Guide to valgrind

Written by Nate Hardison & Julie Zelenski

Some of the most difficult C bugs come from mismanagement of memory: allocating the wrong size, using an uninitialized pointer, accessing memory after it was freed, overrunning a buffer, and so on. The symptom often doesn't give you much to go on-- an intermittent segmentation fault, a mysteriously stomped-on value at a place and time far removed from the original error. Tracing the observed problem back to underlying root cause can be challenging. Valgrind is here to help! You run a program under valgrind to enable extensive checking of memory allocations and accesses and it provides a report with detailed information about the context and circumstances of each error.

Memory errors != memory leaks

Memory issues come in two flavors: memory *errors* and memory *leaks*. Valgrind reports on both with equal vigor, but please don't you conflate errors with leaks or assume them equally important. When a program dynamically allocates memory and forgets to later free it, it creates a leak. A memory leak generally won't cause a program to misbehave, crash, or give wrong answers. A memory leak is not an urgent situation, just a little detail to eventually get around to resolving. A memory error, on the other hand, is a red alert. Reading uninitialized memory, writing past the end of a piece of memory, accessing freed memory, and other memory errors are serious crimes with potentially catastrophic consequences. Memory errors should never be treated casually or ignored. Although this guide describes about how to use Valgrind to find both, keep in mind that errors are by far the primary concern.

Running a program under Valgrind

Like the debugger, Valgrind runs on your executable, so be sure you have compiled an up-to-date copy of your program. Running under valgrind can be as simple as just prefixing the program command like this:

```
valgrind ./myprogram julie chris
```

which starts up valgrind and runs the program inside of it. In this case, "./myprogram" is the path to the program and it is given two arguments julie and chris. Valgrind will execute your program under its watchful eye (expect it to run much more slowly because of the extra checks) and when it finishes, Valgrind will prints a summary of its memory usage. If all goes well, it'll look something like this:

```
==4649== ERROR SUMMARY: 0 errors from 0 contexts
==4649== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4649== malloc/free: 10 allocs, 10 frees, 2640 bytes allocated.
==4649== For counts of detected errors, rerun with: -v
==4649== All heap blocks were freed -- no leaks are possible.
```

This is what you're shooting for: no errors and no leaks. Another useful metric is the number of allocations and total bytes allocated. If these numbers are the same ballpark as our sample (you can run solution under valgrind to get a baseline), you'll know that your memory efficiency is right on target.

Finding memory errors

Memory errors can be truly evil. The more overt ones cause spectacular crashes, but even then it can be hard to pinpoint how and why the crash came about. More insidiously, a program with a memory error can still seem to work correctly because you manage to get "lucky" much of the time. After several "successful" outcomes, you might wishfully write off what appears to be a spurious catastrophic outcome as a figment of your imagination, but depending on luck to get the right answer is not a good strategy. Running under valgrind can help you track down the cause of visible memory errors as well as find lurking errors you don't even yet know about.

Each time valgrind detects an error, it prints information about what it observed. Each item is fairly terse-- the kind of error, the source line of the offending instruction, and a little info about the memory involved, but often it is enough information to direct your attention to the right place. Here is an example of valgrind running on a buggy program:

```
==4651== Invalid write of size 1
==4651==    at 0x80486A4: main (myprogram.c:58)
==4651==  Address 0x4449054 is not stack'd, malloc'd or (recently) free'd
==4651==
==4651== ERROR SUMMARY: 1 errors from 1 contexts
==4651== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4651== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==4651== For counts of detected errors, rerun with: -v
==4651== All heap blocks were freed -- no leaks are possible.
```

The ERROR SUMMARY says there is one error, an invalid write of size 1 (byte, that is). The bad write operation was observed at line 58 in myprogram.c. Let's look at the code:

```
56     ...
57     char *copy = malloc(strlen(buffer));
58     strcpy(copy, buffer);
59     ...
```

Looks like an instance of the classic strlen + 1 bug. The code doesn't malloc enough space for the '\0' character, so when strcpy went to write it at frag[strlen(buffer)], it accessed memory beyond the end of the malloc'ed piece. Despite the code being clearly wrong, it often may appear to "work" because malloc commonly rounds up the requested size to the nearest multiple of 4 or 8 and that extra space may cover the shortfall. "Getting away with it" can lead you to a false sense of security about the code being correct. The next run might get a strange crash that you might write off as a fluke. But vigilantly using valgrind can inform you of the error so you can find and fix it, rather than wait for an observed symptom that may be hard to reproduce.

