Interpreted C++: Is that a thing?

A journey through LLVM/clang-based C++ JITting

Javier López-Gómez for the ROOT team

`using std::cpp`, 2024-04-25

About me

Javier Lopez-Gomez

jalopezg-git   https://jalopezg.dev/

javier.lopez.gomez AT proton.me

Summary of the last 5+ years (compilers-wise)...

- 2017–2020: PhD in Computer Science and Technology (ARCOS-UC3M)
  - Prototype implementation of C++ contracts (clang)
  - Research internship at CERN in 2019: Definition shadowing in cling
- 2020–2023: Senior Fellow (SFT, CERN)
  - More cling – but also RNTuple and general contributions to the ROOT project
- **2024–(currently): Senior Compiler Engineer (Zimperium, Inc.)**
  - Software obfuscation that operates directly AArch64 binaries

Contents

Introduction

Spoiler: how does C++ interpretation look like?

You can do

```
[cling]$  template <typename T>
  T f(T a, T b) {
    return a + b;
  }
[cling]$  f(42, 6)
(int) 48
```

And then

```
[cling]$  std::string S{"Hello,"};
[cling]$  f(S, std::string{"␣world!"})
(std::basic_string<char, std::char_traits<char>, std::allocator<char> >) "Hello,␣world!"
```

But also the abomination below

```
[cling]$  std::vector<int> f{1, 2, 3};
[cling]$  f
(std::vector<int> &) { 1, 2, 3 }
```

Spoiler: how does C++ interpretation look like?

You can do

```
[cling]$  template <typename T>
  T f(T a, T b) {
    return a + b;
  }
[cling]$  f(42, 6)
(int) 48
```

And then

```
[cling]$  std::string S{"Hello,"};
[cling]$  f(S, std::string{"␣world!"})
(std::basic_string<char, std::char_traits<char>, std::allocator<char> >) "Hello,␣world!"
```

But also the abomination below

```
[cling]$  std::vector<int> f{1, 2, 3};
[cling]$  f
(std::vector<int> &) { 1, 2, 3 }
```

Spoiler: how does C++ interpretation look like?

You can do

```
[cling]$  template <typename T>
  T f(T a, T b) {
    return a + b;
  }
[cling]$  f(42, 6)
(int) 48
```

And then

```
[cling]$  std::string S{"Hello,"};
[cling]$  f(S, std::string{"␣world!"})
(std::basic_string<char, std::char_traits<char>, std::allocator<char> >) "Hello,␣world!"
```

But also the abomination below

```
[cling]$  std::vector<int> f{1, 2, 3};
[cling]$  f
(std::vector<int> &) { 1, 2, 3 }
```

Why bothering to interpret C++?

- Many languages already offer a REPL (Read-Eval-Print-Loop) even if not designed to be interpreted, e.g. C#
- It aids a lot while learning the language: try things out!
- Iterative / exploratory prototyping
- Write 'scripts' that make use of C/C++ libraries
- ...

Current state of affairs

- Cling is built on Clang and LLVM 13 (enabling support for C++20)
- CUDA support
- Allows loading an external library (`.so` / `.dll`) and get access to its symbols, e.g. call a function
- Debugging and profiling of JITed code
- Undo steps
- Protection against invalid memory accesses, e.g. dereferencing a pointer that points to unmapped memory

Current state of affairs

But also some features that one would expect from an interpreter (even if that's not ISO C++)...

- Top-level statements
- Print the result of expression evaluation
- `auto` synthesizing, i.e. `foo = 42.0;` is equivalent to `auto foo = 42.0;` (if `foo` not declared before) DEPRECATED
- Support for redefining entities, e.g.

```
int foo = 0;
std::string foo{"Hello!"};
```

Foundations

- LLVM and clang to the rescue!

