Interpreting C++20, with profiling and debugging

26th International Conference on Computing in High Energy and Nuclear Physics

Philippe Canal (FNAL), Javier Lopez-Gomez (CERN), Guilherme Amadio (CERN), Jonas Hahnfeld (CERN), Axel Naumann (CERN), Vassil Vassilev (Princeton)

CHEP2023, 11/05/2023
Cling is a C++ "interpreter" (actually, an incremental compiler), based on LLVM/clang.

- Using LLVM/clang provides a solid infrastructure for C++ parsing / optimizations.
- clang used as a library; cling does additional processing, e.g. to parse top-level statements.

```
// TopLevelStatement.C
sin(12) // so ill-formed C++ but essential to Cling!
```

- Some unique features: value printing, entity redefinition, null ptr checking...

```
root [] int i = 0; ++i // value printing
(int) 1
root [] std::string i{"cling"}; // entity redefinition
```
Cling is a C++ "interpreter" (actually, an incremental compiler), based on LLVM/clang.

- Using LLVM/clang provides a solid infrastructure for C++ parsing / optimizations
- clang used as a library; cling does additional processing, e.g. to parse top-level statements

```cpp
// TopLevelStatement
sin(12) // so ill-formed C++ but essential to Cling!
```

- Some unique features: value printing, entity redefinition, null ptr checking...

```cpp
root [] int i = 0; ++i
(root [] (int) 1
root [] std::string i{"cling"};
```

**Foundational role:**
HENP’s Python binding (cppyy + PyROOT), ROOT I/O, and ROOT’s GUI system (old and web) depend on cling!
The foundations of cling are being upstreamed under the name clang-repl

**Goal**

Reduce cling to HENP-specific features

- Everything else should be part of llvm / clang
- Including patches to llvm (DONE) and clang

Once finalized, there are plans to rebase cling atop clang-repl

- **DONE** Much of incremental interpretation already part of clang-repl since llvm15
- **DONE** Since llvm16: clang supports running statements on the global scope the way cling does, but with a more robust frontend and backend support
- **WIP** We are currently working on landing cling::Value (RFC) and cling’s CUDA backend (RFC)
The new InterOp package provides interoperability primitives to aid bridging C++ with dynamic languages such as Python.

We are actively migrating Cppy (powering ROOT’s PyROOT) to enable faster and more accurate automatic Python to C++ bindings.

Figure 1: Time taken and memory used during class template instantiation for `std::tuple<double, double, ...>` and `std::vector<...<std::vector<double>>>`
Better Numba integration through libInterop

Basic support for integration with Numba was added to Cppy.

This allows the use of Python and C++ together without compromising performance.

Through libInterop this integration can be improved by using the same LLVM backend for both Numba and Cppy, allowing for duplicate code removal and better inlining.
Recent cling features

- The upgrade to LLVM13 brought C++20 (concepts, ...) support into cling!

```cpp
#include <concepts>
#include <vector>

template<typename T> concept Hashable = requires(T a) {
    { std::hash<T>{a} } -> std::convertible_to<std::size_t>;
};

template<Hashable T> void f(const T&) {}

f(std::vector<int>{});
```

- Since ROOT v6.20 (cling-0.7): allow redefining an entity, even as a different kind

```cpp
root [] int i = 0
(int)0
root [] float i(float x) { return x + 1; } // Note that 'i' is now a function
root [] i(12.0f)
(float) 13.0000f
```
Recent cling features

- The upgrade to LLVM13 brought C++20 (concepts, ...) support into cling!

```cpp
#include <concepts>
#include <vector>

template<typename T>
concept Hashable = requires(T a) {
    std::hash<T>{a} -> std::convertible_to<std::size_t>
};

template<Hashable T>
void f(const T&)

f(std::vector<int>{})
```

- Since ROOT v6.20 (cling-0.7): allow redefining an entity, even as a different kind

```cpp
root [] int i = 0 (int) 0
root [] float i(float x) { return x + 1; } // Note that 'i' is now a function
root [] i(12.0f) (float) 13.0000f
```
Recent cling features

- The upgrade to LLVM13 brought C++20 (concepts, ...) support into cling!

```cpp
#include <concepts>
#include <vector>

template< typename T> concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template< Hashable T> void f(const T&) {}

f(std::vector<int>{})
```

- Since ROOT v6.20 (cling-0.7): allow redefining an entity, even as a different kind

```cpp
root [] int i = 0
(int) 0
root [] float i(float x) { return x + 1; } // Note that 'i' is now a function
root [] i(12.0f)
(float) 13.0000f
```
ARM Aarch64 was already supported with cling.

But Apple’s ARM was... different!

- Different ABI, broken backtrace library in macOS, broken exception handling in JIT, ...
- We worked with Apple + LLVM community to add full support

Generally, lots of problems with macOS + Xcode, e.g. breaking changes between macOS 13.2 and 13.3; Xcode 13 and 14, etc.
Debugging interpreted code

Cling now emits debug symbols (Linux / macOS), allowing the use of a standard debugger to, e.g. single-step on interpreted code\(^1\)!

```
$ export CLING_DEBUG=1
$ gdb --args root.exe -l tutorials/hsimple.C
(gdb) break hsimple
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (hsimple) pending.
(gdb) r
root [0]
Processing tutorials/hsimple.C...

Breakpoint 1, hsimple (getFile=0) at tutorials/hsimple.C:36
36 TString filename = "hsimple.root";
(gdb) n
37 TString dir = gROOT->GetTutorialDir();
(gdb)
```

It is recommended to use ROOT \(\geq\)v6.28/04; older versions are known to have a bug

---

\(^1\)It is recommended to use ROOT \(\geq\)v6.28/04; older versions are known to have a bug
Cling can also emit symbol maps for `perf` (on Linux), enabling the profiling of interpreted code, e.g.

```
$ export CLING_PROFILE=1
# Run macro hsimple.C and gather performance counters
$ perf record -g -e cycles -- root.exe -l -q tutorials/hsimple.C
```

Flamegraphs can be generated from the recorded profile as follows\(^2\):

```
$ perf script --no-demangle | c++filt -p | stackcollapse-perf.pl --all | flamegraph.pl > output.svg
```

---

\(^2\) `stack-collapse-perf.pl` and `flamegraph.pl` are part of [https://github.com/brendangregg/FlameGraph](https://github.com/brendangregg/FlameGraph)
Profiling interpreted code

Cling can also emit symbol maps for perf (on Linux), enabling the profiling of interpreted code, e.g.

```
$ export CLING_PROFILE=1
# Run macro hsimple.C and gather performance counters
$ perf record -g -e cycles -- root.exe -l -q tutorials/hsimple.C
```

Flamegraphs can be generated from the recorded profile as follows:\(^2\):

```
$ perf script --no-demangle | c++filt -p | stackcollapse-perf.pl --all | flamegraph.pl > output.svg
```

Caveat: JIT symbols do not get demangled by perf. Instead, they are manually demangled using c++filt

\(^2\) stack-collapse-perf.pl and flamegraph.pl are part of https://github.com/brendangregg/FlameGraph
Profiling interpreted code

Figure 2: FlameGraph showing Python and JITted code (df102_NanoAODDimuonAnalysis.py)
Cling now comes with C++20, GCC 13 and macOS 13.3, etc. support: everything as current as it gets!

Debugging and profiling of interpreted code is now possible!

Now supporting Apple’s ARM and RISC-V architectures; continued Power support underway with help from IBM

Since 2022: cling foundations have been upstreamed to the LLVM community under the name clang-repl
  - There are plans to rebase Cling on top of clang-repl in the future
Thanks!