Interactive C++ for Data Science

VASSIL VASSILEV

Speaker

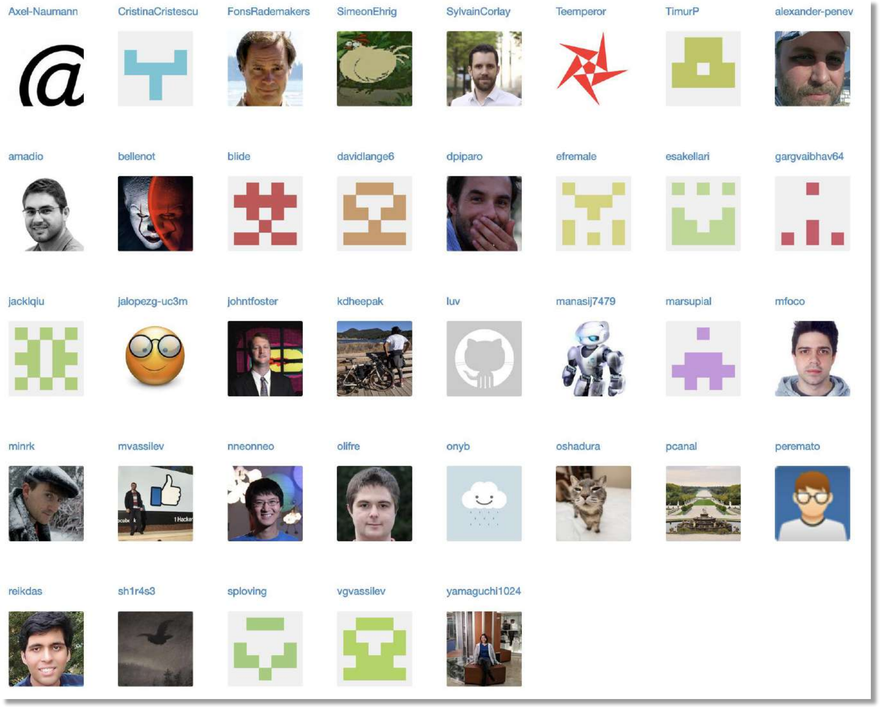
Vassil Vassilev, Research Software Engineer, Princeton/CERN

Outline

- Introduction
  - Key insights of Interactive C++
    - Execution of statements, Execution Results, Entity redefinitions, Error Recovery, Code Undo
  - Interpreting C++. Tools and technology
- Interactive, interpretative C++ for Data Science
  - Eval-style programming, C++ in notebooks, CUDA C++
- Beyond just interpreting C++
  - Template instantiation on demand, Language Interop on Demand
- Interpreter/Compiler as a service
  - Extensions, Automatic differentiation on the fly, Lifelong Optimization
- Conclusion

Acknowledgement & Disclaimer

- This talk includes technologies developed by various individuals and organizations in the area of interpretative C++ since 1998
- This talk is about work conducted by me but also the work of dozens colleagues and contributors from many domains in science and industry. In the slides I have tried to mention individuals and organizations where possible.
- Any characterizations, mischaracterizations, emphasis and errors are solely mine and do not necessarily represent the views of other individuals or organizations.



The current work is partially supported by National Science Foundation under Grant OAC-1931408. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Introduction

Despite its power, C++ is often seen as difficult to learn, difficult to teach and inconsistent with rapid application development/exploratory programming.

Exploration and prototyping is slowed down by the long edit-compile-run cycles during development.

Is C++ a “compiled” language?
Is C++ an “experts’” language?

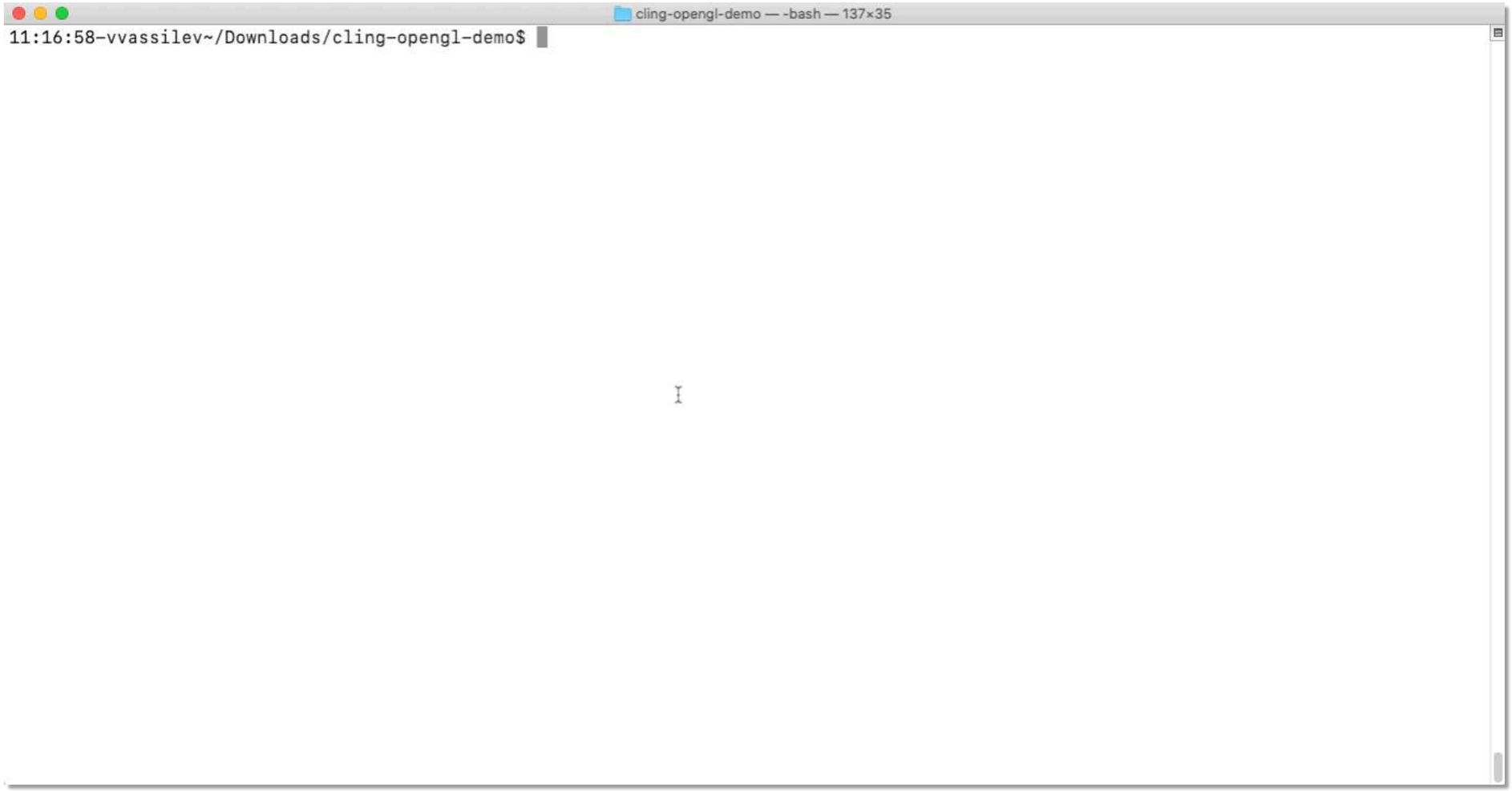
Choice of translation

The language translation should be an implementation choice.

