

CMPE-380: Applied Programming

Laboratory Exercise 03

Valgrind, getopt & getopt_long

Table of Contents

Pre-Lab – 10 pts	4
Part 1 - Valgrind.....	4
Why You Should Use Valgrind	4
Tips & Good Practices.....	5
Valgrind Examples	5
Example 1: Find General Programming Errors	5
Part 2: Parameter Management via getopt & getopt_long_only	7
What is getopt?	7
How to Use getopt?	9
getopt_long_only.....	9
Interactive Exercises – 30 pts.....	11
Example 1 - Find Memory Leaks	11
Example 2 – more leaks.....	12
Example 3 – get_opt.....	13
Example 4 – get_opt_long.....	14
Summary of deliverables	14
Assignment- 60 pts.....	15
Objective	15
Part 1 – Fix memory leaks in your Darray code.	16
Part 2- Add get_opt_long features	16
Grading Criteria	17

Notes	17
Laboratory Grading Sheet	18

Valgrind

Pre-Lab – 10 pts

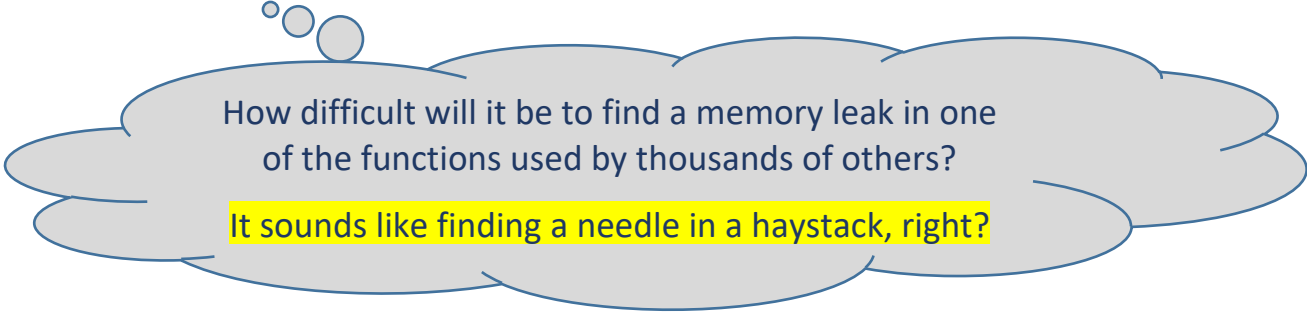
In this lab session, we will review Valgrind that is used to find memory management/threading bugs and profile programs. In addition, we will tackle the issue of command line argument parsing.

Part 1 - Valgrind

Valgrind is an open source tool set.

Using Valgrind, you can easily detect memory management and threading bugs that are not easily detectable with traditional trace and debug or code review sessions.

Consider a software system, with millions of lines of code for which thousands of developers are working. (Windows10 is around ~ 50 million lines).



How difficult will it be to find a memory leak in one of the functions used by thousands of others?

It sounds like finding a needle in a haystack, right?

Why You Should Use Valgrind

- It helps you to detect memory leaks and threading bugs easily. You can really save hours of testing/debugging time.
- It helps you find unfinalized variables which can cause unexpected results.
- Using its profiling features it can help you to speed up your program and increase its overall performance (run faster, consume less resources!).

- It is not a programming language specific tool. It can work with any language. Why?
Because, it is using binary code with debug symbols.
- Because it is open source, it can even be customized for corporate specific needs.

Tips & Good Practices

- You should use Valgrind as part of your automated tests. This way, you can make it part of your test process and ensure that your source code is ready for deployment.
- Valgrind can slow code execution by 100X or more so you will want to consider your test cases carefully.

Valgrind Examples

Example 1: Find General Programming Errors

Review and then compile the provided code **lab_a.c** with the following:

```
gcc -Wall -O0 -std=c99 -g lab_a.c -o lab_a.      Ignore the warnings.
```

Notice: The “height” variable on line 11 is not initialized or set in the code. On line 12 we are using a printf format string with one parameter but haven’t provided any parameter. On line 13 use the print the height variable. Line 23 uses the unsafe strcpy().

- 1) Run the code: **./lab_a** notice the garbage output and core dump (crash).

Now run valgrind with the following commands:

```
valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./lab_a
```

Review at the output from valgrind for the first phrase “uninitialized value of size”, notice it indicates our error is on line 13.

