cppyy

CaaS monthly meeting – 09/02/21

Wim Lavrijsen
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
Examples of Runtime Behavior
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
Runtime Template Instantiations

```cpp
struct MyClass {
    MyClass(int i) : fData(i) {}
    virtual ~MyClass() {}
    virtual int add(int i) {
        return fData + i;
    }
    int fData;
};
```

```python
>>> import cppyy.gbl as CC
>>> v = 
...    CC.std.vector[CC.MyClass]()
...    ...
>>> for i in range(10):
...    ...    v.emplace_back(i)
...    ...
>>> len(v)
10
>>> for m in v:
...    ...    print(m.fData, end=' ')  
...    ...
0 1 2 3 4 5 6 7 8 9
>>> 
```
 Runtime Cross-Inheritance

• Cling’s JIT compiles generated trampolines
  – All proper C++ base classes can be inherited from
    • No need to select a subset of likely base classes
  – Only trampoline methods actually overridden
  – No overhead added to bound base class
  – Memory managed (copy, move, assign, destruct)
```
struct MyClass {
    MyClass(int i) : fData(i) {} 
    virtual ~MyClass() {}
    virtual int add(int i) {
        return fData + i;
    }
    int fData;
};

>>> import cppyy.gbl as CC
>>> class PyMyClass(CC.MyClass):
...     def add(self, i):
...         return self.fData + 2*i
...
>>> m = PyMyClass(1)
>>> CC.callb(m, 2)
5
```
The obvious next step ...

- Cross-inheritance allows Python classes in C++
  - Uniquely identifiable, memory managed
- C++ classes can be used as template argument
- Emergent property: *Python classes in templates!*
Thus obvious next step ...

```python
>>> import cppyy.gbl as CC
>>> class PyMyClass(CC.MyClass):
    ...
        def __init__(self, d, extra):
            ...
                super(PyMyClass, self).__init__(d)
                self.extra = extra
        ...
        def add(self, i):
            ...
                return self.fData + \
                ...                     self.extra + 2*i
            ...
    ...

>>> v = \n    ...
    ...             CC.std.vector[PyMyClass]() ...
>>> v.push_back(PyMyClass(4, 42))
>>> v.back().add(17)
80
>>> }
```

```c
struct MyClass {
    MyClass(int i) : fData(i) {}
    virtual ~MyClass() {};
    virtual int add(int i) {
        return fData + i;
    }
    int fData;
};
```
Runtime Automatic Fallbacks

- Cling instantiates templates at runtime
- But Python types do no map uniquely, example:

<table>
<thead>
<tr>
<th>Python</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>type int</td>
<td>int8_t, uint8_t, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, int64_t, uint64_t, ...</td>
</tr>
</tbody>
</table>

- Solution: automatically fallback as needed
\texttt{template<typename T>}
\begin{verbatim}
T passT(T t) {
    return t;
}
\end{verbatim}

```python
>>> import cppyy.gbl as CC
>>> type(1)
<class 'int'>
>>> CC.passT(1)
1
>>> CC.passT.__doc__
'int ::passT(int t)'
>>> type(2**64-1)
<class 'int'>
>>> CC.passT(2**64-1)
18446744073709551615
>>> CC.passT.__doc__
'unsigned long long ::passT(unsigned long long t)'
```
Runtime Callbacks

• Cling’s JIT compiles generated wrappers
  – Type checked and memory managed
    • Errors (exceptions) can trace through both Python and C++
    • Python manages lifetime, C++ manages resources
      – Note: manage manually if C++ stores the function ptr
  – Supports C++ function pointers and `std::function`

• Python can pass any callable
  – Functions, lambda’s, objects implementing `__call__`
  – Bound C++ functions and methods (w/o wrapper)
typedef int (*P)(int);

int callPtr(P f, int i) {
    return f(i);
}

typedef std::function<int(int)> F;

int callFun(const F& f, int i) {
    return f(i);
}

>>> import cppyy.gbl as CC
>>> def f(val):
...     return 2*val
...
>>> CC.callPtr(f, 2)
4
>>> CC.callFun(f, 3)
6
>>> CC.callPtr(lambda i: 5*i, 4)
20
>>> CC.callFun(lambda i: 6*i, 4)
24
>>>
Runtime Templated Callbacks

