How to Use Valgrind

1 What is Valgrind? How do I run it?

Valgrind is a tool to help you catch memory related errors. In particular, it can catch memory leaks (recall that a memory leak is a location in memory that you allocated through `malloc` but did not de-allocate through a call to `free`), and illegal memory dereferences. As mentioned in class, illegal memory dereferences may cause a segmentation fault or they may not produce any immediately visible errors. Valgrind is there to help you find the root cause of all of these errors.

Important: Valgrind is a tool designed to work with C programs. If you try to use it on a C0 program, you may get nonsensical error messages.

Important: For Valgrind to give helpful information, you need to compile your C program with the `-g` option.

In order to run valgrind on an executable file, all you have to do is add the word `valgrind` in front of it. For example, if you have an executable file that you would ordinarily run using the first line below, then to run it with valgrind you would just have to use the second line instead

```
% ./a.out
% valgrind ./a.out
```

Assuming that your code has no memory errors or memory leaks, valgrind will print out something similar to what is depicted below. If your code has memory errors or memory leaks, then those will appear along with the below output.

```
==30173== Memcheck, a memory error detector
==30173== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30173== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==30173== Command: ./a.out
==30173==
==30173==
==30173== HEAP SUMMARY:
==30173==     in use at exit: 0 bytes in 0 blocks
==30173==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==30173==
==30173== All heap blocks were freed -- no leaks are possible
==30173==
==30173== For counts of detected and suppressed errors, rerun with: -v
==30173== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The important things to look at to know if your code is correct are **HEAP SUMMARY** and **ERROR SUMMARY**. In the heap summary, assuming correct code, you should always see the message `in use at exit: 0 bytes in 0 blocks`. In the error summary you should always see the message `ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)`.

If you do have errors, you should expect them to appear between the line specifying the command (in the above output, `Command: ./a.out`) and the start of the heap summary. If your code has memory leaks, details about those memory leaks will appear below the heap summary (in particular, see Section Memory Leaks below).

2 When should I use Valgrind?

You should run Valgrind when:

- You get undefined behavior (running program gives unexpected results, running multiple times gives different results)
- You get a segmentation fault
- You see scary memory output after running a C executable
- You are about to submit C code that should be leak and error-free
- Always!

3 Backtraces

One of the most important tools that valgrind uses to communicate to you, the user, is a "backtrace". A backtrace is essentially a listing of all of the function calls made in order to get to the specific line where an error occurred. To understand how it works, let's take a look at an example:

```
==2797==    at 0x400555: f1 (example_file.c:7)
==2797==    by 0x400572: f2 (example_file.c:12)
==9892==    by 0x40053E: main (main_file.c:4)
```

This backtrace tells us that:

- The error occurred at line 7 of the file `example_file.c` inside the function `f1` ...
- ... which was called on line 12 of `example_file.c` inside the function `f2` ...
- ... which was called on line 4 of `main_file.c` inside the function `main`

Note that your backtraces will always have the main function at the bottom since every C program starts in the main function.

Important: The rest of this guide will assume an understanding of how to read a backtrace. If the above confused you, then **now** is a good time to post a question asking for clarification.

The rest of this guide will focus on specific types of errors that valgrind can detect. For each error, we will start by providing a sample program, and then show the valgrind output for that program. Then, from the valgrind output we will deduce what the error is in the original program. We leave it as an exercise for the reader to try to fix the program.

4 Memory Leaks

Memory Leak Sample Program:

```
#include <stdlib.h>

int *f1() {
    int *ip = malloc(sizeof(int));

    *ip = 3;
    return ip;
}

int f2() {
    int *internal = f1();

    return *internal;
}

int main() {
    int i = f2();
    return i;
}
```

If we then compile and run this using valgrind, we get the following error:

```
==14132== HEAP SUMMARY:
==14132==     in use at exit: 4 bytes in 1 blocks
==14132==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==14132==
==14132== LEAK SUMMARY:
==14132==    definitely lost: 4 bytes in 1 blocks
==14132==    indirectly lost: 0 bytes in 0 blocks
==14132==      possibly lost: 0 bytes in 0 blocks
==14132==    still reachable: 0 bytes in 0 blocks
==14132==         suppressed: 0 bytes in 0 blocks
==14132== Rerun with --leak-check=full to see details of leaked memory
```

From the leak summary, it is clear that we have a memory leak. Specifically we have lost "4 bytes in 1 block". This means that we made one call to malloc for a chunk of memory 4 bytes big. Re-running as suggested in the last line (e.g. `valgrind --leak-check=full ./a.out`) we get the following additional information:

```
==3601== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3601==    at 0x4C29E63: malloc (vg_replace_malloc.c:309)
==3601==    by 0x40053E: f1 (example_file.c:4)
==3601==    by 0x400572: f2 (example_file.c:12)
==3601==    by 0x400590: main (example_file.c:18)
```

What this chunk of the output is telling is is that:

- We asked for 4 bytes of memory (which is enough to store one integer!)
- These 4 bytes of memory were allocated by `malloc`, which was called on line 4 of our file (inside `f1`).
- `f1` itself was called by `f2` (on line 12)

Looking at our file, we see that line 4 indeed allocates some memory for a pointer, which it then returns. Looking at the calling function, we then see that `f2` forgets to free `internal` before returning to main. There's our memory leak!

5 Invalid Memory Access

Bad Memory Access Sample Program:

```
1  #include <stdlib.h>
2
3  int *f1() {
4      int *ip = malloc(sizeof(int));
5
6      *ip = 3;
7      return ip;
8  }
9
10 int f2() {
11     int *internal = f1();
12
13     int left = internal[0];
14     int right = internal[2];
15     free(internal);
16
17     return left + right / 2;
18 }
19
20 int main() {
21     int i = f2();
22     return i;
23 }
```

If we then compile and run this using valgrind, we get the following error:

```
==12751== Invalid read of size 4
==12751==    at 0x4005C6: f2 (example_file.c:14)
==12751==    by 0x4005FE: main (example_file.c:21)
==12751==  Address 0x5205048 is 4 bytes after a block of size 4 alloc'd
==12751==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==12751==    by 0x40058E: f1 (example_file.c:4)
==12751==    by 0x4005B4: f2 (example_file.c:11)
==12751==    by 0x4005FE: main (example_file.c:21)
```

To break down what this message is saying, let us first look at the first 3 lines. We see that the error itself is that we are reading "a 4 byte thing" (e.g. *Invalid read of size 4*). Furthermore, it is letting us know that the line that causes this invalid read is line 14 of our program.

The second half of the error message (e.g. the next 5 lines) is providing some extra, useful information. First, it tells us that the location we are trying to read from is "4 bytes after a block of size 4 alloc'd". This means that the start of where we are reading is 4 bytes past the end of an allocated block of size 4. Then, the error message points out where this block was allocated (specifically, on line 4, in `f1`).

Looking at our code, we see that the error makes sense. When we access `internal[2]`, we are reading an int (aka "a 4 byte thing") from a location 8 bytes past the start of the pointer internal. Looking at the allocation line, we see that the amount of space that was allocated was only 4 bytes (e.g. **`sizeof(int)`**) meaning that the location we are reading from is 4 bytes past the end of the block allocated for internal! In summary our issue is reading past the end of an "array"!

6 Invalid Free

Invalid Free Sample Program:

```
1  #include <stdlib.h>
2
3  int *f1() {
4      int *ip = malloc(sizeof(int));
5
6      *ip = 3;
7      return ip;
8  }
9
10 int f2() {
11     int *internal = f1();
12     void *other = (void*)internal;
13
14     int result = *internal;
15     int *result2 = &result;
16
17     free(internal);
18     free(other);
19     free(result2);
20
21     return result;
22 }
23
24 int main() {
25     int i = f2();
26     return i;
27 }
```

If we then compile and run this using valgrind, we get the following errors:

```
==31964== Invalid free() / delete / delete[] / realloc()
```

```
==31964==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==31964==    by 0x4005E9: f2 (example_file.c:18)
==31964==    by 0x40060C: main (example_file.c:25)
==31964==  Address 0x5205040 is 0 bytes inside a block of size 4 free'd
==31964==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==31964==    by 0x4005DD: f2 (example_file.c:17)
==31964==    by 0x40060C: main (example_file.c:25)
==31964==  Block was alloc'd at
==31964==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==31964==    by 0x40058E: f1 (example_file.c:4)
==31964==    by 0x4005B4: f2 (example_file.c:11)
==31964==    by 0x40060C: main (example_file.c:25)
==31964==
==31964== Invalid free() / delete / delete[] / realloc()
==31964==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==31964==    by 0x4005F5: f2 (example_file.c:19)
==31964==    by 0x40060C: main (example_file.c:25)
==31964==  Address 0x1ffefff474 is on thread 1's stack
==31964==  in frame #1, created by f2 (example_file.c:10)
```

Let's take each error separately. The first error tells us that the call to free on line 18 is invalid. Looking at the other parts of the error message, we see that the address that we attempted to free was "0 bytes inside a block of size 4 free'd". In other words, we are trying to free a pointer twice! The previous `free` can be found by examining the backtrace underneath, and we find that we had freed it just the previous line! Furthermore, we are informed (should this be necessary) that the block that we are attempting to free twice was allocated on line 4.

Once again, this makes sense. First, we know that `other` points to the same thing to `internal` points to (by line 11). Secondly we know that `free` frees the memory region pointed to by the thing we passed in. Thus, passing in other, causes free to (incorrectly) attempt to free the same memory region twice!

The second error tells us that the call to free on line 19 is also invalid. Looking at the last two lines of the error message, we see that the address we are trying to free does not point to a location in memory we allocated, but rather to a location on the stack. In other words, we are trying to free a pointer to a local variable. The last line of the error message also points out that this local variable that we are pointing to is likely a local variable found in `f2` (which starts on line 10).

Examining our code, we see that the thing we are freeing (`result2` is indeed a pointer to a local variable). In fact, it is a pointer to `result`!

7 Uninitialized Values

Uninitialized Values Sample Program:

```
1 #include <stdlib.h>
2
3 int *f1() {
4     int *ip = malloc(sizeof(int));
5
6     return ip;
```

```
7  }
8
9  int f2() {
10     int *internal = f1();
11     int other = 3;
12
13     if(*internal < 5) {
14         other = *internal;
15     }
16
17     free(internal);
18
19     return other;
20 }
21
22 int main() {
23     int i = f2();
24     return i;
25 }
```

If we then compile and run this using valgrind, we get the following errors:

```
==27751== Conditional jump or move depends on uninitialised value(s)
==27751==    at 0x4005BF: f2 (example_file.c:13)
==27751==    by 0x4005EC: main (example_file.c:23)
==27751==
==27751== Syscall param exit_group(status) contains uninitialised byte(s)
==27751==    at 0x4EFCC09: _Exit (in /usr/lib64/libc-2.17.so)
==27751==    by 0x4E70CAA: __run_exit_handlers (in /usr/lib64/libc-2.17.so)
==27751==    by 0x4E70D36: exit (in /usr/lib64/libc-2.17.so)
==27751==    by 0x4E5955B: (below main) (in /usr/lib64/libc-2.17.so)
```

The first error is telling us that we have a "conditional jump or move" that depends on an uninitialized value. A "conditional jump or move" is just any if statement or while loop guard or any other conditional that may appear in your code. Looking at line 13 (which is the line at the start of the backtrace), we see that this conditional jump is `if(*internal < 3)`.

Unfortunately valgrind does not by default tell us any information about the value that was uninitialized. However, we can use the `--track-origins=yes` flag (e.g. by running `valgrind --track-origins=yes` to get the following extra information:

```
==19257==  Uninitialised value was created by a heap allocation
==19257==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==19257==    by 0x40058E: f1 (example_file.c:4)
==19257==    by 0x4005AA: f2 (example_file.c:10)
==19257==    by 0x4005EC: main (example_file.c:23)
```

This tells us that the value that was uninitialized was created on line 4 - the allocation of `ip`. Looking in the function, we find that nobody ever initialized the value that ip points to! Remember that in C, malloc will not initialize values for you - you have to make sure that everything is initialized yourself.

The second error seems a lot more confusing, but all it is really saying is that the value that we return from `main` is at least partially uninitialized. Since the value we return from main depends on `*internal`, this makes sense. The backtrace here is not particularly helpful.

8 Bad Permissions

Bad Permissions Sample Program:

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

char *f1() {
    char *sp = "Hello";
    return sp;
}

int main() {
    char *s1 = f1();
    printf("Received String: %s\n", s1);
    s1[0] = 'C';
    printf("Changed String: %s\n", s1);
    return 0;
}
```

When we compile and run the code, we will end up with:

```
Received String: Hello
Segmentation fault (core dumped)
```

That's not great! Let's run with valgrind to get more information about the error:

```
==11199== Process terminating with default action of signal 11 (SIGSEGV)
==11199==  Bad permissions for mapped region at address 0x400620
==11199==    at 0x40056F: main (example_file.c:13)
```

This doesn't give us a lot of information, but it does give us a line number to look at. We can tak a look at line 13 to see that our error comes from running `s1[0] = 'C'`.

So, we're trying to write to a string, but don't have permissions to do so. This suggests that s1 is a **read-only string** - and looking through our example, this aligns with how f1 currently returns a pointer to a string in the DATA section of memory.

A simple fix to this would be using heap-allocated memory in `f1` instead (which we'll need to be sure is freed in the right place!), or copying the string returned by `f1` into a writable array on the stack or in the heap.

