code as shown in Listing 1.3.

Listing 1.3: Revised fac_1.c code

```

1  /*
2   * Compute the factorial number for a given integer
3   */
4
5  #include<stdio.h>
6
7  long factorial(int n);
8
9  int main()
10 {
11     int n = 0;
12     printf ("Please input a integer:\n");
13     scanf ("%d", &n);
14     long val=factorial(n);
15     printf ("%ld\n", val);
16     return 0;
17 }
18
19 long factorial(int n)
20 {
21     long result = 1;
22     while(n>0)
23     {
24         result *= n;
25         n--;
26     }
27     return result;
28 }

```

Re-compile and test the program again. Is the program running properly now? If so, put copies of the output into your report make sure you save your now functioning program.

Summary. In the above example, we introduced the commonly used GDB commands such as “run, break, next, step, watch” and “continue”, all of which are powerful and important commands for debugging. Only part of the usage of these commands are shown in the example, and we suggest you learn more about their usage via Google searches. In fact, you will find detailed commands and their usage by googling “GDB cheat sheet”. We omit the introduction to the “GDB cheat sheet” here but show an explicit example to illustrate some of the frequently used GDB commands instead, hoping to make you learn GDB more easily in a practical way.

1.6 Introduction to Valgrind

In this section, we introduce a very useful tool suited for debugging and profiling Linux programs, Valgrind, to facilitate C programming in terms of pointers and memory. More information can be found on its official website: <http://valgrind.org/info/>.

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behavior. Memcheck is extremely useful for learning how pointers work in C. The rest of this guide gives the bare bones information you need to start detecting memory errors in your program with Memcheck. For full documentation of Memcheck and the other tools, please read the Valgrind user manual on its website <https://valgrind.org/docs/manual/manual.html>.

1.6.1 Running example

We use the following source code as the example in this section. This is contained in the lab repository as `val_test.c` and can be seen in Listing 1.4.

Listing 1.4: `val_test.c` code

```
1  #include <stdlib.h>
2
3  void f(void) {
4      int* x = malloc(10 * sizeof(int));
5      x[10] = 0;          /* problem 1: heap block overrun */
6  } /*end f*/             /* problem 2: memory leak -- x not freed*/
7
8  int main(void) {
9      f();
10     return 0;
11 } /*end main*/
```

1.6.2 Prepare the program

Like using GDB to debug your program, the "-g" option is also needed if you want to use Valgrind to detect your program. For example, assume the filename of the source code above is "test.c," then the compiling command to enable Valgrind is as follows:

```
gcc test.c -g
```

The output program is "a.out".

1.6.3 Running your program under Memcheck

For the example above, the following command is suggested to run the program with Valgrind:

```
valgrind --leak-check=yes ./a.out
```

By default, Valgrind displays the output through the terminal. If you want to store the output in a file, e.g. "valgrind.out", use the following command:

```
valgrind --leak-check=yes ./a.out 2>valgrind.out
```

Go ahead and do that and store your output as "valgrind.out". You should see on your terminal something similar to what is shown below.

```

==183== Memcheck, a memory error detector
==183== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==183== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==183== Command: ./a.out
==183==
==183== Invalid write of size 4
==183==    at 0x108668: f (val_test.c:6)
==183==    by 0x108679: main (val_test.c:11)
==183== Address 0x522f068 is 0 bytes after a block of size 40 alloc'd
==183==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==183==    by 0x10865B: f (val_test.c:5)
==183==    by 0x108679: main (val_test.c:11)
==183==
==183==
==183== HEAP SUMMARY:
==183==    in use at exit: 40 bytes in 1 blocks
==183==    total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==183==
==183== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==183==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==183==    by 0x10865B: f (val_test.c:5)
==183==    by 0x108679: main (val_test.c:11)
==183==
==183== LEAK SUMMARY:
==183==    definitely lost: 40 bytes in 1 blocks
==183==    indirectly lost: 0 bytes in 0 blocks
==183==    possibly lost: 0 bytes in 0 blocks
==183==    still reachable: 0 bytes in 0 blocks
==183==    suppressed: 0 bytes in 0 blocks
==183==
==183== For counts of detected and suppressed errors, rerun with: -v
==183== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

In the Valgrind output, most error messages look like the following, which describes problem 1, the heap block overrun:

```

===189== Invalid write of size 4
===189==    at 0x108668: f (val_test.c:6)
===189==    by 0x108679: main (val_test.c:11)
===189== Address 0x522f068 is 0 bytes after a block of size 40 alloc'd
===189==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
===189==    by 0x10865B: f (val_test.c:5)
===189==    by 0x108679: main (val_test.c:11)

```

Here's a few notes about this error message:

- The 189 is the process ID; it's usually unimportant and it will change from machine to machine and even from each run.
- The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling you where the problem occurred. Reading them from the bottom up can help.
- The code addresses (eg. 0x108668) are usually unimportant, but occasionally crucial for tracking down weirder bugs.

- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with `malloc()` on line 6 of `val_test.c`.

It's worth fixing errors in the order they are reported, as later errors can be caused by earlier errors. Failing to do this is a common cause of difficulty with Memcheck. Memory leak messages look like this:

```
==183== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==183==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==183==    by 0x10865B: f (val_test.c:5)
==183==    by 0x108679: main (val_test.c:11)
```

The stack trace tells you where the leaked memory was allocated. Memcheck cannot tell you why the memory leaked, unfortunately. (Ignore the "`vg_replace_malloc.c`", that's an implementation detail.) There are several kinds of leaks. The two most important categories are:

- “definitely lost”: your program is leaking memory – Try to fix it!
- “probably lost”: your program is leaking memory, unless you're doing funny things with pointers (such as moving them to point to the middle of a heap block).

1.7 A Summary of Errors Valgrind can report

Despite considerable sophistication under the hood, Memcheck can only really detect two kinds of errors: use of illegal addresses, and use of undefined values. Nevertheless, this is enough to help you discover all sorts of memory-management nasties in your code. This section presents a quick summary of what error messages mean.

1.7.1 Illegal read / Illegal write errors

For example:

```
Invalid read of size 4
  at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
  by 0x40B07FF4: read_png_image__FP8QImageIO (kernel/qpngio.cpp:326)
  by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
  Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't. In this example, the program did a 4-byte read at address `0xBFFFF0E0`, somewhere within the system-supplied library `libpng.so.2.1.0.9`, which was called from somewhere else in the same library, called from line 326 of `qpngio.cpp`, and so on.

Memcheck tries to establish what the illegal address might relate to, since that's often useful. So, if it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was free'd at. Likewise, if it should turn out to be just off the end of a malloc'd block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was malloc'd.

In this example, Memcheck can't identify the address. Actually, the address is on the stack but for some reason this is not a valid stack address – it is below the stack pointer.

Note that Memcheck only tells you that your program is about to access memory at an illegal address. It can't stop the access from happening. So, if your program makes an access which normally would result in a segmentation fault, your program will still suffer the same fate – but you will get a message from Memcheck immediately prior to this. In this particular example, reading junk on the stack is non-fatal, and the program stays alive.

1.7.2 Use of uninitialized values

For example:

```
Conditional jump or move depends on uninitialised value(s)
  at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
  by 0x402E8476: _IO_printf (printf.c:36)
  by 0x8048472: main (tests/manuel1.c:8)
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
```

An uninitialized-value use error is reported when your program uses a value which hasn't been initialized – in other words, is undefined. Here, the undefined value is used somewhere inside the `printf()` machinery of the C library. This error was reported when running the following small program:

```
int main()
{
    int x;
    printf ("x = %d\n", x);
}
```

It is important to understand that your program can copy junk (uninitialised) data to its heart's content. Memcheck observes this and keeps track of the data, but does not complain. A complaint is issued only when your program attempts to make use of uninitialized data. In this example, `x` is uninitialized. Memcheck observes the value being passed to `_IO_printf` and then to `_IO_vfprintf`, but makes no comment. However, `_IO_vfprintf` has to examine the value of `x` so it can turn it into the corresponding ASCII string and it is at this point that Memcheck complains.

Sources of uninitialized data tend to be:

- Local variables in procedures which have not been initialized, as in the example above.
- The contents of malloc'd blocks, before you write something there.

1.7.3 Illegal frees

For example:

```
Invalid free()
  at 0x4004FFDF: free (vg_clientmalloc.c:577)
  by 0x80484C7: main (tests/doublefree.c:10)
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
  by 0x80483B1: (within tests/doublefree)
Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
  at 0x4004FFDF: free (vg_clientmalloc.c:577)
  by 0x80484C7: main (tests/doublefree.c:10)
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
  by 0x80483B1: (within tests/doublefree)
```

Memcheck keeps track of the blocks allocated by your program with `malloc`, so it can know exactly whether or not the argument to `free` is legitimate or not. Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address free'd. If, as here, the address is one which has previously been freed, you will be told that making duplicate frees of the same block is easy to spot.

