# Dynamic Allocation Issues



Valgrind's Memcheck tool is quite useful for finding problems related to dynamic allocation—whether **malloc** and **free** in C, or, if you go on to learn about C++, **new**/**new[]**, and **delete**/**delete[]**. As with other regions of memory, Memcheck will explicitly track which addresses are valid and which are not. It also tracks exactly what pointers were returned by **malloc**, **new**, and **new[]**, and places some invalid space on each side of the allocated block.

From all this information, Memcheck can report a wide variety of problems. First, if an access goes just past the end of a dynamically allocated array, Memcheck can detect this problem. Second, Memcheck can detect double freeing pointers, freeing the incorrect pointer, and mismatches between allocations and deallocations (*e.g.,* **delete**ingmemory allocated with **malloc**, or mixing up **delete[]** with **delete**, which would be an error for a C++ programmer). For example, suppose we wrote the following (obviously buggy) code:

```
1

2

int * ptr = malloc(sizeof(int));

ptr[1] = 3;
```

If we run this inside of Memcheck, it reports the following error:

```
1

2

3

4

5

==5465== Invalid write of size 4

==5465==    at 0x40054B: main (outOfBounds.c:8)
```

```
==5465==  Address 0x51fc044 is 0 bytes after a block of size 4 alloc'd

==5465==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)

==5465==    by 0x40053E: main (outOfBounds.c:7)
```

The error first tells us the problem (we made an invalid write of 4 bytes), with a stack trace indicating where that invalid write happened (in main, on line 8 of the file outOfBounds.c). The second part tells us what invalid address our program tried to access and the nearest valid location. In this case, it reports it as "0 bytes after" (meaning in the first invalid byte past a valid region) "a block of size 4" (meaning how much space was allocated into that valid region). Memcheck then reports where that valid region of memory was allocated (in main on line 7, by calling malloc).

Note that if we recently freed a block of memory, Memcheck will report proximity to that block of memory, even though it is no longer valid. For example, if we write:

```
1

2

3

int * ptr = malloc(sizeof(int));

free(ptr);

ptr[0] = 3;
```

Then Memcheck will report the invalid write as being inside of the freed block (and tell us where we freed the block):

```
1

2

3

4

5

==5486== Invalid write of size 4
```

```
==5486==    at 0x4005A3: main (outOfBounds2.c:9)

==5486==  Address 0x51fc040 is 0 bytes inside a block of size 4 free'd

==5486==    at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)

==5486==    by 0x40059E: main (outOfBounds2.c:8)
```

Valgrind's Memcheck will also check for memory leaks. However, by default it only reports a summary of the leaks, which is not useful for finding and fixing the problems. If you have memory leaks, you will want to run with the **--leak-check=full**option. When you do so, Memcheck will report the location of each allocation which was not freed. You can then use this information to figure out where you should free that memory.

Note that when running Valgrind's Memcheck with GDB, you can run the leak checker at any time with the monitor command **monitor leak_check full reachable any**.

## Completed