pybind11 Documentation

Release 1.8.1

Wenzel Jakob

Oct 24, 2021

Contents


pybind11

CHAPTER 1

About this project

pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. Its goals and syntax are similar to the excellent Boost.Python library by David Abrahams: to minimize boilerplate code in traditional extension modules by inferring type information using compile-time introspection.

The main issue with Boost.Python—and the reason for creating such a similar project—is Boost. Boost is an enormously large and complex suite of utility libraries that works with almost every C++ compiler in existence. This compatibility has its cost: arcane template tricks and workarounds are necessary to support the oldest and buggiest of compiler specimens. Now that C++11-compatible compilers are widely available, this heavy machinery has become an excessively large and unnecessary dependency.

Think of this library as a tiny self-contained version of Boost.Python with everything stripped away that isn't relevant for binding generation. Without comments, the core header files only require ~2.5K lines of code and depend on Python (2.7 or 3.x) and the C++ standard library. This compact implementation was possible thanks to some of the new C++11 language features (specifically: tuples, lambda functions and variadic templates). Since its creation, this library has grown beyond Boost.Python in many ways, leading to dramatically simpler binding code in many common situations.

1.1 Core features

The following core C++ features can be mapped to Python

- Functions accepting and returning custom data structures per value, reference, or pointer
- Instance methods and static methods
- Overloaded functions
- Instance attributes and static attributes
- Exceptions
- Enumerations
- Iterators and ranges

- Callbacks
- Custom operators
- STL data structures
- Smart pointers with reference counting like `std::shared_ptr`
- Internal references with correct reference counting
- C++ classes with virtual (and pure virtual) methods can be extended in Python

1.2 Goodies

In addition to the core functionality, pybind11 provides some extra goodies:

- It is possible to bind C++11 lambda functions with captured variables. The lambda capture data is stored inside the resulting Python function object.
- pybind11 uses C++11 move constructors and move assignment operators whenever possible to efficiently transfer custom data types.
- It's easy to expose the internal storage of custom data types through Pythons' buffer protocols. This is handy e.g. for fast conversion between C++ matrix classes like Eigen and NumPy without expensive copy operations.
- pybind11 can automatically vectorize functions so that they are transparently applied to all entries of one or more NumPy array arguments.
- Python's slice-based access and assignment operations can be supported with just a few lines of code.
- Everything is contained in just a few header files; there is no need to link against any additional libraries.
- Binaries are generally smaller by a factor of 2 or more compared to equivalent bindings generated by Boost.Python.
- When supported by the compiler, two new C++14 features (relaxed constexpr and return value deduction) are used to precompute function signatures at compile time, leading to smaller binaries.
- With little extra effort, C++ types can be pickled and unpickled similar to regular Python objects.

1.3 Supported compilers

1. Clang/LLVM (any non-ancient version with C++11 support)
2. GCC (any non-ancient version with C++11 support)
3. Microsoft Visual Studio 2015 or newer
4. Intel C++ compiler v15 or newer

CHAPTER 2

First steps

This sections demonstrates the basic features of pybind11. Before getting started, make sure that development environment is set up to compile the included set of examples, which also double as test cases.

2.1 Compiling the test cases

2.1.1 Linux/MacOS

On Linux you'll need to install the **python-dev** or **python3-dev** packages as well as **cmake**. On Mac OS, the included python version works out of the box, but **cmake** must still be installed.

After installing the prerequisites, run

```
cmake .
make -j 4
```

followed by

```
make test
```

2.1.2 Windows

On Windows, use the CMake GUI to create a Visual Studio project. Note that only the 2015 release and newer versions are supported since pybind11 relies on various C++11 language features that break older versions of Visual Studio. After running CMake, open the created `pybind11.sln` file and perform a release build, which will will produce a file named `Release\example.pyd`. Copy this file to the `example` directory and run `example\run_test.py` using the targeted Python version.

Note: When all tests fail, make sure that

1. The Python binary and the testcases are compiled for the same processor type and bitness (i.e. either **i386** or **x86_64**)
2. The Python binary used to run `example\run_test.py` matches the Python version specified in the CMake GUI. This is controlled via the `PYTHON_EXECUTABLE` `PYTHON_INCLUDE_DIR`, and `PYTHON_LIBRARY` variables.

See also:

Advanced users who are already familiar with Boost.Python may want to skip the tutorial and look at the test cases in the `example` directory, which exercise all features of pybind11.

