Cling – The LLVM-based Interpreter

Vassil Vassilev

Why Do We Need Cling?

Cling's advantages:

- Full C++ support
  - STL + templates
  - Path to C++0x
- Planned massive reduction of dictionaries
- Easier and smoother transition between interpreted and compiled code
- Easy maintenance

What Is Cling?

- An interpreter – looks like an interpreter and behaves like an interpreter
  Cling follows the read-evaluate-print-loop (repl) concept.
- More than interpreter – built on top of a compiler (clang) and compiler framework (LLVM)
  Contains interpreter parts and compiler parts. More of an interactive compiler or an interactive compiler interface for clang

What Cling Depends On?

- LLVM

  “The LLVM Project is a collection of modular and reusable compiler and toolchain technologies...”

  - More than 120 active contributors

    Apple, ARM, Google, Qualcomm, QuIC, NVidia, AMD and more

  - ~250 commits/week

- Clang

  “The goal of the Clang project is to create a new C, C++, Objective C and Objective C++ front-end for the LLVM compiler.“

  - More than 100 active contributors

    Apple, ARM, AMD and more

  - ~150 commits/week

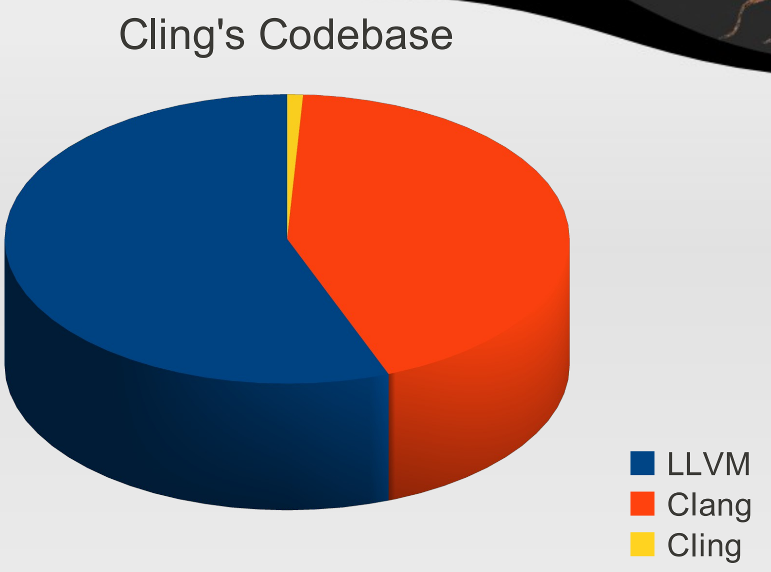
** Stats from last year until 14.10.2011*

Cling's Codebase

LLVM – 430K SLOC*

Clang – 333K SLOC*

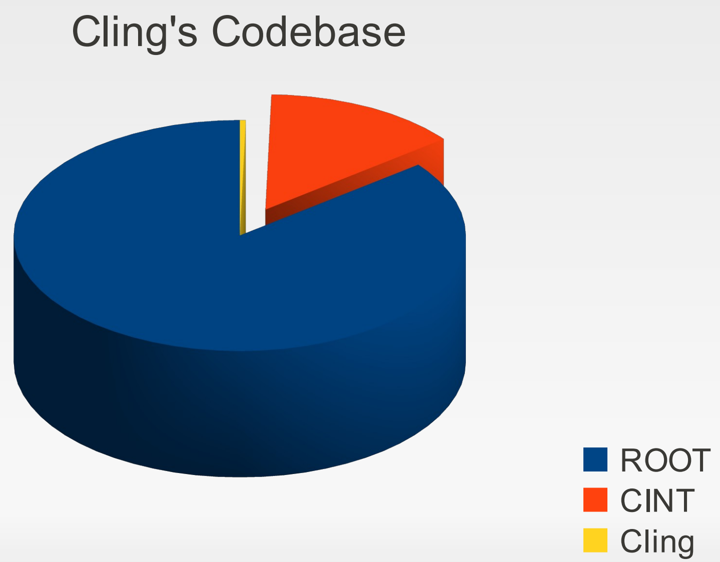
Cling – 7K SLOC*



Other ROOT – 1400K SLOC*

CINT+Reflex – 230K SLOC*

Cling – 7K SLOC*



** By 12.10.2011. No testsuites included. C and C++ only.*
Credits: generated using David A. Wheeler's 'SLOCCount'

Additional Features

- Just-in-time compiler (JIT)
- Extra platform support
- World class performance and optimizations
- OpenCL
- Expressive diagnostics
- ...

Expressive Diagnostics

- ## Column numbers and caret diagnostics

CaretDiagnostics.C:4:13: warning: ***'.*' specified field precision is missing a matching 'int' argument***

```
printf("%.*d");
        ~~^~
```

- ## Range highlighting

RangeHighlight.C:14:39: error: ***invalid operands to binary expression ('int' and 'A')***

```
return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                    ~~~~~~~~~~~~~ ^ ~~~~~~~~
```

- ## Fix-it hints

FixItHints.C:7:27: warning: ***use of GNU old-style field designator extension***

```
struct point origin = { x: 0.0, y: 0.0 };
                        ^~
                         .x =
```

Improving Cling Step-By-Step

Cling prototype in 2010:

- Shortcomings of the existing prototype were analyzed

  Source-to-source manipulations, variable initializers not managed correctly, redundant re-parsing, low efficiency, ...

