CS 246E Fall 2021 — Tutorial 3

September 21, 2021

Summary

1 Valgrind

1.1 Introduction to Valgrind

You may recall from CS136 that Seashell warned you when you had a memory leak or other memory-related error. However, this doesn't happen in CS246E. This is because C/C++ compilers are generally actually not able to detect these errors, but there are extra tools that help us identify certain common classes of errors. For CS246E, we will focus on Valgrind, a tool which helps us identify some of these errors.

1.2 The -g flag

The tools we'll see in this tutorial require a few things to work effectively:

- First, they need to know which files your functions are declared in, which line number each piece of code is on, and so forth. Without them, error messages might be hard to read. However, this information is normally not stored in a program executable
- Second, they need to make sure the code that the computer is running is the code you actually wrote. Most compilers will perform optimizations which make your code faster while still doing the same thing. This is usually useful, but might be confusing when debugging because the debugging tool might report errors in code you never wrote!

Passing the `-g` flag to `g++` both turns off optimizations and stores source information:

```
$ g++ -std=c++14 -g myprogram.cc -o myprogram
```

1.3 Running valgrind to identify bugs

Suppose you were executing a program as follows:

```
$ ./myprogram myarg1 myarg2 < myredirect
```

To run Valgrind, simply precede your command with `valgrind`. Any flags you wish to pass to Valgrind must come before the program.

```
$ valgrind myvalgrindarg1 myvalgrindarg2 ./myprogram myarg1 myarg2 < myredirect
```

To see what exactly Valgrind does, try running the following program both without and with Valgrind:

```
int main() {
    int *p = nullptr;
    *p = 3; //This should crash
    return *p;
}
```

Without Valgrind, this simply gives us the elusive segmentation fault. However, with Valgrind, we are given useful information about where the error occurred and why it occurred.

1.4 Memory errors

Valgrind can detect a class of errors we usually call memory errors. These include:

- Memory leaks (memory is allocated but never freed)
- Invalid reads (a pointer is dereferenced to read from, but the pointer points to invalid memory)
- Invalid writes (a pointer is dereferenced to write to, but the pointer points to invalid memory)
- Mismatched deletes (a pointer is deleted using a different operator than we used to allocate it)
- Double frees (a pointer is deleted twice)
- Uninitialized values (a variable is read before it is first written to)
- And many more

The remaining sections of the Valgrind tutorial give examples of programs with various types of memory errors and how to interpret the error messages that Valgrind gives as useful information to help you fix them.

1.5 Memory leaks - sample1.cc

Consider the following code:

```
int main() {
    int *elements = new int[3];
    return 0;
}
```

This code has a memory leak: `elements` is never deleted. Without Valgrind these might be very hard to notice, let alone fix. Valgrind doesn't do too good of a job helping us fix them, but it at least reports any memory leaks which happened with some basic information. Valgrind divides memory leaks into four categories:

- *Directly lost* leaks are your most common type of leak. This is a piece of memory to which no more pointers remain but which hasn't been deallocated. You'll definitely need to fix each one of these explicitly
- *Indirectly lost* leaks are usually leaks that happen as a consequence of other leaks. For example, if you leak a linked list, the first node will be directly lost (since it is not pointed to) and the remaining nodes will be indirectly lost (since they are at least pointed to by other nodes). Assuming your linked list has a correctly implemented destructor, correctly deleting the first node will prevent the entire list from leaking, thus also fixing the indirect leaks
- *Still reachable* leaks refer to memory which is still allocated, but also still pointed to, at the end of the program. There are good reasons to do this (since the OS will still reclaim memory), but they're beyond the scope of this course. For our purposes, we assume that still reachable memory is leaked and don't allow it, with one exception: the standard library leaks 72,704 bytes of still reachable memory in one block, so you don't have to be worried about this
- *Possibly lost* leaks refer to indirectly lost or still reachable memory where the pointer points partway through the leaked memory rather than to the front of it. These are almost always memory leaks (and the pointer is a coincidence), but certain low-level programs may do this explicitly and can ignore these leaks. For our purposes they are memory leaks and must be fixed

By default, Valgrind won't tell you anything more than that. If you send Valgrind the `--leak-check=full` flag, it will try to tell you where the leaked memory originated, but the hard part of finding and fixing the leaks still has to be done yourself.

Try running the code above with Valgrind and see what information it gives you about memory leaks.