Language design could affect the choice of translation and the choice of translation can affect language design.

What would it enable if we start interpreting C++?

Interactive C++



Video Credits: A. Penev

The invisible compile-run cycle aids interactive use offering a different programming experience and enhanced productivity. It becomes trivial to orient a shape, choose size and color or compare to previous settings.

Exploratory Programming

- An effective way to gain understanding of code/algorithms
- Interactive probing of data and interfaces makes complex libraries accessible to users
- Significantly reduces the edit-compile-run cycle

Exploratory Programming

- Languages which enable exploratory programming tend to have interpreters to shorten the compile-run cycle but generally have a noticeable cost
- Language designers who acknowledge the use case of exploratory programming may also put syntax sugar to improve convenience and terseness
- Performance is mitigated nowadays by just-in-time (JIT) or ahead-of-time (AOT) compilation technology

Interactive C++. Key Insights

- Incremental Compilation
- Handling errors
  - Syntactic
  - Semantic
- Execution of statements
- Displaying execution results
- Entity redefinition

```
[cling] #include <vector>
[cling] std::vector<int> v = {1,2,3,4,5};

[cling] std.sort(v.begin(), v.end());
input_line_1:1:1: error: unexpected namespace
name 'std': expected expression
std.sort(v.begin(), v.end());
^

[cling] std::sort(v.begin(), v.end());
[cling] v // No semicolon
(std::vector<int> &) { 1, 2, 3, 4, 5 }

[cling] std::string v = "Hello World"
(std::string &) "Hello World"
```

Interactive C++. Key Insights

- Eval-style programming – enables embedding in frameworks

```
[cling]$ #include <cling/Interpreter/Value.h>
[cling]$ #include <cling/Interpreter/Interpreter.h>
[cling]$ int i = 1;
[cling]$ cling::Value V;
[cling]$ gCling->evaluate("++i", V);
[cling]$ i
(int) 2
[cling]$ V
(cling::Value &) boxes [(int) 2]
[cling]$ ++i
(int) 3
[cling]$ V
(cling::Value &) boxes [(int) 2]
```

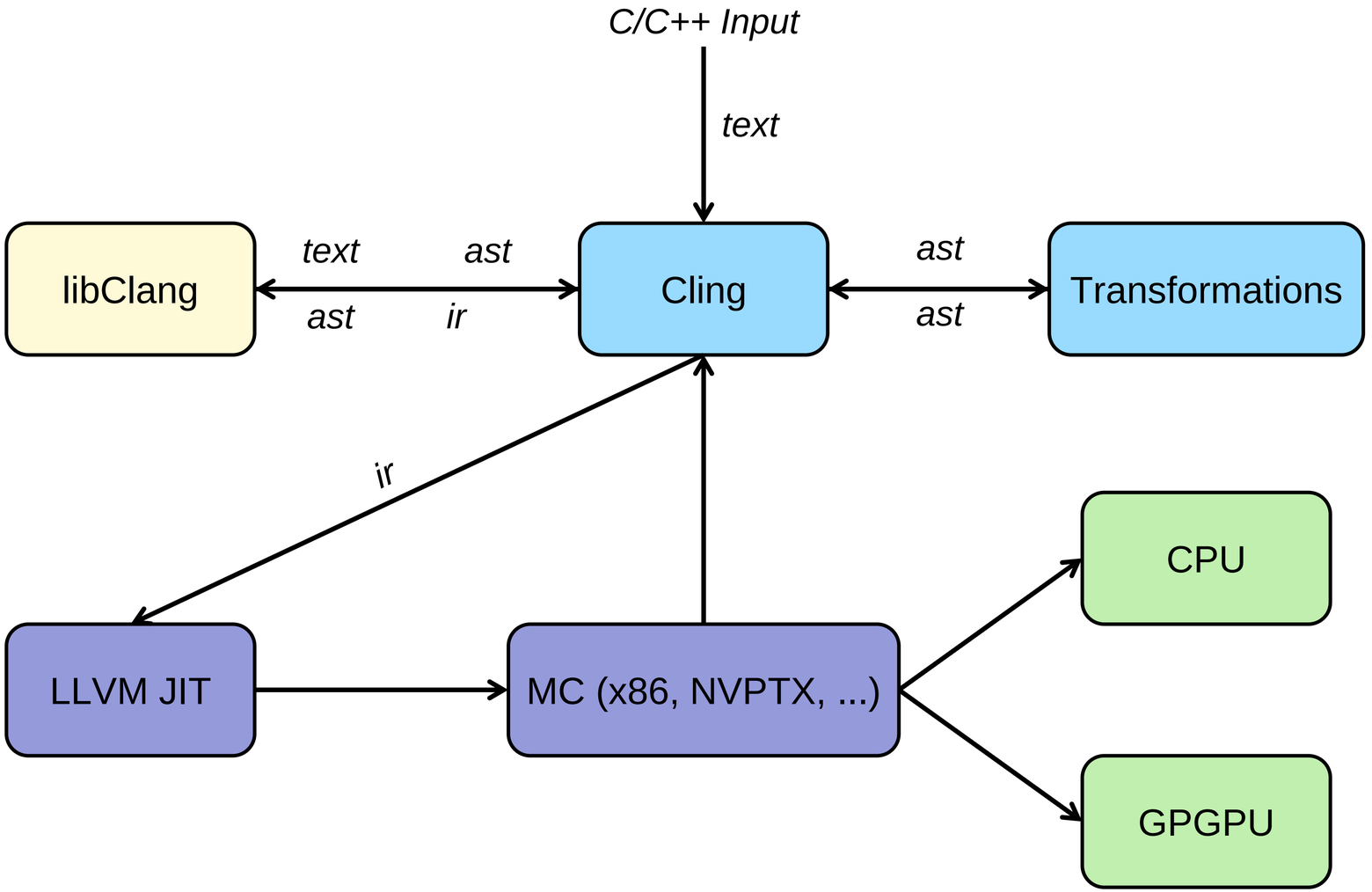
Interpreters for C++

- Ch – a proprietary cross-platform C and C++ interpreter and scripting language environment, originally designed by Harry H. Cheng as a scripting language for beginners to learn mathematics, computing, numerical analysis (numeric methods), and programming in C/C++. Last release 2017

- CINT – a command line C/C++ interpreter that was originally developed by Masaharu Goto and later included in the ROOT data analysis software package. Last release 2013

- Cling – an LLVM-based C/C++ interpreter developed in the field of high-energy physics to replace CINT in ROOT. Last release 2021

Interpreting C++. Cling



Interpreting C++. Cling

- Cling has originally developed in the field of high energy physics to enable interactivity, dynamic interoperability and rapid prototyping capabilities to C++ developers.
- Cling supports the full C++ feature set including the use of templates, lambdas, and virtual inheritance.
- Cling adds a small set of extensions in C++ to allow interactive exploration and makes the language more welcoming for use.
- Cling compiles C++ code incrementally and relies on JIT compilation
- Cling enables exploratory programming for C++

Interactive C++ in Data Science

Data Science

- Data science is a "concept to unify statistics, data analysis, informatics, and their related methods" in order to "understand and analyze actual phenomena" with data. Considered by some as the “fourth paradigm” of science next to empirical, theoretical and computational science.

