Bloomberg

P2911R0 - Python Bindings with Value-Based Reflection

Authors: Adam Lach, Jagrut Dave
Last Updated: Jun 11, 2023
Status: In progress

Abstract

Python/C++ bindings are heavily used in numerical calculation packages such as NumPy. The goal of this paper is to discuss the benefits and challenges of using value-based reflection (P2320 / P1240R2) to simplify creating C++ Python bindings. A previous attempt at simplifying Python bindings using reflection focused on Boost.Python and macro-based reflection, and can be found in the appendix. This paper uses contemporary value-based reflection, which has a path forward towards being accepted into the C++ standard, and is aimed at pybind11, a popular open-source Python library for binding existing C++ code to Python.

Introduction

Python bindings can be created by the means of the Python/C API[1]. It is, however, rarely used directly in practice. Instead, wrapper libraries like Boost.Python or pybind11 are frequently used[2]. For that reason, and for the sake of simplicity, this paper will focus on using C++ reflection to simplify creating Python bindings on top of pybind11.

Use Case

We have used a very simple implementation of an order crossing engine in C++, with a carefully tailored implementation to cover many common bindings applications like:

- Enumerations
- Data members
- Function members
- Constructors

[1] See https://docs.python.org/3/c-api/index.html
[2] There are other ways to create Python bindings like Cython or SWIG, which are not considered in this paper since they are not good candidates to be used with C++ reflection

- Inheritance
- Function overloads
- Nested type aliases
- Operators

In this document, we will present and discuss a subset of the above applications, which, in our opinion, are the most important and informative.

Enumerations

Enumerations are a very frequently used example of how reflection facilities can improve C++ code bases. To not break with this tradition, we demonstrate how this simple application of reflection can benefit writing Python bindings.

Considering a simple nested enumeration:

```C/C++
struct Execution {

      enum class Type {
          new_,
          fill,
          partial,
          cancelled,
          rejected
      };

};
```

The typical bindings code would look like:

```C/C++
py::enum_<Execution::Type>(binding scope, "Type")
   .value("new_"     , Execution::Type::new_)
   .value("fill"     , Execution::Type::fill)
   .value("partial"  , Execution::Type::partial)
   .value("cancelled", Execution::Type::cancelled)
   .value("rejected" , Execution::Type::rejected);
```

From the above, it can be observed that;

- There is plenty of repetition

- If `Execution::Type` is modified the bindings code has to be updated manually
- If an enumeration value is added to `Execution::Type` no compiler error / warning will be emitted hence bindings code can easily diverge from the bound code
- The names of individual enumerations have to be repeated as strings, which is prone to typos that cannot be detected by the compiler

With reflection, we can automate the task:

```C/C++
bind_enum<Execution::Type>(binding scope);
```

Where `bind_enum` can be implemented as[3]:

```C/C++
template<typename T>
std::string basename() {

    auto name = std::string{name_of(^T)}; // ^T reflects type T

    if (size_t pos = name.rfind(':');
        pos != std::string::npos) {
        return name.substr(name.rfind(':') + 1);
    }
    return name;
}

template<typename EnumT, typename Scope>
void bind_enum(Scope& s) {
    auto enum_ = py::enum_<EnumT>(s, basename<EnumT>().c_str());

    // members_of() produces an iterable list
    // of all the members of an enumeration.
    // `template for` iterates over a range at compile time.

    template for (constexpr auto e : members_of(^EnumT)) {
        enum_.value(name_of(e), [:e:]); // [:e:] un-reflects e
    }
}
```

[3] https://cppx.godbolt.org/z/T445M639n

Note that the reflection-based implementation does not suffer from any of the shortcomings mentioned earlier.

Data members

Binding public data members is seemingly a straightforward task.

