Implement value printing in clang-repl

Jun Zhang (jun@junz.org)

Mentor: Vassil Vassilev

Overview

clang-repl is the upstream version of Cling Interpreter, which only implements a subset of features in Cling. In this proposal, we try to bring value printing, a very useful feature that enables users to know the detailed information of the expressions that users have inputted. TL;DR: we want clang-repl to do something like this:

```
[0] int X = 42;
[1] X
(int&) 42
[2] std::vector<int> V = {1,2,3};
[3] V
(std::vector<int>&) {1,2,3}
[4] "Hello, world!"
(const char [14]) "Hello, world!"
[5] 18 + 24
(int) 42
```

Motivation

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 EB of scientific data and is heavily used during data analysis of exabytes of particle physics data from the Large Hadron Collider (LHC) and other particle physics experiments. One of the major challenges we face to foster that community is our cling-related patches in LLVM and Clang forks. The benefits of using the LLVM community standards for code reviews, release cycles, and integration has been mentioned a number of times by our "external" users.

The aim of this project is to reduce the number of patches in the cling-related forks of LLVM and Clang. In addition, it will help move an essential feature of cling in LLVM upstream, providing less technical debt for HEP and also exposing it to world-class compiler experts, possibly improving and maintaining the feature further.

Introduction

First of all, I'd like to introduce some necessary terminologies: `Expression` and `Value`. In programming languages, a value is the representation of some entity that can be manipulated by a program. ***The value consists of the type and the real value of that type.***
For example:

```
int X = 42;
```

X is a value, the type is `int&`, and the real value is `42`.

"expression" is a combination of values and functions that are combined and interpreted by the compiler to create a new value. So below are all expressions in C++:

```
X; // single lvalue.
42; // single rvalue
16 + 26 // binary operator, 2 values involved, and will create a new
value.
```

Print primitive types

This is the simplest case we can consider. In general, clang-repl should be able to produce:

```
clang-repl> int X = 42;
clang-repl> X
(int) 42

clang-repl> "Hello, world!"
(const char [14]) "Hello, World!"

clang-repl> auto* Y = &X;
clang-repl> Y
(int *) 0x7f6662bc2000
```

Print the content of STL containers

Now the situation gets a little bit complex, as there's too much information that worth to be printed. Thus, I proposed that we only print the information that most people care about by default.

```
clang-repl> std::vector<int> V {1,2,3};
clang-repl> V
(std::vector<int> &) { 1, 2, 3 }

clang-repl> std::map<int,std::string> M = {{1, "Apple"}, {2, "Banana"}};
clang-repl> M
```

```
(std::map<int, std::string> &) { 1 => "Apple", 2 => "Banana" }

clang-repl> std::tuple<int,std::string, double> T = {42, "Repl", 3.14};
clang-repl> T
(std::tuple<int, std::string, double> &) { 42, "Repl", 3.1400000 }
```

Print user-defined types

It's merely impossible to support every kind of type, especially what our users have defined. It's also possible that the users can have different ideas about what value should be printed exactly. Thus, not only do we need a way to support the user-preferred approach of printing a value, but also it should have the highest priority. In this case, clang-repl should have a plugin that allows users to define their approach to printing their own classes, and overwrite the default way of printing STL facilities. Also, if we can't find a guideline about how to print a specific class, we should just fall back and print its memory address.

Implementation Plan

General idea

Before we jump into the implementation of the value printing feature, let's discuss clang-repl's pipeline. Briefly, clang-repl's incremental parser will parse the C++ code and produce a `PartialTranslationUnit` that contains a `TranslationUnitDecl` and a `llvm::Module`, then we feed the `llvm::Module` to the ORC JIT to get the result. That said no matter what, the first thing we do is always parsing. So let's take the simplest example and see what we got from the parser.

```
cat test.cpp
int X = printf("dummy");
```

we can dump its AST information by doing:
`clang++ -Xclang -ast-dump -fsyntax-only test.cpp`

```
`-VarDecl 0x5590669edc80 <line:3:1, col:23> col:5 X 'int' cinit
  `-CallExpr 0x5590669ede30 <col:9, col:23> 'int'
    |-ImplicitCastExpr 0x5590669ede18 <col:9> 'int (*)(const char *,
...)' <FunctionToPointerDecay>
    | `-DeclRefExpr 0x5590669eddc8 <col:9> 'int (const char *, ...)'
lvalue Function 0x5590669edae8 'printf' 'int (const char *, ...)'
    `-ImplicitCastExpr 0x5590669ede58 <col:16> 'const char *'
<ArrayToPointerDecay>
      `-StringLiteral 0x5590669edda8 <col:16> 'const char[6]' lvalue
"dummy"
```

We can observe that the init part of the `VarDecl` is a `CallExpr`, not its real value: 5. To know the actual value of `X`, we must execute the code, in our case JIT will take care of it.