2.2 Creating bindings for a simple function

Let's start by creating Python bindings for an extremely simple function, which adds two numbers and returns their result:

```
int add(int i, int j) {
    return i + j;
}
```

For simplicity[1], we'll put both this function and the binding code into a file named `example.cpp` with the following contents:

```
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

namespace py = pybind11;

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    m.def("add", &add, "A function which adds two numbers");

    return m.ptr();
}
```

The *`PYBIND11_PLUGIN()`* macro creates a function that will be called when an `import` statement is issued from within Python. The next line creates a module named `example` (with the supplied docstring). The method *`module::def()`* generates binding code that exposes the `add()` function to Python. The last line returns the internal Python object associated with `m` to the Python interpreter.

Note: Notice how little code was needed to expose our function to Python: all details regarding the function's parameters and return value were automatically inferred using template metaprogramming. This overall approach and the used syntax are borrowed from Boost.Python, though the underlying implementation is very different.

---

pybind11 is a header-only-library, hence it is not necessary to link against any special libraries (other than Python itself). On Windows, use the CMake build file discussed in section *Building with CMake*. On Linux and Mac OS, the above example can be compiled using the following command

[1] In practice, implementation and binding code will generally be located in separate files.

```
$ c++ -O3 -shared -std=c++11 -I <path-to-pybind11>/include `python-config --cflags --
↪ldflags` example.cpp -o example.so
```

In general, it is advisable to include several additional build parameters that can considerably reduce the size of the created binary. Refer to section *Building with CMake* for a detailed example of a suitable cross-platform CMake-based build system.

Assuming that the created file `example.so` (`example.pyd` on Windows) is located in the current directory, the following interactive Python session shows how to load and execute the example.

```
$ python
Python 2.7.10 (default, Aug 22 2015, 20:33:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import example
>>> example.add(1, 2)
3L
>>>
```

2.3 Keyword arguments

With a simple modification code, it is possible to inform Python about the names of the arguments ("i" and "j" in this case).

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i"), py::arg("j"));
```

arg is one of several special tag classes which can be used to pass metadata into *module::def()*. With this modified binding code, we can now call the function using keyword arguments, which is a more readable alternative particularly for functions taking many parameters:

```
>>> import example
>>> example.add(i=1, j=2)
3L
```

The keyword names also appear in the function signatures within the documentation.

```
>>> help(example)

....

FUNCTIONS
    add(...)
        Signature : (i: int, j: int) -> int

        A function which adds two numbers
```

A shorter notation for named arguments is also available:

```
// regular notation
m.def("add1", &add, py::arg("i"), py::arg("j"));
// shorthand
using namespace pybind11::literals;
m.def("add2", &add, "i"_a, "j"_a);
```

The _a suffix forms a C++11 literal which is equivalent to *arg*. Note that the literal operator must first be made visible with the directive `using namespace pybind11::literals`. This does not bring in anything else from the `pybind11` namespace except for literals.

2.4 Default arguments

Suppose now that the function to be bound has default arguments, e.g.:

```
int add(int i = 1, int j = 2) {
    return i + j;
}
```

Unfortunately, pybind11 cannot automatically extract these parameters, since they are not part of the function's type information. However, they are simple to specify using an extension of *arg*:

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i") = 1, py::arg("j") = 2);
```

The default values also appear within the documentation.

```
>>> help(example)

....

FUNCTIONS
    add(...)
        Signature : (i: int = 1, j: int = 2) -> int

        A function which adds two numbers
```

The shorthand notation is also available for default arguments:

```
// regular notation
m.def("add1", &add, py::arg("i") = 1, py::arg("j") = 2);
// shorthand
m.def("add2", &add, "i"_a=1, "j"_a=2);
```

2.5 Supported data types

The following basic data types are supported out of the box (some may require an additional extension header to be included). To pass other data structures as arguments and return values, refer to the section on binding *Object-oriented code*.

