P2911R1 - Python Bindings with Value-Based Reflection

Authors: Adam Lach , Jagrut Dave
Last Updated: Sep 18, 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 and P1240R2) to simplify creating C++/Python bindings. A previous attempt at simplifying Python bindings using reflection, focused on Boost.Python and macro-based reflection, 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. Familiarity with value-based reflection APIs (P2320 and P1240R2) and pybind11 are assumed.

Updates since P2911R0

- Clarification about out of scope items.
- Discussion on reflecting function parameter names, and when that should raise an invalid reflection error.
- Provided a better example of dangerous behavior caused by default bindings
- Detailed discussion on expanding a range of reflected entities to produce a range of their names, meta::name_of/names_of functions.
- Added a section on overloaded operators bindings.
- Added a section comparing the Classdesc framework with value-based reflection and future work.
- Grammatical improvements.

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 the sake of simplicity, this paper will focus on using C++ reflection to simplify creating Python bindings on top of pybind11.

Out of Scope

- The Python/C API. Our research has not dwelled on the C API that allows for writing Python extensions. Though such an approach may provide performance benefits, we believe that most developers would prefer using the ergonomic API provided by pybind11, which is friendly to modern C++.
- For binding data members, pybind11 requires taking their address. However, this approach will not work with bit fields. One way to circumvent that problem is to generate setters and getters for such data members and bind them as properties[3]. Automating that using reflection should not pose any difficulties.

Use Cases

We have used a 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
- Inheritance
- Function overloads
- Nested type aliases
- Operators

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

[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.
[3] See https://pybind11.readthedocs.io/en/stable/classes.html#instance-and-static-fields

Enumerations

Enumerations are a common example of how reflection facilities can improve C++ code. To not part with this tradition, we start with an example showing Python bindings for an enum.

```
C/C++
struct Execution {

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

};
```

Typical bindings 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 seen that

- There is plenty of repetition.
- If `Execution::Type` is modified, then 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[4]:

```
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.
    }
}
```

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

Data members

Binding public data members is a seemingly straightforward task. Consider a simple aggregate type:

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

```
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);
```

It can be seen that:

- The names of data members have to be repeated as strings, which are prone to typos that cannot be detected by the compiler.
- Since side and quantity are mutable public data members, 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[5] as:

```
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 constexpr (has_const_type(e)) {
```

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

```
                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. However, there is an important caveat related to 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 select data members. We discuss bindings customization in more detail in the Conclusions section.

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 member function 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;
};
```

Naive bindings code could look like:

```
C/C++

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

However, this bindings implementation might not be ideal since, by default, pybind11 will copy `std::vector<Order>` into a Python list object[6] 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 cases of the latter, it is assumed that an object will be kept alive until at least one handle to that object exists. However, with our implementation of `CrossingEngine`, 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 reasonable to expect the following Python code to work correctly:

```
Python

def execute_and_get_remaining_asks(orders):
    engine = CrossingEngine()
```

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

```
    for order in orders: engine.cross(order)
    return engine.getAsks()

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

It might happen[7] 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 become a dangling reference. In order to address this shortcoming, it is possible to use `return_value_policy::reference_internal` instead of a 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 bindings for the basic case.

```
C/C++

bind_mem_fn<CrossingEngine>(binding_scope);
```

Where `bind_mem_fn` can be implemented[8] as:

```
C/C++

template<typename ClassT, typename Scope>
void bind_mem_fn(Scope& s) {
    template for (constexpr auto e : member_fn_range(^ClassT)) {
        if constexpr (is_public(e) &&
```

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

```
                    !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 that is used to disambiguate different overloads of the same function.

However, it is not possible 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 section.

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.

Consider 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, excluding enum bindings (which were discussed before), looks as follows:

```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 as follows:

```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<...typename [: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;
- The `type_of` function will not work for this use case. Instead, the `types_of` function should be used, but it is not proposed for P1240 yet; and
- The syntax for expanding a range of reflections into a parameter list of names seems a bit clunky; we will discuss this in more detail in the Challenges section.

At first glance, the above implementation is straightforward. However, using parameter names for keyword arguments is problematic. Parameter names are not part of a C++ function's signature and can change between declaration and definition, or among multiple declarations.

Consider the following code:

```
C/C++

struct X {
   X(int name);
};

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

The parameter name that should be provided while reflecting on the input parameter of `X::X`, when both the declaration and definition are visible, is ambiguous. The only publicly available implementation of P2320 returns names of parameters of the definition when queried directly, i.e., `param_range(^X::X)`. However, it returns the names of parameters of the declaration when accessed indirectly via a reflection of the enclosing class, i.e., `param_range(*member_fn_range(^X).begin())`[9]. This problem becomes even more severe when free functions are considered, since they can have multiple declarations with completely different parameter names. We discuss this in more detail in the Conclusions section.