- 2) Fix the bug online 13 by initializing the height to 6 and re-compile.
- 3) Re-run the stand-alone code **./lab_a**. Notice that the “age” printf bug is still there.

- 4) Rerun with valgrind, notice there was no valgrind error with the “age” printf. Valgrind can detect memory access errors but not all errors. The combination of compiler warnings AND valgrind errors must be used to produce quality code.
- 5) Fix the error on line 12 by using the age variable in the print statement.
- 6) Re-run the stand-alone code `./lab_a`. Notice that the “age” and “height” printf statements are good and the “This string” message seems good but then core dumps (crashes)
- 7) Rerun with valgrind, notice there are still numerous “invalid write of size” messages pointing to line 23.
- 8) Replace the strcpy() function with the proper safe function with the correct length value.
- 9) Re-compile and re-run the stand alone code and then valgrind the code, notice no errors, no warnings and no crash.

Debugging

Using the notes from the previous lab and your corrected lab_a.c code, demonstrate your gdb knowledge. Use the “gdb logging” feature to save the results of your gdb sessions as “**`gdb.log`**” and include it in your tar file.

- 1) Start gdb with logging enabled
- 2) Set a break point at “main”, then run
- 3) Single step over using “n” until you get the strncpy line
- 4) Print the value of “text_p” before you type “n” (it should be “”)
- 5) Single step over and type “text_p” again
- 6) Set a break point at your “free()” line (b 26) and then “continue” to that point
- 7) Step over “n” the free() and then print the value of “text_p”
- 8) Step again and print the value of “text_p”
- 9) Continue “c” and your code will exit normally.

Show your corrected **lab_a.c** code and **gdb.log** file to your TA

Part 2: Parameter Management via getopt & getopt_long_only

The standard “C” command line parsing tools are torture to use due to all the error checking required! Historically, programmers had to individually process the argv[] variables passed into their programs having to audit for all the possible ways a user might enter data.

```
int main(int argc, char *argv[])
```

Where: int argc gives the parameter count

char *argv[] points to the stored parameters

Solution:

Use standard libraries: GNU getopt and GNU getopt_long_only

What is getopt?

Getopt() is an **iterative single character** command line option parsing function utilizing global variables and short hand function call options. It supports boolean arguments, optional arguments with parameters and arbitrary additional arguments.

```
int getopt (int argc, const char *argv[], const char *options)
```

argc and argv are main program parameters and options is a list of characters each representing a single character option.

Example:

“xyz:t:”: There are 4 parameters, -x, -y, -z, and -t where
-z has a required and t has an optional argument.

Getopt global variables and function parameters are shown in Figure 1 and Figure 2

Return Values:

- If an option takes a parameter value, that value is pointer to the external variable **optarg**.
- Getopt returns -1 if there are no more options to process.
- Returns ‘?’ when there is an unrecognized option and it stores that into external variable **optopt**.
- Returns ‘?’ if an option requires a value and no value is provided.

getopt() global variables

Predefined Variable	Description
int opterr	How to handle unknown or invalid parameters. If non-zero - print an error message to the standard error (default) If zero - Don't print any messages, but return the character '?' to indicate an error.
int optopt	If an invalid option is specified or a required option is missing, this variable will contain the offending character option. (note: "int" not "char")
int optind	The index of the next argv element to be processed. This variable can be used to determine where the remaining non-option arguments begin. The initial value is 1.
char *optarg	pointer to the option argument (if any)

Requires: unistd.h.

Figure 1: getopt parameters

getopt() function call

int getopt (int argc, const char *argv[], const char *options)

Parameter	Description
argc and argv	The normal "main" parameters
options	<p>A string that specifies the valid option characters for this program.</p> <p>An option character can be followed by a colon ":" to indicate that it takes a required argument.</p> <p>An option character can be followed by two colons "::" indication the argument is optional</p> <p>e.g.</p> <p>"abc:" - There are 3 parameters, -a -b & -c. -c has a required argument</p> <p>"abc::" - There are 3 parameters, -a -b & -c. -c has a optional argument</p>

Figure 2: getopt function call

How to Use getopt?

