



EP-SFT

Software Frameworks and Tools

EP-SFT

Software Frameworks and Tools



PRINCETON
UNIVERSITY

Hands-on Using Clang as a library: Incremental Compilation and Language Interoperability

Aaron Jomy

for the Compiler-Research group, ROOT Team @CERN

EuroLLVM 2026, Dublin

- ❑ Learn how to use Clang as a library
- ❑ Build a simple C++ REPL with `clang::Interpreter`
- ❑ See how we can bridge compiled and interpreted code
- ❑ Develop building blocks for a basic Compiler-as-a-Service
- ❑ Use the CaaS layer to bridge Python and C++

Requirements:

LLVM21 installation, CMake ≥ 3.13 and a C++17 compiler

Python3 for exercises 4 and 5.

Build:

```
git clone https://github.com/compiler-research/eurollvm26-tutorial.git
cd eurollvm26-tutorial && mkdir build && cd build
cmake -DLLVM_DIR=/usr/lib/llvm-21/lib/cmake/llvm \
      -DClang_DIR=/usr/lib/llvm-21/lib/cmake/clang ..
make
```

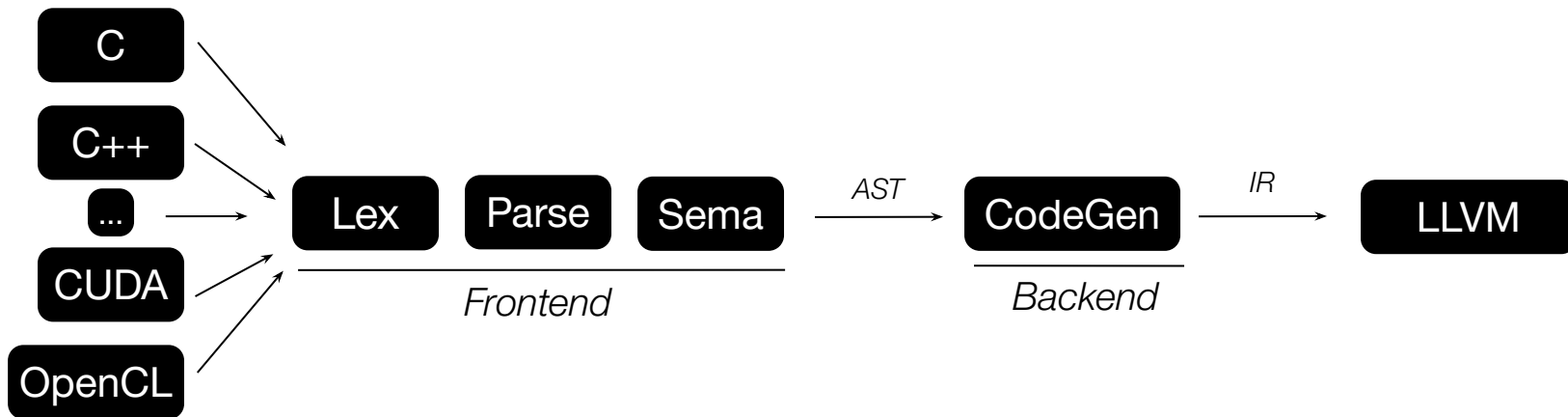
Substitute whichever /usr/lib/llvm-21 with your install:

- mac: `$(brew --prefix llvm@21)/lib/cmake/...`
- Source build: `path/to/build/lib/cmake/...`

- Clang is a compiler which supports C, C++, Objective-C, and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript, CUDA, SYCL, and HIP frameworks.
- Just like LLVM, Clang is built by a set of reusable components and can be used as a library.



Clang takes input pipes it through a frontend and a backend and produces machine code



ex1: Using Clang as a Library

AST

Tooling

```
#include "clang/AST/Comment.h"
#include "clang/AST/DeclTemplate.h"
#include "clang/Tooling/Tooling.h"
...
auto ASTU = tooling::buildASTFromCodeWithArgs(Code,
{"-std=c++20"});
auto &C = ASTU->getASTContext();
auto *TU = C.getTranslationUnitDecl();
TU->dump();
```

