Valgrind Memcheck

for more tricks - https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks

Written by Nate Hardison, Julie Zelenski and Chris Gregg, with modifications by Nick Troccoli

Click here (https://youtu.be/A5Rc4AwdaOA) for a walkthrough video.

Valgrind Memcheck is a tool that detects memory leaks and memory errors. 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. These types of errors are tricky, as they can provide little debugging information, and tracing the observed problem back to underlying root cause can be challenging. Valgrind is here to help!

Memory Errors Vs. Memory Leaks

error is more serious

Valgrind reports two types of issues: memory *errors* and memory *leaks*. 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, and is not an urgent situation. 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 can have significant 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, and memory leaks can generally be resolved later.

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. Run it like this, for example, if your program is named `memoryLeak`:

```
$ valgrind ./memoryLeak
```

Valgrind will then start up and run the specified program inside of it to examine it. If you need to pass command-line arguments, you can do that as well:

```
$ valgrind ./memoryLeak red blue
```

When it finishes, Valgrind will print 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 allocate 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, but 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.

For an example, let's take this `memoryLeak.c` program:

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>

const int ARR_SIZE = 1000;

int main() {
    // create an array of ARR_SIZE ints
    int *intArray = malloc(sizeof(int) * ARR_SIZE);

    // populate them
    for (int i=0; i < ARR_SIZE; i++) {
        intArray[i] = i;
    }

    // print one out
    // first, seed the random number generator
    srand(time(NULL));
    int randNum = rand() % ARR_SIZE;

    printf("intArray[%d]: %d\n", randNum, intArray[randNum]);

    // end without freeing!
    return 0;
}
```

If we compile and run this, here is the output we get:

```
$ gcc -g -Og -std=gnu99 memoryLeak.c -o memoryLeak
$ ./memoryLeak
intArray[645]: 645
```

This seems to be ok, but let's run it under Valgrind, just to be sure:

```
$ valgrind ./memoryLeak
==32251== Memcheck, a memory error detector
==32251== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==32251== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==32251== Command: ./memoryLeak
==32251==
intArray[782]: 782
==32251==
==32251== HEAP SUMMARY:
==32251==     in use at exit: 4,000 bytes in 1 blocks
==32251==   total heap usage: 2 allocs, 1 frees, 5,024 bytes allocated
==32251==
==32251== LEAK SUMMARY:
==32251==    definitely lost: 4,000 bytes in 1 blocks
==32251==    indirectly lost: 0 bytes in 0 blocks
==32251==      possibly lost: 0 bytes in 0 blocks
==32251==    still reachable: 0 bytes in 0 blocks
==32251==         suppressed: 0 bytes in 0 blocks
==32251== Rerun with --leak-check=full to see details of leaked memory
==32251==
==32251== For counts of detected and suppressed errors, rerun with: -v
==32251== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

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. (note that it says 2 allocs even though we only call `malloc` once. Why? Because `srand/time` allocate memory in their implementations, but they free it as well!). 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. Here, the program `memoryLeak.c` requests memory from the heap and then ends without freeing the memory. This is a memory leak, and `valgrind` correctly finds the leak: "definitely lost: 4,000 bytes in 1 blocks"

Valgrind categorizes leaks using these terms:

see categorized leaks in next 2 pages

- *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.

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.

Given that leaks generally do not cause bugs, and incorrect deallocation can, we recommend that you do not worry about freeing memory until your program is finished. Then, you can go back and deallocate your memory as appropriate, ensuring correctness at each step.

If you want more information, you need to include the options `--leak-check=full` and `--show-leak-kinds=all`

```
$ valgrind --leak-check=full --show-leak-kinds=all ./memoryLeak
==4599== Memcheck, a memory error detector
==4599== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4599== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4599== Command: ./memoryLeak
==4599==
intArray[2]: 2
==4599==
==4599== HEAP SUMMARY:
==4599==     in use at exit: 4,000 bytes in 1 blocks
==4599==   total heap usage: 2 allocs, 1 frees, 5,024 bytes allocated
==4599==
==4599== 4,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4599==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4599==    by 0x4005A3: main (memoryLeak.c:9)
==4599==
==4599== LEAK SUMMARY:
==4599==    definitely lost: 4,000 bytes in 1 blocks
==4599==    indirectly lost: 0 bytes in 0 blocks
==4599==      possibly lost: 0 bytes in 0 blocks
==4599==    still reachable: 0 bytes in 0 blocks
==4599==         suppressed: 0 bytes in 0 blocks
==4599==
==4599== For counts of detected and suppressed errors, rerun with: -v
==4599== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Because we compiled with the `-g` flag, `valgrind` is able to tell us exactly where in our program the leak was created:

```
==4599==    by 0x4005A3: main (memoryLeak.c:9)
```

In other words, on line 9 in `memoryLeak.c`, we allocated the memory and it was never freed. This can be super-helpful when debugging your programs!

For more information about Valgrind, also check out the full **valgrind manual (http://valgrind.org/docs/manual/manual.html)**.

Frequently Asked Questions

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.

---

1 Memory Errors and Valgrind

Note that all code used in this tutorial relies on undefined behaviour and may or may not work the same way on your machine.

1.1 Valgrind Basics

- Memory errors (e.g. memory leaks, uninitialized values) are a common problem in C/C++.
- Although your operating system will free any memory leaks when the program ends, many programs run for a long time (eg. browsers, operating systems, video games, word processors) and may slow down or crash due to leaks and other errors.
- It can be really hard to tell where and why these errors are actually happening.
- valgrind is a program designed to help us with this task.
- To run valgrind, we simply need to put valgrind before our normal bash command:

  ```
  valgrind ./exec
  ```

- Your program will run much slower, and extra information printed on stderr will tell you about potential problems.
- If you compile with the -g option, you'll get line number information for debugging purposes.

1.2 Memory Leaks

- You'll notice that some of valgrind's output talks about heap usage. This lets you know if your program is leaking any memory.
- Don't worry if it says that there are 72,206 bytes of "still reachable" memory: this is normal and is the fault of the C++ standard library.
- Memory leaks are broken down into four categories:
  - "Definitely lost" blocks are blocks that you forgot to delete and you no longer can. For example, if you forget to delete some heap memory which is only pointed to by a local variable in a function, then the memory is definitely lost once the function ends.
  - "Indirectly lost" blocks are blocks that weren't deleted because you didn't delete a "definitely lost" block. For example, if you don't delete a linked list, the first node is definitely lost and the rest are indirectly lost. These usually go away when you fix any definite leaks.
  - "Possibly lost" blocks are usually blocks that you forgot to delete and are definitely lost. Occasionally, they might actually be blocks which are still reachable. You will usually not encounter these unless you've made pointer arithmetic errors in unusual places.
  - "Still reachable" blocks are blocks that you forgot to delete but, up until the end of the program, still could've deleted them if you remembered. For example, if a global variable contains a pointer to some heap memory, that memory is still reachable.
- If you pass the --leak-check=full option, more detailed information on memory leaks will be printed.
- Note that other arguments need to be passed to valgrind *b*efore the program is!

- Note: new int[0] returns allocated memory and will cause a memory leak if not deleted! Furthermore, dereferencing this pointer results in undefined behaviour.

1.3 Segmentation Faults

- Most memory errors which aren't memory leaks end up resulting in a segmentation fault.
- A segmentation fault is raised when the operating system realizes that your program is trying to access memory that it shouldn't have access to.
- This is very coarse, so unfortunately if you are accessing memory that you do have access to–but that you don't want to access–a segmentation fault might not be raised.
- When a segmentation fault is raised, valgrind will print a *s*tack trace: a listing of all the functions that were called up until the fault, and which lines they were on (if you passed the -g option when compiling).
- The most familiar example of a segfault happens when you attempt to dereference a null pointer:

```
int *i = NULL;
*i = 3;
```

1.4 Silent Memory Errors

- Unfortunately, some memory errors can happen without any effect on the observable outcome of our program.
- This doesn't mean they aren't important. Why?
  - They might sometimes, but not always, alter observable behaviour.
  - After the program is changed they may begin to alter its behaviour.
  - They may result in code whose behaviour is different when recompiled, especially on a different system or compiler.
- Valgrind catches these errors for us, and may sometimes end our program even when it would normally keep running.
- Types of common memory errors which usually won't be reported:
  - Double free
  - Uninitialized values
  - Off-by-one pointer errors
  - Dangling pointers