LLVM-based C++ Interpreter For ROOT

Axel Naumann, CERN

Lukasz Janyst, CERN

Philippe Canal, Fermilab

Why Interpreter?

Prompt: same as compiled language!

I/O: type database, members, object from type name

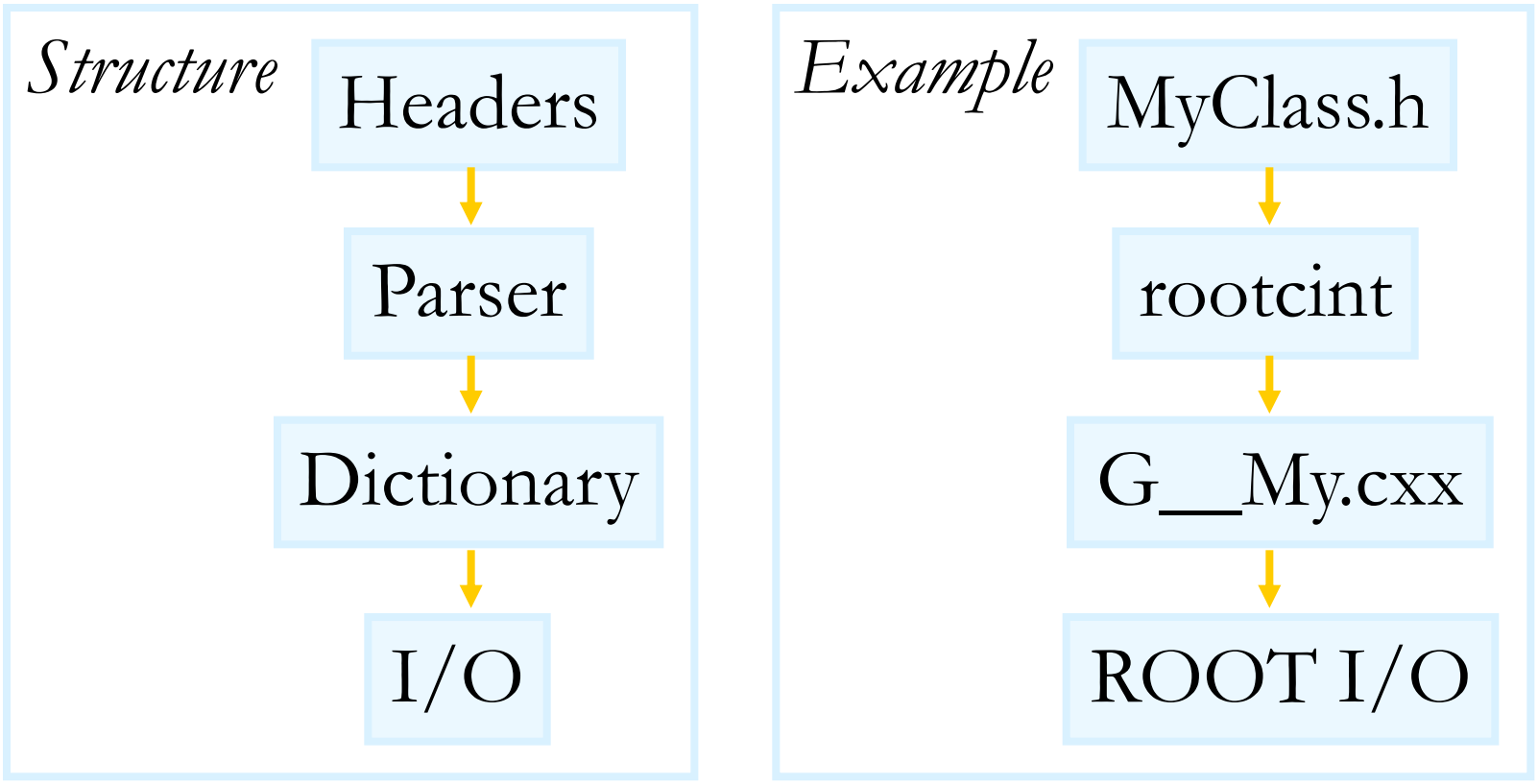
Signal/slot, Plugins: call function given a string