Considering a simple aggregate type:

```C/C++
struct Order {
    int side = 1;
    size_t quantity = 0;
};
```

The typical bindings code would look like:

```C/C++
py::class_<Order>(binding scope, "Order")
    .def_readwrite("side", &Order::side)
    .def_readwrite("quantity", &Order::quantity);
```

From the above, it can be observed that:

- The names of data members have to be repeated as strings, which is prone to typos that cannot be detected by the compiler
- Since side and quantity members are mutable public data members then it is reasonable to provide both read and write access from Python

With reflection, we can automate the task:

```C/C++
bind_mem_var<Order>(binding scope);
```

Where `bind_mem_var` can be implemented[4] as:

[4] https://cppx.godbolt.org/z/3efezYqEE

C/C++

```
template<typename ClassT, typename Scope>
void bind_mem_var(Scope& s) {
    template for (constexpr auto e : data_member_range(^ClassT)){
        constexpr auto name = name_of(e);
        if constexpr (is_public(e) && !is_static_data_member(e)){
            if (has_const_type(e)) {
                s.def_readonly(name, &[:e:]);
            } else {
                s.def_readwrite(name, &[:e:]);
            }
        }
    }
}
```

Note that the reflection-based implementation does not suffer from any of the shortcomings mentioned earlier. There is an important caveat though, which is the choice of the default behavior. Frequently, Python bindings expose a more limited API than that offered by the underlying C++ code. In that case it would be beneficial to allow some way of customizing the behavior of `bind_mem_var` for selected data members. We discuss bindings customization in more detail in the conclusions sections.

Member functions

Bindings for member functions can be exposed in a similar way to data members due to existing quasi reflection capabilities of C++. Specifically it is possible to reflect on the return type and argument types of a function member using existing C++ features.

Considering a partial implementation of an order crossing engine:

C/C++

```
struct CrossingEngine {
    std::vector<Order> const& getAsks() const { return asks; }
    std::vector<Order> const& getBids() const { return bids; }

private:
    std::vector<Order> asks;
```

```
    std::vector<Order> bids;
};
```

As expected, a naive bindings code could look like:

C/C++

```
py::class_<CrossingEngine>(binding scope, "CrossingEngine")
    .def("getAsks", &CrossingEngine::getAsks)
    .def("getBids", &CrossingEngine::getBids);
```

This bindings implementation might not be ideal however, since by default pybind11 will copy `std::vector<Order>` into a python list object[5] every time `getAsks` or `getBids` is invoked from Python.

We can customize our implementation to avoid copying the return values as follows:

C/C++

```
PYBIND11_MAKE_OPAQUE(std::vector<Order>);
py::class_<CrossingEngine>(binding scope, "CrossingEngine")
    .def("getAsks", &CrossingEngine::getAsks,
                    return_value_policy::reference)
    .def("getBids", &CrossingEngine::getBids,
                    return_value_policy::reference);
```

Note the need for `PYBIND11_MAKE_OPAQUE` and `return_value_policy::reference` policy.

While this implementation solves the problem of unwanted data copies it introduces yet another problem which is more subtle. It stems from the difference in object lifetime management in C++ and Python. In case of the latter it is assumed that an object will be kept alive until at least one handle to that object exists. With our implementation of `CrossingEngine`, however, the references returned by `getAsks` and `getBids` will only be valid as long as the `CrossingEngine` object is alive. This has to be taken into account when creating bindings.

For example, it is perfectly reasonable to expect the following Python code to work correctly:

[5] This default approach is quite sensible as it avoids lifetime issues between Python and C++ and makes the resulting Python APIs more pythonic.

Python

```
def execute_and_get_remaining_asks(orders):
    engine = CrossingEngine()
    for order in orders: engine.cross(order)
    return engine.getAsks()

remaining_asks = execute_and_get_remaining_asks(orders)
print(remaining_asks)
```

It might happen[6] that `engine` will be garbage collected before `print(remaining_asks)` is called. As a consequence the C++ object representing `CrossingEngine` instance will be destroyed and `remaining_asks` will be a dangling reference. In order to address this shortcoming it is possible to use `return_value_policy::reference_internal` instead of plain `return_value_policy::reference`.