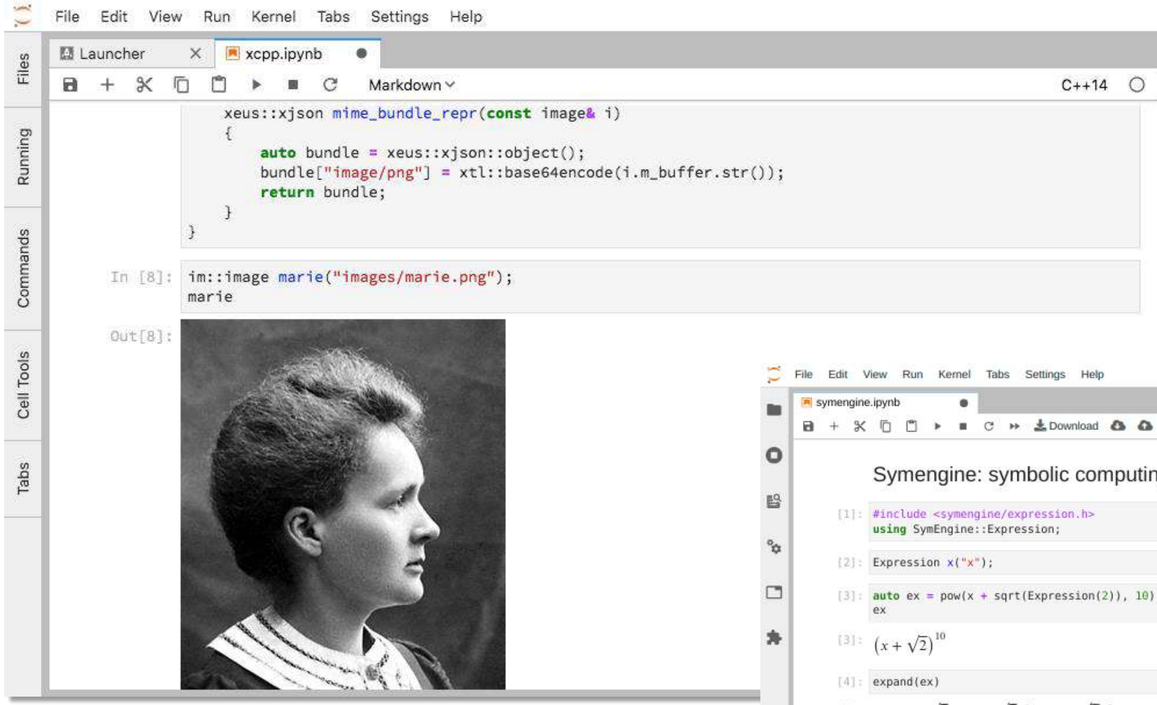
- Per LinkedIn, there has been a 650% increase in data science jobs since 2012.

- The U.S. Bureau of Labor Statistics sees strong growth in the data science field and predicts the number of jobs will increase by about 28% through 2026 (that is roughly 11.5 million new jobs).

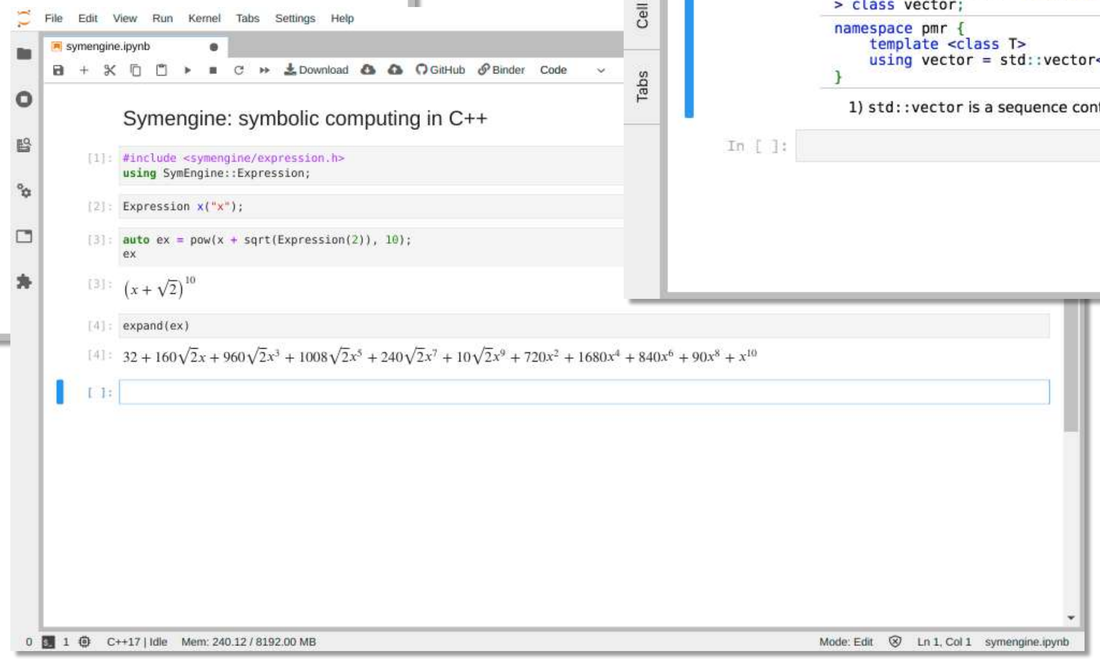
Xeus-Cling. C++ in Notebooks

- Xeus-Cling is a Cling-based Jupyter Notebook kernel
- If you use C++ in Jupyter you use xeus-cling
- Developed and maintained by the private company QuantStack