Data type	Description	Header file
`int8_t`, `uint8_t`	8-bit integers	`pybind11/pybind11.h`
`int16_t`, `uint16_t`	16-bit integers	`pybind11/pybind11.h`
`int32_t`, `uint32_t`	32-bit integers	`pybind11/pybind11.h`
`int64_t`, `uint64_t`	64-bit integers	`pybind11/pybind11.h`
`ssize_t`, `size_t`	Platform-dependent size	`pybind11/pybind11.h`
`float`, `double`	Floating point types	`pybind11/pybind11.h`
`bool`	Two-state Boolean type	`pybind11/pybind11.h`
`char`	Character literal	`pybind11/pybind11.h`
`wchar_t`	Wide character literal	`pybind11/pybind11.h`
`const char *`	UTF-8 string literal	`pybind11/pybind11.h`
`const wchar_t *`	Wide string literal	`pybind11/pybind11.h`
`std::string`	STL dynamic UTF-8 string	`pybind11/pybind11.h`
`std::wstring`	STL dynamic wide string	`pybind11/pybind11.h`
`std::pair<T1, T2>`	Pair of two custom types	`pybind11/pybind11.h`
`std::tuple<...>`	Arbitrary tuple of types	`pybind11/pybind11.h`
`std::reference_wrapper<...>`	Reference type wrapper	`pybind11/pybind11.h`
`std::complex<T>`	Complex numbers	`pybind11/complex.h`
`std::array<T, Size>`	STL static array	`pybind11/stl.h`
`std::vector<T>`	STL dynamic array	`pybind11/stl.h`
`std::list<T>`	STL linked list	`pybind11/stl.h`
`std::map<T1, T2>`	STL ordered map	`pybind11/stl.h`
`std::unordered_map<T1, T2>`	STL unordered map	`pybind11/stl.h`
`std::set<T>`	STL ordered set	`pybind11/stl.h`
`std::unordered_set<T>`	STL unordered set	`pybind11/stl.h`
`std::function<...>`	STL polymorphic function	`pybind11/functional.h`
`Eigen::Matrix<...>`	Dense Eigen matrices	`pybind11/eigen.h`
`Eigen::SparseMatrix<...>`	Sparse Eigen matrices	`pybind11/eigen.h`

CHAPTER 3

Object-oriented code

3.1 Creating bindings for a custom type

Let's now look at a more complex example where we'll create bindings for a custom C++ data structure named `Pet`. Its definition is given below:

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void setName(const std::string &name_) { name = name_; }
    const std::string &getName() const { return name; }

    std::string name;
};
```

The binding code for `Pet` looks as follows:

```
#include <pybind11/pybind11.h>

namespace py = pybind11;

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("setName", &Pet::setName)
        .def("getName", &Pet::getName);

    return m.ptr();
}
```

`class_` creates bindings for a C++ *class* or *struct*-style data structure. `init()` is a convenience function that takes the types of a constructor's parameters as template arguments and wraps the corresponding constructor (see the *Custom constructors* section for details). An interactive Python session demonstrating this example is shown below:

```
% python
>>> import example
>>> p = example.Pet('Molly')
>>> print(p)
<example.Pet object at 0x10cd98060>
>>> p.getName()
u'Molly'
>>> p.setName('Charly')
>>> p.getName()
u'Charly'
```

See also:

Static member functions can be bound in the same way using `class_::def_static()`.

3.2 Keyword and default arguments

It is possible to specify keyword and default arguments using the syntax discussed in the previous chapter. Refer to the sections *Keyword arguments* and *Default arguments* for details.

3.3 Binding lambda functions

Note how `print(p)` produced a rather useless summary of our data structure in the example above:

```
>>> print(p)
<example.Pet object at 0x10cd98060>
```

To address this, we could bind an utility function that returns a human-readable summary to the special method slot named `__repr__`. Unfortunately, there is no suitable functionality in the `Pet` data structure, and it would be nice if we did not have to change it. This can easily be accomplished by binding a Lambda function instead:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def("setName", &Pet::setName)
    .def("getName", &Pet::getName)
    .def("__repr__",
        [](const Pet &a) {
            return "<example.Pet named '" + a.name + "'>";
        }
    );
```

Both stateless[1] and stateful lambda closures are supported by pybind11. With the above change, the same Python code now produces the following output:

```
>>> print(p)
<example.Pet named 'Molly'>
```

[1] Stateless closures are those with an empty pair of brackets `[]` as the capture object.

3.4 Instance and static fields

We can also directly expose the `name` field using the `class_::def_readwrite()` method. A similar `class_::def_readonly()` method also exists for `const` fields.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name)
    // ... remainder ...
```

This makes it possible to write

```
>>> p = example.Pet('Molly')
>>> p.name
u'Molly'
>>> p.name = 'Charly'
>>> p.name
u'Charly'
```

Now suppose that `Pet::name` was a private internal variable that can only be accessed via setters and getters.