There are different kinds of memory errors that you may see in the valgrind reports. The most common are:

- `Invalid read/write of size X` The program was observed to read/write X bytes of memory that was invalid. Common causes include accessing beyond the end of a heap block, accessing memory that has been freed, or accessing into an unallocated region such as from use of a uninitialized pointer.
- `Use of uninitialised value` or `Conditional jump or move depends on uninitialised value(s)` The program read the value of a memory location that was not previously written to, i.e. uses random junk. The second more specifically indicates the read occurred in the test expression in an if/for/while. Make sure to initialize all of your variables! Remember that just declaring a variable doesn't put anything in its contents--if you want an int to be 0 or a pointer to be NULL, you must explicitly state so. Note that Valgrind will silently allow a program to propagate an uninitialized value along from variable to variable; the complaint will only come when(if) it eventually uses the value which may be far removed from the root of the error. When tracking down an uninitialized value, run Valgrind with the additional flag `--track-origins=yes` and it will report the entire history of the value back to the origin which can be very helpful.
- `Source and destination overlap in memcpy()` The program attempted to copy data from one location to another and the range to be read intersects with the range to be written. Transferring data between overlapping regions using `memcpy` can garble the result; `memmove` is the correct function to use in such a situation.
- `Invalid free()` The program attempted to free a non-heap address or free the same block more than once.

Memory errors in your submission can cause all sorts of varied problems (wrong output, crashes, hangs) and will be subject to significant grading deductions. Be sure to swiftly resolve any memory errors by running Valgrind early and often!

Finding memory leaks

You can ask Valgrind to report on memory leaks in addition to errors. When you allocate heap memory, but don't free it, that is called a leak. For a small, short-lived program that runs and immediately exits, leaks are quite harmless. For a project of larger size and/or longevity, a repeated small leak can eventually add up. For CS107, we will expect you to deallocate all memory at the end of program execution. Even though our programs are small enough to not be incapacitated by leaks, we want you to build the skills to properly handle the manual allocation and deallocation required by C. We reserve a few grading points to be earned for plugging all your leaks, but just a few, as leaks are pretty small potatoes in the grand scheme of things.

To check for leaks, you need to include the options `leak-check=full` and `--show-leak-kinds=all` in the valgrind command, as shown below. (You might want to define a shorthand alias (/class/cs107/guide/unix.html#faq) for such a long-winded command.)

```
valgrind --leak-check=full --show-leak-kinds=all program argument(s)
```

Here's the report from a program with leaks:

```
==5942== ERROR SUMMARY: 0 errors from 0 contexts
==5942== malloc/free: in use at exit: 12 bytes in 1 blocks.
==5942== malloc/free: 250 allocs, 249 frees, 12476 bytes allocated.
==5942== For counts of detected errors, rerun with: -v
==5942== searching for pointers to 1 not-freed blocks.
==5942== checked 51,452 bytes.
==5942==
==5942==
==5942== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5942==    at 0x43BC3C0: malloc (vg_replace_malloc.c:149)
==5942==    by 0x804863D: main (myprogram.c:51)
==5942==
==5942== LEAK SUMMARY:
==5942==    definitely lost: 12 bytes in 1 blocks.
==5942==      possibly lost: 0 bytes in 0 blocks.
==5942==    still reachable: 0 bytes in 0 blocks.
==5942==         suppressed: 0 bytes in 0 blocks.
```

It's pretty easy to tell when there's a leak: the alloc/free counts don't match up and you get a LEAK SUMMARY section at the end. Valgrind also gives a little data about each leak -- how many bytes, how many times it happened, and where in the code the original allocation was made. Multiple leaks attributed to the same cause are coalesced into one entry that summarize the total number of bytes across multiple blocks. The report above shows one leak of size 12 bytes in 1 block. That block was allocated by malloc in myprogram.c, line 51, and never freed.

Valgrind categorizes leaks using these terms:

- *definitely lost*: heap-allocated memory that was never freed to which the program no longer has a pointer. Valgrind knows that you once had the pointer, but have since lost track of it. This memory is definitely orphaned.
- *indirectly lost*: heap-allocated memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, the subsequent nodes would be indirectly lost.
- *possibly lost*: heap-allocated memory that was never freed to which valgrind cannot be sure whether there is a pointer or not.
- *still reachable*: heap-allocated memory that was never freed to which the program still has a pointer at exit (typically this means a global variable points to it).

