Valgrind Workshop

Kaylin Finke

May 2, 2019

Contents

Chapter 1

What is Valgrind?

The Valgrind website [valgrind.org] says, “Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.” This description however is not exceptionally useful to somebody who just wants to know when and how to use Valgrind. If you’ve used a debugger before such as GDB youll quickly see that the sort of utility provided by such debuggers is not the functionality provided by Valgrind. In fact, though we won’t go through that here, you can use Valgrind and GDB together and they complement each other. Valgrind tools are geared toward analysis of your programs as opposed to the way you would debug a program using other tools. For example, where GDB lets you view in great detail what your program is doing, Valgrind attempts to analyze the behavior of your program and tell you where your program is doing things it shouldn’t (or conversely, not doing something it should).

1.1 Getting Started

Before you continue please ensure you're logged in to a computer running Linux. While Valgrind is available for many systems and there are ports to many variants of Unix as well as ways to get it to run on Windows, the primary development target for Valgrind is Linux and we'll be working with Valgrind in a Linux environment. You should also have a copy of the examples we're going to work through in this presentation. I've made all the code I'll be running available to you in my home directory and you can copy it into yours. The following command will do just that creating a new directory named valgrindworkshop in your home directory and placing the examples there. You can name the directory whatever you like if valgrindworkshop is not your favorite name.

```
cp -R ~finkec/valgrind_demo/examples ~/valgrindworkshop
```

The Valgrind suite contains a variety of tools for helping improve your programs. These utilities are not specific to the C programming language though some of them are largely designed for programs where memory is managed manually like C, C++, and Ada. We will be specifically focusing on the tool Memcheck in this workshop and as such we'll be sticking to the C language. While Memcheck is only a small fraction of what Valgrind can do, it's probably the most useful tool in regards to the software development courses here at Western. In order to talk about Memcheck we'll need to first quickly go over the memory model, paying special attention to the stack and heap, so we can better understand the output Valgrind gives us.

Chapter 2

The Virtual Memory Model

2.1 The Stack

In CS247 you learned, or will learn, about the virtual memory layout of a Unix process. The important parts for this presentation are the stack and the heap. In particular we want to understand the basics of how a function call works and how we can use the stack to understand where we are in the execution of a program. We'll also need to quickly go over the library calls malloc and free to understand how they give us access to dynamic memory in the heap. Consider the diagram on the following page:

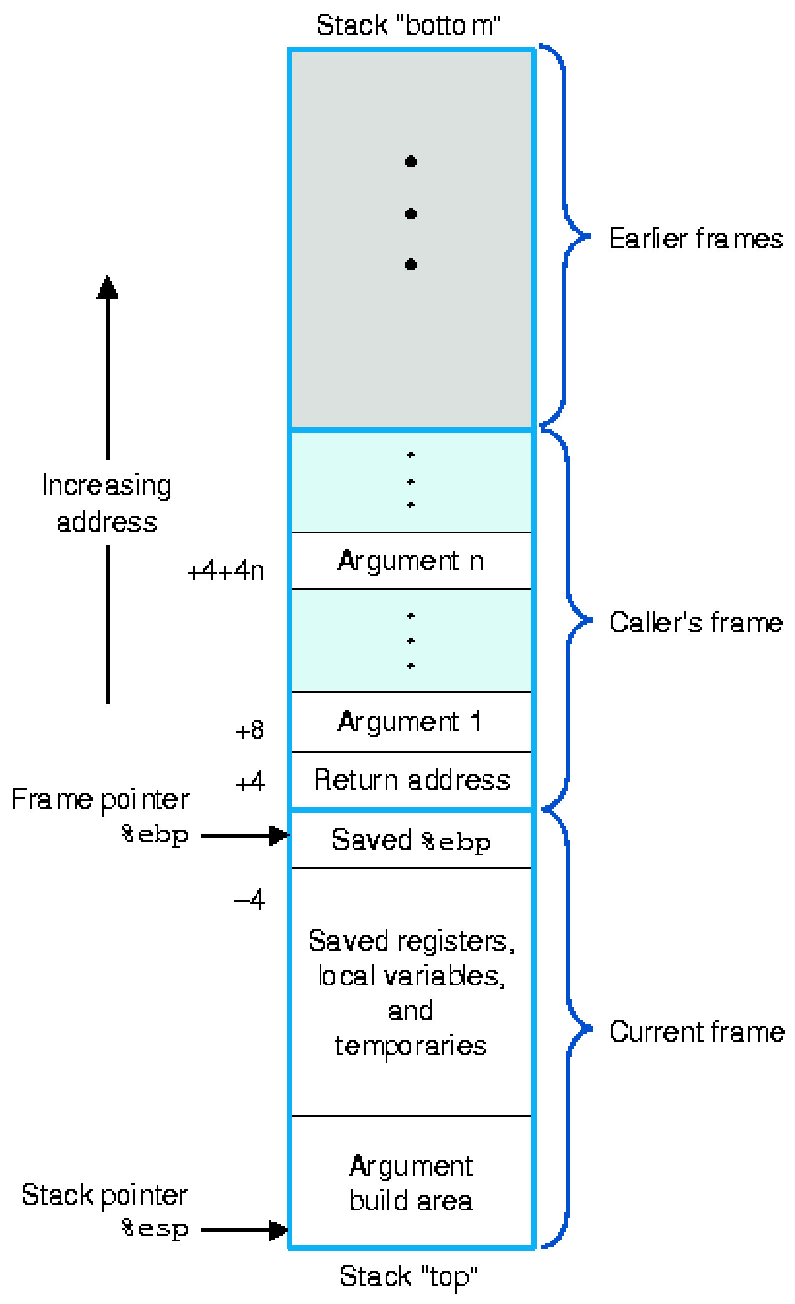

Figure 2.1: The Call Stack

2.1.1 Stack Traces

As we call functions in our program, stack frames providing storage for local variables and function parameters are created growing the stack downward. By looking at the stack at any moment, and keeping track of line numbers in the code, we can know where we are in the execution of a program. The important thing to remember for this presentation is that the stack can tell us not just which function we're in but how we got there. To demonstrate this I've written the following buggy program to give us a nice stack trace to read. We'll use GDB here to show the stack trace but the idea is the same. The following code prints out the arguments on the command line until it crashes, presuming there were arguments:

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

void print_string(char** string_ptr) {
  int i = 0;
  while('\0' != (*string_ptr)[i]) {
    printf("%c", (*string_ptr)[i++]);
  }
  printf("\n");
  print_string(string_ptr+1);
}

int main(int argc, char* argv[]) {
  if(argc > 1) {
    print_string(argv+1);
  }
  return EXIT_SUCCESS;
}
```

You can compile the program by typing:

```
make stacktrace
```

which will give you the program named stacktrace. In order to actually view the stack trace we'll get back when the program crashes we'll use the GNU debugger. Type:

```
gdb stacktrace
```

to open GDB and then type

```
run Hello World!
```

You'll get the following output:

```
Starting program:
    /home/finkec/valgrind_demo/examples/stacktrace Hello World!
Hello
World!

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400510 in
    print_string (string_ptr=0x7fffffffe5a0) at stacktrace.c:6
6         while('\0' != (*string_ptr)[i]) \{
```

This indicates our program crashes. In order to figure out how we got there, we can view the stack trace. Typing

```
backtrace
```

Rewards us with:

```
#0  0x0000000000400510 in print_string
    (string_ptr=0x7fffffffe5a0) at stacktrace.c:6
#1  0x0000000000400575 in print_string
    (string_ptr=0x7fffffffe598) at stacktrace.c:10
#2  0x0000000000400575 in print_string
    (string_ptr=0x7fffffffe590) at stacktrace.c:10
#3  0x00000000004005b5 in main
    (argc=3, argv=0x7fffffffe588) at stacktrace.c:15
```

Here the debugger is telling us our program crashed in the function print_string at line 6, but it's also telling us the stack frames at the time of the error. Using this information we can see how we got to the point where we had an error which gives us a better understanding of our program's behavior. Valgrind's output is structured in a similar manner to GDB's backtrace output so being able to understand such information is important. By dissecting the above we can see that the function main called the function print_string, which called print_string, which called print_string again, where we finally encountered the error. With the ability to read stack traces we are just about ready to jump into using Valgrind, but before we do we should find some motivation for using Memcheck by remembering how dynamic memory works. At this point you can quit GDB by typing:

```
quit
```

and confirming you wish to quit. We won't be using it again in this demonstration.

2.2 The Heap

In order to understand the meaning behind Valgrind's output we will also need to understand the heap. The heap is dynamic memory available to us as we need it, and in C unlike Java it's up to the programmer to manage these resources. A call to a function like malloc requests a chunk of dynamic memory which needs to be explicitly freed when our program is finished using it. When our program loses track of a piece of dynamic memory without freeing it we have a memory leak. When we access memory outside of a chunk we've requested we have a buffer overrun. We'll use Valgrind to catch these and other sorts of errors related to dynamic memory. Let's take a look at the whole memory layout:

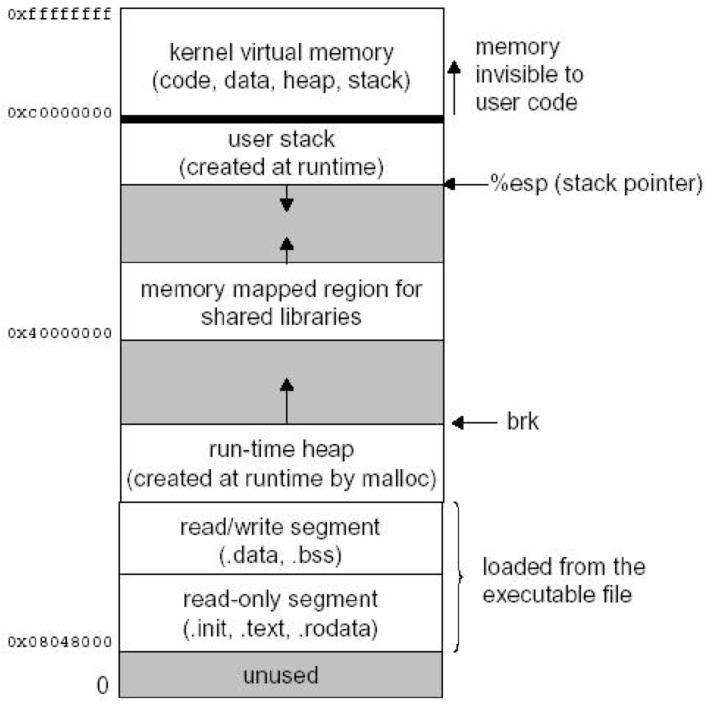

Figure 2.2: Unix Memory Layout

Here you can see the section labeled runtime heap situated in low memory above the data segments and program text. The label brk indicates the frontier of the heap's storage called the break which malloc and free may move by calling the system calls brk and sbrk. If the program's break becomes too high in memory our program will be unable to obtain more memory and may be unable to function correctly. If we run out of bounds of memory we're given we may access data past the break and our program may crash. These are far from the only consequences of improper use of heap memory, but they're certainly bad and it would be good to avoid them. Sadly, C has no real way to tell us when we've run past the end of a chunk of memory presuming it's in a segment we're allowed to access, and no way of knowing when we've forgotten to free memory after we've finished using it. Fortunately, Valgrind can tell us these things. Let's work through some examples together.

Chapter 3

Using Valgrind

3.1 Valgrind options

When using Valgrind to debug memory, the following options will be useful to us. You can read about them in the manual using

```
man 1 valgrind
```

I'm going to use the following options during this demonstration and I recommend you use them as well. They'll give you the most useful output without being too esoteric.

3.1.1 verbose

The verbose option: This will print out a lot of information but some of this is warnings about what Valgrind calls “unusual program behavior” which we certainly want to see. Use

```
–verbose
```

or

```
-v
```

3.1.2 leak-check

This instructs Valgrind to do a search for memory leaks in the program when the program finishes running. It will then print out a full summary of the data it founds in addition to the error counts. We’ll want this behavior in order to find memory leaks. Use

```
–leak-check=full
```

3.1.3 track-origins

This option instructs Valgrind to search for the origin of uninitialized memory used in “dangerous ways” such as inside an if statement or as part of a test for a loop exit condition. This will give us a great place to look for bugs related to undefined program behavior and it’s a good idea to enable it. Use

```
–track-origins=yes
```

3.1.4 tool

The tool option instructs Valgrind to use a specific tool; in this case we'll be using Memcheck. While Memcheck is far from the only tool in the Valgrind suite, it's actually the default tool so specifying Memcheck is not actually required. Use

```
–tool=memcheck
```

or leave it blank. It makes no difference. This should be all the options you'll need for debugging most simple programs. We'll now move on to using Memcheck to track down bugs in the included example programs.

3.2 Example 1

Take a look at the source code for example 1. You can open the file with a graphical editor by typing:

```
emacs example1.c&
```

Which will open the program source code with Emacs. You can specify your favorite editor instead of Emacs if you'd prefer. I'll use Emacs for these examples.

If you're familiar with C you can read through the source and see what this program does. It takes in arguments from the command line and passes them into a function that will print them. That function puts the arguments into a linked list, prints out the arguments from the linked list, and frees the linked list before returning and exiting. We're going to try to find bugs in it. Begin by building the program with the command

```
make example1
```

This will compile the program into an executable you can run named example1. Running it we can see it works correctly. Type

```
./example1 This is an example program.
```

And you can observe it prints out a list of the words we passed to it, separated by newlines. It seems to be working correctly, but we should use Valgrind to test if there are errors. Issue the following command:

```
valgrind -v --leak-check=full --track-origins=yes ./example1
```

We are, in these examples, most interested in the output summary after the program finishes. Valgrind spits out a ton of output in verbose mode but the end of the output should look like this:

```
==17910== HEAP SUMMARY:
==17910==     in use at exit: 0 bytes in 0 blocks
==17910==   total heap usage: 0 allocs,
                0 frees, 0 bytes allocated
==17910==
==17910== All heap blocks were freed -- no leaks are possible
==17910==
==17910== ERROR SUMMARY: 0 errors from 0 contexts
                (suppressed: 0 from 0)
==17910== ERROR SUMMARY: 0 errors from 0 contexts
                (suppressed: 0 from 0)
```

This tells us that we didn't call malloc nor free, and trivially freed all of the 0 blocks of memory we allocated. It also tells us there were no errors related to invalid data accesses, and that the ID of the process was 17910 in this case. This brings us to a very important point. Your test cases have to find the errors. If you want to find a memory leak in your program, you have to actually leak memory. If you think you have a buffer overrun in your program, you need a test case where your program overruns the buffer. Fortunately it's not as hard as testing your program with GDB; you need only make the test cases, not the correct output. Valgrind figures out the correct behavior for you.

Let's re-run example 1 through Valgrind with some parameters this time. Type

```
valgrind -v –leak-check=full –track-origins=yes ./example1 This is an example program.
```

to pass some arguments to the program we're running. Valgrind gives us the following output:

```
==18027== HEAP SUMMARY:
==18027==     in use at exit: 28 bytes in 5 blocks
==18027==   total heap usage: 10 allocs,
                5 frees, 108 bytes allocated
==18027==
==18027== Searching for pointers to 5 not-freed blocks
==18027== Checked 63,064 bytes
==18027==
==18027== 5 bytes in 1 blocks are definitely lost
                in loss record 1 of 2
==18027==    at 0x4C2BBCF: malloc
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==18027==    by 0x4EC33C9: strdup (strdup.c:42)
==18027==    by 0x400680: make_list (example1.c:14)
==18027==    by 0x40076A: print_args (example1.c:43)
==18027==    by 0x4007E1: main (example1.c:58)
==18027==
==18027== 23 bytes in 4 blocks are definitely lost
                in loss record 2 of 2
==18027==    at 0x4C2BBCF: malloc
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==18027==    by 0x4EC33C9: strdup (strdup.c:42)
==18027==    by 0x4006AA: make_list (example1.c:18)
==18027==    by 0x40076A: print_args (example1.c:43)
==18027==    by 0x4007E1: main (example1.c:58)
==18027==
==18027== LEAK SUMMARY:
==18027==    definitely lost: 28 bytes in 5 blocks
==18027==    indirectly lost: 0 bytes in 0 blocks
==18027==      possibly lost: 0 bytes in 0 blocks
==18027==    still reachable: 0 bytes in 0 blocks
==18027==         suppressed: 0 bytes in 0 blocks
==18027==
==18027== ERROR SUMMARY: 2 errors from 2 contexts
                (suppressed: 0 from 0)
==18027== ERROR SUMMARY: 2 errors from 2 contexts
                (suppressed: 0 from 0)
```

This time Valgrind tells us we've leaked memory. From the stack trace we can see that at both lines 14 and 18 our program called strdup and before freeing the linked list structure we forgot to free the copy of the string. You'll notice that sadly Valgrind can't tell us where we lost the memory. This unfortunately means that in order to track down memory leaks we'll need to employ our normal debugging techniques, but at least we know leaks exist. In this case freeing current_node->data before calling the delete_node routine resolves our memory leak issue. Modify free_list in example1.c to look like the following:

```c
void free_list(node_t *list) {
  node_t *current_node;

  while(NULL != list) {
    current_node = list;
    list = list->next;
    free(current_node->data);
    delete_node(current_node);
  }

}
```

Re-running Valgrind on the code tells us that we indeed fixed that memory leak by changing the free_list function. Let's check out a different perhaps wrong solution to the problem in example 2.

3.3 Example 2

Example 2 is very similar to example 1. In fact the code behaves the same it's just been modified to demonstrate a different type of memory error Valgrind will catch. Now that you've seen a correct solution to fixing example 1, the different fix in example 2 might seem quite suspect. If you wish to view the code for example 2 feel free to open example2.c in your favorite text editor.

Build example 2 by typing:

```
make example2
```

and run it with Valgrind just as we did example 1 by typing:

```
valgrind -v --leak-check=full --track-origins=yes
    ./example2 This is an example program.
```

Valgrind will spit out a summary once the program terminates that looks not unlike the following:

```
==30931== HEAP SUMMARY:
==30931==     in use at exit: 0 bytes in 0 blocks
==30931==   total heap usage: 10 allocs,
                10 frees, 108 bytes allocated
==30931==
==30931== All heap blocks were freed -- no leaks are possible
==30931==
==30931== ERROR SUMMARY: 5 errors from 1 contexts
                (suppressed: 0 from 0)
==30931==
==30931== 5 errors in context 1 of 1:
```