Overloaded Operators

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

Binding operators is a special problem. It is complicated by the fact that overloaded operators can exist as member functions of a class, as inline friend functions, and as free functions, and are subject to both ADL and visibility checks.

Considering a simple class `X` with a set of associated `operator+` overloads

```
C/C++

namespace xns {

struct X {
    int v = 0;

    friend X operator+(X const& lhs, int rhs) {
        return X{lhs.v + rhs};
    }
};

X operator+(xns::X const& lhs, X const& rhs) {
    return X{lhs.v + rhs.v};
}

} // ::xns

namespace yns {

struct Y { double v = 0.; };

Y operator+(xns::X const& lhs, Y const& rhs) {
    return Y{lhs.v + rhs.v};
}

Y operator+(Y const& lhs, double rhs) {
    return Y{lhs.v + rhs.v};
}

} // ::yns
```

A naive bindings implementation could look like:

```
C/C++

py::class_<xns::X>{}
    .def("__add__", &xns::X::operator+) // Error: cannot take address of
an inline friend function

    .def("__add__", static_cast<
                        yns::Y (*)(xns::X const&, yns::Y const&)
                    >(&yns::operator+))
    .def("__add__", static_cast<
                        yns::Y (*)(xns::X const&, double)
                    >(&yns::operator+))
    .def("__add__", &xns::operator+);
```

It is possible to improve on this with pybind11 helpers[10]:

```
C/C++

py::class_<xns::X>{}
    .def(py::self + py::self)
    .def(py::self + int{})
    .def(py::self + double{})
    .def(py::self + yns::Y{});
```

With that, all overloads are detected and bindings are created correctly.

While attempting to automate operator bindings, the immediate challenge is how to detect all of the reachable `operator+` overloads. The scalable reflection paper (P1240R2) does not mention any specific facilities that would allow us to do that. We have therefore resorted to scanning all namespaces recursively starting from the global namespace. It has to be noted, however, that based on the initial feedback from compiler implementers, it seems feasible to propose and implement the discovery of overloaded operators by name, even with ADL lookup. This would most likely be limited to non-template functions since the discovery of those cannot be done reliably without knowing all possible parameter types up front.

With reflection, we can automate the task as follows:

[10] https://pybind11.readthedocs.io/en/stable/advanced/classes.html#operator-overloading

```
C/C++

template<typename T, typename Scope>
void bind_operators(Scope& scope) {
    bind_namespace_operators<^::>(scope);
    bind_member_operators<^T>(scope);
}
```

Where `bind_namespace_operators` could be implemented as[11]:

```
C/C++

template<info refl, typename T>
void bind_namespace_operators(py::class_<T>& cls) {

  template for (constexpr auto e : member_range(refl)) {

    if constexpr (is_function(e)) {
      constexpr auto rng = param_range(e);
      constexpr auto len = detail::distance(rng.begin(), rng.end());

      if constexpr (len == 2) {
        constexpr auto param1_t = type_of(*params_range.begin());
        constexpr auto param2_t = type_of(*(++params_range.begin()));

        if constexpr (addable<T, typename [:param2_t:]>
                      && same_as<typename [:param1_t:], T) {
          cls.def(py::self + typename [:param2_t :]{});

        } else
        if constexpr (addable<typename [:param1_t:], T>
                      && same_as<typename [:param2_t:], T>){
          cls.def(typename [:param1_t:]{} + py::self);
        }
      }
    } else if constexpr (is_namespace(e)) {
      bind_namespace_operators<e>(cls);
    }
```

[11] https://cppx.godbolt.org/z/7q4jaxsrn

```
    }
  }

  template<typename Lhs, typename Rhs>
  concept addable = requires (Lhs l, Rhs r) { l + r; };
```

While this works for the overloads residing in the namespace `yns`, the friend inline operator of `xns::X` could not be detected. This seems to be an unintended limitation as the friend inline operator should either be discoverable as a namespace member or as a class member.

While the approach we have taken to automate operator bindings is unlikely to scale well enough for any practical application, we have determined that there are use cases, especially ones involving templated code, where static reflection facilities will be insufficient to perform a task completely. In cases like this it would be useful to allow emitting diagnostic information at compile time (i.e. to the compiler output) which are not warnings or errors. Fortunately, there is already work going on[12] to make that possible.

