Adopting CppInterOp in cppyy

Vipul Cariappa
Mentors: Vassil Vassilev, Aaron Jomy, Wim Lavrijsen, Jonas Rembser

My Project

Cppyy is Python-C++ bindings interface based on Cling/LLVM.

Adopting CppInterOp as an backend alternative to the cling interpreter in Cppyy.

The main motivation is to directly use the LLVM's clang-REPL for C/C++ incremental compilation, instead of depending on cling.

The current cppyy uses lots of string manipulation. This string manipulation increases the performance overhead and affects the code readability and debuggability.

Progress

Counted as passing tests

Status	Early Sep	18 Dec 2024
Pass	276	330
XFail	227	171
Skip	1	3

I had set a goal to fix 60 to 100 tests by December.

I have managed to approximate fix 55 tests till now.

“Hello, World” in cppyy

C++

```cpp
#include <iostream>
#include <string>

void greet(std::string name="C++") {
    std::cout << "Hello, " << name << "!" << std::endl;
}
```

Python

```python
import cppyy
from cppyy import gbl
from cppyy.gbl import std


gbl.greet()
gbl.greet("Python")
```

```
Hello, C++!
Hello, Python!
```

Non-Exhaustive List of Features Enabled

With code examples

List all available declarations within a scope

C++

```cpp
namespace myscope {
    long prod(long a, long b) { return a * b; }
    long prod(long a, long b, long c) { return a * b * c; }
    int a = 10;
    int b = 40;
    enum smth { ONE, TWO };
    enum smth my_enum = smth::ONE;
}
```

Python

```python
myscope = gbl.myscope
attributes_myscope = dir(myscope)

# filter python internal attributes
imp_attributes_myscope = filter(lambda x: not (x.startswith("__") and x.endswith("__")), attributes_myscope)

print(*imp_attributes_myscope, sep="\n")
```

```
a
b
my_enum
prod
smth
```

Enabled 1 test

Lookup static and static constexpr attributes

C++

```cpp
class MyClass {
public:
    static int x;
    static constexpr int y = 10;
};
int MyClass::x = 5;
```

Python

```python
MyClass = gbl.MyClass
c = MyClass()
print(f"{c.x = }\n{c.y = }")
```

```
c.x = 5
c.y = 10
```

Enabled 3 tests

Cache hits

Python

```
  1  print(f"{MyClass is gbl.MyClass = }")
```

```
MyClass is gbl.MyClass = True
```

Enabled 12 tests

Access nested attributes of anonymous records

C++

```cpp
class MyAnonymousNestedClass {
public:
    struct {
        int x;
        int y;
    };
    MyAnonymousNestedClass(int x, int y) : x(x), y(y) {}
};
```

Python

```python
MyAnonymousNestedClass = gbl.MyAnonymousNestedClass
ac = MyAnonymousNestedClass(2, 3)
print(f"{ac.x = }\n{ac.y = }")
```

```
ac.x = 2
ac.y = 3
```

Enabled 2 tests

Access anonymous enums

C++

```
namespace my_enum_scope {
    enum {
        PYTHON,
        CPP,
        CPPYY,
        CPPINTEROP
    };
}
```

Python

```
my_enum_scope = gbl.my_enum_scope
print(f"{my_enum_scope.CPPYY = }")
```

```
my_enum_scope.CPPYY = 2
```

Enabled 1 test

Conversions of references

C++

```cpp
void inplace_concat(std::string &x, std::string y) {
    x += y;
}
```

Python

```python
s1 = std.string("C++")
s2 = std.string(" 💗 Python")
gbl.inplace_concat(s1, s2)
print(f"{str(s1) = }")
```

```
str(s1) = 'C++ 💗 Python'
```

Enabled 8 tests

Nested template instantiation & Instantiation from string

C++

```cpp
template<class T>
class MyTemplatedClass {
public:
    T item;
    MyTemplatedClass(T item) : item(item) {}
};
```

Python

```python
MyTemplatedClass = gbl.MyTemplatedClass
vector_of_templatedclass = std.vector[MyTemplatedClass[int]]()
vector_of_templatedclass.emplace_back(1)
vector_of_templatedclass.emplace_back(2)
vector_of_templatedclass.emplace_back(3)

# instantialtion from string
instantialtion_from_string = std.vector["MyTemplatedClass<int>"]()

print(f"{list(vector_of_templatedclass) = }")
```

```
list(vector_of_templatedclass) = [<cppyy.gbl.MyTemplatedClass<int> object at 0x5b30c3995720>,
<cppyy.gbl.MyTemplatedClass<int> object at 0x5b30c3995724>, <cppyy.gbl.MyTemplatedClass<int> object at
0x5b30c3995728>]
```

Enabled 12 tests

Access protected variable through cross-inheritance

C++

```cpp
class MyBaseClass {
protected:
    int x;
public:
    MyBaseClass(int x) : x(x) {}
};
```

Python

```python
class MyDerivedClass(gbl.MyBaseClass):
    def __init__(self, x):
        super(MyDerivedClass, self).__init__(x)

derived = MyDerivedClass(5)
print(f"{derived.x = }")
```

```
derived.x = 5
```

Enabled 1 test

Work Remaining

- Overloaded operator lookups
- `__str__` method for pretty printing C++ classes
- cross-inheritance and type conversions based on class hierarchy
- Complex template instantiation (that require some kind of type inference)

> Amoung 23 total test files, 2 test files contains approximately 100 failing tests. They are `test_stltypes.py` & `test_template.py`

Thank You