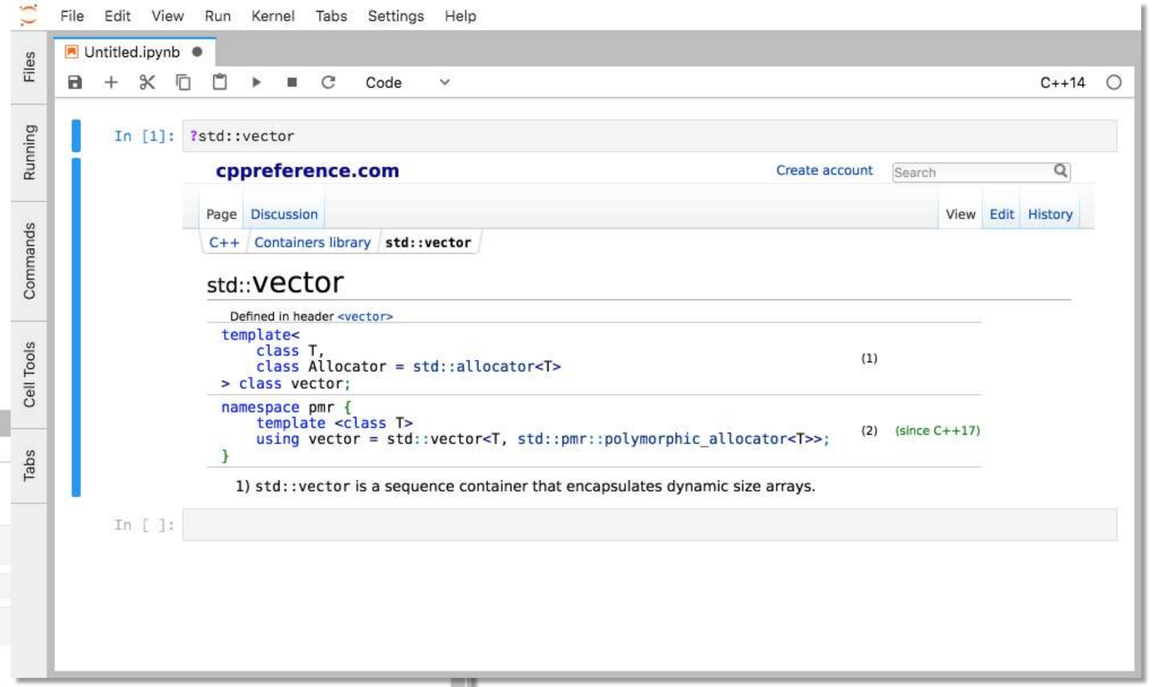
Xeus-Cling. C++ in Notebooks



Visualization of user-defined images

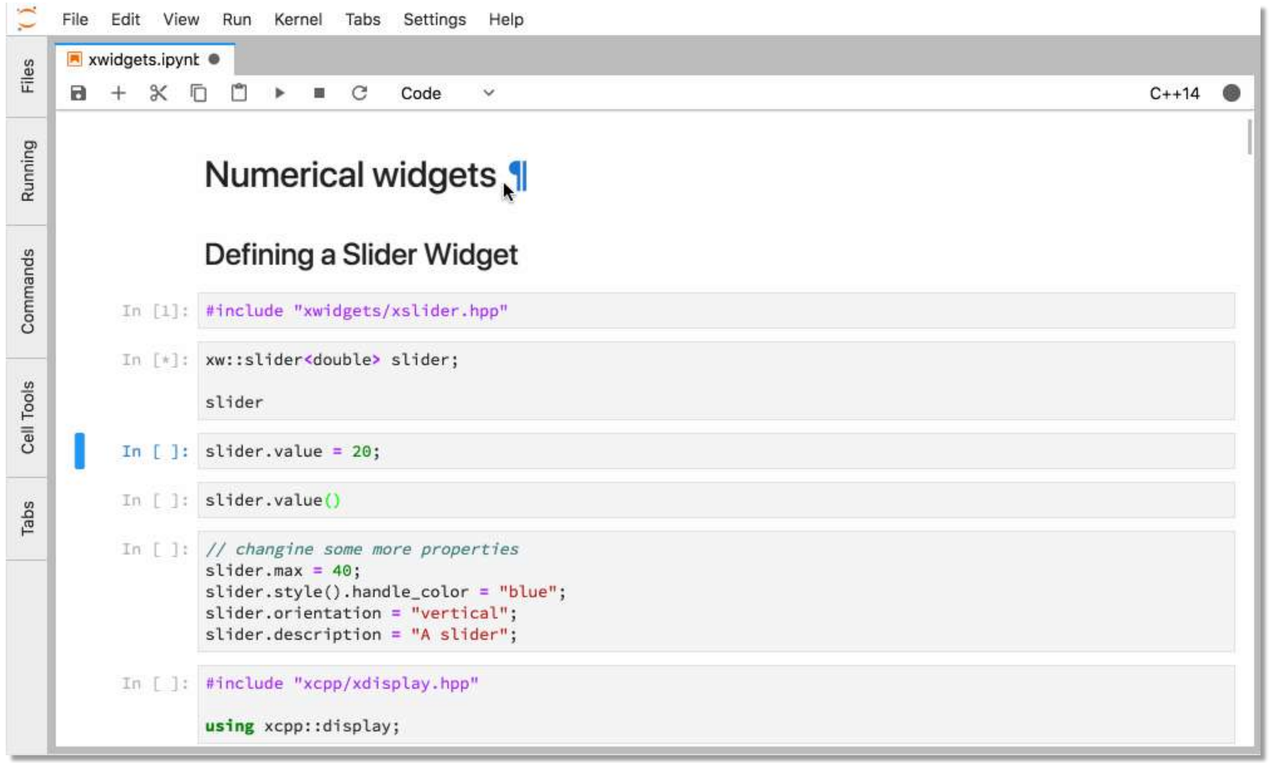


Rich mime type rendering in Jupyter

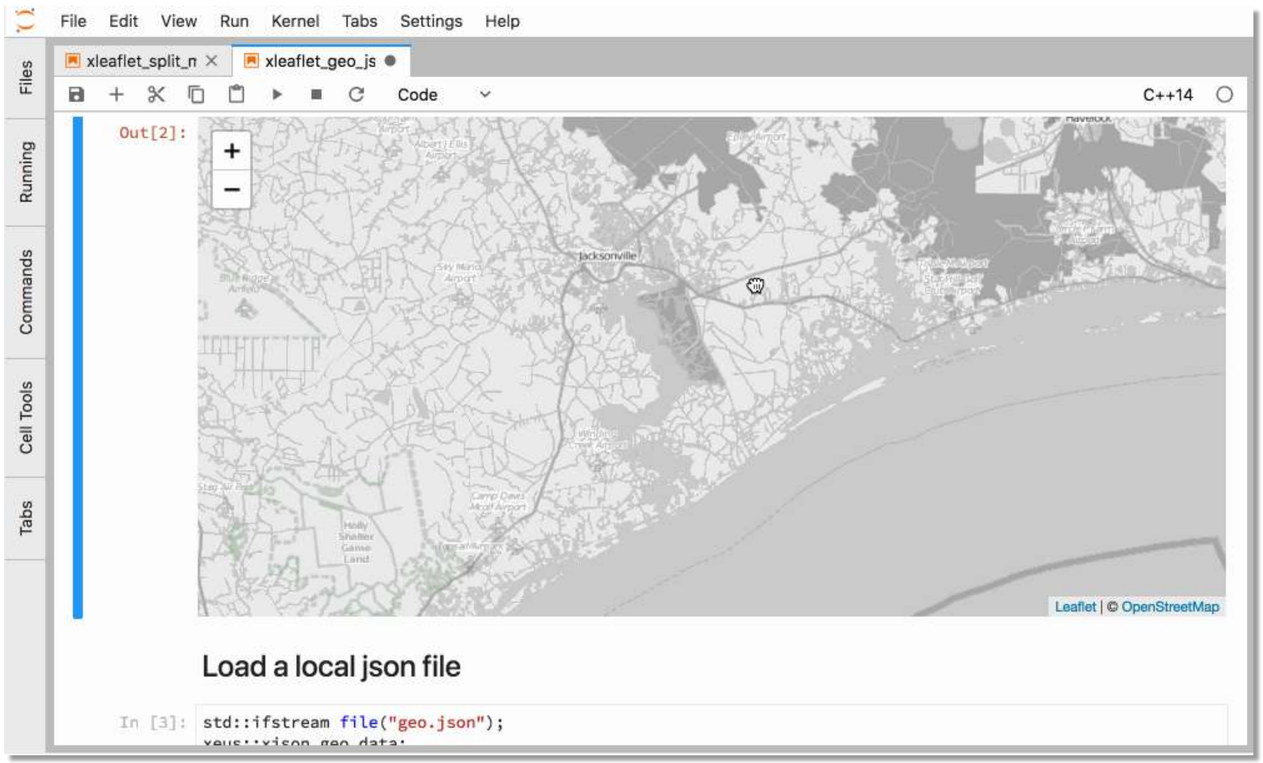


Direct access to documentation

Xeus-Cling. C++ in Notebooks



Xwidgets – User-defined controls



Xleaflet – Interactive Geo Information System

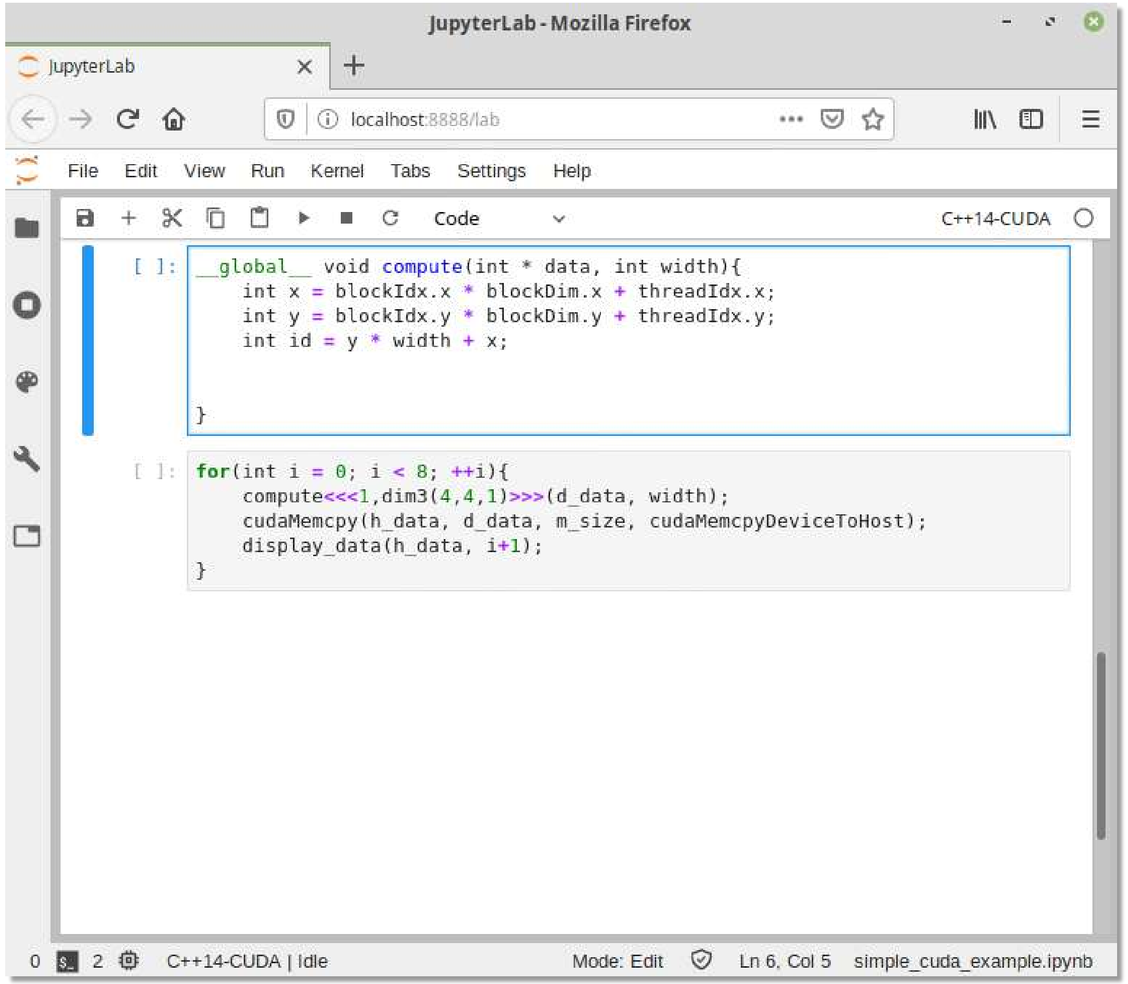
S. Corlay, Quantstack, *Deep dive into the Xeus-based Cling kernel for Jupyter*, May 2021, compiler-research.org

Interactive CUDA C++

- Interactive CUDA backend
- Developed and maintained by HZDR

```
[cling] #include <iostream>
[cling] #include <cublas_v2.h>
[cling] #pragma cling(load "libcublas.so")
// set parameters, allocate memory ...
[cling] __global__ void init(float *matrix, int size){
[cling] ?   int x = blockIdx.x * blockDim.x + threadIdx.x;
[cling] ?   if (x < size)
[cling] ?     matrix[x] = x;
[cling] ?   }
[cling]
[cling] // launching a function direct in the global space