```
==30931== Invalid read of size 8
==30931==    at 0x400723: free_list (example2.c:34)
==30931==    by 0x4007B9: print_args (example2.c:52)
==30931==    by 0x4007E1: main (example2.c:60)
==30931==  Address 0x5202098 is 8 bytes inside a block
                of size 16 free'd
==30931==    at 0x4C2CE2B: free
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==30931==    by 0x400884: delete_node (LinkedList.c:16)
==30931==    by 0x40071E: free_list (example2.c:33)
==30931==    by 0x4007B9: print_args (example2.c:52)
==30931==    by 0x4007E1: main (example2.c:60)
==30931==  Block was alloc'd at
==30931==    at 0x4C2BBCF: malloc
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==30931==    by 0x40080A: new_node (LinkedList.c:7)
==30931==    by 0x400688: make_list (example2.c:15)
==30931==    by 0x40076A: print_args (example2.c:44)
==30931==    by 0x4007E1: main (example2.c:60)
==30931==
==30931== ERROR SUMMARY: 5 errors from 1 contexts
                (suppressed: 0 from 0)
```

In this case the error is not in fact a memory leak. We've freed all our memory but we've incorrectly read from memory we've freed after freeing it. Here we just did things in the incorrect order, but in general this can also indicate that you may have forgot to set a variable to NULL after freeing it, and your program believes since it's not NULL and is a pointer it can safely dereference it elsewhere in your code. Example 2 can be fixed exactly the same way example 1 was fixed since the programs are otherwise identical.

3.4 Example 3

The next 3 examples are much less complex. Depending on how you look at this example there's either one or two errors here. Note that I picked 12 bytes because the string "Hello World\n" is 12 bytes including the trailing newline. Try running example 3 through Valgrind and entering Hello World.

```
valgrind -v –leak-check=full –track-origins=yes ./example3
```

```
Hello World
```

You should receive output not dissimilar to:

```
==31063== HEAP SUMMARY:
==31063==     in use at exit: 0 bytes in 0 blocks
==31063==   total heap usage: 1 allocs,
                1 frees, 12 bytes allocated
==31063==
==31063== All heap blocks were freed -- no leaks are possible
==31063==
==31063== ERROR SUMMARY: 2 errors from 2 contexts
                (suppressed: 0 from 0)
==31063==
==31063== 1 errors in context 1 of 2:
==31063== Invalid read of size 1
==31063==    at 0x4E864B2: vfprintf (vfprintf.c:1642)
==31063==    by 0x4E8CC38: printf (printf.c:33)
==31063==    by 0x40072D: main (example3.c:20)
==31063==  Address 0x520204c is 0 bytes after a block
                of size 12 alloc'd
==31063==    at 0x4C2BBCF: malloc
```

```
             (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31063==    by 0x40068A: main (example3.c:10)
==31063==
==31063==
==31063== 1 errors in context 2 of 2:
==31063== Syscall param read(buf) points to unaddressable byte(s)
==31063==    at 0x4F2F430: __read_nocancel
                (syscall-template.S:81)
==31063==    by 0x400706: main (example3.c:17)
==31063==  Address 0x520204c is 0 bytes after a block
                of size 12 alloc'd
==31063==    at 0x4C2BBCF: malloc
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31063==    by 0x40068A: main (example3.c:10)
==31063==
==31063== ERROR SUMMARY: 2 errors from 2 contexts
                (suppressed: 0 from 0)
```

Note what happened here. Even though example 3 sets the memory we allocated to be nul characters, we actually requested 13 bytes instead of 11. This ensured that the newline in our example input overwrote the null terminator at the end of our buffer causing printf to access memory where it shouldn't have, but there's something even cooler. Valgrind was able to check the parameters of the system call read and tell us that there was a potential for a buffer overrun even if we didn't actually trigger it. We can see this more clearly by re-running valgrind with example 3 but giving it a shorter input string.

```
valgrind -v –leak-check=full –track-origins=yes ./example3
```

```
Hello
```

In this case the item we passed to printf was indeed a string, but Valgrind still complained that though read was told it could read up to 13 bytes, the memory address it was given to store data pointed to a block of only 12 bytes. Fixing our error by changing the line

```
bytes_read = read(STDIN_FILENO, maybe_a_string, string_length+1);
```

to

```
bytes_read = read(STDIN_FILENO, maybe_a_string, string_length-1);
```

resolves both these issues. Make this change and re-run valgrind:

```
valgrind -v –leak-check=full –track-origins=yes ./example3
```

```
Hello World
```

You should get something similar to this:

```
==31266== HEAP SUMMARY:
==31266==     in use at exit: 0 bytes in 0 blocks
==31266==   total heap usage: 1 allocs,
                1 frees, 12 bytes allocated
==31266==
==31266== All heap blocks were freed -- no leaks are possible
==31266==
==31266== ERROR SUMMARY: 0 errors from 0 contexts
                (suppressed: 0 from 0)
==31266== ERROR SUMMARY: 0 errors from 0 contexts
                (suppressed: 0 from 0)
```

With example 3 fixed, let's check out example 4.

3.5 Example 4

This example is just as short as example 3, but it's perhaps the coolest. This example actually relies on Valgrind knowing the rules about using the memcpy family of functions (memmove, memcpy, strcpy, etc). You can view the source code to example 4 in the same way as we've viewed the others. When you're ready, make example 4 and run it through Valgrind. It'll prompt you to enter a string. I recommend "Test Input":

```
make example4
```

```
valgrind -v –leak-check=full –track-origins=yes ./example4
```

```
Test Input
```

Valgrind should give you the following output:

```
==31563== HEAP SUMMARY:
==31563==     in use at exit: 0 bytes in 0 blocks
==31563==   total heap usage: 1 allocs,
                1 frees, 120 bytes allocated
==31563==
==31563== All heap blocks were freed -- no leaks are possible
==31563==
==31563== ERROR SUMMARY: 1 errors from 1 contexts
                (suppressed: 0 from 0)
==31563==
==31563== 1 errors in context 1 of 1:
==31563== Source and destination overlap in
                memcpy(0x5202040, 0x5202045, 12)
==31563==    at 0x4C30573: memcpy@@GLIBC_2.14
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31563==    by 0x4007E6: main (example4.c:21)
==31563==
==31563== ERROR SUMMARY: 1 errors from 1 contexts
                (suppressed: 0 from 0)
```

By checking out the manual for memcpy using

```
man 3 memcpy
```

we can indeed see the description tells us we were using memcpy incorrectly. We should have used memmove.

```
DESCRIPTION
       The  memcpy()  function copies n bytes from memory area
       src to memory area dest.  The memory areas must not
       overlap. Use memmove(3) if the memory areas do overlap.
```

Changing our program to call memmove instead of memcpy fixes one of the bugs in this example, but our first input didn't trigger the other. Let's try it. Re-run Valgrind on example 4 but this time enter a few lines of text when prompted for a string:

```
valgrind -v –leak-check=full –track-origins=yes ./example4
```

```
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
nnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnnn
```

In this case, we've managed to trigger the second bug. Valgrind gives us the following output summary:

```
==31583== HEAP SUMMARY:
==31583==     in use at exit: 0 bytes in 0 blocks
==31583==   total heap usage: 2 allocs,
                2 frees, 428 bytes allocated
==31583==
==31583== All heap blocks were freed -- no leaks are possible
==31583==
==31583== ERROR SUMMARY: 5 errors from 3 contexts
                (suppressed: 0 from 0)
==31583==
==31583== 1 errors in context 1 of 3:
==31583== Invalid read of size 1
==31583==    at 0x4C30710: memcpy@@GLIBC_2.14
            (in/usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31583==    by 0x4007E6: main (example4.c:21)
==31583==  Address 0x5202236 is 2 bytes after a block
                of size 308 alloc'd
==31583==    at 0x4C2DD9F: realloc
            (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31583==    by 0x4EA7DBB: getdelim (iogetdelim.c:106)
==31583==    by 0x400767: main (example4.c:17)
==31583==
==31583==
==31583== 1 errors in context 2 of 3:
==31583== Source and destination overlap in memcpy
                (0x5202100, 0x5202105, 307)
==31583==    at 0x4C30573: memcpy@@GLIBC_2.14
            (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31583==    by 0x4007E6: main (example4.c:21)
==31583==
==31583==
==31583== 3 errors in context 3 of 3:
==31583== Invalid read of size 1
==31583==    at 0x4C3071E: memcpy@@GLIBC_2.14
            (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31583==    by 0x4007E6: main (example4.c:21)
==31583==  Address 0x5202234 is 0 bytes after a block
                of size 308 alloc'd
==31583==    at 0x4C2DD9F: realloc
              (in /usr/lib/valgrind/vgpreload_memcheck
                -amd64-linux.so)
==31583==    by 0x4EA7DBB: getdelim (iogetdelim.c:106)
==31583==    by 0x400767: main (example4.c:17)
==31583==
==31583== ERROR SUMMARY: 5 errors from 3 contexts
                (suppressed: 0 from 0)
```

What happened here is that although the input string fit in the buffer, the length of the input string +5 overran the buffer and our call to memcpy accessed memory outside of the block we'd allocated for our string. Fixing this is left as an exercize to the reader. We're just about done with examples and ready to debug an actual program, but let's look at one more feature that Memcheck has which is unrelated to the heap.

3.6 Example 5

Example 5 is the shortest of all the examples at 3 lines of code. Remember C, unlike Java, doesn't give stack nor heap variables initial values if you fail to assign them. This leaves the character array in example 5 named maybe_a_string with whatever garbage happened to be in memory at the time of the call to main. Running the program, we can see Memcheck warns us about using uninitialized memory in a dangerous way even though that memory is on the stack! In general Memcheck won't catch errors that don't relate to improper use of the heap, but in the case of uninitialized memory it will. Making example 5 and running it with Valgrind:

```
make example5
```

```
valgrind -v –leak-check=full –track-origins=yes ./example5
```

Will get you a ton of errors related to this one bug. Making the array actually a string say by giving maybe_a_string[0] the value of 0 will solve the problem. Without fixing it, here's what Memcheck tells us:

```
==31829== HEAP SUMMARY:
==31829==     in use at exit: 0 bytes in 0 blocks
==31829==   total heap usage: 0 allocs, 0 frees,
              0 bytes allocated
==31829==
==31829== All heap blocks were freed -- no leaks are possible
==31829==
==31829== ERROR SUMMARY: 6 errors from 4 contexts
              (suppressed: 0 from 0)
==31829==
==31829== 1 errors in context 1 of 4:
```

```
==31829== Syscall param write(buf) points to
              uninitialised byte(s)
==31829==    at 0x4F2F490: __write_nocancel
                (syscall-template.S:81)
==31829==    by 0x4EB1B9E: _IO_file_write@@GLIBC_2.2.5
                (fileops.c:1251)
==31829==    by 0x4EB3088: new_do_write (fileops.c:506)
==31829==    by 0x4EB3088: _IO_do_write@@GLIBC_2.2.5
                (fileops.c:482)
==31829==    by 0x4EB223C: _IO_file_xsputn@@GLIBC_2.2.5
                (fileops.c:1319)
==31829==    by 0x4E8572D: vfprintf (vfprintf.c:1673)
==31829==    by 0x4E8CC38: printf (printf.c:33)
==31829==    by 0x400513: main (example5.c:10)
==31829==  Address 0x4026010 is in a rw- anonymous segment
==31829==  Uninitialised value was created by a stack allocation
==31829==    at 0x4004F0: main (example5.c:7)
==31829==
==31829== 1 errors in context 2 of 4:
==31829== Conditional jump or move depends on
              uninitialised value(s)
==31829==    at 0x4EB22B7: _IO_file_xsputn@@GLIBC_2.2.5
                (fileops.c:1289)
==31829==    by 0x4E86471: vfprintf (vfprintf.c:1642)
==31829==    by 0x4E8CC38: printf (printf.c:33)
==31829==    by 0x400513: main (example5.c:10)
==31829==  Uninitialised value was created by a stack allocation
==31829==    at 0x4004F0: main (example5.c:7)
==31829==
==31829== 1 errors in context 3 of 4:
==31829== Conditional jump or move depends on
              uninitialised value(s)
==31829==    at 0x4EB22A9: _IO_file_xsputn@@GLIBC_2.2.5
                (fileops.c:1289)
==31829==    by 0x4E86471: vfprintf (vfprintf.c:1642)
```

```
==31829==    by 0x4E8CC38: printf (printf.c:33)
==31829==    by 0x400513: main (example5.c:10)
==31829==  Uninitialised value was created by a stack allocation
==31829==    at 0x4004F0: main (example5.c:7)
==31829==
==31829== 3 errors in context 4 of 4:
==31829== Conditional jump or move depends on
                uninitialised value(s)
==31829==    at 0x4E864B2: vfprintf (vfprintf.c:1642)
==31829==    by 0x4E8CC38: printf (printf.c:33)
==31829==    by 0x400513: main (example5.c:10)
==31829==  Uninitialised value was created by a stack allocation
==31829==    at 0x4004F0: main (example5.c:7)
==31829==
==31829== ERROR SUMMARY: 6 errors from 4 contexts
                (suppressed: 0 from 0)
```

This is, indeed, an extremely wordy warning that we didn't initialize the character array maybe_a_string on the stack. At this point you've seen examples of the different kinds of messages you'll get as an output summary from Memcheck. You can, at your leisure, use Memcheck to check for bugs in a real program where they might not be so easy to find.

3.7 Where to go from here

You should, at this point be ready to debug a real program using Valgrind and your existing debugging skills. I've provided a buggy binary tree implementation with test cases that prove it doesn't work if you would like to try debugging that, or you're free to use Valgrind on your own C/C++/Ada program and see if you can find bugs there. Source code for the examples we've worked through is available at the end of this document, but it'll be far easier to retrieve the source from my home directory as we did in chapter 1.

.1 Source Code for Examples

.1.1 example1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "LinkedList.h"




node_t* make_list(int argc, char *argv[]) {
  node_t* linked_list = NULL;
  node_t* current_node = NULL;

  for(int i = 1 ; i < argc ; ++i) {
    if(NULL == linked_list){
      linked_list = new_node(strdup(argv[i]));
      current_node = linked_list;
    }
    else {
      current_node->next = new_node(strdup(argv[i]));
      current_node = current_node->next;
    }

  }

  return linked_list;
}

void free_list(node_t *list) {
  node_t *current_node;
  while(NULL != list) {
    current_node = list;
    list = list->next;
    delete_node(current_node);
  }

}


void print_args(int argc, char *argv[]) {
  node_t* linked_list = NULL;
  node_t* current_node = NULL;

  linked_list = make_list(argc, argv);

  current_node = linked_list;
  while(NULL != current_node) {
    printf("%s\n", (char *) current_node->data);
    current_node = current_node-> next;
  }

  free_list(linked_list);

}


int main(int argc, char *argv[]) {

  print_args(argc, argv);

  return EXIT_SUCCESS;

}
```