- Redesign almost from scratch
- Resulted in complete rewrite

Advantages of the New Design

- Rely on the compiler libraries where possible instead of custom implementations
  Reduces the maintenance load. If the implementation is not too specific and makes sense for a compiler we prefer putting it into the compiler codebase and delegate the maintenance...
- More language independent
  The necessary code injections and rewrites are directly in the internal structures of the underlying compiler
- Stability
  The new design enables the implementation of stable error recovery
- Better performance
  Re-parsing only in very few cases

Challenges

- How to combine incompatible concepts like compilation and interpretation
  Many tasks that are trivial for an interpreter become a nightmare for a compiler.
- How to make it user-friendly
  First step should be to adopt the successful usability extensions from CINT.

Challenges

How to make it user-friendly

First step should be to adopt the successful usability extensions from CINT.

- ### Value printer

  The interactive mode obeys the repl concept and there should be easy, interactive and user-extensible access to types and values

- ### Expressions and statements

  CINT-specific C++ extension improving the user interaction with the interpreter from the terminal...

```
[cling]$ sin(1)
(double const) 0.841471
```

```
void wrapper() {
  sin(1);
}
```

Expressions and Statements

- Wrap the input
- Scan for declarations
- Extract the declarations one level up, as global declarations

```
[cling]$ int i = 12; printf("%d\n",i);
[cling]$ printf("%f\n",sin(i));
```

```
int i = 12;
```

```
void wrapper1() {
  int i = 12;
  printf("%d\n",i);
}
```

```
void wrapper2() {
  printf("%f\n", sin(i));
}
```

Challenges

How to combine incompatible concepts like compilation and interpretation

Many tasks that are trivial for an interpreter become a nightmare for a compiler.

- Initialization of global variables

  Cling depends on global variables, which need to be initialized. However, the global variables continue to be added (potentially) with every input line...

- Error recovery

  Even though the user has typed wrong input at the prompt cling must survive, i.e issue an error and continue to work...

- Late binding

  Cling needs to provide a way for symbols unavailable at compile-time a second chance to be provided at runtime...

Error Recovery

- Filled input-by-input from the command line
- Incorrect inputs must be discarded as a whole

```
**** Welcome to the cling prototype! ****
* Type C code and press enter to run it *
* Type .q, exit or ctrl+D to quit       *
*****************************************
[cling]$ int i; error_here; int j;
input_line_5:2:9: error: use of undeclared identifier 'error_here'
 int i; error_here; int j;
        ^

[cling]$ i
input_line_6:2:2: error: use of undeclared identifier 'i'
 i
 ^

[cling]$
```

Late Binding

✔ Defined in the root file

```
{
  TFile F;
  if (is_day_of_month_even())
    F.setName("even.root");
  else
    F.setName("odd.root");
  F.Open();
  hist->Draw();
  hist->Fill(1.5);
  hist->SetFillColor(46);
}
hist->Draw();
```

✘ The root file is gone. Issue an error.

✚ Opens a dynamic scope. It tells the compiler that **cling** will take over the resolution of possible **unknown symbols**

Late Binding

```
{
  TFile F;
  if (is_day_of_month_even())
    F.setName("even.root");
  else
    F.setName("odd.root");
  F.Open();
  gCling->EvaluateT<void>("hist->Draw()", ...);
  ...
}
hist->Draw();
```

- Tell the compiler the symbol will be resolved at runtime
- Wrap it into valid C++ code
- Partially recompile at runtime