- LLVM gives all the infrastructure required to build a compiler
  - Basic data structures, e.g. `llvm::SmallVector`, or `llvm::Twine`
  - An intermediate representation (LLVM IR)
  - Lowering to machine code for many targets: x86_64, ARM7, AArch64, RISC-V, etc.
  - Machine-dependent and machine-independent optimization passes
  - Generation of debug information

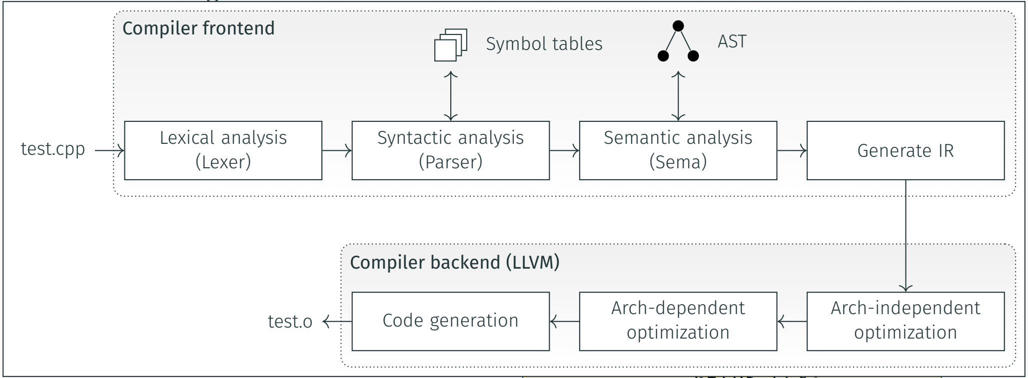
LLVM

- LLVM and clang to the rescue!
- LLVM gives all the infrastructure required to build a compiler
  - Basic data structures, e.g. `llvm::SmallVector`, or `llvm::Twine`
  - An intermediate representation (LLVM IR)
  - Lowering to machine code for many targets: x86_64, ARM7, AArch64, RISC-V, etc.
  - Machine-dependent and machine-independent optimization passes
  - Generation of debug information

And more! See `https://llvm.org/`.

LLVM

- LLVM and clang to the rescue!



And more! See https://

LLVM IR

- LLVM IR may have 3 different representations: **in-memory** structures, assembly-like **plain-text**, or **serialized** form
- Let's play a bit to build the LLVM IR for a simple function!

LLVM IR

```
LLVMContext C;
auto builder = std::make_unique<IRBuilder<>>(C);
auto M = std::make_unique<Module>("main", C);

auto i32 = builder->getInt32Ty();
auto funcTy = FunctionType::get(i32, {i32, i32}, /*isVarArg=*/false);
auto func = Function::Create(funcTy, GlobalValue::LinkageTypes::ExternalLinkage, "func", *M);
auto BB = BasicBlock::Create(C, "entry", func);
builder->SetInsertPoint(BB);
auto addVal = builder->CreateAdd(func->getArg(0), func->getArg(1));
builder->CreateRet(addVal);
M->print(errs(), /*AAW=*/nullptr);
```

LLVM IR

```
LLVMContext C;
auto builder = std::make_unique<IRBuilder<>>(
auto M = std::make_unique<Module>("main", C);

auto i32 = builder->getInt32Ty();
auto funcTy = FunctionType::get(i32, {i32, i3
auto func = Function::Create(funcTy, GlobalVa                        M);
auto BB = BasicBlock::Create(C, "entry", func
builder->SetInsertPoint(BB);
auto addVal = builder->CreateAdd(func->getArg
builder->CreateRet(addVal);
M->print(errs(), /*AAW=*/nullptr);
```

```
; ModuleID = 'main'
source_filename = "main"

define i32 func(i32 %0, i32 %1) {
entry:
    %2 = add i32 %0, %1
    ret i32 %2
}
```