These categorizations indicate whether the program has retained a pointer to the memory at exit. If the pointer is available, it will be somewhat easier to add the necessary free call, but it doesn't change that the fact that all are leaks-- that is, memory that was heap-allocated and never freed. In grading, we will expect that a program that successfully completes will deallocate all heap memory, leaving no leaks of any kind. (We don't expect cleanup of memory leaks if the execution ends early due to a fatal error).

Given that leaks are generally benign and the bugs from incorrect deallocation can be deadly, we strongly recommend that you let your program leak like a sieve while you are working on getting the functionality correct. Only consider looking for and plugging leaks after all of the program's functionality is solid.

Using gdb and Valgrind together

One handy Valgrind trick is the ability to drop into the debugger as soon as it encounters a memory error (not a leak). You do this by specifying the db-attach option when starting valgrind:

```
valgrind --db-attach=yes program argument(s)
```

As soon as it encounters a memory error, Valgrind will ask you if you would like to start up the debugger:

```
==6459== ---- Attach to debugger? --- [Return/N/n/Y/y/C/c] ----
```

Typing either Y or y will throw you into gdb and you get plopped into the running program right at the spot of the memory error. (Sometimes gdb will ask if you would like to kill the running program to which you should answer "no"). You can then use your mad gdb skills to examine the context at the time of the error to better understand what's going on. When you exit the debugger, you will return to valgrind. The db-attach feature is nifty, but realize it's messing with an executing program that has memory errors. The perilous context it is operating in means it sometimes bails on you when the going gets rough, so be prepared to weather a little flakiness when using this feature.

Common questions about Valgrind

My program runs very slowly under Valgrind. Should I be concerned?

No. Valgrind is running your program in a simulated context and monitoring the runtime activity. Depending on how memory-intensive the program is, this extra checking can slow down a program by 2-5x. This is completely expected.

Valgrind says I leaked memory because of a call to malloc() in main(), but I don't call malloc() in main()! What's going on?

This report can also be a result of calling a library function in main() that itself calls malloc() internally. Common examples include fopen() and strdup(). Make sure to fclose any fopened `FILE*` s, and free any strduped `char*` s.

My program runs fine and produces correct output but the Valgrind report shows memory errors. Can I ignore these?

No. An error may not have an observable runtime consequence in some situations but that doesn't mean it doesn't exist. The error is a ticking time bomb that can go off at anytime. Ignoring it and counting on your code continuing to "get lucky" is a risky practice. Make sure your code always runs Valgrind-clean!

My valgrind report includes an ominous-looking entry something like this: "Warning: set address range perms: large range". What is this and do I need to worry about it?

Valgrind prints this warning when an unusually large memory region is allocated, on suspicion that the size may be so large due to an error. If the intention of your code was to allocate a large block, then all is well.

My valgrind report suggests to rerun with `-v` for "counts of suppressed errors". What are these? Should I worry about them?

Pay no attention. Some library code does unusual things which can trigger reports from Valgrind even when operating correctly. Those errors/leaks are suppressed as they are known to be spurious. The `-v` flag causes valgrind to provide verbose commentary about its internal handling of these events. You can safely ignore all suppressed events; no need for you to wade through the verbose chatter.

When I run valgrind with no extra arguments, the ERROR SUMMARY says 0 errors, but the exact same run adding the --leak-check option then reports N errors from N contexts. Do I have errors or don't I?

With leak-check enabled, each distinct leak found by valgrind is included in the count of errors. Without leak-check enabled (the default), it doesn't enumerate/count leaks, so only actual memory errors are reported in the summary count.

I get a "Permission denied" message when I attempt to run a particular executable under valgrind even though I can run the program normally. How do I fix?

Valgrind refuses if you don't have execute permission according to the file mode of the executable. The file mode is mostly irrelevant on our myth systems (the directory-based AFS permissions take precedence), but Valgrind is paying attention anyway. The command `ls -l executable_file` shows the file mode bits, the x's indicate execute permission for owner/group/other. Use the command `chmod a+x executable_file` to enable execute permission for all users.