```
class Pet {
public:
    Pet(const std::string &name) : name(name) { }
    void setName(const std::string &name_) { name = name_; }
    const std::string &getName() const { return name; }
private:
    std::string name;
};
```

In this case, the method `class_::def_property()` (`class_::def_property_readonly()` for read-only data) can be used to provide a field-like interface within Python that will transparently call the setter and getter functions:

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_property("name", &Pet::getName, &Pet::setName)
    // ... remainder ...
```

See also:

Similar functions `class_::def_readwrite_static()`, `class_::def_readonly_static()` `class_::def_property_static()`, and `class_::def_property_readonly_static()` are provided for binding static variables and properties.

3.5 Inheritance

Suppose now that the example consists of two data structures with an inheritance relationship:

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    std::string name;
};

struct Dog : Pet {
```

(continues on next page)

(continued from previous page)

```
    Dog(const std::string &name) : Pet(name) { }
    std::string bark() const { return "woof!"; }
};
```

There are two different ways of indicating a hierarchical relationship to pybind11: the first is by specifying the C++ base class explicitly during construction using the `base` attribute:

```
py::class_<Pet>(m, "Pet")
   .def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

py::class_<Dog>(m, "Dog", py::base<Pet>() /* <- specify C++ parent type */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Alternatively, we can also assign a name to the previously bound `Pet class_` object and reference it when binding the `Dog` class:

```
py::class_<Pet> pet(m, "Pet");
pet.def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

py::class_<Dog>(m, "Dog", pet /* <- specify Python parent type */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Functionality-wise, both approaches are completely equivalent. Afterwards, instances will expose fields and methods of both types:

```
>>> p = example.Dog('Molly')
>>> p.name
u'Molly'
>>> p.bark()
u'woof!'
```

3.6 Overloaded methods

Sometimes there are several overloaded C++ methods with the same name taking different kinds of input arguments:

```
struct Pet {
    Pet(const std::string &name, int age) : name(name), age(age) { }

    void set(int age) { age = age; }
    void set(const std::string &name) { name = name; }

    std::string name;
    int age;
};
```

Attempting to bind `Pet::set` will cause an error since the compiler does not know which method the user intended to select. We can disambiguate by casting them to function pointers. Binding multiple functions to the same Python name automatically creates a chain of function overloads that will be tried in sequence.

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &, int>())
    .def("set", (void (Pet::*)(int)) &Pet::set, "Set the pet's age")
    .def("set", (void (Pet::*)(const std::string &)) &Pet::set, "Set the pet's name");
```

The overload signatures are also visible in the method's docstring:

```
>>> help(example.Pet)

class Pet(__builtin__.object)
 |  Methods defined here:
 |
 |  __init__(...)
 |      Signature : (Pet, str, int) -> NoneType
 |
 |  set(...)
 |      1. Signature : (Pet, int) -> NoneType
 |
 |      Set the pet's age
 |
 |      2. Signature : (Pet, str) -> NoneType
 |
 |      Set the pet's name
```

Note: To define multiple overloaded constructors, simply declare one after the other using the `.def(py::init<...>())` syntax. The existing machinery for specifying keyword and default arguments also works.

3.7 Enumerations and internal types

Let's now suppose that the example class contains an internal enumeration type, e.g.:

```
struct Pet {
    enum Kind {
        Dog = 0,
        Cat
    };

    Pet(const std::string &name, Kind type) : name(name), type(type) { }

    std::string name;
    Kind type;
};
```

The binding code for this example looks as follows:

```
py::class_<Pet> pet(m, "Pet");

pet.def(py::init<const std::string &, Pet::Kind>())
    .def_readwrite("name", &Pet::name)
    .def_readwrite("type", &Pet::type);

py::enum_<Pet::Kind>(pet, "Kind")
    .value("Dog", Pet::Kind::Dog)
```

(continues on next page)

(continued from previous page)

```
    .value("Cat", Pet::Kind::Cat)
    .export_values();
```

To ensure that the `Kind` type is created within the scope of `Pet`, the `pet class_` instance must be supplied to the `enum_.` constructor. The `enum_::export_values()` function exports the enum entries into the parent scope, which should be skipped for newer C++11-style strongly typed enums.

```
>>> p = Pet('Lucy', Pet.Cat)
>>> p.type
Kind.Cat
>>> int(p.type)
1L
```