The key points are:

1. getopt is called in a loop until getopt returns -1 which indicates that there are no other parameters
2. **switch** statement is used to dispatch on the return value of getopt.
3. A second loop is used for the remaining non-option arguments.

getopt_long_only

getopt() only handles SINGLE character parameters which is not user friendly.

Ex: ./lab02a -a

The solution is to use **getopt_long_only()** which extends the getopt() to enable short and **LONG** command line parameters with the same syntax style as getopt()

Ex: ./lab02a -add

The options and parameters of getopt_long_only function are in Figure 3 and Figure 4.

getopt_long “struct option”

- Describes a single long option name.

Element	Description
const char *name	The name of the option
int has_arg	Indicate the option takes an argument. Choices are: no_argument , required_argument and optional_argument .
int *flag int val	These two fields control how to report or act on an option when it occurs. If flag is null then the val is a value which identifies this option. If flag is non-null then it must be the address of an int variable which is the flag for this option. The value in val is the value to store in the flag to indicate that the option was seen.

Figure 3: getopt_long_only struct option

getopt_long_only() function call

```
int getopt_long_only (int argc, const char *argv[], const char  
*shortopts, const struct option *longopts, int *indexptr)
```

Parameter	Description
argc and argv	The normal "main" parameters
shortopts	Same as getopt "options". E.g. "abc:"
struct option *longopts	Pointer to an array of "struct option" objects with the last element identified with the value "0,0,0,0"
int *indexptr	Pointer to a variable containing the longopts[] index. getopt_long_only stores the index of the longopts definition in indexptr. You can get the name of the option with longopts[*indexptr].name. Note: May not be valid if function returns non-zero
Return value	The shortopts character found OR 0 if there is ONLY a longopts parameter

Figure 4: getopt_long_only parameters

Interactive Exercises – 30 pts

Example 1 - Find Memory Leaks

Review and then compile the provided code **lab_b.c** with the following:

```
gcc -Wall -O0 -std=c99 -g -Wall lab_b.c -o lab_b
```

- 1) Run the program **./lab_b** are there any issues? Clearly document your answers in **exercise.txt**.

Now run Valgrind memory check with:

```
valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./lab_b
```

- 2) Are there any issues identified now? How many blocks and bytes were allocated? How much memory was lost? Clearly document your answers in **exercise.txt**.

Fix the code without changing the basic functionality.

- 3) How many blocks and bytes were allocated? How much memory was lost? Clearly document your answers in **exercise.txt**.

Save your fixed code and fixed valgrind output for your TA. Name the valgrind output **lab_b.txt**

```
valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./lab_b > lab_b.txt
```

2>&1

Example 2 – more leaks

Review and then compile the provided code **lab_c.c** with the following:

```
gcc -Wall -O0 -std=c99 -g -Wall lab_c.c -o lab_c
```

- 1) Run the program **./lab_c** are there any issues? Clearly document your answers in **exercise.txt**.

Now run Valgrind memory check with:

```
valgrind --tool=memcheck --leak-check=yes --track-origins=yes ./lab_c
```

- 2) Are there any issues identified now? How many blocks and bytes were allocated? How much memory was lost? Clearly document your answers in **exercise.txt**.

Fix the code without changing the basic functionality.

- 3) How many blocks and bytes were allocated? How much memory was lost? Clearly document your answers in **exercise.txt**.

Save your fixed code and fixed valgrind output for your TA. Name the valgrind output **lab_c.txt**

Example 3 – get_opt

The provided code **lab_d.c** implements a get_opt parsing for “-a” and “-c <string>”. The code prints out the raw arguments provided by the operating systems and then parses the arguments.

Review and then compile the provided code **lab_d.c** with the following:

```
gcc -Wall -O0 -std=c99 -g -Wall lab_d.c -o lab_d
```

Run the program **./lab_d** with the following tests.

- 1) **./lab_d**
- 2) **./lab_d -a**
- 3) **./lab_d -c**
- 4) **./lab_d -c WORD**
- 5) **./lab_d -cWORD** // Notice NO SPACE
- 6) **./lab_d -a -c WORD1 -c WORD2**
- 7) **./lab_d something**

Update the **lab_d.c** code to support an additional flag called “-b” and an additional required parameter flag “-d <string>”. You must print the values of all flags just like the originals.

Rerun the previous test and the following tests and write the output to **lab_d.txt**.