[cling] init<<<blocks, threads>>>(d_A, dim*dim);
[cling] init<<<blocks, threads>>>(d_B, dim*dim);
[cling]
[cling] cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, dim, dim,
dim, &alpha, d_A, dim, d_B, dim, &beta, d_C, dim);
[cling] cublasGetVector(dim*dim, sizeof(h_C[0]), d_C, 1, h_C, 1);
[cling] cudaGetLastError()
(cudaError_t) (cudaError::cudaSuccess) : (unsigned int) 0
```

Interactive CUDA C++



S. Ehrig, HZDR, *Cling's CUDA Backend: Interactive GPU development with CUDA C++*, Mar 2021, compiler-research.org

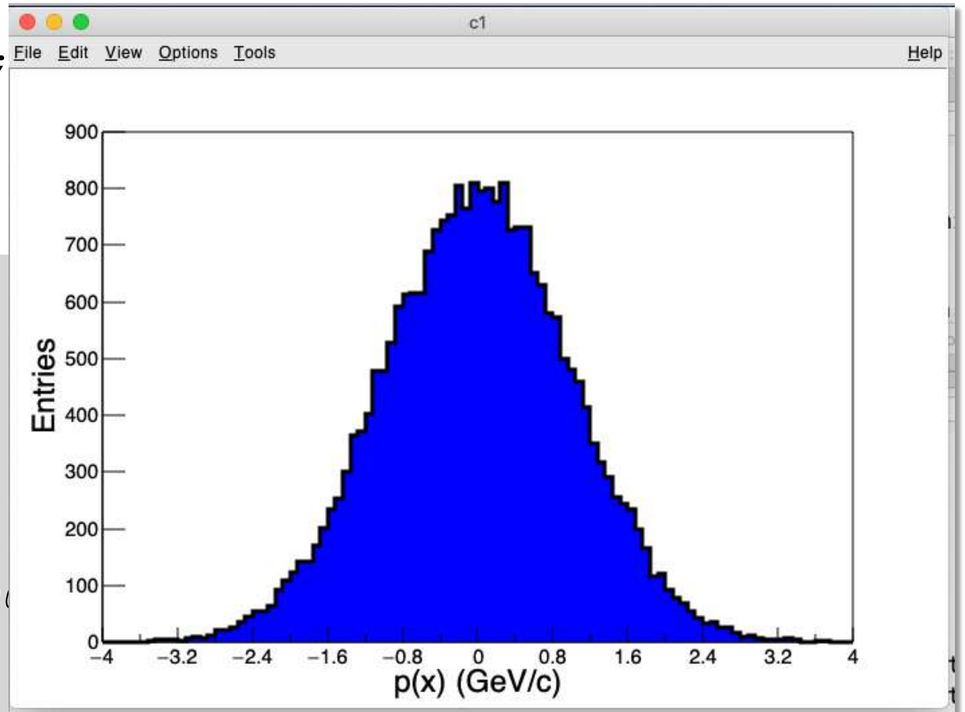
ROOT – Scientific Data Analysis

- Developed and maintained by the field of high-energy physics and organizations such as CERN, FNAL, GSI, University of Nebraska, UC San Diego, Princeton
- Last 5 years the ROOT and Cling technology are used to store around 1EB physics data facilitating more than 1000 scientific publications
- The ROOT data analysis package embeds Cling to enable interactive C++ but also to use it as a reflection information service for data serialization