CHAPTER 4

Advanced topics

For brevity, the rest of this chapter assumes that the following two lines are present:

```
#include <pybind11/pybind11.h>

namespace py = pybind11;
```

4.1 Exporting constants and mutable objects

To expose a C++ constant, use the `attr` function to register it in a module as shown below. The `int_` class is one of many small wrapper objects defined in `pybind11/pytypes.h`. General objects (including integers) can also be converted using the function `cast`.

```
PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");
    m.attr("MY_CONSTANT") = py::int_(123);
    m.attr("MY_CONSTANT_2") = py::cast(new MyObject());
}
```

4.2 Operator overloading

Suppose that we're given the following `Vector2` class with a vector addition and scalar multiplication operation, all implemented using overloaded operators in C++.

```
class Vector2 {
public:
    Vector2(float x, float y) : x(x), y(y) { }

    Vector2 operator+(const Vector2 &v) const { return Vector2(x + v.x, y + v.y); }
    Vector2 operator*(float value) const { return Vector2(x * value, y * value); }
```

(continues on next page)

(continued from previous page)

```
    Vector2& operator+=(const Vector2 &v) { x += v.x; y += v.y; return *this; }
    Vector2& operator*=(float v) { x *= v; y *= v; return *this; }

    friend Vector2 operator*(float f, const Vector2 &v) {
        return Vector2(f * v.x, f * v.y);
    }

    std::string toString() const {
        return "[" + std::to_string(x) + ", " + std::to_string(y) + "]";
    }
private:
    float x, y;
};
```

The following snippet shows how the above operators can be conveniently exposed to Python.

```
#include <pybind11/operators.h>

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    py::class_<Vector2>(m, "Vector2")
        .def(py::init<float, float>())
        .def(py::self + py::self)
        .def(py::self += py::self)
        .def(py::self *= float())
        .def(float() * py::self)
        .def("__repr__", &Vector2::toString);

    return m.ptr();
}
```

Note that a line like

```
.def(py::self * float())
```

is really just short hand notation for

```
.def("__mul__", [](const Vector2 &a, float b) {
    return a * b;
})
```

This can be useful for exposing additional operators that don't exist on the C++ side, or to perform other types of customization.

Note: To use the more convenient `py::self` notation, the additional header file `pybind11/operators.h` must be included.

See also:

The file `example/example3.cpp` contains a complete example that demonstrates how to work with overloaded operators in more detail.

4.3 Callbacks and passing anonymous functions

The C++11 standard brought lambda functions and the generic polymorphic function wrapper `std::function<>` to the C++ programming language, which enable powerful new ways of working with functions. Lambda functions come in two flavors: stateless lambda function resemble classic function pointers that link to an anonymous piece of code, while stateful lambda functions additionally depend on captured variables that are stored in an anonymous *lambda closure object*.

Here is a simple example of a C++ function that takes an arbitrary function (stateful or stateless) with signature `int -> int` as an argument and runs it with the value 10.

```
int func_arg(const std::function<int(int)> &f) {
    return f(10);
}
```

The example below is more involved: it takes a function of signature `int -> int` and returns another function of the same kind. The return value is a stateful lambda function, which stores the value `f` in the capture object and adds 1 to its return value upon execution.

```
std::function<int(int)> func_ret(const std::function<int(int)> &f) {
    return [f](int i) {
        return f(i) + 1;
    };
}
```

After including the extra header file `pybind11/functional.h`, it is almost trivial to generate binding code for both of these functions.

```
#include <pybind11/functional.h>

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    m.def("func_arg", &func_arg);
    m.def("func_ret", &func_ret);

    return m.ptr();
}
```

The following interactive session shows how to call them from Python.

```
$ python
>>> import example
>>> def square(i):
...     return i * i
...
>>> example.func_arg(square)
100L
>>> square_plus_1 = example.func_ret(square)
>>> square_plus_1(4)
17L
>>>
```

Note: This functionality is very useful when generating bindings for callbacks in C++ libraries (e.g. a graphical user interface library).

The file `example/example5.cpp` contains a complete example that demonstrates how to work with callbacks and anonymous functions in more detail.

Warning: Keep in mind that passing a function from C++ to Python (or vice versa) will instantiate a piece of wrapper code that translates function invocations between the two languages. Copying the same function back and forth between Python and C++ many times in a row will cause these wrappers to accumulate, which can decrease performance.