C/C++

```
PYBIND11_MAKE_OPAQUE(std::vector<Order>);
py::class_<CrossingEngine>(binding scope, "CrossingEngine")
    .def("getAsks", &CrossingEngine::getAsks,
                    return_value_policy::reference_internal)
    .def("getBids", &CrossingEngine::getBids,
                    return_value_policy::reference_internal);
```

It is straightforward to automate the bindings for the naive bindings case:

C/C++

```
bind_mem_fn<CrossingEngine>(binding scope);
```

Where `bind_mem_fn` can be implemented[7] as:

C/C++

```
template<typename ClassT, typename Scope>
void bind_mem_fn(Scope& s) {
```

[6] But it doesn't have to, which is even worse.
[7] https://cppx.godbolt.org/z/aMzdfnKdr

```
    template for (constexpr auto e : member_fn_range(^ClassT)) {
        if constexpr (is_public(e) &&
                      !is_special_member_function(e)) {
            constexpr auto name = name_of(e);
            if constexpr (is_nonstatic_member_function(e)) {
                s.def(name, py::overload_cast<
                                ...[:type_of(param_range(e)):]...
                            >(&[:e:]));
            } else {
                s.def_static(name, &[:e:]);
            }
        }
    }
}
```

Note that the `py::overload_cast<...>` is just a `static_cast<..>` in disguise used to disambiguate different overloads of the same function.

It is not possible, however, to solve the problem of unwanted copies and object lifetime management without providing some degree of user customization. We discuss the problem of bindings customization in more detail in the conclusions sections.

Constructors

Constructors are slightly different from member functions since it is not possible to take their address. As a consequence it is not possible to use existing C++ features to inspect the types of their parameters. To circumvent this limitation pybind11 provides a special `pybind11::init<...>` utility.

Considering a partial implementation of an Execution class:

```
C/C++

struct Execution {

    enum class Type { new_, fill, ... }
```

```
    Execution(Order order, Type type);
    Execution(Order order, Type type,
              double price, size_t quantity = 0);

};
```

The typical bindings code would look like (bar the enum which we handled before):

C/C++

```
py::class_<Execution>(binding scope, "Execution")
    .def(py::init<Order, Execution::Type>(),
         py::arg("order"), py::arg("type"))
    .def(py::init<Order, Execution::Type, double, size_t>(),
         py::arg("order"), py::arg("type"),
         py::arg("price"), py::arg("quantity") = 0);
```

While the usage of `init` should not be problematic to decipher, we simply pass all the argument types to the type list of the helper, the usage of `py::arg` allows the bindings module user to use a Python feature - keyword arguments.

With reflection, we can automate the task:

C/C++

```
bind_ctors<CrossingEngine>(binding scope);
```

Where `bind_ctors` could be implemented as:

C/C++

```
template<typename ClassT, typename Scope>
void bind_ctors(Scope& s) {
    template for (constexpr auto e : member_fn_range(^ClassT)) {
        if constexpr (is_public(e) && is_constructor(e) &&
                      !is_copy_constructor(e) &&
                      !is_move_constructor(e)) {
            constexpr auto params = param_range(e);
```

```
            s.def(py::init<...[:type_of(params):]...>(),
                  py::arg(name_of(^...[:params:]))...);
        }
    }
}
```

Note that the implementation of bind_ctors cannot be validated with the lock3 implementation of p2320 since it lacks pack splicing capabilities. Also note that the syntax for expanding reflections range into a list parameter names seems a bit clunky; we will discuss this in more detail in the Challenges section.

At first glance, the above implementation is straightforward and simple. However, the usage of parameter names for keyword arguments is problematic. This is due to the fact that parameter names are not part of a function signature and can change between declaration and definition.

Considering the following code:

C/C++

```
struct X {
   X(int name);
};

X:X(int different_name) { (void)different_name; };
```

It is not immediately clear which parameter name should be provided while reflecting on parameters of `X::X` when both the declaration and definition are visible. The only publicly available implementation of p2320 always returns the names of parameters of the declaration[8]. This problem becomes even more evident when free functions are considered since they can have multiple declarations with completely different parameter names. We discuss this problem in a bit more detail in the Conclusions section.

Overloaded Operators

Binding operators is a special problem. That is because they could be free functions and are subject to both ADL and visibility checks.

[8] This is true even if the reflection is done inside the definition of `X::X`

[To be continued...]

Conclusions

Advantages

We have determined that:

1. As expected, it is possible to achieve significant (~95%) boilerplate code reduction;
2. The usage of reflection has the potential to reduce the likelihood of error in many cases (e.g., enum bindings); and
3. Most of the bindings can be reasonably automated with carefully selected default behaviors (i.e., we have leveraged the defaults specified by pybind11).

