Back to Resources

Valgrind

by Tamara Nelson-Fromm

Valgrind

Valgrind is an analysis tool that can detect cases of poor memory management. Using Valgrind, you can more quickly find the causes of segmentation faults, memory leaks, and misused memory.

Using Valgrind

Once you've compiled your program, you can run it with Valgrind using the following command:

```
valgrind ./[executable name]
```

This will run your program, and also output a report that looks something like this:

```
==13715== Command: ./mp_traversals
==13715==
==13715== Conditional jump or move depends on uninitialised value(s)
==13715==    at 0x40080F: main (main.cpp:20)
==13715==
==13715== HEAP SUMMARY:
==13715==     in use at exit: 20,572,080 bytes in 2 blocks
==13715==   total heap usage: 65,665 allocs, 65,663 frees, 2,132,319,481 bytes allocated
==13715==
==13715== LEAK SUMMARY:
==13715==    definitely lost: 48 bytes in 1 blocks
==13715==    indirectly lost: 0 bytes in 0 blocks
==13715==      possibly lost: 20,572,032 bytes in 1 blocks
==13715==    still reachable: 0 bytes in 0 blocks
==13715==         suppressed: 0 bytes in 0 blocks
==13715== Rerun with --leak-check=full to see details of leaked memory
==13715==
==13715== For counts of detected and suppressed errors, rerun with: -v
==13715== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind's report will include any errors, and then detail the amounts of memory created (allocated), deleted (freed), and the number of calls (allocs) that create all of that memory. As bytes of memory are still in use at exit, this program has memory leaks.

Note: in order to see line numbers and function details in Valgrind, you must compile your executable using the debug flag. (`-g`) You won't need to worry about this in CS 225 when using the included makefiles, as they already include this flag.

Memory Leaks

If Valgrind reports that you have memory leaks, you can run:

```
valgrind --leak-check=full ./[executable name]
```

Which will output a report that also details the source of that leaked memory:

```
==27013== 48 bytes in 1 blocks are definitely lost in loss record 1 of 2
==27013==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==27013==    by 0x43CF79: operator new(unsigned long) (in /home/tln2/cs225git/mp4/mp4)
==27013==    by 0x409E62: FloodFilledImage::animate(unsigned int) const (FloodFilledImage.cpp:56)
==27013==    by 0x407B5A: main (main.cpp:37)
==27013==
==27013== 20,572,032 bytes in 1 blocks are possibly lost in loss record 2 of 2
==27013==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==27013==    by 0x43CF79: operator new(unsigned long) (in /home/tln2/cs225git/mp4/mp4)
==27013==    by 0x4226F9: cs225::PNG::_copy(cs225::PNG const&) (PNG.cpp:26)
==27013==    by 0x4229BF: cs225::PNG::PNG(cs225::PNG const&) (PNG.cpp:46)
==27013==    by 0x409E8F: FloodFilledImage::animate(unsigned int) const (FloodFilledImage.cpp:56)
==27013==    by 0x407B5A: main (main.cpp:37)
==27013==
==27013== LEAK SUMMARY:
==27013==    definitely lost: 48 bytes in 1 blocks
==27013==    indirectly lost: 0 bytes in 0 blocks
==27013==      possibly lost: 20,572,032 bytes in 1 blocks
==27013==    still reachable: 0 bytes in 0 blocks
==27013==         suppressed: 0 bytes in 0 blocks
==27013==
==27013== For counts of detected and suppressed errors, rerun with: -v
==27013== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Valgrind will report cases of memory being directly and indirectly lost. Memory that is directly lost has to do with variables that are allocated to the heap but never deleted. Indirectly lost memory is memory that was not directly created by a single statement, such as the indices in an array, but is still is lost when direct memory is. As indirectly lost memory has a connection to directly lost memory, fixing direct memory leaks will often (but not always) also fix the indirect leaks.

Looking at the backtrace on both of these leaks, it can be seen that they both come from a variable created at line 56 in FloodFilledImage.cpp. With this kind of output, it is likely if that variable is then freed up (with the correct kind of delete statement) at the end of its use, these leaks will be patched.

Errors

There are many possible errors Valgrind can report, and details on all can be found in the Valgrind User Manual. Here are a few of the common errors you might encounter in this class, what they mean, and what might be done to fix them:

1. `Use of uninitialised value`
   - This error is due to a variable that has not been initialized being dereferenced or otherwise accessed.
   - Usually this is fixed by giving the variable a starting value that makes sense in the program's implementation.
2. `Conditional jump or move depends on uninitialised value(s)`
   - This is similar to above, but instead of the variable only being accessed, its value is being used in a loop or some other form of moving through values.
   - Again, this can be fixed by initializing the value.
3. `Invalid free() / delete / delete []`
   - This error occurs when the program attempts to free memory that cannot be freed. Either this memory is not allocated on the heap, or our has already been freed.
   - To fix this error, pay attention to what memory has been allocated and freed previously.
4. `Mismatched free() / delete / delete []`
   - A mismatched free occurs when one type of memory structure is freed using a different kind of delete statement. Most commonly, this is when `delete` is called on an array, or `delete []` is called on a single value.
5. `Invalid read`
   - An invalid read is reported when a program attempts to read from memory that does not exist, or that it does not have access to. In this class, this will almost always be memory that does not exist on the heap.
   - Make sure the memory accessed exists within the program! This kind of error can include, but is not limited to, `NULL` or stack memory being dereferenced, a non-existent memory address being read from, or a program reading beyond the bounds of an array or other linear data structure.
6. `Invalid write`
   - Similarly to an invalid read, an invalid write occurs when a program attempts to write to a memory address that does not exist, or that the program does not have access to.
   - Make sure that the variable being written to exists within the bounds of the program, and that it has not previously been deleted or set to `NULL`.