

CS515 – Lab 2

LISTEN, KID, HOW MANY TIMES DO I HAVE TO EXPLAIN IT TO YOU...BUGS DON'T BECOME PROGRAMMERS.

The purpose of this lab is to familiarize you with a debugger, in this case GNU gdb, and also with the Valgrind analysis tool. You will be working with C++ arrays and C-style strings.

Copy all files from `~cs515/public/2L` into a 2L directory you create on agate (using `ssh` & `scp`).

Part 1. Intro to debugging with gdb

A debugger is a program that runs other programs and helps the user to detect run time errors (instead of compile time errors). A debugger allows one to

- Look at the value of any variable and expression while the program is running
- Pause execution when the program reaches a desired statement (done by setting a “breakpoint”).
- Step statement-by-statement through a program. (also called “single-stepping”)
- Find out what line the program crashed on
- And much more...

Typically, a debugger is run within an IDE. However, we will be looking at the command-line debugger `gdb` in this lab. Note that `gdb` is probably not available on your laptop, so it can only be run on agate. We will cover only some very basic aspects. There is much more for you to explore on your own.

In the starter code, we have a `debug0.cpp` program that should print out 3 “hello”s and 3 “bye”s. However, if you compile and run the program, you get only 3 “hello”s and 1 “bye” printed.

Let’s use the `gdb` debugger and follow the simple steps below to find the bug.

Step 1. Compile the C++ program with debugging option `-g` option.

```
$g++ -g debug0.cpp -o debug0
```

The `-g` option tells the compiler to generate debugging information, so the created executable `debug0` is ready to be used for debugging.

Step 2. Launch the `gdb` debugger (`gdb`) as shown below

```
$gdb debug0
```

The debugger will start and you will get the `gdb` prompt below and ready for debugging.

```
(gdb)
```

Step 3. Use `start` command to start running the program with a temporary breakpoint at the beginning of the main function. (The `run` command would start the program without a breakpoint)

```
(gdb) start
Temporary breakpoint 1, main () at debug0.cpp:5
5         int i = 3;
```

Next, we can step through the program using the `step` command. You may hit return for the rest of steps after the first command.

```
(gdb) step
6         int *ptr = &i;
(gdb)
7         int **pptr = &ptr;
(gdb)
9         while((*ptr)--)
(gdb)
10        cout << "hello" << endl;
(gdb)
```

Try stepping through the whole program. You may already figure out what was the problem. But let's continue to learn some more debugging.

Step 4. We can set a breakpoint in the program where you suspect the errors are, so that the program will pause during execution at that point and we can look around.

At `gdb` prompt, set up a breakpoint in the source code using the **break** (or **b**) command:

Syntax: **break** line_number

To determine what line number to use when setting a breakpoint, you can print the program with the **list** (or **l**) command. If you don't specify a number, it will only list 10 lines at a time, starting from the beginning, and continuing from the last time you ran the **list** command. Let's specify a range...

```
(gdb) list 1, 50
1      #include <iostream>
2      using namespace std;
3
4      int main() {
5          int i = 3;
6          int *ptr = &i;
7          int **pptr = &ptr;
8
9          while((*ptr)--)
10             cout << "hello" << endl;
11             for(int j = 0; j < 3; j++);
12             cout << "bye" << endl;
13     }
14
```

Go ahead and set two breakpoints in the program, one at line 9 and one at line 11.

```
(gdb) break 9
Breakpoint 1 at 0x100000d22: file debug0.cpp, line 9.
(gdb) break 11
Breakpoint 2 at 0x100000d41: file debug0.cpp, line 11.
(gdb)
```

Step 5. Execute the C++ program in `gdb` debugger

Syntax: **run** [args]

You can now start running the program using the **run** command. Note: The `start` command does the equivalent of setting a temporary breakpoint at the beginning of the main procedure and then invoking the `run` command.

```
(gdb) run
Starting program: labs/2L-gdb_and_valgrind/start/debug0

Breakpoint 2, main () at debug0.cpp:9
9      while((*ptr)--)
(gdb)
```

The program will start running, but stops at the first break point. At this point, you have the `gdb` prompt again for debugging.

Step 6. Display variable values with the **print** (or **p**) command

Syntax: **print** {variable}

At the current breakpoint, let's examine the value of all variables.

```
(gdb) print i
$1 = 3
(gdb) print &i
$2 = (int *) 0x7fff5fbffa40
(gdb) p ptr
$3 = (int *) 0x7fff5fbffa40
(gdb) p &ptr
$4 = (int **) 0x7fff5fbffa38
(gdb) p *ptr
$5 = 3
```

```
(gdb) p *pptr
$6 = (int *) 0x7fff5fbffa40
(gdb) p **pptr
$7 = 3
```

Pay attention to the values printed and try to understand pointers and the pointer operators (* and &).

