# Notes from CPSC 223 Valgrind Sessions

Sessions held February 17 and 18, 2020 by Emma Pierce-Hoffman

## What is Valgrind?

Valgrind is a tool that helps find mistakes in memory management, like memory leaks, reading/writing past the end of an allocated block of memory, or use of an uninitialized value.

## Why should I care about Valgrind?

Valgrind can help you find many bugs in your code related to memory allocation. Some of these bugs will cause your program to have the wrong output, or crash from a segmentation fault, while others cause only intermittent issues (ie. printing of garbage values in uninitialized arrays) or are not detectable. Valgrind's descriptive output can be extremely helpful in locating and understanding these bugs.

Also, the public and private tests will run Valgrind on your program, and you will lose points if Valgrind finds errors or memory leaks, so you should make testing with Valgrind part of your workflow.

## Other resources on Valgrind

A number of helpful resources are available on the class web page, including this one: https://zoo.cs.yale.edu/classes/cs223/current/Valgrind. This resource talks about Valgrind in general and then gives examples of common errors and explanations of what they mean.

## How do I run Valgrind?

Valgrind is already installed on the zoo, so after you compile your program (for example, with source code named program.c and executable named program), simply run:

```
valgrind --leak-check=full ./program [any additional command line arguments]
```

For HW3, that might look something like this:

```
echo "aaabb" | valgrind --leak-check=full ./strwrs -Qn aa a abb aab
```

## Example program with Valgrind errors

The rest of this lesson will use an example C program that contains memory allocation errors. To follow along interactively, copy and paste the example (buggy) code into your own C file on the zoo, then correct each error as we go through the explanations. It will be very satisfying when you end up with a bug-free program!

### Example code:

```c
#define _GNU_SOURCE // Recall: define GNU_SOURCE necessary for strdup
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
      // Make a character array of all the lowercase letters in the
alphabet
      // Then print it as a string
      char *alphabet;
      alphabet = malloc(sizeof(char) * 27);
      for(int i = 0; i < 26; i++) {
            alphabet[i] = (char) i + 97;
      }


      printf(alphabet);
      putchar('\n');

      // Copy 1st 10 letters over to first10 and print
      char *first10 = malloc(11);
      strncpy(first10, alphabet, 10);

      printf(first10);
      putchar('\n');

      // Make a copy of that and print it
      char *first10copy = strdup(first10);
      printf("Copy: %s\n", first10copy); // it will print successfully -
null-terminated
```

```
	free(alphabet);
	free(first10);

	// Make an integer array of digits 0-9
	// Then print the odd ones
	int *digits;
	digits = malloc(sizeof(int) * 9);
	for(int j = 0; j < 9; j++) {
		digits[j] = j;
	}

	for(int k = 0; k <= 9; k++) {
		if(digits[k]%2 != 0) {
			printf("Odd number! %d\n", digits[k]);
		}
	}

	free(alphabet);

}
```

Desired/Correct output:

```
abcdefghijklmnopqrstuvwxyz
abcdefghij
Copy: abcdefghij
Odd number! 1
Odd number! 3
Odd number! 5
Odd number! 7
Odd number! 9
```

Desired/Correct output with valgrind:

```
elp34@~/223ula/Valgrind$ valgrind --leak-check=full ./correct
==2305== Memcheck, a memory error detector
==2305== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2305== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
```

```
==2305== Command: ./correct
==2305==
abcdefghijklmnopqrstuvwxyz
abcdefghij
Copy: abcdefghij
Odd number! 1
Odd number! 3
Odd number! 5
Odd number! 7
Odd number! 9
==2305==
==2305== HEAP SUMMARY:
==2305==     in use at exit: 0 bytes in 0 blocks
==2305==   total heap usage: 5 allocs, 5 frees, 1,113 bytes allocated
==2305==
==2305== All heap blocks were freed -- no leaks are possible
==2305==
==2305== For lists of detected and suppressed errors, rerun with: -s
==2305== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Actual output (initially):

```
elp34@~/223ula/Valgrind$ ./example
abcdefghijklmnopqrstuvwxyz
abcdefghij
Copy: abcdefghij
Odd number! 1
Odd number! 3
Odd number! 5
Odd number! 7
```

Actual Valgrind output (initially):

```
elp34@~/223ula/Valgrind$ valgrind --leak-check=full ./example
==7896== Memcheck, a memory error detector
==7896== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7896== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==7896== Command: ./example
==7896==
```