Challenges

We have determined that:

1. bindings customization facilities cover the differences in language features between Python and C++;
2. some reflection features, like parameter name reflection, can be dangerous;
3. in some corner cases, reflection-based automation can hide problems and give a false sense of security; and
4. expanding reflections range into a list of its elements' names

In the next sections we will discuss the various challenges in more detail.

Customization

There are at least two general categories where bindings customization would be needed:

- overriding and/or improving binding defaults
- bridging the gap between languages

The first point can be visualized using a simple example of public data members bindings which should specify the allowed access type (read-only vs. read-write). While the default approach is easy to establish - if, and only if, the data member is non-const should the data member be writeable from Python. That is, however, not necessarily appropriate under all circumstances. In

practice, it is not an infrequent situation where Python code should either not be allowed to modify a public data member or should not have access to it altogether.

The second point is more about the specific features that both languages do and do not support. Some notable examples might be keyword arguments and garbage collection in Python and function overloading and polymorphism in C++. While pybind11 does a reasonably good job at providing facilities for bridging that gap, they typically require additional work. Some of that work can be automated, e.g., function overloading, but some might require manual intervention, e.g., specifying reference management policy.

It should be clear at this point that some user customization is necessary for any reflection-based Python bindings implementation. We can think of approaching that problem in two ways:

- Library specific hooks
- Custom attributes

Library Specific Hooks can be implemented in a multitude of ways. In our opinion, one of the simplest would be creating a constexpr list of modification for reflected entities, like in the example below:

C/C++

```
constexpr auto customizations = {
    {^CrossingEngine::getAsks,
     return_value_policy::reference_internal},
    {^Order::side,
     value_access_policy::readonly},
};

bind_class<CrossingEngine>(scope, customizations);
bind_class<Order>(scope, customizations);
```

On the positive side, with this approach it is possible to create and customize bindings of a code base that the bindings implementer has no control over. On the negative side, the customizations are disjointed from the C++ code that is being bound; therefore, there is a high risk of the two diverging and, as a consequence, introducing errors.

The other option is attaching custom attributes to the actual code that is being bound:

```C/C++
struct CrossingEngine {
    [[refl_bind::return_policy("reference_internal")]]
    std::vector<Order> const& getAsks() const { return asks; }

    [[refl_bind::return_policy("reference_internal")]]
    std::vector<Order> const& getBids() const { return bids; }

    ...
};
```

On the positive side, with this approach customizations would naturally evolve alongside the code. On the negative side, adding user-defined attributes requires control of the source code that is the subject of bindings and, what is probably more important, adding the support for user-defined attributes to the C++ language, lifting the requirement of ignorability of attributes[9], and adding support for reflecting on attributes.

We believe that both approaches to customizations are valuable in their own right, with user-defined attributes being less error prone, and, therefore, preferable wherever applicable.

Parameter Names

Python's keyword arguments allow specifying function parameter names and their values at the point where a function is called. This feature improves readability and so is used quite heavily. Therefore, it is desirable to make keyword arguments automatically available with C++/Python bindings. The natural way of doing so is to reflect on parameter names. However, this is dangerous. C++ parameter names are not part of the function signature and can change between function declaration and definition – and even across different declarations of the same function. The code below[10] illustrates this problem:

```C/C++
#include <experimental/meta>
#include <iostream>

using namespace std::experimental::meta;

// declaration 1
```

[9] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2552r0.pdf
[10] https://cppx.godbolt.org/z/coq6KhvdK

```
void func(int x, int y);

void print_func_params1() {
    std::cout << "func param names are: ";
    template for (constexpr auto e : param_range(^func)) {
        std::cout << name_of(e) << ", ";
    }
    std::cout << "\n";
}

// declaration 2
void func(int a, int b);

void print_func_params2() {
    std::cout << "func param names are: ";
    template for (constexpr auto e : param_range(^func)) {
        std::cout << name_of(e) << ", ";
    }
    std::cout << "\n";
}

int main() {
    print_func_params1(); // prints: func param names are: x, y,
    print_func_params2(); // prints: func param names are: a, b,
}
```

It is easy to see how using parameter names reflection makes the code fragile, as implementers do not expect that changing forward declaration parameter names will impact the output of the program in any way.