Step 7. Running the program

There are three `gdb` commands that control how the program continues to run at a break point:

- **next** (or **n**): Debugger will execute the next line as a single instruction.
- **step** (or **s**): Same as **next**, but does not treat function as a single instruction, instead goes into the function and executes it line by line.
- **continue** (or **c**): Debugger will continue executing until the next break point.

Let's try the **next** command to execute the next line in program:

```
(gdb) next
10          cout << "hello" << endl;
```

Then, use the **continue** command to go to next break point

```
(gdb) continue
Continuing.
hello

Breakpoint 2, main () at debug0.cpp:9
9          while ((*ptr)--)
```

Since we have set a breakpoint at the **while** statement, program will stop at this breakpoint for each loop iteration. We need to type 3 more **continue**'s to reach the second break point at line 11.

```
(gdb) continue
Continuing.
...

Breakpoint 3, main () at debug0.cpp:11
11         for(int j = 0; j < 3; j++);
(gdb)
```

Here, we will execute the program line by line using the **next** command.

```
(gdb) next
12          cout << "bye" << endl;
(gdb) next
bye
13      }
(gdb) next
0x0000003427c20700 in __libc_start_main () from /lib64/libc.so.6
```

Here it shows the program finished before ever stepping into the **for** loop. A careful examination of the loop statement helped us notice there is an extra semicolon at the end of the **for** statement, which made an empty **for**-loop. Bug found!

Finally, we can quit the debugger with the **quit** command.

```
(gdb) quit
```

Part 2. Debug Exercise

Use **gdb** to debug the starter code file **debug1.cpp**. The program is supposed to calculate the factorial of an input number, but it is buggy. Your task is to find the bug(s) and fix them. It is assumed the keyboard input is always a valid non-negative integer number.

Part 3: Program debugging. You will debug all memory errors found in `debug2.cpp`.

The program `debug2.cpp` contains several memory errors. We will use `gdb` to debug the program and fix all the bugs. First, we need to compile the program with debug information included. The `makefile` supplied in the starting code will make the debug-executable using (using the `-g` compiler option) if you tell it to make the “debug2” target.

```
%make debug2
```

The program is *supposed* to take one integer command line argument, create an array of the given size, populate the array with random numbers and display all array elements. Next, the program is supposed to double the array size, populate it again with new random values and display the new values. A sample run of the correct program is shown below (the current program crashes).

```
$ ./debug2 5
54 73 18 92 55
52 20 67 6 14 38 87 19 77 31
```

Use `gdb` to examine memory errors:

We will use `gdb` to figure out why the starting code crashes with memory errors. First, start the debugger, and run the program with 5 as the command-line argument

```
$ gdb ./debug2
(gdb) run 5
```

At the end of the error output, you will see a message like this:

```
...
Program received signal SIGABRT, Aborted.
0x000000392fc349c8 in raise () from /lib64/libc.so.6
```

Let's take a backtrace to figure out the problem (you can type either “where” or “backtrace”):

```
(gdb) backtrace
#0  0x00007ffff71819fb in raise () from /lib64/libc.so.6
#1  0x00007ffff7183800 in abort () from /lib64/libc.so.6
#2  0x00007ffff71c7bb1 in __libc_message () from /lib64/libc.so.6
#3  0x00007ffff71d2a59 in _int_free () from /lib64/libc.so.6
#4  0x00007ffff71d83be in free () from /lib64/libc.so.6
#5  0x0000000000400c38 in doubleArray (ptr=0x7ffffffffffe240, n=5) at debug2.cpp:55
#6  0x0000000000400afa in main (argc=2, argv=0x7ffffffffffe3a8) at debug2.cpp:34
```

The `gdb` command `backtrace` will show the function activation frames on the call stack. Here we can see the program crashed at the point where `free()` is called. The function call sequence is `main()->doubleArray()->free()`. We are only interested in the user code, not the library function `free()`, so we look into stack frame 5.

```
(gdb) frame 5
#4  0x0000000000400c98 in doubleArray (ptr=0x7ffffffffffe240, n=5) at debug2.cpp:55
55      delete ptr;
```

We can get more information of the activation from using the `info` command. `info frame` displays information about the current stack frame. `info locals` displays the list of local variables and their values for the current stack frame, and `info args` displays the list of arguments.

```
(gdb) info args
ptr = 0x7ffffffffffe240
n = 5
(gdb) info local
size = 5
tmp = 0x7ffffffffffe240
```

It looks like the program crashed at `delete ptr` at line 55 within `doubleArray()`, and `ptr` contains some address value. So why do we get a segmentation fault? (Hint: `delete` can only be used to free up dynamic memory on the heap.) Well, there are other issues in the program and it will be your responsibility to fix all the errors in this program. But first, we'll introduce you to another useful tool called `valgrind` to help detect and debug the problems.


