Preserving Type Information in Memory Usage Measurement of C++ Programs

Zsolt Parragi, Zoltan Porkoláb

Memory profilers are essential tools to understand the dynamic behavior of complex modern programs. They help to reveal memory handling details: the wheres, the whens and the whats of memory allocations, helping progammers find memory leaks and optimizing their programs.

Most heap profilers provide sufficient information about which part of the source code is responsible for the memory allocations by showing us the relevant call stacks. The sequence of allocations inform us about their order. Heap profilers for languages with strong reflection capabilities can also show type information on the allocated objects.

C++ has no run or compile time reflection, and certain widely used constructs – for example, allocators in the standard library – hide information even more. Theoretically the information is still there, and could be recovered based on the source code, build command list and stacktrace, but this is impossible in practice.

Previous solutions used the preprocessor and operator overloading to solve this problem. Unfortunately the complex syntax of new and delete restricts the use of this method: using dynamically allocated arrays, placement new or calling the new or delete operators directly requires source code modification, even in the standard library.

In this article we will report on a type preserving heap profiler for C++ based on the LLVM/Clang tooling library which resolves the above limitations. Our technique is based on source to source compilation resulting in standard compliant C++ code. The instrumented source can be built by the same compilers as the original, without any dependency on debug symbols or run time type information.

Using this method users can tell how much memory was used of std::vector objects – separately for every template parameter – in every point of the run, when, where and how they were allocated and freed. Having such a type information in hand programmers can identify critical classes responsible for memory usage more easily and can perform optimizations based on evidence rather than speculations.

This information also helps in the detection of memory related bugs: for example heavy use of casts might result in calling the wrong destructor, resulting in otherwise hard to debug or detect program errors.

The tool is publicly available at http://typegrind.github.io

References

[1] J. Mihalicza, Z. Porkoláb, and A. Gabor, Type-preserving heap profiler for C++ (2011)