Tutorial 5: Debugging memory errors in C++ using Valgrind

Code that looks like this may be familiar:

```
int main() {
  int *elements = new int[3];
  return 0;
}
```

Obviously, your code will not this short. Memory errors and undefine d behaviors will be harder to spot in real situations. They often go unnoticed or crash the program with a non-descriptive message, since C and C++ are quite perimissive on what you can do.

Valgrind

- A tool for finding undefined behaviors and memory errors
- Can do basic leak checking, but will also inform you of the presence of many other memory errors and undefined behaviors.
- Error messages can be confusing, but contain lots of information.
- How can we use this to our advantage?

Before we begin

Make sure that you compile you code using `-g` option:
`g++-5 --std=c++14 -g <source-files>`
This adds debug information to your executable so that valgrind and gdb can display line number information for errors.

Code sample 2

```
int main() {
  int * elements = new int[3];
  elements[3] = 4;
  cout << elements[3] << endl;
  delete [] elements;
  return 0;
}
```

1. What's wrong with this code?
2. What happens when we run this code?
3. How can we find this error despite it working fine?

Code sample 3

```
int main() {
  int i;
  if (i) {
    cout << "We did it!" << endl;
  } else {
    cout << "We didn't do it." << endl;
  }
  return 0;
}
```

1. What's wrong with this code?
2. What happens when we run this code?
3. Why might this be a problem?

Code sample 4

```
int main() {
  int elements[3];
  delete [] elements;
  return 0;
}
```

Luckily, g++ warns us when we try to compile this. Why? Can we write our code in such a way as to cause this problem without a convenient compiler warning?

Code sample 4, version 2

```
void helper(int *mem) {
  delete [] mem;
}
int main() {
  int elements[3];
  helper(elements);
  return 0;
}
```

What do you think this will do when we run it?

Code sample 5

```
struct S {
  int elements[3];
};

int main() {
  S *s = new S;
  delete [] s->elements;
  return 0;
}
```

1. What's wrong with this code?
2. What do you expect to happen when we run it?
3. Many memory errors might go unnoticed without valgrind. That doesn't mean they have no negative effects.

Some notes

- Valgrind may not catch every error, so don't assume your program must be free of bugs just because valgrind doesn't identify any!
- You may find valgrind's `--error-exitcode` option useful for intergrating it into runSuite.

Some notes

- In the student environment and many other environments, Valgrind will report that there are some "still reachable" memory. For example:

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 72,704 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- This is acceptable as long as the error summary says `0 errors from 0 contexts`. For details see here.