CSE 333
Section 2
Valgrind Notes
For use with sec2_Valgrind slides

Some Buggy Code:

- [Note: the new line above main was added to make the slide read better, but this means the line numbers in main are one larger than what valgrind reports. The important point here is that if you forget to recompile before you run valgrind then you can get confused when trying to debug, even if you only changed white space.]
- main:
  - Note that atoi is not the most robust way to parse an int. It returns zero if it does not understand the input, which is often not what you want. See sscanf or strtol.

Valgrind Output:

- "Invalid write of size 4"
  - You wrote to a memory address that you were not supposed to. The "size 4" indicates you wrote something that is 4 bytes. This could be:
    - int, long, or long long if your system has 4 byte int, long, or long long;
    - int32_t or uint32_t;
    - void * if your system uses 4 byte pointers;
    - some struct if it happens to be 4 bytes (including the padding);
    - etc...
  - Notice the stack trace leading up to the line that caused the error. The root cause of your bug is not necessarily in there, but it is the first place to look.
  - "Address 0x51d2068 is 0 bytes after a block of size 40 alloc'd"
    - This means you wrote outside of the memory malloc gave you. This message usually means you are writing past the end of an array. Specifically, you are writing one element past the end (i.e. "0 bytes after").
    - Notice once again the stack trace. This says where you got the memory from, i.e. where the allocation happened sometime in the past.
  - Looking at RangeArray.c:14, the error is probably the "<=" in the for loop. That would make sense given that the error message is commonly seen with going past the end of an array, and we only wrote one element past the end.
- "Invalid read of size 4"
  - This same thing is going on as above, but you are reading instead of writing.
  - "Address 0x51d2068 is 0 bytes after a block of size 40 alloc'd"
    - Notice that 0x51d2068 is the same as the previous error message. This means we are reading the same data we wrote earlier.
  - Looking at warmup.c:22 (actually 23), the error is probably the "<=" in the for loop. The calculation of length is the same as in the previous error, so it makes sense that it would break here too.
- "1 2 3 4 5 6 7 8 9 10 11"
  - This is the output of the program. Remember that valgrind is running the code; it is not really looking at the source. It can only find errors that actually happen while the program is running.

- "HEAP SUMMARY"
    - Basically, if there is anything in use when the program exits, then we are not happy with you. For one-off programs that no one else will ever see, you can sometimes ignore this. However, if any other person will see/run your code then you should fix this. We usually state that valgrind must exist without errors or leaks for anything you submit, but some exercises might play with uninitialized memory (take a close look at “Your code must” in ex2).
- "LEAK SUMMARY"
    - You can look into the documentation for what each of these mean and why they are different. "indirectly lost" probably means that thing you leaked was pointing to something else that was malloc'd (which is now also leaked). For example, you called free directly on some data structure instead of calling the proper Cleanup function.

Code Fix:

- Note that the comment on RangeArray should indicate that the client should free the pointer it returns (if not NULL).
- "XXX We must check this explicitly"
    - Valgrind did not find this. It is a logical bug, but notice the length calculation can be negative. This means malloc will get a negative argument, which is an actual bug that needs to be fixed.
- "XXX... malloc return..."
    - Valgrind did not find this either. Never forget that malloc can return NULL.
- "XXX... off-by-one..."
    - Valgrind found this.

Code Fix (cont.):

- The rest of these are straightforward. Valgrind did not find the first, but it did find the others.

Illegal Reads/Writes:

- It is always a bad idea for a pointer to point to invalid memory. Reading or writing invalid memory is always a bug.
- Note that free is handed the correct pointer. This is not a bug.
- The compiler will trust any cast, so casting a constant to a pointer is perfectly valid. However, it is pointing to invalid memory so using it is a bug.

Illegal Frees:

- "Address is not stack'd, malloc'd or (recently) free'd"
    - Valgrind does not know anything about the pointer you are trying to use. This likely comes up if you dereference something that is not a pointer. Remember that C does not guarantee a Segfault: anything can happen.
- "Invalid free()..." -> "Address 0x51d2050 is 12 bytes after a block of size 4 alloc'd"
    - Unlike the previous error, valgrind recognizes the address as something "nearby" to a malloc'd pointer and is trying to be helpful by telling where the malloc happened.