GSoC, 2023
Program @CERN-HSF

Student : Smit Shah
Mentors : Vassil Vassilev, Baidyanath Kundu

Enable cross-talk between Python and C++ kernels in xeus-clang-REPL by using Cppyy

So what actually is cross-talk and its use ?

jupyter Python-Cpp-Integration-Demo (unsaved changes)

Logout

File Edit View Insert Cell Kernel Help Not Trusted C++11

Run Code

Declaring variables in C++

In [1]:
```
extern "C" int printf(const char*,...);
```

In [2]:
```
int new_var1 = 12;
int new_var2 = 25;
int new_var3 = 64;
```

Running Python with C++ variables

In [3]:
```
%%python

from time import time,ctime
print('This is printed from Python: Today is', ctime(time()))
python_array = [1, 2, new_var1, new_var2, new_var3]
print(python_array)
```

```
This is printed from Python: Today is Tue Oct 25 11:38:08 2022
[1, 2, 12, 25, 64]
```

In [4]:
```
%%python

new_python_var = 1327
```

In [5]:
```
auto k = printf("new_python_var = %d\n", new_python_var);
```

```
new_python_var = 1327
```

Use of intercommunication of C++ and Python

1] Leveraging C++ Performance

2] Access to Established C++ Libraries

3] Python's Rapid Prototyping and Ease of Use

4] Code Reusability

So how do we achieve above intercommunication?

cppyy

1] cppyy is an automatic, run-time, Python-C++ bindings generator, for calling C++ from Python and Python from C++.

2] cppyy is built on top of the Cling C++ interpreter, which provides C++ code parsing and execution capabilities.

3] cppyy delivers above tasks without any language extensions, intermediate languages, or the need for boiler-plate hand-written code

```
# continue the decoration on the C++ side, by adding an operator+ overload
cppyy.cppdef("""
namespace Math {
    Integer2 operator+(const Integer2& left, const Integer1& right) {
        return left.m_data + right.m_data;
    }
}""")
```

```
# now use that fresh decoration (it will be located and bound on use):
k = i2 + i
print(k, i2.m_data + i.m_data)
```

```
55 55
```

```
>>> cppyy.cppdef("""
... int sumit1(const std::vector<int>& data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }
... int sumit2(std::vector<int> data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }
... int sumit3(const std::vector<int>&& data) {
...   return std::accumulate(data.begin(), data.end(), 0);
... }""")
...
True
>>> cppyy.gbl.sumit1(range(5))
10
>>> cppyy.gbl.sumit2(range(6))
16
>>> cppyy.gbl.sumit3(range(7))
21
```

CPPInterOp

1] CPPInterOp is a Clang-based C++ Interoperability library

2] It is a compiler service designed to access C++ code and obtain all relevant information, such as all variable declarations, functions, classes, etc.

3] It implements numerous modules to extract all information from C++ code.

Example

```
size_t SizeOf(TCppScope_t scope) {
    assert (scope);
    if (!IsComplete(scope))
      return 0;

    if (auto *RD = dyn_cast<RecordDecl>(static_cast<Decl*>(scope))) {
      ASTContext &Context = RD->getASTContext();
      const ASTRecordLayout &Layout = Context.getASTRecordLayout(RD);
      return Layout.getSize().getQuantity();
    }

    return 0;
  }
```



```
TEST(ScopeReflectionTest, SizeOf) {
  std::vector<Decl*> Decls;
  std::string code = R"(namespace N {} class C{}; int I; struct S;
                        enum E : int; union U{}; class Size4{int i;};
                        struct Size16 {short a; double b;};
                       )";
  GetAllTopLevelDecls(code, Decls);
  EXPECT_EQ(InterOp::SizeOf(Decls[0]), (size_t)0);
  EXPECT_EQ(InterOp::SizeOf(Decls[1]), (size_t)1);
  EXPECT_EQ(InterOp::SizeOf(Decls[2]), (size_t)0);
  EXPECT_EQ(InterOp::SizeOf(Decls[3]), (size_t)0);
  EXPECT_EQ(InterOp::SizeOf(Decls[4]), (size_t)0);
  EXPECT_EQ(InterOp::SizeOf(Decls[5]), (size_t)1);
  EXPECT_EQ(InterOp::SizeOf(Decls[6]), (size_t)4);
  EXPECT_EQ(InterOp::SizeOf(Decls[7]), (size_t)16);
}
```

cppyy-backend

1] cppyy-backend package helps in generation of dictionary consisting all C++ modules so that it can be accessed easily during runtime.

2] It also understands python compiler and depending on the C++ service required during runtime, it takes help of CPPInterOp.

3] Hence it combines information from CPPInterOp and binds it under one name ‘cppyy’

Example

```
size_t Cppyy::SizeOf(TCppType_t klass)
{
    return InterOp::SizeOf(klass);
}
```

snappify.com

```
Cppyy::TCppType_t Cppyy::GetType(const std::string &name, bool enable_slow_lookup /* = false */) {
    static unsigned long long var_count = 0;

    if (auto type = InterOp::GetType(getSema(), name))
        return type;

    if (!enable_slow_lookup) {
        if (name.find("::") == std::string::npos)
            throw std::runtime_error("Calling Cppyy::GetType with qualified name '"
                                     + name + "'\n");
        return nullptr;
    }
```

snappify.com

Summary

1] To implement modules in CPPInterOp in order to have access over C++ code and depending on requirements of our user(cppyy).

2] Connecting CPPInterOp and cppyy-backend so that user can have control over it.

3] Generate binding of C++ and python in cppyy and test it

4] Test passing approx. 185/504

Thank You