Interest even outside HEP

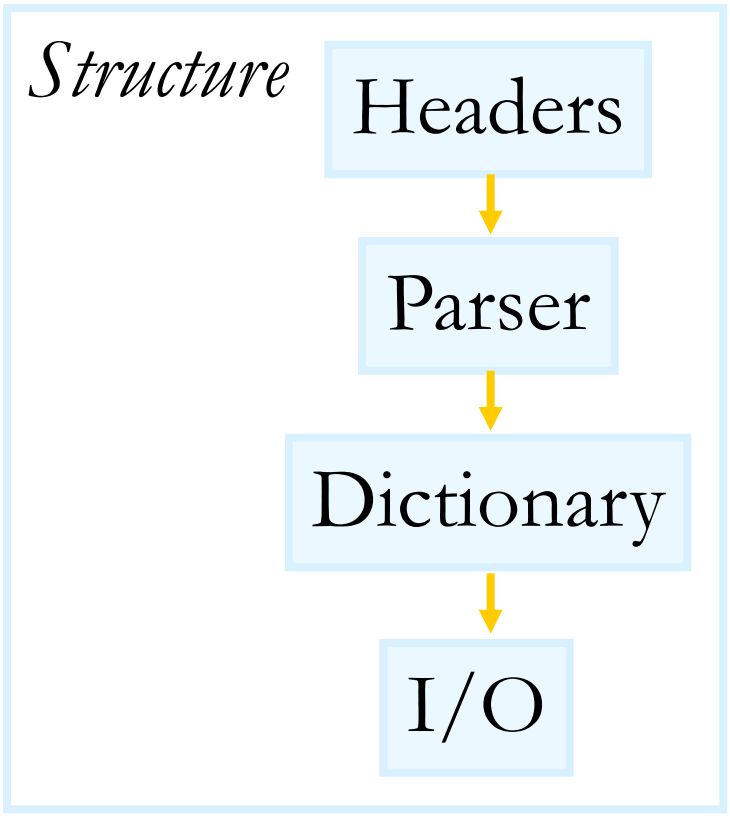
Main items:

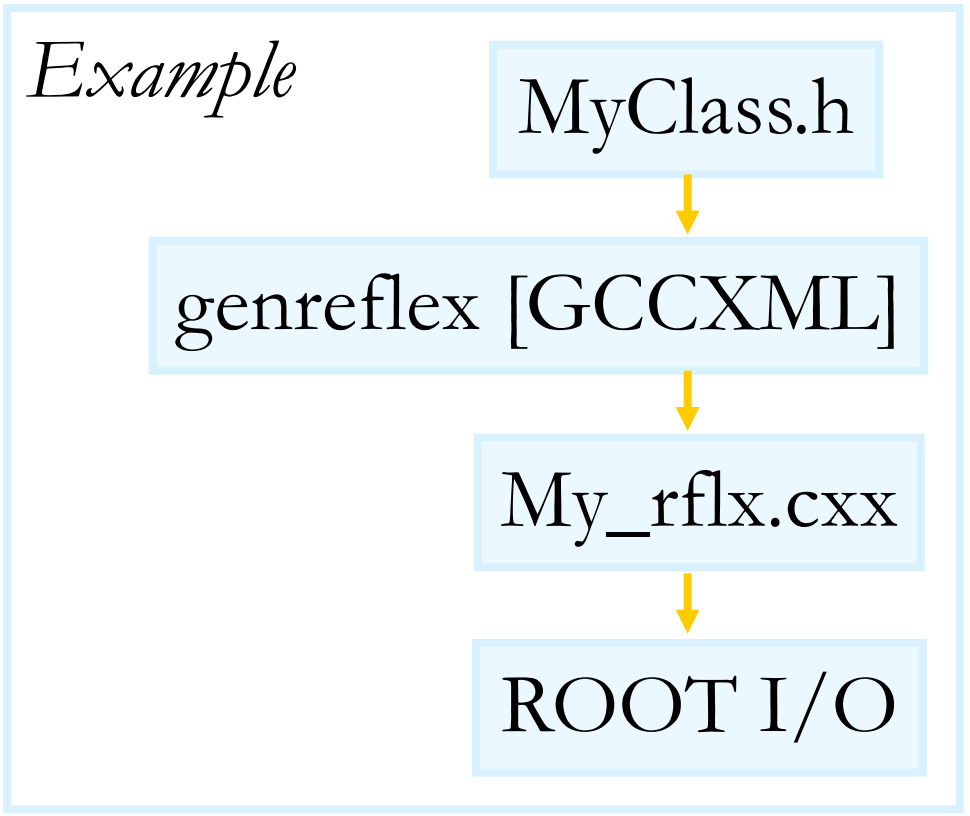
- Reflection
  (types, members, sizes…)
- Calling compiled functions

Overview Of Reflection Data



Overview Of Reflection Data





Current Reflection Gathering

CINT

- Limited parser (templates, operators, overloading, lookup, file handling, STL / sys headers)

GCCXML/ genreflex

- Inhomogeneous: GCC / XML / Python / C++
- Slow
- Extremely difficult to influence / fix; involves three parties (GCC, GCCXML, us), different interests

Function Calls From String, Current

Maintain table `"function_name"` → `&Func_stub`

Stubs generated as part of dictionary:

```
Func_stub(vector<void*> args) {
   function_name((int)args[0]); }
```

Given `"func(0)"` call function stub, pass arguments.

```
funcmap["func"](make_vector(0))
```

Needs one stub per function:

40% of dictionaries' size!