LLVM can also JIT IR to current target's machine code...[1]

```
int main(int argc, char *argv[]) {
  using namespace llvm;
  InitializeNativeTarget();
  InitializeNativeTargetAsmPrinter();

  /* CREATE IR AS IN PREVIOUS SLIDE */

  auto EE = EngineBuilder(std::move(M)).setEngineKind(llvm::EngineKind::JIT).create();

  using FuncPtr_t = uint32_t (*)(uint32_t, uint32_t);
  auto pFunc = (FuncPtr_t)EE->getFunctionAddress("func");
  auto ret = pFunc(42, 7);
  errs() << "\nfunc() returned " << ret << "\n";

  return 0;
}
```

[1] Full example: https://github.com/jalopezg-git/slides-using_stdcpp_2014/blob/master/code/llvm-ir.cpp

LLVM JIT / ORC

LLVM can also JIT IR to current target's machine code...[1]

```
int main(int argc, char *argv[]) {
  using namespace llvm;
  InitializeNativeTarget();
  InitializeNativeTargetAsmPrinter();

  /* CREATE IR AS IN PREVIOUS SLIDE */

  auto EE = EngineBuilder(std::move(M)).setEngineKind(llvm::EngineKind::JIT).create();

  using FuncPtr_t = uint32_t (*)(uint32_t, uint32_t);
  auto pFunc = (FuncPtr_t)EE->getFunctionAddress("func");
  auto ret = pFunc(42, 7);
  errs() << "\nfunc()␣returned␣" << ret << "\n";

  return 0;
}
```

```
func() returned 49
```

[1] Full example: https://github.com/jalopezg-git/slides-using_stdcpp_2014/blob/master/code/llvm-ir.cpp

Clang as a library

- Clang is basically a frontend that parses C / C++ / ObjectiveC and generates LLVM IR, i.e.
  - It does lexical / gramatical / semantic analysis on the source code + builds an AST
  - LLVM takes over from there
- E.g., the simple code...

```c
extern int puts(const char *s);

int main(void) {
    puts("Hello, world!");
    return 0;
}
```

[2] Get this with `clang -c -Xclang -ast-dump -o /dev/stdout input.c`

[3] Get this with `clang -S -emit-llvm -o /dev/stdout input.c`

Clang as a library

```c
extern int puts(const char *s);

int main(void) {
    puts("Hello,␣world!");
    return 0;
}
```

Has AST representation[2]

```
|-FunctionDecl 0x563c61b0bda0 </tmp/simple.c:1:1, col:30> col:12 used puts 'int (const char *)' extern
| `-ParmVarDecl 0x563c61b0bcd0 <col:17, col:29> col:29 s 'const char *'
`-FunctionDecl 0x563c61b0bf60 <line:3:1, line:6:1> line:3:5 main 'int (void)'
  `-CompoundStmt 0x563c61b0c188 <col:16, line:6:1>
    |-CallExpr 0x563c61b0c100 <line:4:5, col:25> 'int'
    | |-ImplicitCastExpr 0x563c61b0c0e8 <col:5> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x563c61b0c038 <col:5> 'int (const char *)' Function 0x563c61b0bda0 'puts' 'int (const char *)'
    | `-ImplicitCastExpr 0x563c61b0c140 <col:10> 'const char *' <NoOp>
    |   `-ImplicitCastExpr 0x563c61b0c128 <col:10> 'char *' <ArrayToPointerDecay>
    |     `-StringLiteral 0x563c61b0c098 <col:10> 'char[14]' lvalue "Hello, world!"
    `-ReturnStmt 0x563c61b0c178 <line:5:5, col:12>
      `-IntegerLiteral 0x563c61b0c158 <col:12> 'int' 0
```

[2] Get this with `clang -c -Xclang -ast-dump -o /dev/stdout input.c`

[3] Get this with `clang -S -emit-llvm -o /dev/stdout input.c`

Clang as a library

```
extern int puts(const char *s);

int main(void) {
    puts("Hello,␣world!");
    return 0;
}
```

And LLVM IR representation[3]

```
; ModuleID = '/tmp/simple.c'
source_filename = "/tmp/simple.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [14 x i8] c"Hello,␣world!\00", align 1

; Function Attrs: nofree nounwind sspstrong uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 @puts(ptr noundef nonnull dereferenceable(1) @.str)
  ret i32 0
}

; Function Attrs: nofree nounwind
declare noundef i32 @puts(ptr nocapture noundef readonly) local_unnamed_addr #1
```