9 Dealing with long Valgrind output

Because Valgrind prints out a message every single time a memory error is encountered, the length of the valgrind output can be very long. Even with a medium-sized program with only a couple of

errors, the length of the valgrind output can be substantial. For instance, take this - admittedly silly - program that performs some recursion:

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int *f1() {
5      int *ip = malloc(sizeof(int));
6      return ip;
7  }
8
9  int inner_fn(int *p) {
10     printf("Inner function called with value %i\n", *p);
11     if(*p <= 3) {
12         return *p;
13     }
14     p[1] = p[0] / 2;
15     int *ip = f1();
16     *ip -= p[1] - 1;
17
18     return *p + inner_fn(ip);
19 }
20
21 int main() {
22     int *p = f1();
23     *p = 10;
24     int i = inner_fn(p);
25     return i;
26 }
```

For clarity, the errors here are that `f1` does not initialize its pointer and that `inner_fn` accesses a pointer out of bounds on line 14 and 16. If you run this program, the full valgrind output will be:

```
==4892== Memcheck, a memory error detector
==4892== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4892== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4892== Command: ./a.out
==4892==
Inner function called with value 10
==4892== Invalid write of size 4
==4892==    at 0x4005E7: inner_fn (example_file.c:14)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==  Address 0x5205044 is 0 bytes after a block of size 4 alloc'd
==4892==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==4892==    by 0x40058E: f1 (example_file.c:5)
==4892==    by 0x400644: main (example_file.c:22)
==4892==
==4892== Invalid read of size 4
==4892==    at 0x400605: inner_fn (example_file.c:16)
==4892==    by 0x40065E: main (example_file.c:24)
```

```
==4892==  Address 0x5205044 is 0 bytes after a block of size 4 alloc'd
==4892==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==4892==    by 0x40058E: f1 (example_file.c:5)
==4892==    by 0x400644: main (example_file.c:22)
==4892==
==4892== Conditional jump or move depends on uninitialised value(s)
==4892==    at 0x4E81C5E: vfprintf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E8A4A8: printf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4005BC: inner_fn (example_file.c:10)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==
==4892== Use of uninitialised value of size 8
==4892==    at 0x4E7F32B: _itoa_word (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E835B0: vfprintf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E8A4A8: printf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4005BC: inner_fn (example_file.c:10)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==
==4892== Conditional jump or move depends on uninitialised value(s)
==4892==    at 0x4E7F335: _itoa_word (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E835B0: vfprintf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E8A4A8: printf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4005BC: inner_fn (example_file.c:10)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==
==4892== Conditional jump or move depends on uninitialised value(s)
==4892==    at 0x4E835FF: vfprintf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E8A4A8: printf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4005BC: inner_fn (example_file.c:10)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==
==4892== Conditional jump or move depends on uninitialised value(s)
==4892==    at 0x4E81D2B: vfprintf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E8A4A8: printf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4005BC: inner_fn (example_file.c:10)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==
==4892== Conditional jump or move depends on uninitialised value(s)
==4892==    at 0x4E81DAE: vfprintf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E8A4A8: printf (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4005BC: inner_fn (example_file.c:10)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
```

```
==4892==
Inner function called with value -4
==4892== Conditional jump or move depends on uninitialised value(s)
==4892==    at 0x4005C6: inner_fn (example_file.c:11)
==4892==    by 0x400629: inner_fn (example_file.c:18)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==
==4892== Syscall param exit_group(status) contains uninitialised byte(s)
==4892==    at 0x4EFCC09: _Exit (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E70CAA: __run_exit_handlers (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E70D36: exit (in /usr/lib64/libc-2.17.so)
==4892==    by 0x4E5955B: (below main) (in /usr/lib64/libc-2.17.so)
==4892==
==4892==
==4892== HEAP SUMMARY:
==4892==     in use at exit: 8 bytes in 2 blocks
==4892==   total heap usage: 2 allocs, 0 frees, 8 bytes allocated
==4892==
==4892== LEAK SUMMARY:
==4892==    definitely lost: 8 bytes in 2 blocks
==4892==    indirectly lost: 0 bytes in 0 blocks
==4892==      possibly lost: 0 bytes in 0 blocks
==4892==    still reachable: 0 bytes in 0 blocks
==4892==         suppressed: 0 bytes in 0 blocks
==4892== Rerun with --leak-check=full to see details of leaked memory
==4892==
==4892== Use --track-origins=yes to see where uninitialised values come from
==4892== For lists of detected and suppressed errors, rerun with: -s
==4892== ERROR SUMMARY: 10 errors from 10 contexts (suppressed: 0 from 0)
```

The best strategy to approach a long series of error messages like this is to try to attack only the first error message. In this case, that is

```
==4892== Invalid write of size 4
==4892==    at 0x4005E7: inner_fn (example_file.c:14)
==4892==    by 0x40065E: main (example_file.c:24)
==4892==  Address 0x5205044 is 0 bytes after a block of size 4 alloc'd
==4892==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==4892==    by 0x40058E: f1 (example_file.c:5)
==4892==    by 0x400644: main (example_file.c:22)
```

Once you fix the first error message, generally speaking the number of error messages reduces significantly.