Run the compiler on
given code.

```
| -ClassTemplateDecl 0x7f83db895f48 <input.cc:4:1, line:5:40> col:8
ComplexNumber
| | -TemplateTypeParmDecl 0x7f83db895dd0 <line:4:10, col:19> col:19
| | -CXXRecordDecl 0x7f83db895e98 <line:5:1, col:40> col:8 struct ComplexNumber
| | ...
| | | -FullComment 0x7f83dc00c200 <line:2:4, col:52>
| | | | -ParagraphComment 0x7f83dc00c1d0 <col:4, col:52>
| | | | | -TextComment 0x7f83dc00c1a0 <col:4, col:52>
Text=" This is the documentation for the ComplexNumber."
```

ex2: A Simple C++ REPL

Frontend

Interpreter

ex2: Incremental Compilation in Clang

// Initialize our builder class

```
clang::IncrementalCompilerBuilder CB;
```

-fplugin=my_plugin.so

```
CB.SetCompilerArgs({"-std=c++20"}); // pass `-xc` for C.
```

Standard compiler flags

// Create the incremental compiler instance

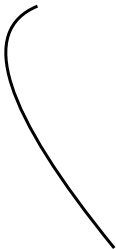
```
auto CI = ExitOnErr(CB.CreateCpp());
```

Creates an incremental
compiler instance

// Create the interpreter instance

```
auto Interp = ExitOnErr(Interpreter::create(std::move(CI)));
```

```
llvm::LineEditor LE("eurolvm26-repl");  
while (std::optional<std::string> Line = LE.readLine()) {  
    if (*Line == "%quit")  
        break;  
    if (auto Err = Interp->ParseAndExecute(*Line)) {  
        ...  
    }  
}
```



Adds a partial translation unit

ex3: Bridging Compiled and Interpreted C++

Frontend

Interpreter

- Transporting results from compiled to interpreted code and back
- Frontend support for taking control of object lifetime
- Frontend support handling C++ semantics such as non-assignable types

An efficient, ref-counted, small-buffer optimized facility to transport results
Supports pretty printing and type conversion operations

// Create a value to store the transport the result from JIT.

clang::Value V;

Interp->ParseAndExecute(R"(extern "C" int sq(int x){return x*x;}
sq(12))", &V);

printf("From JIT: square(12)=%d\n", V.getInt());

// Or just get a function pointer and call it in compiled code:

auto SymAddr = ExitOnErr(Interp->getSymbolAddress("sq"));

auto sqPtr = SymAddr.toPtr<int(*)>();

printf("From compiled code: square(13)=%d\n", sqPtr(13));

ex4: Building a Compiler-as-a-Service

Frontend

Interpreter

Programmatically Instantiating C++ Templates

Frontend

Interpreter

- Create a library containing CaaS building blocks and create an interface for it
- Build a C binary to programmatically instantiate and call a C++ template function
- Teach Python to express C++: `cpp.myclass.myfunc["int"]`

Programmatically Instantiating C++ Templates

Frontend

Interpreter

```
#include <typeinfo>

class A {};
struct B {
    template <typename T>
    static void callme(T) {
        printf(" Instantiated with [%s] \n", typeid(T).name());
    }
};
```

```
void Clang_Parse(const char* Code);
```

```
void* Clang_LookupName(...);
```

————— Returns a declaration given a string

```
unsigned Clang_GetFunctionAddress(...);
```

————— Returns the in-memory address of JIT'd function

```
void* Clang_CreateObject(...);
```

————— Allocates storage for a declaration

```
void* Clang_InstantiateTemplate(...);
```

```
int main(int argc, char **argv) {
    Clang_Parse(Code);
    Decl_t TemplatedClass = Clang_LookupName("B", /*Context=*/0);

    // Instantiate B::callme with the given types
    Decl_t Instantiation = Clang_InstantiateTemplate(TemplatedClass, "callme", "A");

    // Get the symbol to call
    typedef void (*fn_def)(void*);
    fn_def callme_fn_ptr = (fn_def) Clang_GetFunctionAddress(Instantiation);

    // Create objects of type A
    Decl_t T = Clang_LookupName("A", /*Context=*/0);
    void* NewA = Clang_CreateObject(T);

    callme_fn_ptr(NewA);
}
```

Let's build a Python wrapper for the the functions in ex4-lib:

- InteropLayerWrapper — Responsible for instantiating a C++ template
- TemplateWrapper — Finds and matches C++ template arguments
- CallCppFunc — Invokes the C++ template function