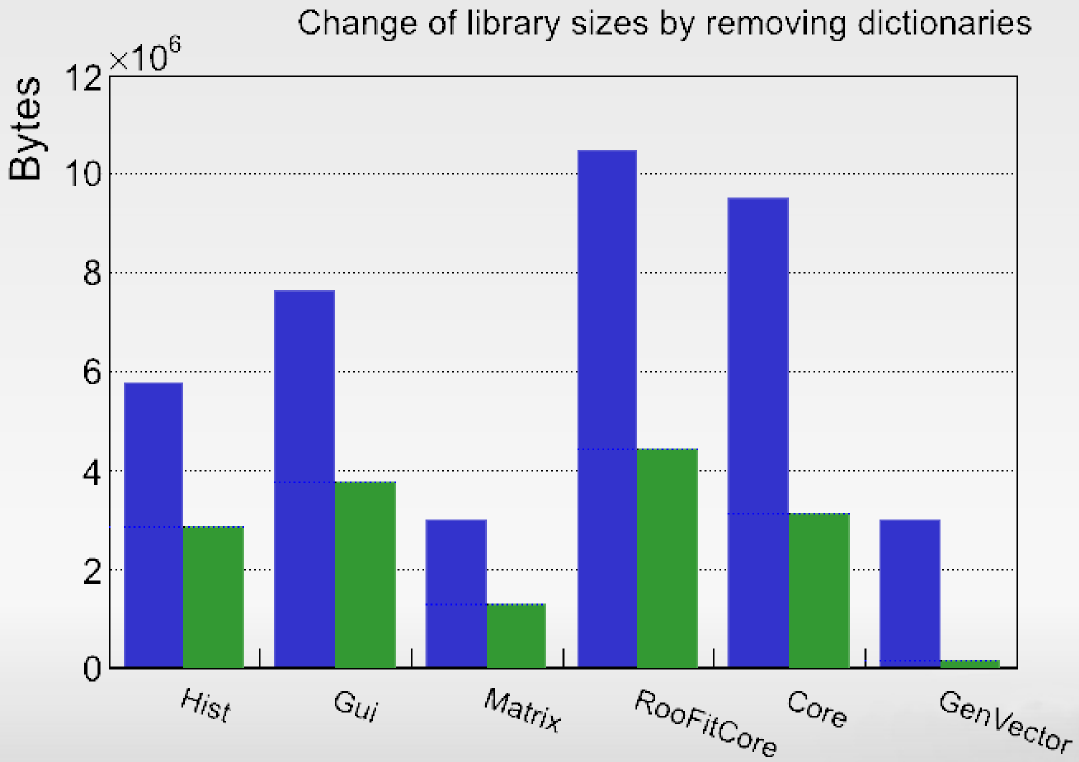
Challenges

- *Error recovery*
  Even though the user has typed wrong input at the prompt cling must survive, i.e issue an error and continue to work...
- Initialization of global variables
  Cling depends on global variables, which need to be initialized. However, the global variables continue to be added (potentially) with every input line...
- Late binding
  Cling needs to provide a way for symbols unavailable at compile-time a second chance to be provided at runtime...
- Value printer
  The interactive mode obeys the repl concept and there should be way of easy print value and type of expression in a user-extensible way...
- Expressions and statements
  CINT-specific C++ extension improving the user interaction with the interpreter from the terminal...

IMPLEMENTED

Dictionaries

- No reflection information in C++
  There is no way a C++ interpreter could know what are the detailed contents of a compiled program
- CINT and Reflex dictionaries:
  - Take large fraction of libraries
  - Multiple copies of the dictionary data in the memory



Dictionaries in Cling

- Now the compiler is an interpreter as well!
- JIT enables native calls into libraries
- Query the reflection data from compiler libraries
- Compiled dictionaries should be no longer needed
  - Middle term: Everything but ClassDef goes away!
  - Long term: No dictionaries at all

Library Calls in Cling

Can we avoid re-parsing again and again?

Can we repackage a library's headers?

- Load the lib
- #include the header containing the function definition
- Make the call

```
*****************************************
[cling]$ .L libz.so
[cling]$ #include "zlib.h"
[cling]$ zlibVersion()
(const char * const) "1.2.3.3"
[cling]$
```

Cling @ the LLVM Community

- On 25.07.2011 cling was announced on clang's mailing list as a working C++ interpreter
- People were thrilled and enthusiastic about it
- Lots of excellent comment and suggestions

The Road Ahead

Integration into ROOT

Ongoing and continuous process that needs:

- Experience with ROOT
- Knowledge about cling and clang interfaces

Future: Code Unloading

```
[cling]$  .L Calculator.h
[cling]$  Calculator calc;
[cling]$  calc.Add(3, 1)
[cling]$  2
[cling]$  .U Calculator.h
[cling]$  .L Calculator.h
[cling]$  Calculator calc;
[cling]$  calc.Add(3, 1)
[cling]$  4
```

```
// Calculator.h
class Calculator {
  int Add(int a, int b) {
    return a - b;
  }
...
};
```

```
// Calculator.h
class Calculator {
  int Add(int a, int b) {
    return a + b;
  }
...
};
```

Future: Code Unloading

- Fundamental requirement for ROOT
  This is what drives the rapid development in ROOT...
- Extremely difficult for a compiler
  Teaching an elephant to dance...
- Requires in-depth knowledge of clang internals
  Different phases in the compiler, advanced AST manipulations, inter-procedural analysis, knowledge about LLVM intermediate representation (bitcode), JIT internals, bitcode recompilation...
- Thinking out-of-the-box
  Not often seen problem needs novel way of understanding the compiler libraries...
- We know how to do it!
  Watermarks, dependency analysis, annotation of the corresponding bitcode, generated for the high-level internal structures,...

Cling in ROOT

- Lots of interest from experiments and physicists
- The prototype will be included in the source package of ROOT (the November release)
- The prototype will be an optional interpreter for ROOT

Demo:

C N G

Thank you!

Backup slides

Pre-Compiled Headers

Carefully crafted data structures designed to improve translator's performance:

- Reduce lexical, syntax and semantic analysis
- Loaded “lazily” on demand

Pre-Compiled Headers

Design advantages:

- Loading PCH is significantly faster than re-parsing
- Minimize the cost of reading
- Read times don't depend on PCH size
- Cost of generating PCH isn't large