Dynamic Scopes. Runtime Lookup

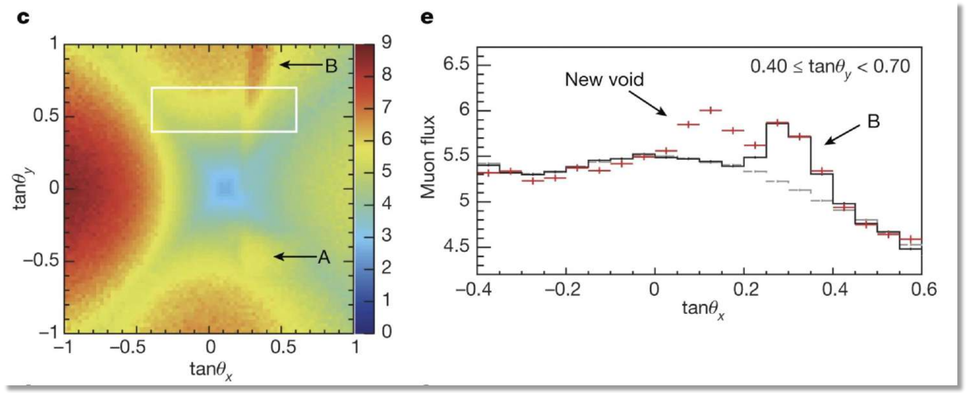
```
gCling->EvaluateT</*ret type*/void>("ntuple->GetTitle()", /*context*/);
```

```
[root] ntuple->GetTitle()
error: use of undeclared identifier 'ntuple'
[root] TFile::Open("tutorials/hsimple.root"); ntuple->GetTitle()
(const char *) "Demo ntuple"
[root] gFile->ls();
TFile**         tutorials/hsimple.root    Demo ROOT file with histograms
 TFile*         tutorials/hsimple.root    Demo ROOT file with histograms
  OBJ: TH1F     hpx    This is the px distribution : 0 at: 0x7fadbb84e39
  OBJ: TNtuple    ntuple    Demo ntuple : 0 at: 0x7fadbb93a890
  KEY: TH1F     hpx;1    This is the px distribution
  [...]
  KEY: TNtuple    ntuple;1    Demo ntuple
[root] hpx->Draw()
```

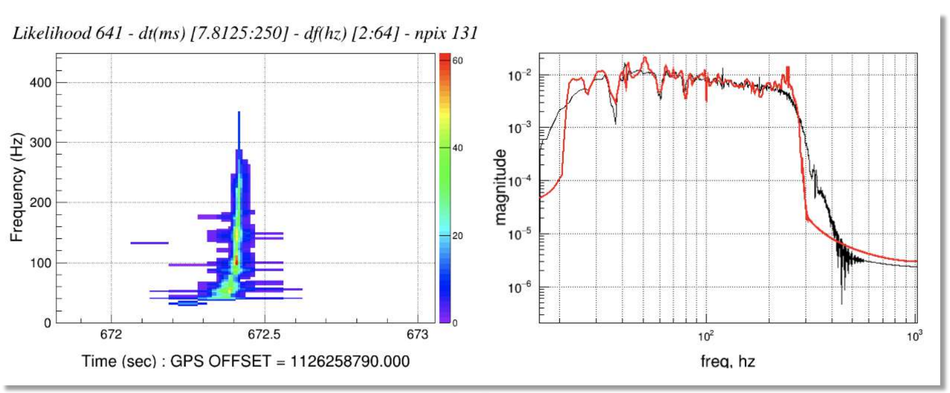


Eval-style programming enables Cling to be embedded in frameworks.

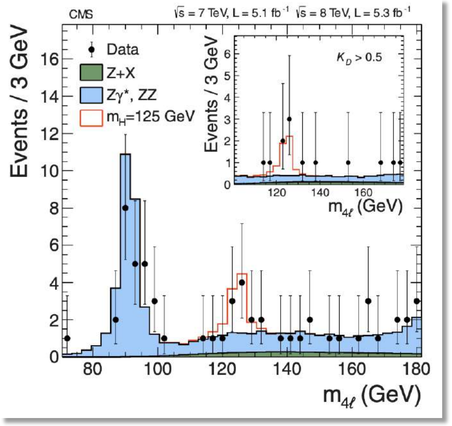
Impact of Interactive C++ in Physics



[1]



[2]



[3]

Scientific breakthroughs such as the discovery of the big void in the Khufu's Pyramid, the gravitational waves and the Higgs boson heavily rely on the ROOT software package

[1] K. Morishima et al, **Discovery of a big void in Khufu's Pyramid by observation of cosmic-ray muons**, *Nature*, *2017*
[2] Abbott et al, **Observation of gravitational waves from a binary black hole merger**. *Physical review letters, 2016*
[3] CMS Collab, **Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC**. *Physics Letters B, 2012*

Key Insights

- Cling is not just a Repl, it is an embeddable and extensible execution system for efficient incremental execution of C++
- Cling is used in several high-performance systems to provide reflection and introspection information
- Cling can produce efficient code for performance-critical tasks where hot-spot regions can be annotated with specific optimization levels
- Cling allows us to decide how much we want to compile statically and how much to defer for the target platform

Beyond Just Interpreting C++

Automatic Language InterOp

Classification of Prior Work

- Static binding generators — SWIG, SIP, Boost.Python, CxxWrap.jl
  They rely on limited parsers and limited type introspection to generate wrapper functions which the other language knows how to call. Advanced C++ features such as reference counting are hard to support.
- Dynamic / Automatic bindings — cppyy, Cxx.jl
  They operate on demand and generate only what's necessary when necessary. They rely on a compiler available at program runtime: cling and the Julia JIT respectively. They are efficient and can support about advanced C++ features such as template instantiation. Harder to implement.

How to define to whether two languages are interoperable?

CaaS Monthly, Oct, 2021 *Interactive C++: A Language InterOp Layer, V. Vassilev* 3

V. Vassilev, Princeton, *Interactive C++: A Language InterOp Layer*, Oct 2021, compiler-research.org

Automatic Language InterOp. Python

cppyy: Yet another Python – C++ binder?!

- Yes, but it has its niche: *bindings are runtime*
  - Python is all runtime, so runtime is more natural
  - C++-side runtime-ness is provided by Cling
- Very complete feature-set (not just "C with classes")
- Good performance on CPython; great with PyPy✷

pip: https://pypi.org/project/cppyy/
conda: https://anaconda.org/conda-forge/cppyy
git: https://github.com/wlav/cppyy
docs: https://cppyy.readthedocs.io/en/latest/

For HEP users: *cppyy in ROOT is an old fork. It won't run all the examples here, doesn't work with PyPy, and has worse performance.*

(✷) PyPy support lags CPython

Runtime Template Instantiations

- Cling instantiates templates at runtime
  - No pre-instantiation/compilation necessary
  - Prevent duplication of standard classes (e.g. STL)
  - No combinatorial explosion (esp. with numeric types)
  - Support for templates of user classes

W. Lavrijsen, LBL, cppyy, Sep 2021, compiler-research.org

Cppyy connects the Cling interpreter to the Python interpreter to co-operate.