4.4 Overriding virtual functions in Python

Suppose that a C++ class or interface has a virtual function that we'd like to to override from within Python (we'll focus on the class `Animal`; `Dog` is given as a specific example of how one would do this with traditional C++ code).

```
class Animal {
public:
    virtual ~Animal() { }
    virtual std::string go(int n_times) = 0;
};

class Dog : public Animal {
public:
    std::string go(int n_times) {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += "woof! ";
        return result;
    }
};
```

Let's also suppose that we are given a plain function which calls the function `go()` on an arbitrary `Animal` instance.

```
std::string call_go(Animal *animal) {
    return animal->go(3);
}
```

Normally, the binding code for these classes would look as follows:

```
PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    py::class_<Animal> animal(m, "Animal");
    animal
        .def("go", &Animal::go);

    py::class_<Dog>(m, "Dog", animal)
        .def(py::init<>());

    m.def("call_go", &call_go);

    return m.ptr();
}
```

However, these bindings are impossible to extend: `Animal` is not constructible, and we clearly require some kind of "trampoline" that redirects virtual calls back to Python.

Defining a new type of `Animal` from within Python is possible but requires a helper class that is defined as follows:

```
class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) {
        PYBIND11_OVERLOAD_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function */
            n_times      /* Argument(s) */
        );
    }
};
```

The macro `PYBIND11_OVERLOAD_PURE()` should be used for pure virtual functions, and `PYBIND11_OVERLOAD()` should be used for functions which have a default implementation.

There are also two alternate macros `PYBIND11_OVERLOAD_PURE_NAME()` and `PYBIND11_OVERLOAD_NAME()` which take a string-valued name argument after the *Name of the function* slot. This is useful when the C++ and Python versions of the function have different names, e.g. `operator()` vs `__call__`.

The binding code also needs a few minor adaptations (highlighted):

```
PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    py::class_<PyAnimal> animal(m, "Animal");
    animal
        .alias<Animal>()
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog>(m, "Dog", animal)
        .def(py::init<>());

    m.def("call_go", &call_go);

    return m.ptr();
}
```

Importantly, the trampoline helper class is used as the template argument to `class_`, and a call to `class_::alias()` informs the binding generator that this is merely an alias for the underlying type `Animal`. Following this, we are able to define a constructor as usual.

The Python session below shows how to override `Animal::go` and invoke it via a virtual method call.

```
>>> from example import *
>>> d = Dog()
>>> call_go(d)
u'woof! woof! woof! '
>>> class Cat(Animal):
```

(continues on next page)

(continued from previous page)

```
...     def go(self, n_times):
...             return "meow! " * n_times
...
>>> c = Cat()
>>> call_go(c)
u'meow! meow! meow! '
```

Please take a look at the *General notes regarding convenience macros* before using this feature.

See also:

The file `example/example12.cpp` contains a complete example that demonstrates how to override virtual functions using pybind11 in more detail.

4.5 General notes regarding convenience macros

pybind11 provides a few convenience macros such as `PYBIND11_MAKE_OPAQUE()` and `PYBIND11_DECLARE_HOLDER_TYPE()`, and `PYBIND11_OVERLOAD_*`. Since these are "just" macros that are evaluated in the preprocessor (which has no concept of types), they *will* get confused by commas in a template argument such as `PYBIND11_OVERLOAD(MyReturnValue<T1, T2>, myFunc)`. In this case, the preprocessor assumes that the comma indicates the beginnning of the next parameter. Use a `typedef` to bind the template to another name and use it in the macro to avoid this problem.

4.6 Global Interpreter Lock (GIL)

The classes `gil_scoped_release` and `gil_scoped_acquire` can be used to acquire and release the global interpreter lock in the body of a C++ function call. In this way, long-running C++ code can be parallelized using multiple Python threads. Taking the previous section as an example, this could be realized as follows (important changes highlighted):

```
class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) {
        /* Acquire GIL before calling Python code */
        py::gil_scoped_acquire acquire;

        PYBIND11_OVERLOAD_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function */
            n_times      /* Argument(s) */
        );
    }
};

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");
```

(continues on next page)

(continued from previous page)

```
    py::class_<PyAnimal> animal(m, "Animal");
    animal
        .alias<Animal>()
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog>(m, "Dog", animal)
        .def(py::init<>());

    m.def("call_go", [](Animal *animal) -> std::string {
        /* Release GIL before calling into (potentially long-running) C++ code */
        py::gil_scoped_release release;
        return call_go(animal);
    });

    return m.ptr();
}
```

4.7 Passing STL data structures

When including the additional header file `pybind11/stl.h`, conversions between `std::vector<>`, `std::list<>`, `std::set<>`, and `std::map<>` and the Python `list`, `set` and `dict` data structures are automatically enabled. The types `std::pair<>` and `std::tuple<>` are already supported out of the box with just the core `pybind11/pybind11.h` header.