[Disclaimer: code is paraphrased]

LLVM

Open Source project with Uni Illinois Urbana-Champaign, started around 2002

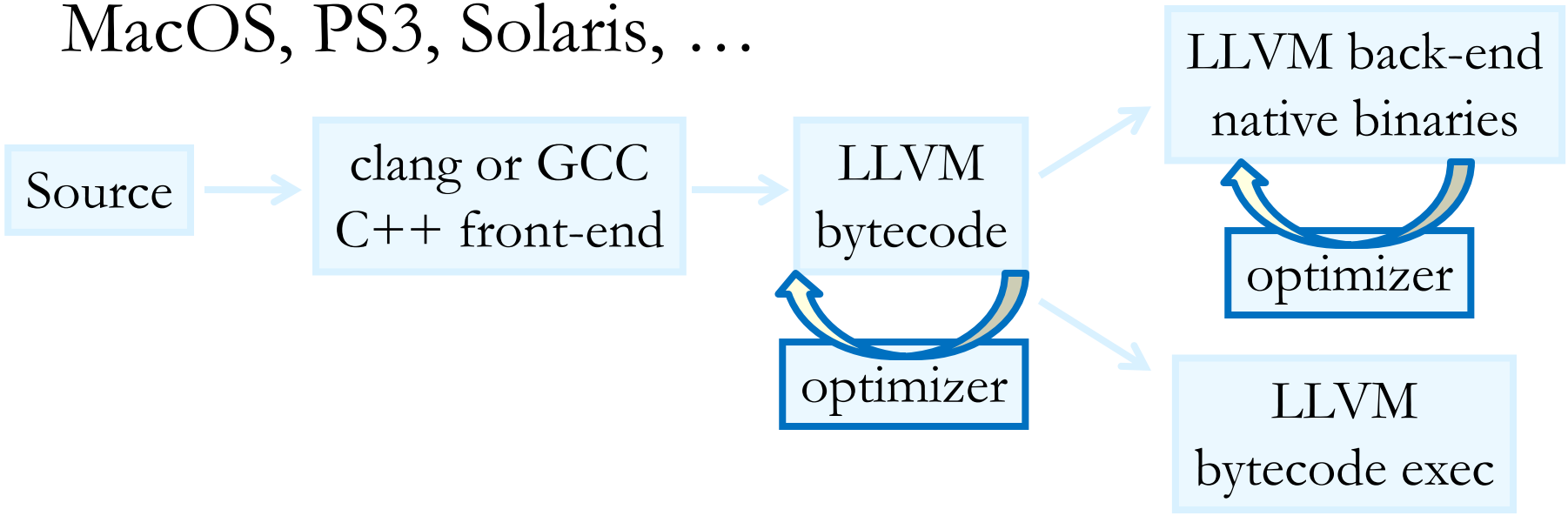
Sponsored by Apple, Adobe, Cray,…

Features:

- drop-in, faster alternative to GCC
- modular design with C++ API
- liberal NCSA open source license
- just-in-time compiler
- very active mailing list, very responsive developers

LLVM

- C++ front-end clang (alternative: GCC front-end)
- bytecode layer with optimizers and execution engine
- back-end for all major platforms: Windows, Linux, MacOS, PS3, Solaris, …



Can already build e.g. ROOT (with GCC front-end)

clang

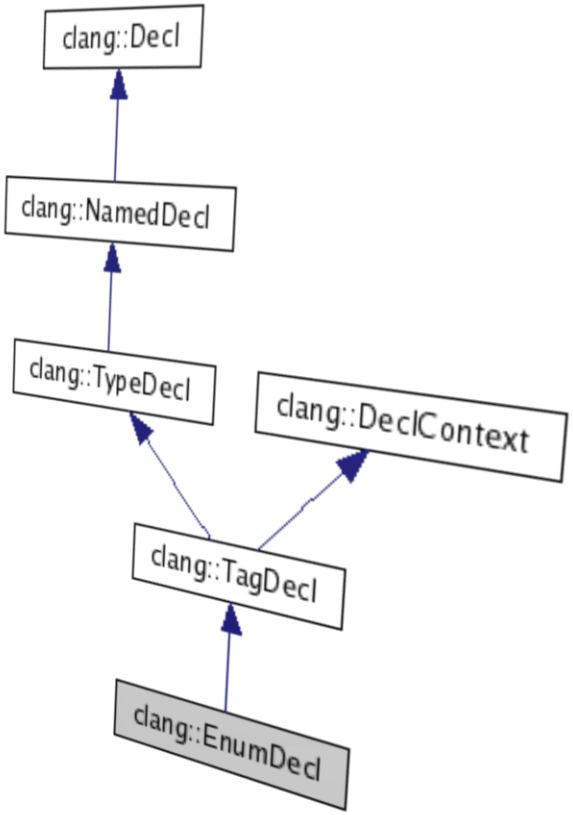
LLVM C[++] front-end: preprocessor etc, result: AST

Clean C++ API: analyze, generate, manipulate code!