Instantiating C++ templates on-demand with Python

```
import ctypes
```

```
libInterop = ctypes.CDLL("ex4-lib.so")
```

```
# tell ctypes which function to call and the expected param/return types.
```

```
_cpp_compile = libInterop.Clang_Parse
```

```
_cpp_compile.argtypes = [ctypes.c_char_p]
```

```
void Clang_Parse(const char* Code);
```

```
def cpp_compile(arg):
```

```
    return _cpp_compile(arg.encode("ascii"))
```

```
# define some classes to play with
```

```
cpp_compile(r"""\
```

```
class A {};
```

```
struct B : public A {
```

```
    template<typename T> static void callme(T*) {
```

```
        printf("Template: %s\n", typeid(T).name());
```

```
    }
```

```
};
```

```
""")
```

```
class InteropLayerWrapper:
```

```
    # Provide a Python wrapper over the interop layer
```

```
    _get_scope = libInterop.Clang_LookupName
```

```
    _get_scope.restype = ctypes.c_size_t
```

```
    _get_scope.argtypes = [ctypes.c_char_p]
```

```
    _construct = libInterop.Clang_CreateObject
```

```
    _construct.restype = ctypes.c_void_p
```

```
    _construct.argtypes = [ctypes.c_size_t]
```

```
    _get_template_ct = libInterop.Clang_InstantiateTemplate
```

```
    _get_template_ct.restype = ctypes.c_size_t
```

```
    _get_template_ct.argtypes = [ctypes.c_size_t, ctypes.c_char_p, ctypes.c_char_p]
```

```
def _get_template(self, scope, name, args):
```

```
    return self._get_template_ct(scope, name.encode("ascii"), args.encode("ascii"))
```

```
def get_scope(self, name):
```

```
    return self._get_scope(name.encode("ascii"))
```

```
void* Clang_LookupName(...);
```

```
void* Clang_CreateObject(...);
```

```
void* Clang_InstantiateTemplate(...);
```

```
class TemplateWrapper:
```

```
    # Find, instantiate and invoke a template function given arguments
```

```
    def __init__(self, scope, name):
```

```
        self._scope = scope
```

```
        self._name  = name
```

```
    def __getitem__(self, *args, **kwds):
```

```
        # Look up the template and return the overload
```

```
        return gIL.get_template(
```

```
            self._scope, self._name, tmpl_args = args)
```

—————→ Overloads index bracket access for array-like entries
my_list[2] calls my_list.__getitem__(2) to return the element at index 2, but here we replace this functionality with a way to specialize our templates as an analog to <>

```
    def __call__(self, *args, **kwds):
```

```
        # find the appropriate overload
```

```
        ol = self.get_template(self._scope, self._name, tpargs = [type(a) for a in args])
```

```
        # Call actual method
```

```
        ol(*args, **kwds)
```

```
class CallCppFunc:
    # Responsible for calling low-level function pointers.
    _get_funcptr = libInterop.Clang_GetFunctionAddress
    _get_funcptr.restype = ctypes.c_void_p
    _get_funcptr.argtypes = [ctypes.c_size_t]                unsigned Clang_GetFunctionAddress(...);

    def __init__(self, func):
        # Ideally this should go through the interop layer to know
        # whether to pass pointer, reference, or value of which type etc.
        proto = ctypes.CFUNCTYPE(None, ctypes.c_void_p)
        self._funcptr = proto(self._get_funcptr(func))

    def __call__(self, *args, **kwargs):
        # See the comment above.
        a0 = ctypes.cast(args[0].cppobj, ctypes.POINTER(ctypes.c_void_p))
        return self._funcptr(a0)
```

```
# Initialize our C++ interoperability layer wrapper
```

```
gIL = InteropLayerWrapper()
```

```
# Create a couple of types to play with
```

```
CppA = type('A', (), {  
    'handle' : gIL.get_scope('A'),  
    '__new__' : cpp_allocate})
```

```
h = gIL.get_scope('B')  
CppB = type('B', (A,), {  
    'handle' : h,  
    '__new__' : cpp_allocate,  
    'callme' : TemplateWrapper(h, 'callme')})
```

```
# Connect to C++ classes
```

```
a = CppA()
```

```
b = CppB()
```

```
# Explicit template instantiation and execution
```

```
b.callme['A'](a)
```

```
# Implicit template instantiation and execution
```

```
b.callme(b)
```