Automatic Language InterOp. Python

Performance Compared to Static Approaches

- No fundamental CPU performance difference

Note carefully that *everything* in Python is runtime: compile-time just means that the bindings *recipe* is compiled, not the actual bindings themselves!

- But heavy Cling/LLVM dependency:
  - ~25MB download cost; ~100MB memory overhead
  - Complex installation (and worse build)

Basic Performance Test: overload

Tool	Execution time (ms/call)✷
C++ (Cling w/ -O2; out-of-line)	1.8E-6
cppyy / pypy-c	0.50
cppyy / CPython	1.25
swig (builtin)	1.29
swig (default)	4.23
pybind11	6.97

⇒ C++ overload is resolved at compile time, not based on dynamic type
⇒ Largest overhead: Python instance type checking (avoidable, but clumsy)
⇒ There is no obvious benefit to "static" over runtime bindings

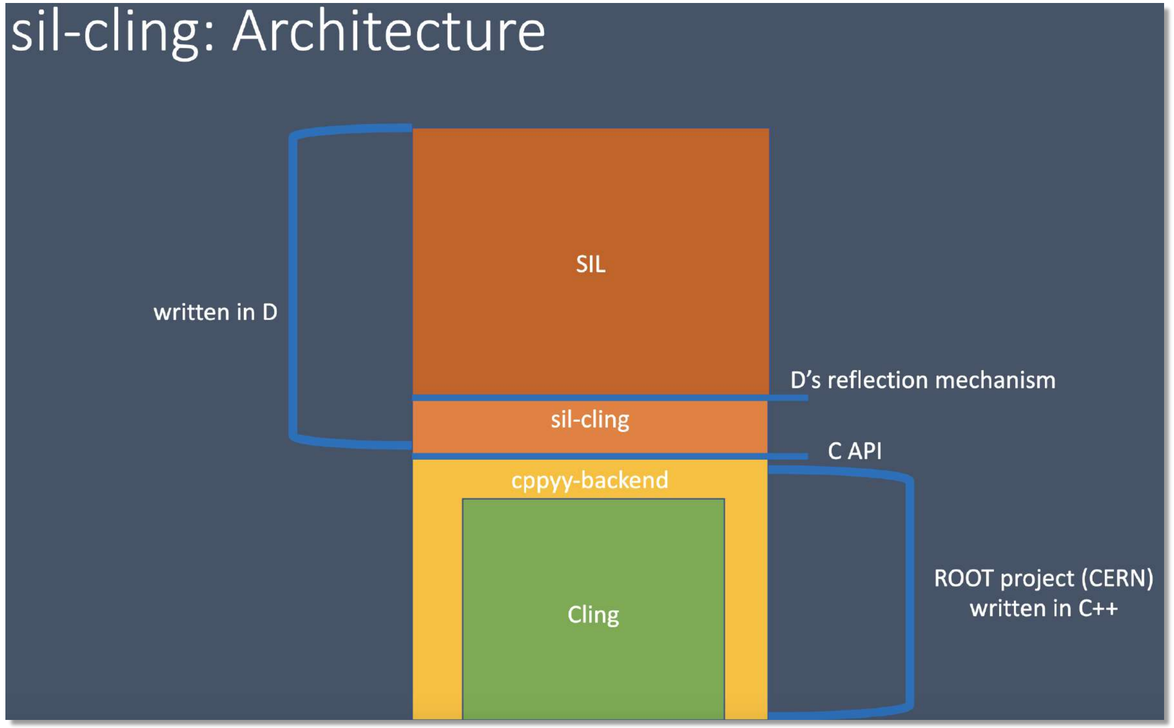
(✷) lower is better

W. Lavrijsen, LBL, cppyy, Sep 2021, compiler-research.org

The approach does not require the project maintainer to bother providing static bindings

Automatic Language InterOp. D

sil-cling: Architecture



sil-cling: Interface with cppyy

- binds with cppyy through the direct inclusion of the latter's C header using dpp

```
1 //cling.dpp
2
3 #include "capi.h"  // cppyy's C header
4
5 // D code ↓
6 import std.string : fromStringz, toStringz;
7
8 string resolveName(string cppItemName)
9 {
10     import core.stdc.stdlib : free;
11     // Calling cppyy_resolve_name ↓
12     char* chars = cppyy_resolve_name(cppItemName.toStringz);
13     string result = chars.fromStringz.idup;
14     free (chars);
15     return result;
16 }
```

```
→  ~ dub run dpp -- cling.dpp --keep-d-files -c
```

A. Militaru, Symmetry Investments, *Calling C++ libraries from a D-written DSL: A cling/cppyy-based approach*, Feb 2021, compiler-research.org

Sil-Cling connects D to C++ via Cppyy at runtime

Automatic Language InterOp. Julia

Features - Interactive C++ REPL

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.1.0 (2019-01-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> using Cxx

C++ > // Press '<' to activate C++ mode

C++ > #include <iostream>
true

C++ > std::cout << "Welcome to Cxx.jl" << std::endl;
Welcome to Cxx.jl

C++ > std::string cxx = "C++";

julia> println("Combine Julia and ", String(icxx"cxx;"))
Combine Julia and C++

julia>
```

How does it work? - Runtime

```
julia> cxx"""
       struct nontrivial { ~nontrivial() { $:(println("I got deleted")); };
       """
true

julia> a = icxx"nontrivial{};"
(struct nontrivial) {
}

julia> a = nothing;

julia> # Some time later

julia> GC.gc()
I got deleted
```

Nesting works also at global scope

Last reference dropped here

Julia GC deletes object => C++ destructor called

K. Fischer, Julia Computing, *A brief history of Cxx.jl*, Aug 2021, compiler-research.org

Cxx.jl, similarly to Cppyy uses incremental compilation (not based on Cling).

Compiler As A Service

Compiler As A Service

The design of Cling, just like Clang, allows it to be used as a library. In the next example we show how to incorporate libCling in a C++ program. The program can by compiled by your favorite compiler. Cling can be used on-demand, as a service, to compile, modify, describe or extend C++ but also use CUDA.

Compiler As A Service

```
/// Call an interpreted function using its symbol address.
void callInterpretedFn(cling::Interpreter& interp) {
  // Declare a function to the interpreter. Make it extern "C"
  // to remove mangling from the game.
  interp.declare("#pragma cling optimize(1)"
      extern \"C\" int cube(int x) { return x * x * x; }");
  void* addr = interp.getAddressOfGlobal("cube");
  using func_t = int(int);
  func_t* pFunc = cling::utils::VoidToFunctionPtr<func_t*>(addr);
  std::cout << "7 * 7 * 7 = " << pFunc(7) << '\n';
}
```