.1.2 example2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "LinkedList.h"



node_t* make_list(int argc, char *argv[]) {
  node_t* linked_list = NULL;
  node_t* current_node = NULL;

  for(int i = 1 ; i < argc ; ++i) {
    if(NULL == linked_list){
      linked_list = new_node(strdup(argv[i]));
      current_node = linked_list;
    }
    else {
      current_node->next = new_node(strdup(argv[i]));
      current_node = current_node->next;
    }

  }

  return linked_list;
}

void free_list(node_t *list) {

  while(NULL != list) {
    free(list->data);
    delete_node(list);
    list = list->next;
  }

}


void print_args(int argc, char *argv[]) {
  node_t* linked_list = NULL;
  node_t* current_node = NULL;

  linked_list = make_list(argc, argv);

  current_node = linked_list;
  while(NULL != current_node) {
    printf("%s\n", (char *) current_node->data);
    current_node = current_node-> next;
  }

  free_list(linked_list);


}


int main(int argc, char *argv[]) {

  print_args(argc, argv);

  return EXIT_SUCCESS;

}
```

.1.3 example3.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>

static const ssize_t string_length = 12;

int main() {
  char *maybe_a_string = malloc(string_length);
  ssize_t bytes_read;

  assert(NULL != maybe_a_string);

  memset(maybe_a_string, '\0', string_length);

  bytes_read = read(STDIN_FILENO,
                    maybe_a_string, string_length+1);

  if(0 <= bytes_read) {
    printf("You entered: %s\n", maybe_a_string);
  }

  free(maybe_a_string);
  return EXIT_SUCCESS;

}
```

.1.4 example4.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>



int main() {
  char *a_string = NULL;
  size_t buffsize = 0;
  ssize_t string_length = 0;

  fprintf(stderr, "Please enter a string:");

  string_length = getline(&a_string, &buffsize, stdin);

  assert(0 <= string_length);

  memcpy(a_string, a_string+5, strlen(a_string));

  fprintf(stderr, "Your modified string is: %s", a_string);
  free(a_string);

  return EXIT_SUCCESS;

}
```

.1.5 example5.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>

int main() {
  char maybe_a_string[12];

  printf("Your string is: %s\n", maybe_a_string);

  return EXIT_SUCCESS;

}
```

.1.6 LinkedList.c

```
#include <assert.h>
#include <stdlib.h>
#include "LinkedList.h"


node_t* new_node(void *data) {
  node_t *new = (node_t *) malloc(sizeof(node_t));
  assert(NULL != new);
  new->next = NULL;
  new->data = data;

  return new;
}

void delete_node(node_t *node) {
  free(node);
}
```

.1.7 LinkedList.h

```
#ifndef _LINKEDLIST_H_
#define _LINKEDLIST_H_
typedef struct node {
  void *data;
  void *next;
} node_t;

node_t* new_node(void *data);
void delete_node(node_t *node);
#endif /*_LINKEDLIST_H_*/
```

.1.8 stacktrace.c

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

void print_string(char** string_ptr) {
  int i = 0;
  while('\0' != (*string_ptr)[i]) {
    printf("%c", (*string_ptr)[i++]);
  }
  printf("\n");
  print_string(string_ptr+1);
}

int main(int argc, char* argv[]) {
  if(argc > 1) {
    print_string(argv+1);
  }
  return EXIT_SUCCESS;
}
```

Bibliography

[1] Figure 2.1 found at http://stackoverflow.com/questions/17900910/why-addresses-of-elements-in-the-stack-are-reversed-in-ubuntu64

[2] Figure 2.2 found at http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/067/6701/6701f1.jpg