1.6 Invalid reads/writes - sample2.cc

Consider the following code:

```
#include <iostream>

int main() {
    int * elements = new int[3];
    elements[3] = 4;
    std::cout << elements[3] << std::endl;
    delete [] elements;
    return 0;
}
```

This code accesses memory past the end of an array. Conventional wisdom says this should cause a segmentation fault, but it often will not! These sort of errors can be hard to detect, and if they're present in your code it may work just fine on one machine or one day, but crash or perform differently on a different machine or different day.

Try running the code above with Valgrind and see what information it gives you about where this error occurred and why it occurred. It should tell you what you tried to do, the location of the pointer you accessed, how far away it was from valid memory, and where the code was in your source file.

1.7 Uninitialized values - sample3.cc

Consider the following code:

```
#include <iostream>

int main() {
    int i;
    if (i) {
        std::cout << "i is nonzero" << std::endl;
    } else {
        std::cout << "i is zero" << std::endl;
    }
    return 0;
}
```

This code uses a variable which was never initialized. It is incorrect to think that uninitialized `int`s are initialized to 0 or some other standard value. Like with invalid reads/writes, the behaviour of this code may depend on the place in your program that this function was called, the time that you ran the program, the machine it was run on, or many other factors.

Try running the code above with Valgrind and see what information it gives you about the error.

1.8 Mismatched deletes - sample4-1.cc and sample4-2.cc

Consider the following code:

```
int main() {
    int elements[3];
    delete [] elements;
    return 0;
}
```

This code deletes memory using a different mechanism (`delete[]`) than the one that allocated it (the stack). Luckily in this case the compiler will give us a warning that we probably don't intend this, but even a trivial change to the code can fool the compiler:

```
void helper(int *mem) {
    delete [] mem;
}
int main() {
    int elements[3];
    helper(elements);
    return 0;
}
```

Try running the code above with Valgrind and see what information it gives you about the error.

2 GDB

2.1 What is GDB?

The **G**NU **D**e**b**ugger is a debugger which helps identify why your code is not behaving the way you expect it to. It's essentially a REPL (Read-evaluate-print loop) which lets you track the progress of your code as it executes and identify when and why errors arise. If you have ever used the debugger provided by your IDE, you are familiar with the core concepts behind GDB: the main difference is that GDB is generally used from the command line rather than through a graphical interface.

2.2 Why use GDB?

- GDB or similar debuggers (such as LLDB for Macs) are available on every platform, whereas Valgrind is linux-only and doesn't have good alternatives on other platforms
- Valgrind only detects a very specifc class of errors (memory errors), and is rarely helpful if, say, you are trying to track down the source of an exception or the reason a variable is set incorrectly
- `printf` debugging can take a long time due to needing to constantly place more print statements and recompile. Furthermore, it may produce lots of useless output that makes it diffcult to identify a problem, and may in fact change, add, or remove bugs in some contexts (you will likely encounter this in CS350 or CS343)
- GDB solves a different problem than Valgrind and `printf` do in many cases: it can help you identify the cause of problems that Valgrind identified the presence of, or can help you determine where you might want to place debug print statements in a very large program

2.3 When should we not use GDB

GDB is a high-powered tool, and as a result has a significant learning curve and may take some time to use simple tasks. If you already know roughly where or what your problem is, `printf` debugging may be quicker and easier. Furthermore, GDB is more or less useless for detecting many of the memory errors Valgrind can identify. GDB is neither the solution to all your problems nor to none of them: it's simply another tool in your debugging toolbox.

2.4 Finding a segfault - example1.cc

At its most basic, GDB can be used in a similar way to valgrind to detect the source of a crash.

2.4.1 Running (and crashing) your first program in GDB

- Programs run in GDB need to be compiled with the `-g` flag for it to give the most information possible (in particular, line number information), so begin by compiling
  `$ g++ -std=c++14 -g example1.cc example1 hidden.cc -o example1`
- The syntax for starting a GDB session is similar to running Valgrind. Run `gdb ./example1` to start GDB
- Unlike with Valgrind, GDB displays a prompt and waits for instructions. We can quit this prompt at any time using the `quit` (or `q` or Ctrl+D) command. We can ask for help using the `help` (or `h`) command. For now, run the program using the `run` (or `r`) command
- Notice how GDB identifies the crash very similarly to how Valgrind would, and displays the line where the program crashed

2.4.2 Displaying basic diagnostic information and navigating the call stack

However, notice that GDB hasn't actually ended yet. This is because GDB does a lot more than just display a bit more information about crashes. It's waiting for us to issue more commands, either to restart the program or to ask it to display more information.