```
// caas-demo.cpp
// g++ ... caas-demo.cpp; ./caas-demo
int main(int argc, const char* const* argv) {
  cling::Interpreter interp(argc, argv, LLVMDIR);

  callInterpretedFn(interp);
  return 0;
}
```

```
vvassilev@vv-nuc ~/.../builddir $ ./caas-demo
7 * 7 * 7 = 343
vvassilev@vv-nuc ~/.../builddir $
```

Compiler As A Service. Extensions

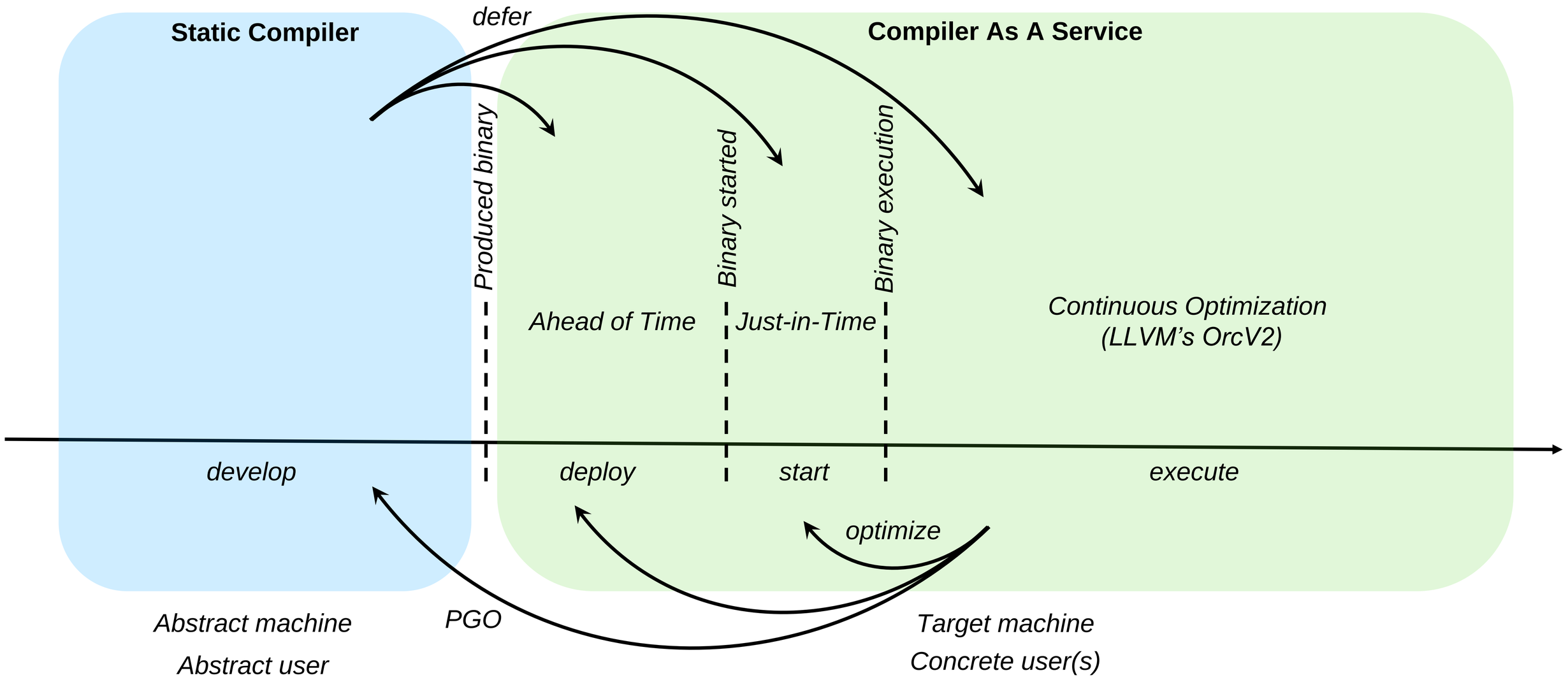
```
int main(int argc, const char* const* argv) {
  std::vector<const char*> argvExt(argv, argv+argc);
  argvExt.push_back("-fplugin=etc/cling/plugins/lib/clad.so");
  cling::Interpreter interp(argvExt.size(), &argvExt[0], LLVMDIR);
  gimme_pow2dx(interp);
  return 0;
}
```

Compiler As A Service. Extensions

```
#include <...>
// Derivatives as a service.
void gimme_pow2dx(cling::Interpreter &interp) {
  // Definitions of declarations injected also into cling.
  interp.declare("double pow2(double x) { return x*x; }");
  interp.declare("#include <clad/Differentiator/Differentiator.h>");
  interp.declare("#pragma cling optimize(2)");
  interp.declare("auto dfdx = clad::differentiate(pow2, 0);");

  cling::Value res; // Will hold the evaluation result.
  interp.process("dfdx.getFunctionPtr();", &res);

  using func_t = double(double);
  func_t* pFunc = res.getAs<func_t*>();
  printf("dfdx at 1 = %f\n", pFunc(1));

  interp.process("dfdx.getCode();", &res);
  printf("dfdx code: %s\n %s\n", res.getAs<const char*>());
}
```

```
vvassilev@vv-nuc ~/.../builddir $ ./caas-demo
dfdx at 1 = 2.000000
dfdx code: double pow2_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}

vvassilev@vv-nuc ~/.../builddir $
```

Lifelong Optimization



Incremental Compilation in LLVM

- In the context of the compiler-as-a-service project we are gradually making it possible to use incremental C++ in Clang

- We have put the infrastructure to required to instantiate and execute a template in 120 LOC

Instantiating a C++ template in C

```
// gcc ... template_instantiate_demo.C
#include "InterpreterUtils.h"

int main(int argc, char **argv) {
  Clang_Parse("void* operator new(__SIZE_TYPE__,
void* __p);"
      "extern \"C\" int printf(const char*,...);"
      "class A {};"
      "\n #include <typeinfo> \n"
      "struct B {"
      "  template<typename T>"
      "  void callme(T) {"
      "    printf(\" Instantiated with [%s] \\n \",
typeid(T).name());"
      " }"
      "};");
  const char * InstArgs = "A*";
  Decl_t T = Clang_LookupName("A");
  Decl_t TemplatedClass = Clang_LookupName("B");
  // ...
  // Instantiate B::callme with the given types
  Delc_t Inst
     = Clang_InstantiateTemplate(TemplatedClass,
"callme", InstArgs);


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

  // Create object of type T
  void* NewT = Clang_CreateObject(T);

  callme_fn_ptr(NewT);

  return 0;
}
```

```
vvassilev@vv-nuc ~/.../cpptemplate $ LD_LIBRARY_PATH="." ./template_instantiate_demo.out
Instantiated with [P1A]
vvassilev@vv-nuc ~/.../cpptemplate $ LD_LIBRARY_PATH="." ./template_instantiate_demo.out "class MyClass1{};" "MyClass1" "MyClass1*"
Instantiated with [P8MyClass1]
vvassilev@vv-nuc ~/.../cpptemplate $
```

Conclusion

- Is C++ a “compiled” language?
- Is C++ an “experts’” language?
- What would it enable if we start interpreting C++?

Thank You!

Selected References

- https://blog.llvm.org/posts/2020-11-30-interactive-cpp-with-cling/
- https://blog.llvm.org/posts/2020-12-21-interactive-cpp-for-data-science/
- https://blog.llvm.org/posts/2021-03-25-cling-beyond-just-interpreting-cpp/
- https://Compiler-Research.org
- https://root.cern
- https://root.cern/cling

Q&A