CS 246 Fall 2018 — Extra Tutorial

October 3rd, 2018

Summary

1 Valgrind

Valgrind is a Linux tool for detecting memory errors in your programs. We will take a look at a variety of errors that Valgrind will find–you should always test your program with Valgrind to catch these errors! When Marmoset runs your code, it uses Valgrind to detect any memory errors.

Valgrind is run as follows:

```
$ valgrind val_arg_1 ... val_arg_N ./myprogram prog_arg_1 ... prog_arg_N
```

Valgrind does not read from standard input, so if your program takes standard input, it will work properly with Valgrind. This means you can use input redirection like normal. For example, if you had files `test.in, test.args` then you could run

```
$ valgrind ./myprogram $(cat $test.args) < $test.in
```

Exercise: Modify `runSuite` to additionally check your program for memory errors using Valgrind.

1.1 Finding Segfaults

At its most basic, Valgrind is a useful way for detecting why a segfault occurs:

```
// basic.cc
void crash() {
    int *p = nullptr;
    *p = 1;             // Dereferencing a nullptr, this will crash!
}
int main() {
    crash();
}
```

When we run this with Valgrind, we will get something like the following:

```
$ g++14 basic.cc -o basic
$ valgrind ./basic
==6483== Invalid write of size 4
```

```
==6483==    at 0x4004E6: crash() (in ../valgrind-examples/basic)
==6483==    by 0x4004F7: main (in ../valgrind-examples/basic)
==6483==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

This tells us we ran into an error in the function `crash()`, by trying to write to the pointer `0x0`, which is `nullptr`.

1.2 Detecting Memory Leaks

A memory leak occurs when memory is allocated but never freed. In CS 246, if your solution has a memory leak, it is considered to be incorrect. Consider the following program:

```
//sample1.cc
int main() {
    int *elements = new int[10];
    for (int i = 0; i < 10; i++)
        elements[i] = i*i;
    for (int i = 0; i < 10; i++)
        std::cout << i << " * " << i << " = " << elements[i] << std::endl;
    return 0;
}
```

While this program will compile, we never delete the memory we allocated to `elements`! If we run it with Valgrind, we get the following:

```
$ g++14 sample1.cc -o sample1
$ valgrind ./sample1
==69296== LEAK SUMMARY:
==69296==    definitely lost: 40 bytes in 1 blocks
==69296==    indirectly lost: 0 bytes in 0 blocks
==69296==      possibly lost: 0 bytes in 0 blocks
==69296==    still reachable: 72,704 bytes in 1 blocks
==69296==         suppressed: 0 bytes in 0 blocks
```

We see that we have 'definitely lost' 40 bytes of memory; that is, 10 ints with 4 bytes per int.

Notice Valgrind breaks our memory into 4 sections:

- *Definitely lost* memory is memory that is no longer pointed to at the end of the program, so we have no way to delete it
- *Indirectly lost* memory is memory that is pointed to, but still cannot be deleted (e.g., if we forget to delete a linked list, every note past the root will be indirectly lost, and the root will be directly lost)
- *Possibly lost* memory is memory for which we have a pointer that points part-way through the memory. This tends to just be a coincidence, and we consider this memory lost.

- *Still reachable* memory is memory that has not been freed, but which we still have a pointer to. There can be good reasons to do this, but in CS 246 you should not leak still reachable memory.

Exception: The standard library leaks 72,704 bytes of still reachable memory. If Valgrind tells you that you leak exactly 72,704 byes of still reachable memory, there is no issue with your code.

Question: Why did Valgrind tell us we leaked 40 bytes of definitely lost memory? We still had the pointer `elements` at the end of the program, shouldn't it be still reachable?

Answer: When `main` returns, `elements` is removed from the stack and the pointer is lost. Try making `elements` global, and see what happens.

1.3 Invalid reads/writes

Consider the following piece of code:

```
//sample2.cc
int main() {
    int *elements = new int[9];
    for (int i = 0; i < 10; i++)
        elements[i] = i*i;
    for (int i = 0; i < 10; i++)
        std::cout << i << " * " << i << " = " << elements[i] << std::endl;

    delete [] elements;
    return 0;
}
```

Now we delete `elements` at the end, however we have introduced a new error: we try to read and write to `elements[9]`, when we only asked for 9 ints (`elements[0]` through `elements[8]`).

What will this code do when we run it? Actually, it will probably work. However, since we don't own that memory, we can't tell for sure. Since this works most of the time, it can be particularly difficult to detect that we have made a mistake of this nature, but fortunately Valgrind is smart enough to detect it:

```
$ g++14 sample2.cc -o sample2
$ valgrind ./sample2
==134705== Invalid write of size 4
==134705==    at 0x400965: main (in ../valgrind-examples/sample2)
==134705==  Address 0x5ac2ca4 is 0 bytes after a block of size 36 alloc'd
==134705==    at 0x4C2E80F: operator new[](unsigned long) (...)
==134705==    by 0x400938: main (in ../valgrind-examples/sample2)
==134705== Invalid read of size 4
==134705==    at 0x40098E: main (in ../valgrind-examples/sample2)
==134705==  Address 0x5ac2ca4 is 0 bytes after a block of size 36 alloc'd
```

```
==134705==    at 0x4C2E80F: operator new[](unsigned long) (...)
==134705==    by 0x400938: main (in ../valgrind-examples/sample2)
```

Here we get two errors from Valgrind: first, an invalid write of size 4 (when we try to set `elements[9] = 9*9`) and second, an invalid read of size 4 (when we try to `cout « elements[9]`).

1.4 Uninitialized Values

Much like in C, if you don't initialize the value of a primitive type in C++, you can't be sure what its value is. Consider the following piece of code:

```
// sample3.cc
#include <iostream>
using namespace std;
int main() {
    int i;
    if (i) {
        cout << "i is not 0" << endl;
    } else {
        cout << "i is 0" << endl;
    }
}
```

If we run this with Valgrind, we get

```
$ g++14 sample3.cc -o sample3
$ valgrind ./sample3
==161541== Conditional jump or move depends on uninitialised value(s)
==161541==    at 0x400852: main (in ../valgrind-examples/sample3)
```

Here, a "conditional jump" is an if statement. Valgrind will detect when this occurs with any conditional jump–for example, while loops.

1.5 Mismatched Deletes

In C++, when you allocate memory with a particular method, it is expected you will use its companion method to free it; for example, use `malloc` with `free`, `new` with `delete`, and `new[]` with `delete[]`. If you try to mix things up, you will get a very ugly looking error–try running the following:

```
//sample4.cc
void helper(int *p) {
    delete [] p;
}
int main() {
    int x[10];
```

```
    helper(x);
}
```

The reason this gives an error is because we are using `delete[]` to delete memory on the stack. Valgrind will tell us this:

```
$ g++14 sample4.cc -o sample4
$ valgrind ./sample4
==210771== Invalid free() / delete / delete[] / realloc()
==210771==    at 0x4C2F74B: operator delete[](void*) (...)
==210771==    by 0x400684: helper(int*) (in ../valgrind-examples/sample4)
==210771==    by 0x4006AA: main (in ../valgrind-examples/sample4)
==210771==  Address 0xfff00069c is on thread 1's stack
==210771==  in frame #2, created by main (???:)
```

The important part of this error message is to see that we are calling `delete[]` on address `0xfff00069c`, which is on the stack. If we change `int x[10];` to `int *x = new int[10];`, then the code will run without issue.

2 GDB

GDB is short for the **G**NU **D**e**B**ugger. In some sense, it is a kind of REPL for C/C++, similar to Dr. Racket. GDB is a very powerful debugging tool that will enable you to quickly find flaws in your program without having to keep recompiling it.

2.1 When to use GDB

GDB is not always the right tool for the job. If you have a memory error, you should use Valgrind. If you already have a good idea of what the issue is, you may find it faster to turn to the familiar "printf debugging", wherein you litter your code with print statements to see if variables are what you expect them to be. However, for any serious debugging task, GDB will allow you to jump through your code with much more flexibility than printf debugging will.

2.2 Basic GDB

In order to use GDB, you will want to compile with the `-g` option given to `g++`. This option effectively tells the compiler not to optimize your code, as well as ship your program with a copy of the code that other programs can read. Consider the following code and GDB interaction:

```
// basic.cc
#include <iostream>
int main() {
    int x = 5;
```

```
    int y = 2;
    std::cout << "Hello World!" << std::endl;
}
// end basic.cc
$ g++14 basic.cc -o basic -g
$ gdb ./basic
(gdb) break main
(gdb) run
    int x = 5;
(gdb) next
    int y = 2;
(gdb) print x
$1 = 5
(gdb) next
    std::cout << "Hello World!" << std::endl;
(gdb) print x+y
$1 = 7
(gdb) next
Hello World!
(gdb) q
```

Lets take a look at what GDB commands we used here:

Short Form	Long Form	Effect
q	quit	Quit gdb
b	break	Set a breakpoint on the given line/function
r	run	Run the program
n	next	Go to the next line (run the last displayed line)
p	print	Print an expression (Accepts lots of C-like syntax)

2.3 Debugging with GDB

Consider the following code:

```
// example1.cc
#include "example1_hidden.h" // Defines complicated(), sophisticated()

void crash(int *i) {
    *i = 3;
}

void f(int *i) {
    int *j = i;
    j = sophisticated(j);
    j = complicated(j);
```

```
    crash(j);
}

int main(int argc, char *argv[]) {
    f(&argc);
}
```

If `crash` is called with an invalid pointer, the program will crash. When we run it, we see that it does crash, and we can use GDB to confirm that it was because we dereferenced a null pointer:

```
(gdb) break crash
(gdb) run
Breakpoint 1, crash (i=0x0) at example1.cc:4
4   *i = 3
```

We want to find out why `i` is 0 here. Type `up` to move up the call stack, in this case back into the function `f` where `crash` was called as `crash(j)`

```
(gdb) up
#1  0x000000000040052b in f (i=0x7fffffffe6cc) at example1.cc:11
11   crash(j);
(gdb) print j
$1 = (int *) 0x0
```

This tells us that `i` was correctly set when calling `f`, and so there must be an error somewhere within `f`. Set a breakpoint on `f` and restart the program.

```
(gdb) break f
(gdb) run
Breakpoint 2, f (i=0x7fffffffe6cc) at example1.cc:8
8   int *j = i;
(gdb) display j
1: j = (int *) 0x7fffffffe6cc
(gdb) n
9   j = sophisticated(j);
1: j = (int *) 0x7fffffffe6cc
(gdb) n
10   j = complicated(j);
1: j = (int *) 0x7fffffffe6cc
(gdb) n
11   crash(j);
1: j = (int *) 0x0
```

Here we used the `display` command, which makes the value of `j` print after every command. We can see that `j` is set properly, but then becomes 0 after running line 10, `j = complicated(j);`. This is where the error is.

2.4 Another GDB Debugging Example

What can we do in the case where variables don't have the values we want them to, but we have no idea where they are being changed? If we don't know where to look, the above method will not work.

GDB supports "watchpoints", which is the ability to look at a particular location in memory, and stop the debugging session when it changes. By setting a watchpoint on a variable, we can detect not only when we change it directly, but also when it is changed by pointers holding the address of that variable.

Compile `example2` and run it. The output will show us that we have a value `i=7`, when we expect `i=5`.

Try setting a breakpoint on `print_data`, and printing the structure given to ensure the error is not in the print function. Remark that although we are given a pointer, GDB understands C syntax, and so you can do `print *dat` without issue.

Now we will set a watchpoint:

```
(gdb) break main
(gdb) run
(gdb) next
19   some_data dat;
(gdb) watch dat.i
(gdb) continue
Hardware watchpoint 2: dat.i

Old value = 4198000
New value = 5
(gdb) continue
Hardware watchpoint 2: dat.i

Old value = 5
New value = 7
complicated (i=0x7fffffffe6b0) at example2_hidden.cc:7
(gdb) up
#1  0x0000000000400d79 in main (argc=1, argv=0x7fffffffe7b8) at example2.cc:25
25   complicated(ip); //We don't know what this does
```

and then we have located the source of the bug–`ip` must point to `dat.i` in our main function. To summarize, we watched `dat.i` until we found it being changed to something incorrect, then we used `up` to return to the point where we called the function that changed it.

2.5 Additional GDB Commands

Short Form	Long Form	Effect
	info breakpoints	List all break/watchpoints and IDs
d	delete	Delete the given breakpoint ID
disp	display	Set a variable to print every step
undisp	undisplay	Undo a 'display' command
k	kill	Kill the currently running program
c	continue	Resume running the program
	up	Move up the callstack
	down	Move down the callstack
f	finish	Finish the current function