Note: Arbitrary nesting of any of these types is supported.

See also:

The file `example/example2.cpp` contains a complete example that demonstrates how to pass STL data types in more detail.

4.8 Binding sequence data types, iterators, the slicing protocol, etc.

Please refer to the supplemental example for details.

See also:

The file `example/example6.cpp` contains a complete example that shows how to bind a sequence data type, including length queries (`__len__`), iterators (`__iter__`), the slicing protocol and other kinds of useful operations.

4.9 Return value policies

Python and C++ use wildly different ways of managing the memory and lifetime of objects managed by them. This can lead to issues when creating bindings for functions that return a non-trivial type. Just by looking at the type information, it is not clear whether Python should take charge of the returned value and eventually free its resources, or if this is handled on the C++ side. For this reason, pybind11 provides a several *return value policy* annotations that can be passed to the *`module::def()`* and `class_::def()` functions. The default policy is `return_value_policy::automatic`.

Return value policy	Description
`return_value_policy::automatic`	This is the default return value policy, which falls back to the policy `return_value_policy::take_ownership` when the return value is a pointer. Otherwise, it uses `return_value::move` or `return_value::copy` for rvalue and lvalue references, respectively. See below for a description of what all of these different policies do.
`return_value_policy::automatic_reference`	As above, but use policy `return_value_policy::reference` when the return value is a pointer. You probably won't need to use this.
`return_value_policy::take_ownership`	Reference an existing object (i.e. do not create a new copy) and take ownership. Python will call the destructor and delete operator when the object's reference count reaches zero. Undefined behavior ensues when the C++ side does the same..
`return_value_policy::copy`	Create a new copy of the returned object, which will be owned by Python. This policy is comparably safe because the lifetimes of the two instances are decoupled.
`return_value_policy::move`	Use `std::move` to move the return value contents into a new instance that will be owned by Python. This policy is comparably safe because the lifetimes of the two instances (move source and destination) are decoupled.
`return_value_policy::reference`	Reference an existing object, but do not take ownership. The C++ side is responsible for managing the object's lifetime and deallocating it when it is no longer used. Warning: undefined behavior will ensue when the C++ side deletes an object that is still referenced and used by Python.
`return_value_policy::reference_internal`	This policy only applies to methods and properties. It references the object without taking ownership similar to the above `return_value_policy::reference` policy. In contrast to that policy, the function or property's implicit `this` argument (called the *parent*) is considered to be the the owner of the return value (the *child*). pybind11 then couples the lifetime of the parent to the child via a reference relationship that ensures that the parent cannot be garbage collected while Python is still using the child. More advanced variations of this scheme are also possible using combinations of `return_value_policy::reference` and the `keep_alive` call policy described next.

The following example snippet shows a use case of the `return_value_policy::reference_internal` policy.

```
class Example {
public:
    Internal &get_internal() { return internal; }
private:
    Internal internal;
};

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind11 example plugin");

    py::class_<Example>(m, "Example")
        .def(py::init<>())
        .def("get_internal", &Example::get_internal, "Return the internal data",
                             py::return_value_policy::reference_internal);

    return m.ptr();
}
```

Warning: Code with invalid call policies might access unitialized memory or free data structures multiple times, which can lead to hard-to-debug non-determinism and segmentation faults, hence it is worth spending the time to understand all the different options in the table above.

Note: The next section on *Additional call policies* discusses *call policies* that can be specified *in addition* to a return value policy from the list above. Call policies indicate reference relationships that can involve both return values and parameters of functions.

Note: As an alternative to elaborate call policies and lifetime management logic, consider using smart pointers (see the section on *Smart pointers* for details). Smart pointers can tell whether an object is still referenced from C++ or Python, which generally eliminates the kinds of inconsistencies that can lead to crashes or undefined behavior. For functions returning smart pointers, it is not necessary to specify a return value policy.