[2] Get this with `clang -c -Xclang -ast-dump -o /dev/stdout input.c`

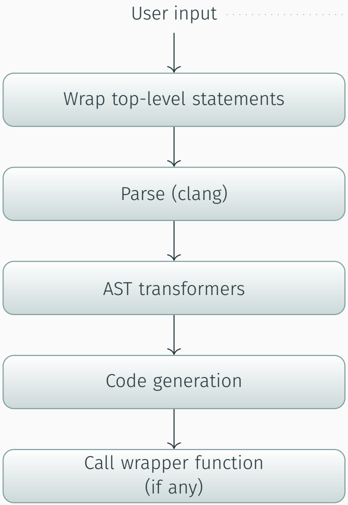
[3] Get this with `clang -S -emit-llvm -o /dev/stdout input.c`

- And it offers libTooling, libclang-cpp, and libclang!
- Meaning we can mostly reuse this[4] and only write a layer on top that does “impedance matching” between ISO and interpreted C++

[4] Modulo some patches required to clang.

The cling C++ interpreter

Cling's pipeline




```
[cling]$  puts("Hello,␣world!");
```

Cling's pipeline

User input

Wrap top-level statements

Parse (clang)

AST transformers

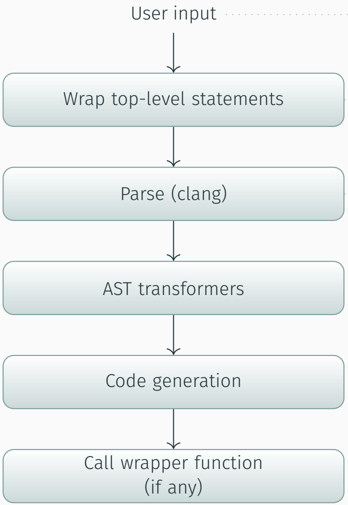
Code generation

Call wrapper function
(if any)

```
[cling]$  puts("Hello,␣world!");
```

```
void __cling_Un1Qu30(void *) {
    puts("Hello,␣world!");
}
```

Cling's pipeline




```
[cling]$  puts("Hello,␣world!");
```

```
void __cling_Un1Qu30(void *) {
    puts("Hello,␣world!");
}
```

```
`-FunctionDecl 0x5648f4bc0900 <input_line_4:1:1, line:4:1> line:1:6 __cling_Un1Qu30 'void (void *)'
  |-ParmVarDecl 0x5648f4bc0850 <col:22, col:28> col:28 vpClingValue 'void *'
  `-CompoundStmt 0x5648f4bc0b08 <col:42, line:4:1>
    |-CallExpr 0x5648f4bc0aa0 <line:2:2, col:22> 'int'
    | |-ImplicitCastExpr 0x5648f4bc0a88 <col:2> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x5648f4bc0a38 <col:2> 'int (const char *)' lvalue Function 0x5648f4519c80 'puts' 'int (const cha
r *)'
    | `-ImplicitCastExpr 0x5648f4bc0ac8 <col:7> 'const char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x5648f4bc0a10 <col:7> 'const char [14]' lvalue "Hello, world!"
    `-NullStmt 0x5648f4bc0ae0 <line:3:1>
```

Cling's pipeline

User input

```
[cling]$ puts("Hello,␣world!");
```

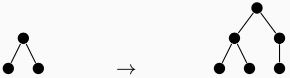
Wrap top-level statements

```
void __cling_Un1Qu30(void *) {
    puts("Hello,␣world!");
}
```

