Handle Execution results in clang-repl

Jun Zhang
jun@junz.org

Mentors:
Vassil Vassilev & David Lange
About me

I’m Jun Zhang, 3rd year undergraduate student major in Software Engineering.

Working in the compiler research team since May 2022.

Last year Google Summer of Code student, Clang/LLVM contributor. (Land ~70 patches)
Project Background
An introduction to clang-repl

clang-repl is an interactive C++ interpreter that allows for incremental compilation, based on Clang and LLVM Orc JIT.

```cpp
#include <iostream>
std::cout << "Hello, world!\n";
Hello,world!
```
Project Goals
Value pretty printing

```c++
clang-repl> int x = 42;
clang-repl> x
(int) 42
clang-repl> std::vector<int> v{1,2,3};
clang-repl> v
(std::vector<int> &)[1, 2, 3]
```
Compiled/Interpreted code interop

```cpp
int Global = 42;
void setGlobal(int val) { Global = val; }
int getGlobal() { return Global; }
Interp.ParseAndExecute("void setGlobal(int val);");
Interp.ParseAndExecute("int getGlobal();");
Value V;
Interp.ParseAndExecute("getGlobal()"); &V;
std::cout << V.getAs<int>() << \"\n\"; // Prints 42
```
The implementation
(Highly inspired by Cling)
Overview

Parser

set a marker in the token

Sema

set a flag in AST

AST Consumer

Synthesize the AST

Execute the given code

LLVM Orc JIT
Top level expressions extension

clang-repl> int x = 42;
clang-repl> x // Missing semicolon.
    Invalid in standard C++, but fine in the incremental C++.

Omit the semi tells clang-repl we want to capture the value of the DeclRefExpr (x)
When parsing an ExprStmt and the last semi is missing, we'll pretend that there's one and set a marker for the late use.
Sema sets a flag after seeing the special token, so we know we should transform the AST before the real CodeGen process.
Code synthesis

clang-repl> x → __clang_Interpreter_SetValue(x); // pseudo code.

DeclRefExpr → CallExpr

```cpp
// This synthesizes a call expression to specialize
+ // function that is responsible for generating the Value.
+ // In general, we transform:
+ // clang-repl> x
+ // To:
+ // 1. If x is a built-in type like int, float.
+ // __clang__Interpreter_SetValueNoAlloc(ThisInterp, OpaqueValue, xQualType, x);
+ // 2. If x is a struct, and a lvalue.
+ // __clang__Interpreter_SetValueNoAlloc(ThisInterp, OpaqueValue, xQualType,
+ // &x);
+ // 3. If x is a struct, but a rvalue.
+ // new __clang__Interpreter_SetValueWithAlloc(ThisInterp, OpaqueValue,
+ // xQualType)) (x);
+ + Expr *Interpreter::SynthesizeExpr(Expr *E) {
```
Value runtime

A value is a container that can carry the arbitrary result of an expression in an endian-independent way with small buffer optimization.

```cpp
class Value {
public:
  clang::QualType* getType(); // Obtain the type information of the expression.
  template<typename T>
  T castAs(); // Cast the value to corresponding type.
  void printType(llvm::raw_ostream& OS);
  void printData(llvm::raw_ostream& OS);
  void print(llvm::raw_ostream& OS);
  void dump() const; // Dump the value, called print(llvm::outs()) internally.
};
```

The value that holds the information of the expression can be passed around after construction.
Pretty print implementation

If the user asks for the Value, we pass it as output parameter.

Or we perform pretty printing: Invoke Value::dump()
Implementation of Value::dump

If the type is a builtin type:

Just print it directly.

Else:

Synthesize a call to another runtime function: PrintValueRuntime(const T*)

```
+  } else {
+    // All fails then generate a runtime call, this is slow.
+    SS << SynthesizeRuntimePrint(V);
+  }
+  return Str;
```
std::string PrintValueRuntime(const T*)

All overloads live in a header, which are included at runtime.

So print a std::vector is equivalent to: PrintValueRuntime(&v);

This means users can write their own overload for their types:

```cpp
clang-repl> struct S{};
clang-repl> std::string PrintValueRuntime(const S* s) { return "My printer!"; }
clang-repl> S{}
(S) "My Printer!"
```
How we make it
Submit the RFC

RFC: Handle Execution Results in clang-repl

TLDR: Synthesize automatic `printf` to print execution results in clang-repl and generalize the approach to use an object used to bridge compiled/interpreted code taking inspiration from what was done in Cling.

**Introduction**

The Cling interpreter is a unique interpretative technology for C++ based on Clang developed by high-energy physics (HEP). It is used to deliver reflection and type information for exabytes of scientific data and is heavily used during data analysis of particle physics data from the Large Hadron Collider (LHC) and other particle physics experiments.

In RFC Moving parts of the Cling REPL in Clang we discussed and shipped the initial incremental compilation facilities into LLVM mainline, called clang-repl.

In this RFC we propose two distinct features and their interaction: automatic `printf` and connecting compiled and interpreted C++ through a class called `Value` as an abstraction layer used to carry expression results and support value pretty printing in clang-repl.

**Goals**

**Automatic `printf`**

One of the key aspects of interactive C++ is exploratory programming which encourages showing execution results on screen easily. Typing every time `printf` or similar is too laborious and too annoying. Taking inspiration from Cling, we could achieve this effect by an extension that lives purely in libc++/Interpreter. We propose to have a special mode to indicate when we want to do value pretty printing: a expression in the global scope (without the semicolon). Coincidently Rust takes a similar approach:

Spend almost one month writing the RFC and discussing the implementation with the community
Submit patches to Phabricator

Made 2 patches in but still one left!
What I learned
Large patches are hard to get it in!

Crazy long revision history!
Writing portable code is hard!

To revert, or not to revert, it’s a question...
Thank you!

Special thanks to:

Aaron Ballman
Axel Naumann
Lang Hames
Richard Smith
Stefan Gränitz