Purpose of this Lab

This lab demonstrates use of valgrind, a tool that helps you find memory leaks, uninitialized variables, and bad memory accesses in your programs.

To accomplish this, we use a number of example programs. These are trivial programs, for teaching purposes, so you will probably be able to just glance at the programs and find the problems without using valgrind. However, imagine a ten thousand line program with dozens of conditional memory allocations and deallocations—that might be a bit more difficult.

Files

This tutorial uses a number of source files. Get them from the web at ~cs253/Labs/Valgrind.

Example #1

In these examples, the shell prompt is "%".

Consider the first program, exp1.cc. Examine it. If you see a problem with it, do *not* shout it out—it's not a contest:

```
% cat exp1.cc
```

Compile the first program:

```
% g++ -Wall exp1.cc
```

Run it. All appears well:

```
% ./a.out
```

Run it again with valgrind:

```
% valgrind ./a.out
```

Amongst all that, you see the distressing message `definitely lost: 100 bytes in 1 blocks`
and the instruction `Rerun with --leak-check=full to see details of leaked memory`. Do so:

```
% valgrind --leak-check=full ./a.out
```

That's still quite noisy; use the `-q` (quiet) option:

```
% valgrind -q --leak-check=full ./a.out
```

That last line isn't very helpful. Recompile with `-g` (debug information) so that valgrind can pinpoint which line allocated the leaked memory, and re-run valgrind:

```
% g++ -Wall -g exp1.cc
% valgrind -q --leak-check=full ./a.out
```

The problem, of course, is that 100 bytes were allocated, but never deallocated. Add code to `exp1.cc` to deallocate the memory, recompile and run with valgrind to show that the memory is now properly deallocated.

Example #2

Compile and run `exp2.cc`:

```
% g++ -Wall -g exp2.cc
% ./a.out
```

It seems to run just fine, but the code is clearly flawed. What's the problem? Why didn't you get a segmentation fault?

Now, run it with valgrind:

```
% valgrind -q --leak-check=full ./a.out
```

Read and understand the messages.

Examples #3–7

Compile and run `exp3.cc` through `exp7.cc` with valgrind, and understand the problems.

Example #8

Compile and run `exp8.cc`, with valgrind, and understand the problem. Note that when you run the program *without* valgrind, the memory allocation library is smart enough to detect that something bad has occurred. However, valgrind's error message is *much* better than the message without valgrind.

Example #9

Compile and run `exp9.cc`, with valgrind, and understand the problem. Looks like valgrind can't find all problems!