Parse (clang)

```
`-FunctionDecl 0x5648f4bc0900 <input_line_4:1:1, line:4:1> line:1:6 __cling_Un1Qu30 'void (void *)'
  |-ParmVarDecl 0x5648f4bc0850 <col:22, col:28> col:28 vpClingValue 'void *'
  `-CompoundStmt 0x5648f4bc0b08 <col:42, line:4:1>
    |-CallExpr 0x5648f4bc0aa0 <line:2:2, col:22> 'int'
    | |-ImplicitCastExpr 0x5648f4bc0a88 <col:2> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x5648f4bc0a38 <col:2> 'int (const char *)' lvalue Function 0x5648f4519c80 'puts' 'int (const cha
r *)'
    | `-ImplicitCastExpr 0x5648f4bc0ac8 <col:7> 'const char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x5648f4bc0a10 <col:7> 'const char [14]' lvalue "Hello, world!"
    `-NullStmt 0x5648f4bc0ae0 <line:3:1>
```

AST transformers



Code generation

Call wrapper function (if any)

Cling’s pipeline

User input

↓

Wrap top-level statements

↓

Parse (clang)

↓

AST transformers

↓

Code generation

↓

Call wrapper function
(if any)

```
[cling]$  puts("Hello,␣world!");
```

```
void __cling_Un1Qu30(void *) {
    puts("Hello,␣world!");
}
```

```
`-FunctionDecl 0x5648f4bc0900 <input_line_4:1:1, line:4:1> line:1:6 __cling_Un1Qu30 'void (void *)'
  |-ParmVarDecl 0x5648f4bc0850 <col:22, col:28> col:28 vpClingValue 'void *'
  `-CompoundStmt 0x5648f4bc0b08 <col:42, line:4:1>
    |-CallExpr 0x5648f4bc0aa0 <line:2:2, col:22> 'int'
    | |-ImplicitCastExpr 0x5648f4bc0a88 <col:2> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x5648f4bc0a38 <col:2> 'int (const char *)' lvalue Function 0x5648f4519c80 'puts' 'int (const cha
r *)'
    | `-ImplicitCastExpr 0x5648f4bc0ac8 <col:7> 'const char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x5648f4bc0a10 <col:7> 'const char [14]' lvalue "Hello, world!"
    `-NullStmt 0x5648f4bc0ae0 <line:3:1>
```

```
@.str = private unnamed_addr constant [14 x i8] c"Hello,␣world!\00", align 1

; Function Attrs: nofree nounwind sspstrong uwtable
define dso_local void @__cling_Un1Qu30(ptr nocapture noundef readnone %0) local_unnamed_addr {
  %2 = tail call i32 @puts(ptr noundef nonnull dereferenceable(1) @.str)
  ret void
}
```

Cling's pipeline

User input

```
[cling]$  puts("Hello,␣world!");
```

Wrap top-level statements

```
void __cling_Un1Qu30(void *) {
    puts("Hello,␣world!");
}
```

Parse (clang)

```
`-FunctionDecl 0x5648f4bc0900 <input_line_4:1:1, line:4:1> line:1:6 __cling_Un1Qu30 'void (void *)'
  |-ParmVarDecl 0x5648f4bc0850 <col:22, col:28> col:28 vpClingValue 'void *'
  `-CompoundStmt 0x5648f4bc0b08 <col:42, line:4:1>
    |-CallExpr 0x5648f4bc0aa0 <line:2:2, col:22> 'int'
    | |-ImplicitCastExpr 0x5648f4bc0a88 <col:2> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x5648f4bc0a38 <col:2> 'int (const char *)' lvalue Function 0x5648f4519c80 'puts' 'int (const cha
r *)'
    | `-ImplicitCastExpr 0x5648f4bc0ac8 <col:7> 'const char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x5648f4bc0a10 <col:7> 'const char [14]' lvalue "Hello, world!"
    `-NullStmt 0x5648f4bc0ae0 <line:3:1>
```

AST transformers

Code generation

```
@.str = private unnamed_addr constant [14 x i8] c"Hello,␣world!\00", align 1

; Function Attrs: nofree nounwind sspstrong uwtable
define dso_local void @__cling_Un1Qu30(ptr nocapture noundef readnone %0) local_unnamed_addr {
  %2 = tail call i32 @puts(ptr noundef nonnull dereferenceable(1) @.str)
  ret void
}
```