1.7.4 Passing system call parameters with inadequate read/write permissions

Memcheck checks all parameters to system calls. If a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressable and has valid data, ie, it is readable. And if the system call needs to write to a user-supplied buffer, Memcheck checks that the buffer is addressable.

After the system call, Memcheck updates its administrative information to precisely reflect any changes in memory permissions caused by the system call. Here's an example of a system call with an invalid parameter:

```
#include <stdlib.h>
#include <unistd.h>
int main( void )
{
    char* arr = malloc(10);
    (void) write( 1 /* stdout */, arr, 10 );
    return 0;
}
```

You will get the following complaint:

```
Syscall param write(buf) contains uninitialised or unaddressable byte(s)
  at 0x4035E072: __libc_write
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
  by 0x80483B1: (within tests/badwrite)
  by <bogus frame pointer> ???
Address 0x3807E6D0 is 0 bytes inside a block of size 10 alloc'd
  at 0x4004FEE6: malloc (ut_clientmalloc.c:539)
  by 0x80484A0: main (tests/badwrite.c:6)
  by 0x402A6E5E: __libc_start_main (libc-start.c:129)
  by 0x80483B1: (within tests/badwrite)
```

The reason is because the program has tried to write uninitialized junk from the malloc'd block to the standard output.

1.7.5 Overlapping source and destination blocks

The following C library functions copy some data from one memory block to another (or something similar): `memcpy()`, `strcpy()`, `strncpy()`, `strcat()`, `strncat()`. The blocks pointed to by their `src` and `dst` pointers aren't allowed to overlap. Memcheck checks for this. For example:

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492==    at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492==    by 0x804865A: main (overlap.c:40)
==27492==    by 0x40246335: __libc_start_main (./sysdeps/generic/libc-start.c:129)
==27492==    by 0x8048470: (within /auto/homes/njn25/grind/head6/memcheck/tests/overlap)
==27492==
```

You don't want the two blocks to overlap because one of them could get partially trashed by the copying.

We have introduced the basic information for you to use Valgrind to debug your C program pointers. Please try to use this powerful tool to help you finish the current and future labs. More information on additional checks Valgrind can make, as well and help understanding its output can be found in the manual or online.

1.8 Unit Testing

Debugging our code can allow us to eliminate compile and run time errors. But, some run-time errors are difficult to catch. That is because you may run a simple test of your code and only do a few iterations. Or, you may not "push" your code to the outer limits. The reality is though, you don't know what kind of data your program may get. The user may provide incorrect data or the data file itself may have values out of range.

Our first line of defense with this is to of course stop it before it happens. Check the data coming in and if it is outside of our range then simply reject it. Ask the user to re-input the data or don't accept the incoming

data stream. This is more difficult though when it comes to numerical computations though. The data coming in can be perfectly fine but as we process the data we exceed a memory allocation or even something like a divide by zero.

We can also have errors due to things like rounding errors, which we will cover in another lecture talking about the IEEE 754 standard. It's also possible that the way that we executed our algorithm was just simply wrong. This happens even with the best programmers, so don't feel bad when it happens. In all cases though, we need to thoroughly test our code.

Unit testing is where we try to test every statement we have as much as we can. We test it with a variety of test outcomes and if we are lucky, maybe even an existing code. Perhaps for example you have code that is well tested, but is a little slow. You worked on optimizing some new code, but it needs to still provide the same results as the original. In this case we can test with that known results.

In most cases though we are going to either do hand calculations or other calculations and then compare those results with the results in our code. We can use random numbers to generate a huge range of inputs and compare the outputs.

In many ways, you have already been doing unit testing. Most of the labs that we have provided you have included scripts that have tested your code. That is how we have been grading your code, we have known working programs and values and we test your program with ours. If it matches then you pass, and if not, well, the struggle begins.

For this lab we are going to use an open source unit tester called μ unit (micro-unit). What is nice with μ unit is that it is compact and doesn't require any installing. We will cover a few features in this write-up, but I would strongly encourage you to visit their web site at <https://nemequ.github.io/munit/#>. As you can see it can do quite a bit, but we will mostly work with a few functions for this lab. This lab is not designed to teach you everything about unit testing or debugging. But hopefully there will be some handy tools and tricks and you will want to explore more on your own.

1.8.1 μ unit Basics

One of μ unit's core features is the assert function. The C language actually has an assert function that is used to add diagnostic information to an error file. μ unit adds some additional features however. With assert we are telling the program to do a compare between values. Sounds simple right? Well, for things like an integer it is, as you can see in Listing 1.5

Listing 1.5: Example of assert in numerical_test.c