Using the Valgrind tool

Valgrind detects memory errors during the run of a program. Make sure to compile the code with the `-g` option so `valgrind` can have access to information such as line numbers, and then run `valgrind` on the resulting executable (like, `valgrind ./a.out`). In the starter code, there are three short programs to help you understand `valgrind` output messages. Take a look at the `valgrind` output and the code file for each program to see what each error looks like in the `valgrind` output.

Error 1. Use of uninitialized memory. (`error1.cpp`)

`valgrind` output:


```
==60830== Memcheck, a memory error detector
==60830== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==60830== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==60830== Command: a.out
==60830==
==60830== Use of uninitialised value of size 8
==60830==    at 0x4005BE: main (ex1.cpp:3)
==60830==
==60830== Invalid read of size 1
==60830==    at 0x4005BE: main (ex1.cpp:3)
==60830== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==60830==
==60830== Process terminating with default action of signal 11 (SIGSEGV)
    [... other lines removed to keep this easier to read...]
==60830== For counts of detected and suppressed errors, rerun with: -v
==60830== Use --track-origins=yes to see where uninitialised values come from
==60830== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```



Error 2. Reading/writing dynamic memory after it has been freed. (`error2.cpp`)

`valgrind` output:

```
==61926== Memcheck, a memory error detector
==61926== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==61926== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==61926== Command: ./a.out
==61926==
==61926== Invalid write of size 1
==61926==    at 0x4006A3: main (ex2.cpp:4)
==61926== Address 0x4c49c80 is 0 bytes inside a block of size 100 free'd
==61926==    at 0x4A08634: operator delete[](void*) (in
/usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==61926==    by 0x40069E: main (ex2.cpp:3)
==61926==
    [... other lines removed to keep this easier to read...]
==61926== For counts of detected and suppressed errors, rerun with: -v
==61926== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```



Error 3. Reading/writing beyond the boundary of a dynamic array. (error3.cpp)

valgrind output:

```
[... Standard header lines removed to keep this short...]
==63400== Command: ./a.out
==63400==
==63400== Invalid write of size 4
==63400==    at 0x400694: main (ex3.cpp:3)
==63400==    Address 0x4c49c8c is 4 bytes after a block of size 8 alloc'd
==63400==    at 0x4A077FD: operator new[](unsigned long) (in
/usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==63400==    by 0x400687: main (ex3.cpp:2)
[... other lines removed to keep this easier to read...]
==63400== For counts of detected and suppressed errors, rerun with: -v
==63400== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Error 4. Memory leak. (error4.cpp)

valgrind output:

```
[... Earlier lines removed to keep this short...]
==64564== HEAP SUMMARY:
==64564==    in use at exit: 72,804 bytes in 2 blocks
==64564==    total heap usage: 2 allocs, 0 frees, 72,804 bytes allocated
==64564==
==64564== LEAK SUMMARY:
==64564==    definitely lost: 100 bytes in 1 blocks
==64564==    indirectly lost: 0 bytes in 0 blocks
==64564==    possibly lost: 0 bytes in 0 blocks
==64564==    still reachable: 72,704 bytes in 1 blocks
==64564==    suppressed: 0 bytes in 0 blocks
==64564== Rerun with --leak-check=full to see details of leaked memory
==64564==
==64564== For counts of detected and suppressed errors, rerun with: -v
==64564== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Here we can see more details of the leak by using the `--leak-check=full` option. E.g.

```
$ valgrind --leak-check=full ./a.out
```

Other common causes for memory errors:

- Call `delete` on non-dynamic memory.
- Call `delete` multiple times to release the same heap memory block.
- Forgot to close file streams.