- For now, let's check out the value of the `i` variable using `print i` (some aliases for the `print` command are `p` and `inspect`). Sure enough, it's `NULL`. Since `i` was a parameter, we can't immediately see why it's `NULL`
- To find out why `i` is `NULL`, let's look at the caller of `crash()`. First, we can use the `backtrace` command (or `bt`) to see the call stack at the time of the crash, which may help for more complicated programs. Running `backtrace`, we see that our parent was `f`. It also shows us all the parameters passed to every function in the call stack
- It looks like `i` was set correctly in `f`. To find out more, we're going to need to examine the context in which crash was called. For now, let's visit `f` by using the `up` command (which goes up the call stack). If we want to get back, we can use the `down` command
- Printing `j`, it looks like (unsurprisingly) it was `NULL` when crash was called. However, looking at the source, we can't immediately see whether sophisticated or complicated was responsible

2.4.3 Navigating a running program using breakpoints and stepping

We're going to need to run our program again to learn any more. This time, we know that something seems to be going wrong in `f`, so we need to tell GDB that we want to pay special attention to `f`.

- First, we want GDB to stop the program and hand us control when it gets to `f`. To do this, we'll set a *breakpoint* at `f`. We can do this using the `break` (or `breakpoint`, or `bp`) command. breakpoint accepts numerous different kinds of arguments, including function names and line numbers. For now, we use `break f`
- Next let's run the program again. This time, it stops when `f` is called and waits for further instructions. It displays the line which it will run next: note that this line hasn't actually run yet, leading to the somewhat confusing fact that `j = 0x0` even though `i` isn't
- We might want to see more of the source code than we currently can. Using `list 10` shows us the 10 lines around (half above, half below) the current position
- We can step forward one line by using the `next` (or `n`) command. Sure enough, `j` has now been set to `i`
- Using `next` once again, notice that `next` skips the body of sophisticated and goes to the next line of `f`. `print j` tells us that `j` is still correct at this stage
- This time, let's get a closer look at what's happening. We can step into functions by using the `step` (or `s`) command which works the same as `next`, other than that it doesn't skip functions
- Here we see the problem: `complicated()` is returning `nullptr`!
- To finish executing the program, we can use the `continue` (or `c`) command, which will cause the program to run without further input from GDB until it ends or reaches the next breakpoint. Sure enough, it crashes. We can also just end the program at any time using the `kill` (or `k`) command or by quitting GDB

2.5 Finding out why the value of a variable is wrong - example2.cc

Next, let's consider a different program, `example2.cc`. In this case, running the program shows us that the value of `dat->i` is wrong. Unfortunately the program doesn't crash here, so there's no convenient place for us to be warned that something went wrong.

- First, let's just double-check the contents of `dat`. Set a breakpoint at print data and run the program. Unfortunately, `print dat` is not very enlightening, since it just gives us the address the pointer points to
- Luckily, GDB understands C-like syntax. To view `dat->i`, we can use the command `print dat->i`. To view all of `dat`, we can use the command print `*dat`
- We could verify that the program works correctly if `dat->i` is changed back to 5 by using the `set` command: in this case, `set dat->i = 5`. Running the program verifies that it now works correctly
- Unfortunately, this tells us nothing we didn't already know. We could step through the program line-by-line to try to see something going wrong, but in this example the program is (supposedly) very long and, more importantly, the error is not going to turn out to be caused by someone saying something of the form `dat->i = 7`
- As a result, we're going to set a watchpoint on `i`. First, set a breakpoint on `main()` and re-run the program, then step once so that `dat` exists. Now set a watchpoint on `i` using `watch dat->i` (or `w dat->i`). A watchpoint is like a breakpoint, but instead of stopping at a specific line of code, it stops when the value at a position in memory changes
- Now resume the program. Notice that the program stops when `dat->i` is about to be assigned the value 5, all good thus far

- Resuming again, the program stops at an unexpected point: when `complicated()` changes `ip`. Apparently `ip` was pointing to `dat->i` all along, which caused our bug

2.6 More commands of interest

There are a number of commands that we didn't have the chance to use in these short examples.

- The `display` (or `disp`) command works like the `print` command, but also runs every time the program stops (at a breakpoint, for example). This is useful if you need to constantly display particular values and are getting tired of typing `print` all the time
- The `delete` (or `del` or `d`) command can be used to remove breakpoints, watchpoints and so forth. To delete breakpoints, you'll need to know their number. You can see all of your breakpoints using the `info` command with `info breakpoints`
- Breakpoints can also be temporarily disabled and subsequently re-enabled using the `disable` and `enable` commands
- The `finish` command continues until the end of the current function.
- The `step` and `next` commands can optionally be supplied with a number of lines to step through
- Pressing enter (i.e. supplying a blank command to GDB) repeats the previous command