Call wrapper function (if any)

[If ∃ top-level statement, get a pointer to the wrapper function and do indirect call]

Cling's pipeline

User input

```
[cling]$ puts("Hello,␣world!");
```

Wrap top-level statements

```
void __cling_Un1Qu30(void *) {
    puts("Hello,␣world!");
}
```

Parse (clang)

```
`-FunctionDecl 0x5648f4bc0900 <input_line_4:1:1, line:4:1> line:1:6 __cling_Un1Qu30 'void (void *)'
  |-ParmVarDecl 0x5648f4bc0850 <col:22, col:28> col:28 vpClingValue 'void *'
  `-CompoundStmt 0x5648f4bc0b08 <col:42, line:4:1>
    |-CallExpr 0x5648f4bc0aa0 <line:2:2, col:22> 'int'
    | |-ImplicitCastExpr 0x5648f4bc0a88 <col:2> 'int (*)(const char *)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x5648f4bc0a38 <col:2> 'int (const char *)' lvalue Function 0x5648f4519c80 'puts' 'int (const cha
r *)'
    | `-ImplicitCastExpr 0x5648f4bc0ac8 <col:7> 'const char *' <ArrayToPointerDecay>
    |   `-StringLiteral 0x5648f4bc0a10 <col:7> 'const char [14]' lvalue "Hello, world!"
    `-NullStmt 0x5648f4bc0ae0 <line:3:1>
```

AST transformers

Code generation

```
local_unnamed_addr {
```

Call wrapper function (if any)

[If a top-level statement, get a pointer to the wrapper function and do indirect call]

- Deferred (implicit) template instantiations **must** be emitted; we do that by forcing a end-of-TU event!
- CodeGen: also involves linking (external symbol resolution, etc.)

Transactions

- AST is built incrementally
- **Transaction:** declarations that were parsed and emitted in a single step
  - User-provided declarations
  - Implicit template instatiations
  - Deserialized declarations from a C++ module
- And allows undoing it. That's useful, e.g. after a failed parse

Extensions

- Most extensions are implemented as an AST transformer
- Currently, there is support for
  - `auto` synthesizing, e.g. `foobar = 42.0f;`
  - Protection against invalid pointer deferencing, e.g.

```
[cling]$  *((int *)0xff00ff00) = 0;
Error in <HandleInterpreterException>: Trying to access a pointer that points to an
    invalid memory address.
Execution of your code was aborted.
ROOT_prompt_6:1:2: warning: invalid memory pointer passed to a callee:
*((int *)0xff00ff00) = 0;
 ^~~~~~~~~~~~~~~~~~~
```

  - Shadowing of definitions

```
[cling]$  int foobar = 0;
[cling]$  std::string foobar() { return "A␣string!"; }
```

  - Printing / capturing the value of an expression

```
[cling]$  foobar()
(std::string) "A␣string!"
```

Extensions

- Most extensions are implemented as an AST transformer
- Currently, there is support for
  - `auto` synthesizing, e.g. `foobar = 42.0f;`
  - Protection against invalid pointer deferencing, e.g.

```
[cling]$  *((int *)0xff00ff00) = 0;
Error in <HandleInterpreterException>: Trying to access a pointer that points to an
     invalid memory address.
Execution of your code was aborted.
ROOT_prompt_6:1:2: warning: invalid memory pointer passed to a callee:
*((int *)0xff00ff00) = 0;
 ^~~~~~~~~~~~~~~~~~~
```

  - Shadowing of definitions

```
[cling]$  int foobar = 0;
[cling]$  std::string foobar() { return "A␣string!"; }
```

  - Printing / capturing the value of an expression

```
[cling]$  foobar()
(std::string) "A␣string!"
```