But how can we track the change of variable `X`? Well, let's not overthink it and step back to where we originally at. What we really want to do is just the pseudo-code below:

```
[0] int X = printf("dummy");
[1] printValue(X);
```

Here's the pseudo-code for `printValue`:

```
template<typename T>
void printValue(const T& value) {
    std::cout << ConvertTypeToStr<T>() << " " << value << "\n";
 }
```

That said, all we really need to do is manually insert a function call that dump the value for us.

Code parsing

The first issue we should consider is how we should determine if the user wants to print the expression. Languages like Rust already give a good answer: We can omit the semicolon so clang-repl knows that this should be dealt with differently.

When we see the last character of the input is not a semicolon, we can turn on the value printing mode and start to do some magic.

Transform the AST

Just as mentioned above, this should be where the code injection happens. So in this stage, we have a `DeclRefExpr(X)`, like below:

```
DeclRefExpr 0x162a808 <line:5:2> 'int' lvalue Var 0x162a648 'X' 'int'
```

So the easiest approach we can think about is manually inserting some C++ code that does the printing job. However, it at least has one serious problem: Since we have to parse that code every time we try to print something, it's super expensive. So instead of sacrificing the performance, we can put the code that does the work externally, and only synthesize a `CallExpr` that only transfers the control out.

We can define an interface like the below:

```
void PrintValue(QualType* Ty, auto Val);
```

Do the real work

`PrintValue(QualType* Ty, auto Val)` just forwards things, now we have everything and should start printing the value. To abstract more, we can consider introducing a new class `Value` and making `print` become a behavior of it. Here's some pseudo-code.

```
class Value {
  std::variant<...> Data;
  QualType* Type; // no ownership
public:
  template<typename T>
  Value(QualType* Ty, T Val);
  void dump() const;
  void dump(llvm::raw_ostream& Out) const;
};

void PrintValue(QualType* Ty, auto Val) {
  Value Val = CreateValue(Ty, Val); // omit some details for now.
  Val.dump();
}
```

Timeline

Week 1-2	Get familiar with the code and learn more details on how we do this in Cling
Week 3-4	Make clang-repl capable parsing expressions (code without semicolon), but nop for now.
Week 5-6	Design the Value class that can store the type and its real value.
Week 7-8	Inject code after parsing and transforming the AST.
Week 9-12	Implement print for some basic primitive types.
Week 13-14	Polish the patch and send it to the Phabricator.
Week 15-16	Implement user-defined types of support.
Week 17-18	Implement the default way of printing STL facilities.
Week 19-20	Start looking at our additional deliverables if everything goes smoothly. We need to profile the usage of all preallocated source locations and find out why sparse vector doesn’t work.

Week 21-22	Communicate with mentors and upstream code owners and discuss possible ways to reduce memory usage based on our previous profiling results.
Week 23-24	Develop a patch and submit it to the Phabricator.
Week 25-26	Prepare a presentation

Additional Deliverables

Within the context of this project we plan to investigate how to mitigate performance issues coming from the way the C++ Modules are implemented in Clang models source locations. That would have a direct impact on the C++ Modules adoption in CMSSW. In addition, we will continue upstreaming essential patches from the Cling's LLVM forks into LLVM mainline.

About Me

I'm a software engineering undergraduate currently in my 3rd year of study. This year I contributed to the ROOT project as a GSoC contributor and focused on improving the performance of Clang modules usage in ROOT. I'm also a contributor to Clang and have already did a lot of contributions to clang-repl, including fixing existing bugs and adding new features. In my spare time, I like digging into the internals of Clang, so I have a good understanding of Clang AST.

Availability

~20 hours/week, more during college holidays.

TimeZone

CST(UTC+8000) 6 hours faster than EET.

Preferred working duration

December 1st 2022 - May 31st 2023

Personal Motivation

After participating in the GSoC this year, I got a better understanding of compiler internals. Also, I'm very happy to work in the compiler research team, and hope can pick up more exciting topics to work on. What's more, I have left some work unfinished this summer, and I really wish I can complete them.