```
==7896== Conditional jump or move depends on uninitialised value(s)
==7896==    at 0x48402DE: strchrnul (vg_replace_strmem.c:1396)
==7896==    by 0x48EC764: __find_specmb (printf-parse.h:108)
==7896==    by 0x48EC764: __vfprintf_internal (vfprintf-internal.c:1322)
==7896==    by 0x48D938E: printf (printf.c:33)
==7896==    by 0x4011C6: main (example.c:17)
==7896==
==7896== Conditional jump or move depends on uninitialised value(s)
==7896==    at 0x48EC7F8: __vfprintf_internal (vfprintf-internal.c:1334)
==7896==    by 0x48D938E: printf (printf.c:33)
==7896==    by 0x4011C6: main (example.c:17)
==7896==
abcdefghijklmnopqrstuvwxyz
==7896== Conditional jump or move depends on uninitialised value(s)
==7896==    at 0x48402DE: strchrnul (vg_replace_strmem.c:1396)
==7896==    by 0x48EC764: __find_specmb (printf-parse.h:108)
==7896==    by 0x48EC764: __vfprintf_internal (vfprintf-internal.c:1322)
==7896==    by 0x48D938E: printf (printf.c:33)
==7896==    by 0x401207: main (example.c:24)
==7896==
==7896== Conditional jump or move depends on uninitialised value(s)
==7896==    at 0x48EC7F8: __vfprintf_internal (vfprintf-internal.c:1334)
==7896==    by 0x48D938E: printf (printf.c:33)
==7896==    by 0x401207: main (example.c:24)
==7896==
abcdefghij
==7896== Conditional jump or move depends on uninitialised value(s)
==7896==    at 0x483BBF8: strlen (vg_replace_strmem.c:461)
==7896==    by 0x49106C2: strdup (strdup.c:41)
==7896==    by 0x40121D: main (example.c:28)
==7896==
==7896== Conditional jump or move depends on uninitialised value(s)
==7896==    at 0x483BBF8: strlen (vg_replace_strmem.c:461)
==7896==    by 0x48EEA1D: __vfprintf_internal (vfprintf-internal.c:1645)
==7896==    by 0x48D938E: printf (printf.c:33)
==7896==    by 0x401237: main (example.c:29)
==7896==
Copy: abcdefghij
Odd number! 1
Odd number! 3
Odd number! 5
Odd number! 7
```

```
==7896== Invalid read of size 4
==7896==    at 0x4012A7: main (example.c:43)
==7896==  Address 0x4a4f5a4 is 0 bytes after a block of size 36 alloc'd
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x401259: main (example.c:37)
==7896==
==7896== Invalid free() / delete / delete[] / realloc()
==7896==    at 0x4839A0C: free (vg_replace_malloc.c:540)
==7896==    by 0x4012EC: main (example.c:48)
==7896==  Address 0x4a4f040 is 0 bytes inside a block of size 27 free'd
==7896==    at 0x4839A0C: free (vg_replace_malloc.c:540)
==7896==    by 0x401243: main (example.c:31)
==7896==  Block was alloc'd at
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x401187: main (example.c:11)
==7896==
==7896==
==7896== HEAP SUMMARY:
==7896==     in use at exit: 47 bytes in 2 blocks
==7896==   total heap usage: 5 allocs, 4 frees, 1,109 bytes allocated
==7896==
==7896== 11 bytes in 1 blocks are definitely lost in loss record 1 of 2
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x49106CE: strdup (strdup.c:42)
==7896==    by 0x40121D: main (example.c:28)
==7896==
==7896== 36 bytes in 1 blocks are definitely lost in loss record 2 of 2
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x401259: main (example.c:37)
==7896==
==7896== LEAK SUMMARY:
==7896==    definitely lost: 47 bytes in 2 blocks
==7896==    indirectly lost: 0 bytes in 0 blocks
==7896==      possibly lost: 0 bytes in 0 blocks
==7896==    still reachable: 0 bytes in 0 blocks
==7896==         suppressed: 0 bytes in 0 blocks
==7896==
==7896== Use --track-origins=yes to see where uninitialised values come
from
==7896== For lists of detected and suppressed errors, rerun with: -s
==7896== ERROR SUMMARY: 10 errors from 10 contexts (suppressed: 0 from 0)
```

## Approach

Wow, that's a lot of errors! It's way too many to address all at once - instead, we'll go one by one, starting from the top (the errors are listed in order of appearance in the program). Sometimes correcting one error will actually fix multiple errors!

## Error #1:

```
==7573== Conditional jump or move depends on uninitialised value(s)
==7573==    at 0x48402DE: strchrnul (vg_replace_strmem.c:1396)
==7573==    by 0x48EC764: __find_specmb (printf-parse.h:108)
==7573==    by 0x48EC764: __vfprintf_internal (vfprintf-internal.c:1322)
==7573==    by 0x48D938E: printf (printf.c:33)
==7573==    by 0x4011C6: main (example.c:17)
```

The first line of the error message (here, "Conditional jump or move depends on uninitialised value(s)) is the category of error detected by Valgrind. What follows is a stack trace, starting at the bottom from main and going up through the nested function calls until the the function where the error was detected.

Here, we see that line 17 of main calls the function `printf`, which then calls several internal functions, and the error is detected during one of those internal functions. We don't need to worry about the internal implementation of `printf`, but importantly, this tells us where to find our error -- line 17 of example.c!

Line 17 attempts to `printf(alphabet)`, but something is going wrong: printf() expects a null-terminated string as an argument, but `alphabet` is not null-terminated! Printf() keeps reading past 'z' looking for the null terminator and encounters an uninitialized value at index 26 of the array. To fix the error, add the following line on line 16 before the call to `printf(alphabet)`:

```
alphabet[26] = '\0';
```

Save the code, re-compile, and re-run with valgrind, and you will see that multiple error messages disappear, and the full alphabet string prints without a problem.

Note that this fix works because the original allocation of space for the `alphabet` string allocates space for 27 chars: 26 letters + 1 null terminator. If space for only 26 characters had been allocated, then adding the line above would have caused an Invalid Write error that looks like this (try it for yourself by changing the 27 to 26 on line 11!):

```
==6008== Invalid write of size 1
```

```
==6008==    at 0x4011BE: main (example.c:16)
==6008==  Address 0x4a4f05a is 0 bytes after a block of size 26 alloc'd
==6008==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==6008==    by 0x401187: main (example.c:11)
```

An Invalid Write means that the program tries to write to space that is outside the block allocated for `alphabet`. The valgrind output specifies that the invalid write occurs on line 16, where we added the code to place the null terminator at `alphabet[26]`. In case you can't figure out which array is the problem, it also tells you that the error pertains to the block of space malloc'd on line 11-- where space for `alphabet` is allocated. Put simply, you can't access index 26 when you've only allocated space for 0-25.

After reverting the code to allocate space for 27 chars for `alphabet`, saving, re-compiling, and re-running the code through Valgrind, let's move on to the next error.

## Error #2:

```
==7573== Conditional jump or move depends on uninitialised value(s)
==7573==    at 0x48402DE: strchrnul (vg_replace_strmem.c:1396)
==7573==    by 0x48EC764: __find_specmb (printf-parse.h:108)
==7573==    by 0x48EC764: __vfprintf_internal (vfprintf-internal.c:1322)
==7573==    by 0x48D938E: printf (printf.c:33)
==7573==    by 0x401207: main (example.c:24)
```

Questions to try to answer based on the Valgrind output for Error #2:

1. What type of error is this?
2. On what line does it occur in example.c?
3. What function call causes the error?

Answers:

1. Conditional jump or move depends on uninitialized value(s)
2. Line 24
3. Printf (specifically, when we look at line 24 of example.c, `printf(first10)`)

It's a very similar error to Error #1! Again, the string passed to `printf()` should be null-terminated but it isn't. Library functions like `strncpy()`, used here to fill in the character array `first10`, may not add a null terminator to the end of a string automatically. It's important to read the documentation carefully.

To fix the error, add the following line on line 23 before `printf(first10)`:

```
first10[10] = '\0';
```

Save, re-compile, and notice that MANY errors disappear. This is because there were previously similar errors related to creating and printing `first10copy`, because `strdup()` and `printf()` both expect to null-terminated strings as arguments. When `strdup()` receives a null-terminated string as an argument, it does copy over the null-terminator in the new string, so those errors are fixed.

## Error #3

```
==7896== Invalid read of size 4
==7896==    at 0x4012A7: main (example.c:43)
==7896==  Address 0x4a4f5a4 is 0 bytes after a block of size 36 alloc'd
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x401259: main (example.c:37)
```

This is a new type of error: Invalid Read. It means the program is trying to read space that is past the end of the space allocated for the digits array.

From the Valgrind output:
On what line of example.c is the invalid write occurring? => Line 43
On what line of example.c was the array in question allocated? => Line 37

Looking at example.c, we see that we allocate space for 9 integers in digits on line 37, but then we try to read at index 9, which is past the end of the array, on line 43. Why is the invalid read of "size 4"? The unit for size is bytes (1 char takes up 1 byte = 8 bits). We are trying to read one integer beyond the end of the array, and one integer (on the zoo) is 32 bits = 4 bytes.

How can we fix this? Keeping in mind that the desired behavior is to populate the array `digits` with the numbers 0-9 and then go through the `digits` array and print out the odd numbers, that means we need space for 10 integers in the array. But we only allocated space for 9 integers and initialized 0-8 in the first for loop. That explains why the final "Odd number! 9" was not being printed.

So, we need to allocate more space and initialize numbers 0-9 in the `digits` array. We can do this by 1) changing line 37 to allocate space for 10 ints:

```
digits = malloc(sizeof(int) * 10);
```

And 2) changing line 38 to initialize all 10 digits:

```
for(int j = 0; j <= 9; j++) {
```

If we just did step 2 without allocating more space, what would happen? => Invalid Write of size 4 on line 38!

Fix it, save it, re-compile it, and re-run it with Valgrind.

## Error #4

```
==7896== Invalid free() / delete / delete[] / realloc()
==7896==    at 0x4839A0C: free (vg_replace_malloc.c:540)
==7896==    by 0x4012EC: main (example.c:48)
==7896==  Address 0x4a4f040 is 0 bytes inside a block of size 27 free'd
==7896==    at 0x4839A0C: free (vg_replace_malloc.c:540)
==7896==    by 0x401243: main (example.c:31)
==7896==  Block was alloc'd at
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x401187: main (example.c:11)
```

The first segment of this error tells us that it is an invalid free on line 48 of main (valgrind identifies it as an invalid free OR delete OR realloc, but then the stack trace below says the error occurs in the call to `free()`). Line 48 contains a call to `free(alphabet)`.

The second segment says that the address we are trying to free was already freed on line 31! That means this error is caused by a double free: we already freed `alphabet`, so the `alphabet` pointer no longer points to any allocated memory, and we can't free it again.

The third segment shows us where the block of memory that the error pertains to was allocated: line 11, where space for `alphabet` is malloc'd.

To fix this problem, delete the second call to `free(alphabet)` on line 48.

Save, re-compile, re-valgrind and move on to…

## Error #5

```
==7896== 11 bytes in 1 blocks are definitely lost in loss record 1 of 2
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x49106CE: strdup (strdup.c:42)
==7896==    by 0x40121D: main (example.c:28)
```

This type of error is a memory leak: 11 bytes of memory are "definitely lost" because they aren't freed by the time the pointer to the array is lost, which, here, is when the program exits. Every

time you allocate memory, you must free it a) before you lose track of the pointer to the beginning of that memory (ie. if it is local to a function and not returned by the function, then before the function returns) or b) before the program exits, whichever comes first.

The Valgrind output tells us that the block of memory that is lost was allocated on line 28 of main:

```
char *first10copy = strdup(first10);
```

But line 28 doesn't contain a call to `malloc()`, so what's going on? `strdup()` actually mallocs space for you, so you have to free memory allocated by `strdup()` (or `getline()`, etc.) just like you would free memory allocated by `malloc()`.

To fix the problem, call `free(first10copy)` at some point after you are done using the `first10copy` array-- for example, on line 30 or 48.

Save, compile, valgrind, and move on to the final error…

## Error #6

```
==7896== 40 bytes in 1 blocks are definitely lost in loss record 2 of 2
==7896==    at 0x483880B: malloc (vg_replace_malloc.c:309)
==7896==    by 0x401259: main (example.c:37)
```

(If you haven't been following along, with the original code this error had 36 bytes lost, but fixing an earlier issue changed it to 40.)

This one is very similar to Example #5, so you ask yourself (figure out each question before moving to the next):

1. What is the problem?
2. On what line of example.c was the memory that was lost allocated?
3. What function call allocated the lost memory?
4. What is the name of the array that was lost? (Referring to example.c)
5. Why does it say there are 40 bytes lost?
6. How do you fix the error?

Answers:

1. It is a memory leak: the block of memory was not freed before the program exited.
2. 37
3. `malloc()`
4. `digits`
5. The `digits` array allocates space for 10 integers (now that we've changed it from 9!), and each integer is 4 bytes => 40 bytes total.

6. Add a line to `free(digits)` on line 49.

Save, re-compile, and re-run and voilà! All the bugs are gone.

## Corrected example code:

The full corrected source code is provided for reference:

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

int main()
{
	// Make a character array of all the lowercase letters in the
alphabet
	// Then print it as a string
	char *alphabet;
	alphabet = malloc(sizeof(char) * 27);
	for(int i = 0; i < 26; i++) {
		alphabet[i] = (char) i + 97;
	}

	alphabet[26] = '\0'; // add in without changing malloc to 27
characters --> invalid write
	printf(alphabet);
	putchar('\n');

	// Copy 1st 10 letters over to first10 and print
	char *first10 = malloc(11);
	strncpy(first10, alphabet, 10);
	first10[10] = '\0';
	printf(first10);
	putchar('\n');

	// Make a copy of that and print it
	char *first10copy = strdup(first10);
	printf("Copy: %s\n", first10copy); // it will print successfully -
null-terminated
```

```
	free(alphabet);
	free(first10);

	// Make an integer array of digits 0-9
	// Then print the odd ones
	int *digits;
	digits = malloc(sizeof(int) * 10);
	for(int j = 0; j < 10; j++) {
		digits[j] = j;
	}

	for(int k = 0; k < 10; k++) {
		if(digits[k]%2 != 0) {
			printf("Odd number! %d\n", digits[k]);
		}
	}

	free(first10copy); // must free things from strdup!
	free(digits);
}
```