Usage of Valgrind

Let's use Valgrind to find the memory leak and out-of-boundary problems given in the example at https://github.com/PacktPublishing/Expert-C-2nd-edition/blob/main/Chapter14/ch14_dynamic_analysis.cpp.

To use Valgrind for dynamic analysis, the following steps need to be performed (consider the fatc that valgrind is supported only on Linux and if you want to install it on Windows you should install WSL on your Winodws and then do the commands below):

1. First, we need to install `valgrind`. We can do this using the following command:

```
sudo apt install valgrind //for Ubuntu, Debian, etc.
```

2. Once it has been installed successfully, we can run `valgrind` by passing the executable as an argument, along with other parameters, as follows:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
--verbose --log-file=valgrind-out.txt ./myExeFile myArgumentList
```

3. Next, let's build this program, as follows:

```
g++ -o ch14_dyn -std=c++11 -Wall ch14_dynamic_analysis.cpp
```

4. Then, we run `valgrind`, like so:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
--verbose --log-file=log.txt ./ch14_dyn
```

Finally, we can check the contents of `log.txt`. The bold and italic lines indicate the memory leak's location and size. By checking the address `(0x4844BFC)` and its corresponding function name (`main()`), we can see that this `malloc` is in the `main()` function:

```
... //ignore many lines at beginning
by 0x108A47: main (in /home/nvidia/wus1/Chapter-13/ch14_dyn)
==18930== Uninitialised value was created by a heap allocation
==18930== at 0x4844BFC: malloc (in /usr/lib/valgrind/vgpreload_memcheckarm64-linux.so)
... //ignore many lines in middle
==18930== HEAP SUMMARY:
==18930== in use at exit: 40 bytes in 1 blocks
==18930== total heap usage: 3 allocs, 2 frees, 73,768 bytes allocated
==18930==
==18930== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==18930== at 0x4844BFC: malloc (in /usr/lib/valgrind/vgpreload_memcheckarm64-linux.so)
==18930==
==18930== LEAK SUMMARY:
==18930== definitely lost: 40 bytes in 1 blocks
==18930== indirectly lost: 0 bytes in 0 blocks
==18930== possibly lost: 0 bytes in 0 blocks
==18930== still reachable: 0 bytes in 0 blocks
==18930== suppressed: 0 bytes in 0 blocks
```

Here, we can see that `malloc()` is called to allocate some memory at address `0x4844BFC`. The heap summary section indicates that we have 40 bytes of memory loss at `0x4844BFC`. Finally, the leak summary section shows that there is definitely one block of 40 bytes memory loss. By searching the address value of `0x4844BFC` in the `log.txt` file, we eventually figured out that there is no

`free(p)` line being called in the original code. After uncommenting this line, we redo the `valgrind` analysis so that the leakage problem is now out of the report.

In conclusion, with the help of static and dynamic analysis tools, the potential defects of a program can be greatly reduced automatically. However, to ensure the quality of software, humans must be in the loop for final tests and evaluations. Now, we're going to explore the unit testing, test-driven development, and behavior-driven development concepts in software engineering.