- Modern Python3 supports “annotations”
  - Very commonly used in any modern Python project
  - Type information used by IDEs, static checkers, etc.
    - Unused by (and mostly irrelevant to) the interpreter
  - Dictionary of (strings of) argument and return types
    - Strings are necessary for compound C++ types

- Annotated functions can instantiate templates
  - Incl. bound C++ functions (by definition “annotated”)
Runtime Templated Callbacks

template<typename R,
    typename... U,
    typename... A>
R callT(R(*f)(U...), A&&... a) {
    return f(a...);
}

>>> import cppy.gbl as CC
>>> def f(a: 'int') -> 'double':
    ...    return 3.1415*a
    ...
>>> CC.callT(f, 2)
6.283
>>> def f(a: 'int', b: 'int') |
    ...    return 3*a*b
    ...
>>> CC.callT(f, 6, 7)
126
>>>
Runtime Auto-downcast and Object Identity

• Always cast to the most derived C++ type
  – Involves a fake base and retrieving C++ RTTI
  – Custom RTTI implementation on MS Windows 64b

• Preserve identity Python proxy ↔ C++ instance
  – Eases resource management / prevents dangling_ptrs
  – Guarantees equal hashes for dictionary lookups
    • Alternative: specialize std::hash or __hash__
  – Enables cctor and assignment of cross-derived classes
```cpp
struct Base {
    virtual ~Base() {}
};

struct Derived : public Base {};

Base* passB(Base* b) {
    return b;
}

>>> import cppy.gbl as CC
>>> d = CC.Derived()
>>> b = CC.passB(d)
>>> type(b) == CC.Derived
True
>>> d is b
True
>>>
Runtime Exceptions

• Map exceptions derived from `std::exception`
  – Python exceptions are not `object` instances
  – Python exception classes do not match C++ ones

• Preserves C++ exception types
  – Allows crossing multiple language layers
    • Provides trace showing full call stack

Note: can’t mix compiled & JITed on all platforms
  • To be fixed in a future version of Clang?
Runtime Exceptions

class MyException :
    public std::exception {
public:
    const char* what() const throw() {
        return "C++ failed";
    }
};

void throw_error() {
    throw MyException{};
}

>>> import cppyy.gbl as CC
>>> try:
...    CC.throw_error()
...    except CC.MyException as e:
...        print(e)
...    void ::throw_error() =>
        MyException: C++ failed

>>>
Runtime Unicode

• Python unicode encapsulates code points + codec
  – Defaults to UTF-8 (with BOM check)

• C++ is just all over the place, for example:
  – Byte-encoded w/o codec (e.g. std::string)
  – Code points w/o codec (e.g. std::u16string)
  – Wide char types w/ assumed codec (platform-specific)

• Python’s type str != C++’s std::string
  – Developers still want interchangeable use by default ...
```cpp
#include <string>

namespace std {

template<class T>
std::string to_str(const T& chars) {
    char buf[12]; int n = 0;
    for (auto c : chars)
        buf[n++] = char(c);
    return std::string(buf, n-1);
}

std::string utf8_chinese() {
    auto chars = {0xe4, 0xb8, 0xad, 0xe6, 0x96, 0x87, 0};
    return to_str(chars);
}

std::string gbk_chinese() {
    auto chars = {0xd6, 0xd0, 0xce, 0xc4};
    return to_str(chars);
}