Already used for C and ObjC

C++ still in development:
contribute and influence!

Apple: “production quality in 2011”
approx 200 commits / week,
by far most for C++



LLVM + clang + X = cling

Requirements: **all CINT features, no user level changes** (prompt, ACLiC, gInterpreter etc)

Benefits of cling:

- ✓ Interpreter is based on production-grade parser and optimizers; benefit from compiler improvements
- ✓ ACLiC (.L file.C+) will use JIT instead of native compiler / linker / loader: in-memory compilation!
- ✓ Modularity eases maintenance
- ✓ Apply >10 years of C++ interpreter to new design

Auto-dlopen, -#include, -Declaration

CINT: loads lib for class on first usage (rootmap)

clang: analyze AST, extract types, load needed libraries

CINT: needs no #include for types with dictionary

clang: intercept parser errors on unknown type; inject type's header into AST and re-parse

CINT: auto-declares variables in assignment

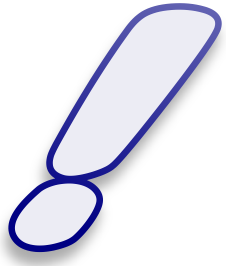
clang: patch in "auto" keyword as error handling

Function Calls With clang / LLVM

LLVM resolves missing symbols in memory from:

- shared libraries
- code already compiled in memory
- code not yet compiled: just-in-time (JIT) compile it!

JIT already available on X86, PowerPC, ARM, Alpha with Linux (32 / 64 bit), MacOS X, Win32.



No stubs anymore, 40% smaller dictionaries!

Also speeds up performance critical interpreted code!

The Challenges

As compiler, LLVM expects all code to be available.

cling on the other hand:

1. must allow iterative loading

```
cling[0] .L func0.C
cling[1] int i = func0();
cling[2] .L func1.C
cling[3] i = func1();
```

Compilers parse all,
then compile, then link.

Solution: iterative linking of tiny translation units

The Challenges

As compiler, LLVM expects all code to be available.

cling on the other hand:

1. must allow iterative loading
2. must keep stack

```
cling[0] int i = 12;
cling[1] .L times2.C
cling[2] times2(&i);
cling[3] printf("%d\n",i);
24
```

Stack not even set up for compiler.

Solution: need interpreter context to survive incremental linking

The Challenges

As compiler, LLVM expects all code to be available.

cling on the other hand:

1. must allow iterative loading
2. must keep stack
3. must support unloading

```
cling[0] .L func0.C
cling[1] func0();
cling[2] .U func0.C
cling[3] int func0 = 0;
```

Unthinkable for compilers.

Solution: need to modify AST, re-link, track dependencies,…

More Challenges

- Prompt: use prompt namespace, incremental linking
- Dynamic scope: automatic variable names

```
new TFile("f.root");
hist->Draw();
```

transform

```
new TFile("f.root");
cling.delay("hist->Draw()");
```

- "Multithreading" (multiple interpreter objects): make LLVM thread-safe where needed (rare), use new, thread-safe Reflex as reflection database
- PyROOT, ROOT's python interpreter interface : "llvm-py provides Python bindings for LLVM"

Objective

Aim at full-blown replacement for existing solutions:

- parser replaces CINT, GCCXML
- type info from clang replaces rootcint, genreflex
- interpreter replaces CINT
- JIT replaces ACLiC (.L MyCode.C+)

Works with GCC / MSVC / … as native compiler

No need to switch to LLVM as compiler!

Summary

LLVM and clang: exciting and promising!

Compile a list of ingredients for a C++ interpreter and code parser for reflection extraction:
clang + LLVM is an incredibly good match.

Proof-of-concept exists, first steps in prompt, JIT, unloading, calling into shared libraries:

```
http://root.cern.ch/viewvc/branches/dev/cling/
```

CINT will have a competitor!

cling: C++ Interpreter Demo

1.
```
[cling]$ auto s = "hello"; printf("%s\n", s);
```

2.
```
cling errors.h
```

3.
```
[cling]$ .L /usr/lib64/libexpat.so
[cling]$ .x xml.h
```

xml.h

```
#include "/usr/include/expat.h"
#include <stdio.h>

int xml() {
   printf("%s\n", XML_ExpatVersion());
   return 0;
}
```