```
1 ||      assert_int(factor, ==, factor_test);
```

However, things like floats and doubles get a little more tricky. We often may only be looking for a certain level of accuracy. For example, maybe we only want to go to 12 digits of accuracy. With μ unit we can specify how accurate we wish to be. This can be shown in Listing 1.6

Listing 1.6: Example of a double assert in numerical_test.c

```
1 ||      double test_series = ComputeSeriesValue(random_dbl, random_int);
2 ||      double series = ComputeSeriesValue_broke(random_dbl, random_int);
3 ||      assert_double_equal(series, test_series, 12);
```

Another feature that μ unit has is a better random number generator. The default random number generator in C is well, not great. There are however ways to improve it and μ unit uses that by using what we call a "seed value". But what is even more important than that is that μ unit will tell you what that seed value is when it runs its test. Why is that important? Because if we know the seed value, then we can recreate that random number sequence and recreate the error if needed.

We can use μ unit to create both random integers and random float or double values. See Listing 1.7 to see how we generate random integers. We can also use the assert function to check that the random numbers generated fall in the range of values that we wish to have.

Listing 1.7: Example of generating a random integer in numerical_test.c

```
1 | random_int = munit_rand_int_range(1, 20);
2 | munit_assert_int(random_int, >=, 1);
3 | munit_assert_int(random_int, <=, 20);
```

Again, we can use μ unit to also generate random floating point numbers as well. This is a little different however. In order to generate random floats, μ unit will only generate values from 0 to 1. However, we can simply take these values times a multiplier to get larger float values. You can see this in Listing 1.8

Listing 1.8: Example of generating a random float numbers in numerical_test.c

```
1 | random_dbl = munit_rand_double();
2 | random_dbl = random_dbl * 1000;
3 | munit_assert_double(random_dbl, >=, 0.0);
4 | munit_assert_double(random_dbl, <=, 1000.0);
```

For this lab, this is what we will focus on. Of course μ unit is capable of more including parameters, nested suites, a powerful CLI, and other features. One feature you can see for yourself is that it will also provide CPU times as well. As you know, CPU time is not an absolute metric for determining program performance, but it can certainly help. Hopefully though, this brief introduction will spark your interest to look more into Unit Testing or what you can do to test your code to make sure it does what you think it will do.

1.9 Exercises

1.9.1 Exercise 1 - GNUDebugger

Make sure you read through the writeup and follow the directions on using both GDB.

Exercise 1. (5 points) - After following the writeup above on using GDB make sure you have a fixed fac_1.c file.

1.9.2 Exercise 2 - Valgrind

Make sure you read through the writeup and follow the directions on using both Valgrind.

Exercise 2. (5 points) - After following the writeup above on using Valgrind make sure you have a fixed val_test.c file. The program will be tested with Valgrind to make sure there is no memory leaks or heap block over-runs.

1.9.3 Exercise 3 - Valgrind

The valgrind tool can be used to see the cache misses responsible for the dramatic slowdown with selection of the cachegrind tool like this:

```
valgrind --tool=cachegrind ./a.out
```

Try compiling and running the `examples/row_order.c` and `examples/col_order.c` programs with `cachegrind` to see how many hits and misses occur on these relatively small matrices.

Exercise 3. (10 points) - For your report this week, use Valgrind to analyze the cache misses and hits of the example code provided.

In your report you should answer the following:

1. How does the cache performance and run time of the two solutions differ
2. Can you see the impact of the transposition?
3. How would this impact larger programs?
4. Take an educated guess about the big-O complexity of this with and without the transposition
5. How much data would you have to process for this to pay off?

1.9.4 Exercise 4 - More GNUMDebugger

Exercise 4. (10 points) - The code `broke.c` in your lab repo segfaults. Use the debugging skills introduced above to correct the code.

1.9.5 Exercise 5 - Unit Testing

Exercise 5. (20 points) - There are two parts to this exercise. First, you need to modify the Makefile to compile the `numerical_test.c` that uses the provided `libnumerical.so` shared library. Then using the information provided by the `μunit`, fix the errors in the code. Your code should compile with the Makefile and all tests must pass.

For this exercise, we will provide several files. The first is a shared library called `libnumerical.so`. This library has the “correct” implementation of a few numerical functions. We will call these functions in our tests to compare what the value should be and what it is.

Remember from Lab 7 that we need to compile and run our program using this shared library. Your first step should be making changes to the Makefile to compile and run this program. Don’t forget the `-lm` flag and you should be using `-Wall` as well. If you wish to use GNUMDebugger or Valgrind to find the errors, don’t forget the `-g` flag.

The `numerical_test.c` has another feature that may be helpful. To keep the output fairly clean, by default the program will not print out some of the statements in the functions. In order to enable them, you will need to compile your code using the `-DDEBUG` flag (and yes, that is with two “Ds”. This will then enable those in code. However, when done, you should not be outputting debug statements.

Once you get it compiled, you will notice that the `μunit` will report that all tests are currently failing. Fix `numerical_test.c` until you have all tests passing. Once you have them all passing, you are done with this exercise.

I will give one big hint for this. I did not include the source file for the shared library that I provided. I did that on purpose because you could of course just look there and figure out what is wrong and that isn’t the purpose of this lab. But, I did include the header file. That should give you a big hint on what some of the issues are. Happy bug hunting!

1.9.6 Exercise 6

Exercise 6. (20 points) - Read the information below and add to your lab writeup the Algorithm asked for.

One way to help reduce the amount of debugging you do is to have a proper plan before you even write a single piece of code. This is especially important in large or complex problems that you are trying to solve. For this extra lab report, I want you to design an algorithm for the Maclaurin series evaluator. Below is what we are looking for in this part of the writeup.

Algorithm & Design (10 points): How will your program solve the problem? Use the example algorithm block given below to print your pseudo code. Think before you start to code!

Labeled Loops (5 points): What does each loop do? What changes and what is constant?

Complexity (5 points): Is the algorithm efficient? How will the time taken by the computer increase as the input increases?

An example of pseudo-code and the L^AT_EX to generate it is shown below. You can use this as a template for yours.

Algorithm 1 Algorithm for finding the factorial

```

n = ? number to factorial, passed as argument(int)
fact=1 answer and working memory (long)
if n not >= 0 then
    Invalid n, exit with error
end if

while n > 0 do
    fact = fact * n
    n = n - 1
end while
Return n

```

▷ Required variables:

▷ Loop over decreasing n from initial n to 1

```

\usepackage{algpseudocode}
\usepackage{algorithm}

\begin{algorithm}
  \caption{Algorithm for finding the factorial}

  \begin{algorithmic}
    \Statex \Comment {Required variables:}
    \State n = ? number to factorial, passed as argument(int)
    \State fact=1 answer and working memory (long)

    \If{ n not >= 0}
      \State Invalid n, exit with error
    \EndIf

    \Statex \Comment {Loop over decreasing n from initial n to 1}
    \While{ n > 0 }
      \State fact = fact * n
      \State n = n - 1
    \EndWhile
    \State Return n
  \end{algorithmic}
\end{algorithm}

```

1.10 Report

Your report should have your code outputs from the programs you ran above. It should also include copies of the code you wrote. Finally, make sure you explained how you solved the problems for each Exercise in the **Analysis** section. Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors. In addition to what we normally ask, we have an additional requirement which is upping your report score to 80 instead of the normal 50. The extra 30 points will be 10 points from Exercise 3 and 20 points for Exercise 6.

Warning: As stated in the syllabus, programs and reports that do not compile will be awarded zero points! If you have issues with your lab work, utilize your resources (lecture notes, textbooks, lab manuals, personal internet research) and contact a TA for assistance if you are still unable to resolve your issue.

1.11 Wrapping Up

Congratulations! You have finished lab 5. When you are finished with your lab, check that you have everything done below, and enjoy the rest of your day.

- ☐ Read the lab write-up
- ☐ Clone this repo into your Repl.IT, create your develop branch
- ☐ Ensure all exercises are in the correctly named file in your lab directory
- ☐ Ensure the source file compiles with the sample main function without errors
- ☐ Push your changes to the develop branch
- ☐ Complete your lab writeup
- ☐ Open a new Pull Request (and leave it open for the grader to find)