If the program doesn't contain any errors, the valgrind output for the `debug2.cpp` program looks like the following:

```
$ valgrind ./debug2 5
[... Standard header lines removed to keep this short...]
==3592== Command: ./a.out 5
==3592==
66 99 18 67 0
85 7 54 44 92 97 0 54 37 86
==3592==
==3592== HEAP SUMMARY:
==3592==    in use at exit: 0 bytes in 0 blocks
==3592==    total heap usage: 2 allocs, 2 frees, 60 bytes allocated
==3592==
==3592== All heap blocks were freed -- no leaks are possible
==3592==
==3592== For counts of detected and suppressed errors, rerun with: -v
==3592== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Fix all memory errors (including memory leaks) in debug2.cpp

Some notes about segmentation faults vs. bus errors

A *bus error* occurs when you try to dereference a pointer to a memory location that is not aligned correctly. E.g. on a 32-bit machine integers must be aligned on addresses that are multiples of 4. If you dereference a pointer that is not divisible by 4, you get a bus error. The cause of a bus error is usually an attempt to dereference an uninitialized (junk) pointer that's not a multiple of 4.

The operating system and the hardware work together divide up all memory into "segments". Some segments contain your variables. You have read/write access to these segments. Some segments contain constants, or executable code. You typically have read-only access to these segments. Some segments are forbidden for a normal user to access. These might contain operating system data, or data allocated to other users. If you dereference a pointer, and it points (by accident, typically) to a segment that you don't have any access to, you get a segmentation fault. Or, if you try to write into a segment where you have read only access, it will also produce a segmentation fault.

It may be the case that the hardware checks for the alignment of a pointer BEFORE it checks for the segment type. So, a pointer outside your valid memory segments will typically produce a bus error if it is not properly aligned, rather than a segmentation fault (e.g. a pointer to int that's not a multiple of 4.)

Part 4. Programming with arrays and strings.

Implement a function in **myfun.cpp**, as described below and in the comments in the code file.

Implement the function named **myatoi** that takes a string and converts it to an integer value. The function should behave the same as the library function `atoi` in `<cstdlib>`. It should discard any leading whitespace characters until first non-whitespace character is found. Then it should take as many digit and sign characters as possible to form a valid integer number representation and then convert that to integer value.

A valid integer value consists of the following parts:

- (optional) plus or minus sign followed by
- numeric digits

The return value of the function is the integer value corresponding to the string on success. If no conversion can be performed, 0 is returned.

For example, the test cases...

```
cout << myatoi("98") << endl;
cout << myatoi(" 98123") << endl; // leading white space discarded
cout << myatoi("+98") << endl;
cout << myatoi("98a") << endl;    // remaining chars discarded
cout << myatoi("a98") << endl;    // error; non numeric
cout << myatoi("1a98") << endl;   // remaining chars discarded
cout << myatoi("+ 98") << endl;   // error: non-numeric char after +
cout << myatoi(" ") << endl;     // error: empty string
```

would produce the following output:

```
98
98123
98
98
0
1
0
0
```

Again, please write the function definition in the starter file **myfun.cpp**. Do not change the prototype of the function declaration—only provide the function implementation.

Some helpful hints (assume `c` is declared as `char c = '5';` and `b` is declared as `int b;`):

- You can compare characters like this: `c < '9'`
- You can subtract the '0' character to get the numeric equivalent: `b = '8' - '0';` *//(b is 8)*
- You can build numbers up left to right by multiply by 10, then adding:
`b = b * 10 + c - '0';` *//(b is 85)*

The test cases are given in `testdriver.cpp` for you to test the function. You should add more test cases to test your functions. Do **not** add `main()` function in `myfun.cpp`. To compile the `myfun.cpp` file (which doesn't contain a main function), you can use the provided `makefile`, or compile with the `-c` option, so that the source file is compiled as `myfun.o` instead of generating an executable.

```
% g++ -c myfun.cpp
```

Submission:

For this lab, you must submit the following files: `debug1.cpp`, `debug2.cpp`, `myfun.cpp`, the given `makefile`, `testdriver.cpp` and the finished `README`. It is ok to also submit `sampletests.cpp` if you need to, but you could get around it by removing the `sampletests` target from your submitted `makefile`.

Do not turn in executables or object code. You need to make sure `make all` doesn't produce any error for successful submission.

Submit the files in Mimir by following the link for this lab in MyCourses. Here is a list of files you are expected to submit:

```
debug1.cpp debug2.cpp myfun.cpp makefile testdriver.cpp README
```

Important:

You should also include detailed collaboration declaration information in your comments. If you worked in pairs in this lab, each of you must include the partner's name in your program comment. Both you and your partner must complete an individual submission in order to earn a grade. If you work by yourself, you must indicate in the program comments that you have worked on your own independently.

In all your source files, you must include a section of comments as shown below:

```
/** CS515 Lab X
    File: XXX.cpp
    Name: Your Name
    Section: X
    Date: XXX
    Collaboration Declaration:
    Lab Partner: <partner's name from lab session number> or <none>
    Collaboration: <anyone who may have helped that was not the primary partner>
*/
```

Below is the tentative grading scheme for this lab:

directions	100
debug file 1	20
debug file 2 test run 1	10
debug file 2 test run 2	15
debug file 2 Valgrind check	15
myatoi input file (20 tests)	40