Conclusions

Advantages

We have determined that:

1. As expected, it is possible to achieve significant (~95%) boilerplate code reduction, as opposed to manually written bindings code;
2. Using reflection for generating bindings avoids manual errors in many cases (e.g., enum bindings); and
3. Most 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. Feature gaps between between Python and C++, such as lifetime management, could be handled by customizing bindings;

[12] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2758r0.html

2. Some reflection features, like parameter name reflection, can be dangerous and must be used with caution;

3. Reflection-based automation is not foolproof. It can hide subtle problems and give a false sense of security; and

4. The syntax for expanding a range of reflections into a parameter list of names is a bit clunky.

In the following sections, we will discuss the various challenges in detail.

Customization

Bindings customization is needed in at least two applications:

- Overriding and/or improving binding defaults
- Bridging the gap between languages

In some cases, automatically produced bindings may be correct, but suboptimal, and cause resource leaks or crashes. Suppose an object is returned from C++ code that holds a raw pointer to a resource, such as a file or a memory buffer. The default behavior of copying out results from C++ to Python would cause a shallow copy of the pointer, assuming the copy constructor is implicitly generated. Now, both Python and C++ have access to the pointer, and could try to manipulate the underlying resource independently. E.g., one side might close the underlying file, while the other expects it to be open, and tries to write to it. In case the pointer held a memory buffer, and concurrent access was allowed, the concurrency control mechanism may not work as it would not account for non-native language access, or the buffer itself could be deleted or re-allocated. Thus, automatically produced bindings could lead to dangerous behavior, and the programmer does need to be aware of the intricate behavior of the C++ classes that are bound.

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, those facilities typically require additional work. Some of that work can be automated, e.g., function overloading, but some might require manual intervention, e.g., specifying the 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 approach that problem in two ways:

- By providing library-specific hooks
- By creating custom attributes

Library-specific hooks can be implemented in a multitude of ways. A simple way is to create a constexpr list of modifications 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 disjoint from the C++ code that is being bound; therefore, there is a risk of the two diverging, and errors being introduced.

Another option is to attach custom attributes to the 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[13], and adding support for reflecting on attributes.

[13] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2552r0.pdf

We believe that both approaches to customizations are valuable in their own right, with user-defined attributes being less error-prone, and, therefore, preferable where 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[14] illustrates this problem:

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

using namespace std::experimental::meta;

// declaration 1
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";
}
```

[14] https://cppx.godbolt.org/z/coq6KhvdK

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

Thus, reflecting on parameter names makes the code fragile, as developers don't expect that changing forward declaration parameter names will impact the output of the program in any way.

A quick survey of different compilers has found that there's no fixed pattern of how function signatures are evaluated across the different declarations of a function that are accessible from a translation unit. Some compilers use the first declaration that they parse, while others use the last. A solution that has gained consensus in SG7 is that if reflecting on parameter names results in more than one name for a parameter, then that reflection should be considered invalid, and a compile time error should be raised. Alternatively, the first or last declaration could be chosen, provided a diagnostic is emitted, letting the developer know about the choice and its line of code.

Another approach for creating unique parameter name lists is to introduce an attribute that explicitly specifies a function declaration or definition that should be used to infer parameter names. A function declared with that attribute, e.g., “infer_param_names” could be used to infer parameter names, rather than any other declaration or definition. This approach assumes that the C++ code to be bound is accessible to the developer who generates Python bindings, and the attribute is used in only one declaration, or the definition. E.g.,

C/C++

```
// A declaration of function foo. Parameter names used here are
ignored.

void foo(int n, int m);


// Another declaration of function foo, with the special
attribute.
```

```
 [[refl_bind::infer_param_names]] // Parameter names seen by
Python are taken from this declaration.

void foo(int apples, int bananas);



// Function definition. Parameter names used here are ignored.

void foo(int a, int b) {

     return a + b;

}
```

In the above example, the function foo()’s parameter name list as seen by Python would be {“apples”, “bananas"}, as indicated by the attribute infer_param_names.

Expanding a Range of Parameters

Consider a use-case where we need to expand a function parameter range into a range of names of those parameters:

```
C/C++

struct X {
    void fun(int y, float z) {};
};
void print(std::vector<py::arg> args) {
    for (auto const& arg : args)
        std::cout << arg.name << ',';
    std::cout << '\n';
}

int main() {
    constexpr auto param_range = param_range(^X::fun);
    print({/* expand to a vector of py::arg(param_name) */});
}
```

There are multiple ways to achieve this.

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

   It is unclear whether this syntax would work, as we see no examples of range pack expansion for a range of reflections, without using the splicing operator in P1240r2.

2. `...py::arg(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[15] that `meta::name_of([:*param_range.begin():])` does not compile.

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

   Similarly applying `meta::name_of(meta::info)` inside a splicing expression also does not compile[16].

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

   This will probably work since `meta::name_of(^[:*param_range.begin():])` compiles fine[17], though the need to utilize ^ operator twice seems a bit clunky `...py::arg(meta::name_of(^[:meta::param_range(^X::fun):]))...`

5. `py::arg(meta::names_of(param_range))`

   This was an alternative proposed during the SG7 meeting where the initial draft of this paper was presented. While the proposed `names_of` library function would produce a list of strings from a range of parameters, it would not be able to generate `py::arg`s from a range of parameters. Simply doing `py::arg(meta::names_of(param_range))` would result in `py::arg(“y”s,“x”s)` instead of the desired `{ py::arg(“y”s), py::arg(“x”s) }`.

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.

[15] https://cppx.godbolt.org/z/rKb5WjGj9
[16] https://cppx.godbolt.org/z/a8ee54Ehe
[17] https://cppx.godbolt.org/z/9qnbn9x5G

2. A type that is passed between two Python/C++ binding libraries has different binary representations between the two.

Point 1 is a ubiquitous problem for many Python features, and we will not be discussing it here. For point 2, the problem can typically occur when bindings are shared across libraries owned by different teams. 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 must 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 only the necessary binary code[18]
- Handles singletons without additional work

Cons

- Build and deployment time increase, all dependent libraries must be available when the integration build starts
- 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 collisions.

Pros

- No library re-use and hence no ABI problems[19]
- Safe to use with the Python Package Index (PyPI)

[18] 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.
[19] Notably pybind11 has an added feature that tries to recognize "compatible" types by additional means, which might still cause ABI compatibility problems.

Cons

- Not possible to share singletons among libraries without additional logic[20]
- Library sizes are larger, as each library comprises both its own binary code and the binary code of all its dependencies
- Bindings cannot be re-used out of the box

Comparison with Classdesc

In the absence of type information from the compiler, the C++ language exposes limited type information to the programmer, and so some form of a preprocessing system is required to obtain this information. Classdesc could be considered a static reflection system. At its core, it is a C++ preprocessor that reads C++ header files and generates overloaded function templates (or functor objects in later versions) that recursively call themselves on members of the class. The collection of overloaded functions is called a descriptor, and they are global functions. Classdesc consists of a simplified C++ parser/code generator along with support libraries implementing a range of reflection tasks, such as serialization and Python bindings. Classdesc classes are integrated with the build system, e.g., with a Makefile, and thus evolve with the evolution of the underlying C++ classes.

Classdesc has been used for generating Python bindings via Boost.Python. pybind11 ‘s roots are also in Boost.Python, and thus the syntax for creating Python bindings is similar in both libraries. Value-based reflection APIs have the benefit of being able to work on meta-information from the compiler itself, without having to pre-process code to produce an additional layer of classes that enable reflection. Classdesc suffers from some implementation nuances, such as not being able to reflect on functions that return pointers, while value-based reflection suffers from issues such as not having a complete set of reflection APIs, or missing or buggy implementation features in the compiler.

In terms of lines of code eliminated, the boilerplate lines of code eliminated by Classdesc are offset by the new code for shim classes needed for specific C++ features, such as containers, const and mutable attributes.

In terms of build times and memory consumed, experiments need to be carried out compiling the same application package using Classdesc vs. value-based reflection, to arrive at comparative numbers. Both approaches lead to increased build times and memory usage.

[20] 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.

Appendix

1. Classdesc: C++ Reflection for Python Binding - https://accu.org/journals/overload/27/152/standish_2682/
2. P2320R0 - The Syntax of Static Reflection - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2320r0.pdf
3. P1240R2 - Scalable Reflection in C++ - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1240r2.pdf
4. Pybind11 - https://pybind11.readthedocs.io/en/stable/
5. Programming for every language, everywhere all at once - CoreCpp ‘22 talk - https://www.youtube.com/watch?v=43Tmqn-sFsk
6. Reflection on attributes: https://wg21.link/p1887