ex5: Transporting clang::Value to Python

Frontend

Interpreter

- In ex4 we could instantiate and call C++ templates from Python — but we couldn't read typed results back
- What if we parse, execute, and capture the result as a typed primitive via `clang::Value`?
- Goal: Python wrapper gets typed return values without writing a ctypes prototype per signature

```
class Math {  
public:  
    template<typename T> T pow2(T x)      { return x * x; }  
    template<typename T> T add(T a, T b) { return a + b; }  
};
```

```
typedef enum {  
    Clang_Value_Void = 0, Clang_Value_Bool,  
    Clang_Value_Int,      Clang_Value_LongLong,  
    Clang_Value_Double,   Clang_Value_Ptr,  
    Clang_Value_Other,    Clang_Value_Error  
} Clang_ValueKind;
```

`Clang_ValueKind Clang_Eval(const char* Expr);` → Wraps `Interpreter::ParseAndExecute(Expr, &Value)`
The tag tells the caller which accessor to read

```
int      Clang_Value_GetInt(void);  
double   Clang_Value_GetDouble(void);  
long long Clang_Value_GetLongLong(void);  
void *    Clang_Value_GetPtr(void);
```

→ Expose the `clang::Value` result

```
class ValueEval:
    _eval = libInterop.Clang_Eval
    _eval.restype = ctypes.c_int
    _eval.argtypes = [ctypes.c_char_p]

    _VOID, _BOOL, _INT, _LONGLONG, _DOUBLE, _PTR, _OTHER, _ERROR = range(8)

    def __init__(self):
        self._readers = {
            self._BOOL: (libInterop.Clang_Value_GetBool,      ctypes.c_int,      bool),
            self._INT:  (libInterop.Clang_Value_GetInt,       ctypes.c_int,      int),
            self._LONGLONG: (libInterop.Clang_Value_GetLongLong, ctypes.c_longlong, int),
            self._DOUBLE: (libInterop.Clang_Value_GetDouble,  ctypes.c_double,   float),
        }

    def __call__(self, expr):
        kind = self._eval(expr.encode("ascii"))
        if kind in (self._VOID, self._OTHER):
            return None
        fn, _, cast = self._readers[kind]
        return cast(fn())
```

makes a `ValueEval` instance callable: takes a C++ expression, dispatches on the kind, returns a typed Python primitive

```
class TemplateWrapper:
    def __init__(self, scope_name, method):
        self._scope_name = scope_name
        self._method = method

    def __getitem__(self, key):
        types = key if isinstance(key, str) else ', '.join(key)
        return _Specialised(self._scope_name, self._method, types)
```

```
class _Specialised:
    def __call__(self, *args):
        return gEval('{}{{{}}}.{}<{}>({})'.format(
            self._scope_name, self._method, self._type_args,
            ', '.join(repr(a) for a in args)))
```

→ Build the expression and call `ValueEval`
`ParseAndExecute` instantiates the template

```
CppMath = type('Math', (), {  
    'pow2': TemplateWrapper('Math', 'pow2'),  
    'add' : TemplateWrapper('Math', 'add'),  
})
```

```
m = CppMath()
```

```
m.pow2['int'](7)  
m.pow2['double'](2.5)  
m.add['int'](20, 22)
```

Same metaclass shape as ex4, but
each call returns a typed Python value

```
gEval('2 + 40')  
gEval('true && false')
```

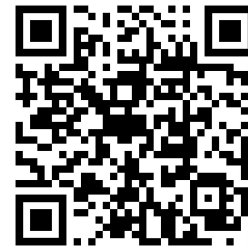
Our compiler-research.org initiatives aim to:

- Make compiler research visible and connected
- Train and mentor the next generation of compiler engineers
- Support impactful open-source R&D
- Build bridges between academia, industry, and scientific domains



We are happy to announce our Clang-focused fellowship program sponsored by CppAlliance:

- Remote, GSoC-like, project-based
- Upto 6-month internships on Clang oriented projects
- Application form: <https://forms.gle/hTCqC2nkTZwqNabc6>



Thank You!



vgvassilev@cern.ch



aaron.jomy@cern.ch



vipul.cariappa.thimmaiah@cern.ch