Extensions

- Most extensions are implemented as an AST transformer
- Currently, there is support for
  - `auto` synthesizing, e.g. `foobar = 42.0f;`
  - Protection against invalid pointer deferencing, e.g.

```
[cling]$  *((int *)0xff00ff00) = 0;
Error in <HandleInterpreterException>: Trying to access a pointer that points to an
     invalid memory address.
Execution of your code was aborted.
ROOT_prompt_6:1:2: warning: invalid memory pointer passed to a callee:
*((int *)0xff00ff00) = 0;
 ^~~~~~~~~~~~~~~~~~~
```

  - Shadowing of definitions

```
[cling]$  int foobar = 0;
[cling]$  std::string foobar() { return "A␣string!"; }
```

  - Printing / capturing the value of an expression

```
[cling]$  foobar()
(std::string) "A␣string!"
```

Extensions

- Most extensions are implemented as an AST transformer
- Currently, there is support for
  - `auto` synthesizing, e.g. `foobar = 42.0f;`
  - Protection against invalid pointer deferencing, e.g.

```
[cling]$  *((int *)0xff00ff00) = 0;
Error in <HandleInterpreterException>: Trying to access a pointer that points to an
     invalid memory address.
Execution of your code was aborted.
ROOT_prompt_6:1:2: warning: invalid memory pointer passed to a callee:
*((int *)0xff00ff00) = 0;
 ^~~~~~~~~~~~~~~~~~~
```

  - Shadowing of definitions

```
[cling]$  int foobar = 0;
[cling]$  std::string foobar() { return "A␣string!"; }
```

  - Printing / capturing the value of an expression

```
[cling]$  foobar()
(std::string) "A␣string!"
```

Debugging / Profiling of JIT'ed code

Cling also allows debugging JITed code and offers integration with Linux's `perf`, e.g.

- A breakpoint on interpreted code can be set and step-into after each statement
- It can generate a symbol file for `perf` – Can be used together with Flamegraph[5]!

[5] Flamegraph: `https://github.com/brendangregg/FlameGraph`

The future of cling: `clang-repl`

- Cling proved to perform okay in the context of the larger ROOT project at CERN
- Let's upstream the foundations of it back to the LLVM community so that
  - The whole community can benefit from it
  - Maintenance is easier in the long term
- `clang-repl`: already in recent versions of LLVM — Thanks, Vassil!
- Slightly different to the design of cling, e.g. modeling of top-level statements is much more rubust

Closing words

Closing words

Key ideas to take home

- Cling enables incremental C++ compilation and JITting
  - It would not be possible without the solid framework provided by LLVM and clang
- Convenient integration with Jupyter notebook via `xeus-cling`
- Try it!

For the curious

- cppyy / libinterop provide interoperability with other languages
  - Enabling crazy things such as injecting the C++ definition of a type `T` and creating an object of type `T` from Python
  - Or even crazier: on-the-fly template instantiation, e.g. a `std::vector`$< T >$ [a][b]

[a] https://cppyy.readthedocs.io/
[b] https://compiler-research.org/libinterop/

How can I get it?

`https://github.com/root-project/cling/`

- If you have ‘llvm‘ installed locally, try `clang-repl`

Thank you!

Backup

Debugging / Profiling of JIT'ed code: HOWTO

Cling also allows debugging JITed code and offers integration with Linux's **`perf`**, e.g.

- A breakpoint on interpreted code can be set and step-into after each statement

```
$ export CLING_DEBUG=1
$ gdb --args cling /tmpl/simple.C
(gdb) break simple
(gdb) r
Starting program: cling /tmp/simple.C

Breakpoint 1, simple () at /tmp/simple.C:4
4    std::cout << "Hello,␣world!" << std::endl;
(gdb) q
```

- It can generate a symbol file for **`perf`** – Can be used together with Flamegraph[6]!

```
$ export CLING_PROFILE=1
$ perf record -g -e cycles -- cling /tmp/simple.C
```

[6] Flamegraph: `https://github.com/brendangregg/FlameGraph`

Debugging / Profiling of JIT'ed code: flamegraph