} // namespace std
```

```python
>>> import cppyy.gbl as CC
>>> CC.utf8_chinese()
中文
>>> CC.gbk_chinese()
b'\xd6\xd0\xce\xc4'
>>> CC.gbk_chinese().decode('gbk')
中文
```
And much more, not directly runtime ...

- Classes, functions, (static) methods, operators, iterators, enums, single/multiple inheritance, shared/unique_ptr, STL pythonizations, ...
- Low-level C support (memory, arrays, ptr math, ...)
- Debug support (e.g. segfault -> Python exception)
- Customize with pythonizations, “freeze” binary distributions, cmake fragments for projects, ...
- See: https://cppyy.readthedocs.io/en/latest/
Yes, okay, runtime is great ... but what about performance?
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)
void empty_call() {}

class Overload {
public:
    double add(int a, int b);
    double add(short a);
    double add(long a);
    double add(int a, int b, int c);
    double add(double a);
    double add(float a);
    double add(int a);
};

// benchmark example:
Overload obj;
for (size_t i=0; i < N; ++i)
    obj.add((double)i);

System:
    Ubuntu 20.04.2 LTS
    AMD EPYC 7702P 64-Core CPU
    1TB of RAM

Setup:
    gcc 9.3.0 (system)
    pytest 6.2.4 (pypi)
    benchmark: 3.4.1 (pypi)

Comparison:
    cppyy 2.1.0 (pypi)
    pybind11 2.7.1 (pypi)
    swig 4.0.1 (system)
    pypy-c 3.7.1 (system)
# Basic Performance Test: empty call

<table>
<thead>
<tr>
<th>Tool</th>
<th>Execution time (ns/call)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ (Cling w/ -O2; out-of-line)</td>
<td>1.5</td>
</tr>
<tr>
<td>cppyy / pypy-c</td>
<td>16</td>
</tr>
<tr>
<td>swig (builtin)</td>
<td>27</td>
</tr>
<tr>
<td>cppyy / CPython</td>
<td>68</td>
</tr>
<tr>
<td>pybind11</td>
<td>68</td>
</tr>
<tr>
<td>swig (default)</td>
<td>104</td>
</tr>
</tbody>
</table>

- Empty global function call is a pure overhead measure (zero work)
- pypy-c slower than C++ b/c of global interpreter lock (GIL) release
- “Builtin” swig trades functionality for speed
- There is no obvious benefit to “static” over runtime bindings

(*) lower is better
## Basic Performance Test: overload

<table>
<thead>
<tr>
<th>Tool</th>
<th>Execution time (ms/call)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ (Cling w/ -O2; out-of-line)</td>
<td>1.8E-6</td>
</tr>
<tr>
<td>cppyy / pypy-c</td>
<td>0.50</td>
</tr>
<tr>
<td>cppyy / CPython</td>
<td>1.25</td>
</tr>
<tr>
<td>swig (builtin)</td>
<td>1.29</td>
</tr>
<tr>
<td>swig (default)</td>
<td>4.23</td>
</tr>
<tr>
<td>pybind11</td>
<td>6.97</td>
</tr>
</tbody>
</table>

⇒ 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
Implementation: Bird’s Eye View
Implementation

• **Python offers hooks for C++ entity lookups, e.g.:**
  – meta-classes for class creation
  – `__getattr__` for resolving attributes
  – `__getitem__/__call__` for template instantiations

• **The hooks call into Cling for name lookup**
  – All initial lookups are always *string-based*

• **Access provided by address or through wrappers**
  – Wrappers are C++ code to easily cover esoteric uses
  – Generalized interfaces to simplify downstream code
cppyy.gbl.MyClass

If "MyClass" is unknown, use __getattr__ to Cling’s lookup.

If "MyClass" is known, return MyClass.

If "MyClass" is unknown, raise Exception.

Templates additionally use either __getitem__ (explicit instantiation) or __call__ (implicit).

Proxy creation calls into Cling for reflection information; wrappers are created on first use.
Wrappers of generated (and JITted) C++ are used to easily cover a range of C++isms, such as linkage of inline functions, overloaded operator new, default arguments, operator lookup, etc., etc. For simple cases in PyPy, direct FFI is used, for improved performance.
Conclusions
Current Limitations

- Complex and heavy Cling/LLVM dependency
- All C++ code enters single, global, translation unit
  - Significant slowdown for templates (Eigen, PCL, ...)
- PyPy/cppyy is significantly behind CPython/cppyy
- No MS Windows port for conda
- Significant Clang JIT limitations on MS Windows
Current Focus

• Add GPU (cuda) support to cppyy
• Bring PyPy / cppyy on par with CPython / cppyy
• Simplify installation / distribution
Runtime Python-C++ bindings are much more functional than similar static approaches (and without loss in performance*)

(*) memory overhead is higher
That's all Folks!