- 1) **./lab_d -d WORD**
- 2) **./lab_d -b**
- 3) **./lab_d -b -c WORD1 -d WORD2**
- 4) **./lab_d -b -d WORD1 -b -c WORD2**
- 5) **./lab_d -c WORD1 -c WORD2**

Example 4 – get_opt_long

The provided code **lab_e.c** implements a get_opt_long parsing for the following short options: "abc:d:f:" and the following long options "verbose", "brief", "add" (no_argument) and "del", "create" (required_argument). Review and then compile the provided code **lab_e.c** with the following:

```
gcc -Wall -O0 -std=c99 -g -Wall lab_e.c -o lab_e
```

Update the lab_e.c code to support additional options: "verb", "append", (no_argument) and "delete", "file" (required_argument)

Rerun the following tests and write the output to **lab_e.txt**.

- 1) ./lab_e
- 2) /lab_e -ver // This does NOT imply adding a "-v" short option
- 3) /lab_e -verb
- 4) /lab_e -verbose
- 5) /lab_e -add
- 6) /lab_e -append
- 7) /lab_e -delete WORD1
- 8) /lab_e -del WORD2
- 9) /lab_e -file WORD3

Summary of deliverables

lab_b.txt, lab_c.txt, lab_d.txt, lab_e.txt, exercise.txt

lab_a.c, lab_b.c, lab_.c, lab_d.c, lab_e.c

Assignment- 60 pts

Objective

To resolve any memory leaks in your previous dynamic array implementation and then add `get_opt_long` command line parsing features. Upload the file **lab03.tar** (available in MyCourses) to the **lab03** working directory. Copy your: `DynamicArrays.h`, `ClassErrors.h`, `DynamicArrays.c` and `simpleTest.c` from your previous assignment.

The **hw3_files.tar** tarball contains:

- build1** – a script to build your original dynamic array code
- mem1** – a script to run Valgrind on your original dynamic array code.
- build2** – a script to build your new `get_opt_long` parsing code.
- test2** – a script to test your new `get_opt_long` parsing code
- mem2** – a script to run Valgrind on new `get_opt_long` parsing code. It should take about 5minutes to execute on proper code.
- engWords.txt**, **shortWords.txt** and **veryShortWords.txt** – testing data
- a testing helper application called: **TestDarray.c**
- solution.txt** contains the expected results to your new `get_opt_long` parsing code.

Note: Be sure to modify your `realloc` code in `pushToDarray()` to grow by **GROWTH_AMOUNT** or your **mem2** test will run **LONG**.

Your `analysis.txt` file must be formatted so that it can be viewed on an 78 character wide display. (e.g. Format the final file by adding a hard CR/LF at column 78 or before).

Part 1 – Fix memory leaks in your Darray code.

Use the **build1** script and **mem1** scripts to rebuild and verify your existing Darray code. Fix any memory leaks you have. Valgrind should not report any memory access errors or leaking memory.

Part 2- Add get_opt_long features

A skeleton testing program called **TestDarray.c** is provided. Add get_opt_long features to the program to implement the following features:

- | | | | |
|-----------|-----------------|----------------|---|
| -i file | or -in file | or -input file | required, the file to process |
| -w number | or -word number | | required, the number of words to print |
| -h | or -help | | print user help and take no other action. |
- e.g. TestDarray -i[n[put]] filename [-w[ord] num] [-h[elp]]

Use the provided scripts **./build2**, **./test2** and **./mem2** to build and run your code.

Note, you may want to **edit the scripts** and comment out the more challenging tests until after you have debugged everything. The final tests MUST use engWords.txt

Grading Criteria

1. (15 points) Correct analysis for Valgrind
2. (30 points) Correct program behavior (error codes, error messages, checking for all important return codes, etc)
3. (15 points) No memory leaks or access errors (you returned all memory back to the heap, no access errors)

Notes

1. Start work on this homework early, especially if you do not have experience using pointers or programming in C.
2. It should be clear that the implementation must be your own (not copied from some remote source in the Internet)

Student Name: _____

Laboratory Grading Sheet

Lab03 - Valgrind, get_opt & get_op_long

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab – corrected lab_a.c and gdb.log file	10		
Interactive Exercise : Example 1	7		
Interactive Exercises: Example 2	7		
Interactive Exercises: Example 3	8		
Interactive Exercise : Example 4	8		
Total	40		

You must turn this signed sheet in at the end of lab to receive credit!