Range Name Expansion

We have encountered a situation where it would be useful to expand a function parameter range into a range of names, such as in the example below:

C/C++

```
struct X {
    void fun(int y, float z) {};
};
```

```cpp
template<typename... V>
void print(V... v) {
    (std::cout << ... << v ) << '\n';
}

int main() {
    constexpr auto param_range = param_range(^X::fun);
    print(/* expand param_range to a list of parameter names */);
}
```

We can see multiple possible ways in which this could be achieved:

1. `meta::name_of(param_range)...`

   It is unclear whether that syntax would work as we see no examples of reflections range pack expansion without using the splicing operator in p1240r2.

2. `meta::name_of(...[:param_range:])...`

   The `meta::name_of(meta::info)` takes a `meta::info` object, so this is unlikely to work; in fact we can confirm[11] that `meta::name_of([:*param_range.begin():])` does not compile.

3. `...[:meta::name_of(param_range):]...`

   Similarly applying `meta::name_of(meta::info)` inside a splicing expression can be confirmed[12] to not compile.

4. `meta::name_of(^...[:param_range:])...`

   This will probably work since `meta::name_of(^[:*param_range.begin():])` compiles fine[13], though the need to utilize ^ operator twice seems a bit clunky `...meta::name_of(^[:meta::param_range(^X::fun):])...`

[11] https://cppx.godbolt.org/z/rKb5WjGj9
[12] https://cppx.godbolt.org/z/a8ee54Ehe
[13] https://cppx.godbolt.org/z/9qnbn9x5G

ABI Compatibility

While the discussion of ABI compatibility is not strictly related to the usage of reflection for creating Python bindings, it has been an important consideration in C++/Python bindings discussion. ABI compatibility issues could occur in two cases:

1. Bindings were created with a Python library version that is incompatible with the Python interpreter that is loading them.
2. A type that is passed between two Python/C++ binding libraries has different binary representations between the two.

Point 1 is an obvious problem and we will not be discussing it here. For point 2, the problem can typically occur when bindings are shared. To visualize this, consider three C++ libraries: A, B, and C, with the caveat that both A and B depend on C. It can be easily observed that if A creates an object of a type X belonging to C, which is subsequently passed to B, both A and B have to use the same binary representation of X. To solve this problem at scale, we can see two approaches - using an integration build or fat bindings.

Integration Build

With this approach, all libraries and their bindings are built from source together and are deployed together. This way, the possibility of having multiple libraries with the same dependency, but different ABI representations, is eliminated

Pros

- Allows bindings to be re-used across libraries
- Each library is comprised of only the necessary binary code[14]
- Handles singletons without additional work

Cons

- Additional deployment time guarantees are necessary
- Can't be safely used out of the box with the Python Package Index (PyPI)

Fat Bindings

With this approach, every binding library statically links its dependencies, hides symbols, and exposes every C++ type as a distinct type in Python, hence avoiding any possible collisions.

[14] Only the code that is the subject of bindings and the bindings code itself. External dependencies and their bindings can be dynamically loaded by Python at runtime.

Pros

- No library re-use and hence no ABI problems[15]
- Safe to use with the Python Package Index (PyPI)

Cons

- Not possible to share singletons among libraries without additional logic[16]
- Library sizes are larger as each library is comprised of its own binary code as well as the code of all its dependencies
- Bindings cannot be re-used out of the box

Appendix

1. C++ Reflection for Python Binding - https://accu.org/journals/overload/27/152/standish_2682/
2. Pybind11 - https://pybind11.readthedocs.io/en/stable/
3. Programming for every language, everywhere all at once - CoreCpp ‘22 talk - https://www.youtube.com/watch?v=43Tmqn-sFsk
4. Reflection on attributes: https://wg21.link/p1887

[15] Notably pybind11 has an added feature that tries to recognize “compatible” types by additional means, which might still cause ABI compatibility problems.
[16] Since each extension links in all their dependencies and hides symbols, each module has its own version of a singleton. We do not know of any generic solution to this problem.