Invalid Reads and Writes

Valgrind’s Memcheck tool will also perform stricter checking of memory accesses (*e.g.* via pointers) than normally occurs when you run your program. In particular, Memcheck will track whether or not every address is valid at any particular time (as well as whether or not that address contains initialized data). Any access to an invalid address will result in an error for Memcheck.

For example, suppose we wrote the following (broken) code:

```
//horribly broken--returns a dangling pointer!
int * makeArray(size_t sz) {
  int data[sz];
  for (size_t i = 0; i < sz; i++) {
    data[i] = i;
  }
  return data;
}
```

Depending on what we do with the return result of this function, our code might either appear to work, or give us rather strange errors. If we read the array in the calling function (*e.g.*, **main**), then we might not immediately observe anything bad (however, if we call other functions, the values in the array might "mysteriously" change). If we ran such code in Valgrind, we would might get an error such as this:

```
==24640== Invalid read of size 4
==24640==    at 0x40060C: main (dangling.c:16)
==24640==  Address 0xfff000340 is just below the stack ptr.
```

Here, Memcheck is telling us that we tried to read 4 bytes from an invalid (currently unallocated) memory location. It gives us a call stack trace for where the invalid read occurred (in this case, it was in on line 16 of the code, which is not shown here). Memcheck tells us what address experienced the problem and gives us the most information it can about where that address is relative to valid regions of memory. In this particular case, the address is just below the stack pointer, meaning that it is in the frame of a function that recently returned. If we had written to the address instead, we would get a message about an "Invalid write of size X."

Note that Memcheck cannot detect all memory-related errors (even though it can detect many that will slip through otherwise). If another function were called (which would allocate a frame in the same address range), the memory would again become valid, and Memcheck would be unable to tell that accesses to it through this pointer are not correct. Likewise, Memcheck may not be able to detect an array-out-of-bounds error because the memory location that is improperly accessed may still be a valid address for the program to access (*e.g.*, part of some other variable).

Note that using **-fsanitize=address** can find a lot of problems of this type that Memcheck cannot. The reason is that **-fsanitize=address**forces extra unused locations between variables and marks them unreadable with the validity bits it uses. Because there is now space invalid space between the variables, the checks inserted by **-fsanitize=address** will detect accesses in between them, such as going out of the bounds of one array.