4.10 Additional call policies

In addition to the above return value policies, further *call policies* can be specified to indicate dependencies between parameters. There is currently just one policy named `keep_alive<Nurse, Patient>`, which indicates that the argument with index `Patient` should be kept alive at least until the argument with index `Nurse` is freed by the garbage collector; argument indices start at one, while zero refers to the return value. For methods, index one refers to the implicit `this` pointer, while regular arguments begin at index two. Arbitrarily many call policies can be specified.

Consider the following example: the binding code for a list append operation that ties the lifetime of the newly added element to the underlying container might be declared as follows:

```
py::class_<List>(m, "List")
    .def("append", &List::append, py::keep_alive<1, 2>());
```

Note: `keep_alive` is analogous to the `with_custodian_and_ward` (if Nurse, Patient != 0) and `with_custodian_and_ward_postcall` (if Nurse/Patient == 0) policies from Boost.Python.

See also:

The file `example/example13.cpp` contains a complete example that demonstrates using `keep_alive` in more detail.

4.11 Implicit type conversions

Suppose that instances of two types `A` and `B` are used in a project, and that an `A` can easily be converted into an instance of type `B` (examples of this could be a fixed and an arbitrary precision number type).

```
py::class_<A>(m, "A")
    /// ... members ...

py::class_<B>(m, "B")
    .def(py::init<A>())
    /// ... members ...

m.def("func",
    [](const B &) { /* .... */ }
);
```

To invoke the function `func` using a variable `a` containing an `A` instance, we'd have to write `func(B(a))` in Python. On the other hand, C++ will automatically apply an implicit type conversion, which makes it possible to directly write `func(a)`.

In this situation (i.e. where `B` has a constructor that converts from `A`), the following statement enables similar implicit conversions on the Python side:

```
py::implicitly_convertible<A, B>();
```

4.12 Unique pointers

Given a class `Example` with Python bindings, it's possible to return instances wrapped in C++11 unique pointers, like so

```
std::unique_ptr<Example> create_example() { return std::unique_ptr<Example>(new␣
↪Example()); }
```

```
m.def("create_example", &create_example);
```

In other words, there is nothing special that needs to be done. While returning unique pointers in this way is allowed, it is *illegal* to use them as function arguments. For instance, the following function signature cannot be processed by pybind11.

```
void do_something_with_example(std::unique_ptr<Example> ex) { ... }
```

The above signature would imply that Python needs to give up ownership of an object that is passed to this function, which is generally not possible (for instance, the object might be referenced elsewhere).

4.13 Smart pointers

This section explains how to pass values that are wrapped in "smart" pointer types with internal reference counting. For the simpler C++11 unique pointers, refer to the previous section.

The binding generator for classes, `class_`, takes an optional second template type, which denotes a special *holder* type that is used to manage references to the object. When wrapping a type named `Type`, the default value of this template parameter is `std::unique_ptr<Type>`, which means that the object is deallocated when Python's reference count goes to zero.

It is possible to switch to other types of reference counting wrappers or smart pointers, which is useful in codebases that rely on them. For instance, the following snippet causes `std::shared_ptr` to be used instead.

```
py::class_<Example, std::shared_ptr<Example> /* <- holder type */> obj(m, "Example");
```

Note that any particular class can only be associated with a single holder type.

To enable transparent conversions for functions that take shared pointers as an argument or that return them, a macro invocation similar to the following must be declared at the top level before any binding code:

```
PYBIND11_DECLARE_HOLDER_TYPE(T, std::shared_ptr<T>);
```

Note: The first argument of `PYBIND11_DECLARE_HOLDER_TYPE()` should be a placeholder name that is used as a template parameter of the second argument. Thus, feel free to use any identifier, but use it consistently on both sides; also, don't use the name of a type that already exists in your codebase.

One potential stumbling block when using holder types is that they need to be applied consistently. Can you guess what's broken about the following binding code?

```
class Child { };

class Parent {
public:
   Parent() : child(std::make_shared<Child>()) { }
   Child *get_child() { return child.get(); }  /* Hint: ** DON'T DO THIS ** */
private:
    std::shared_ptr<Child> child;
};

PYBIND11_PLUGIN(example) {
    py::module m("example");

    py::class_<Child, std::shared_ptr<Child>>(m, "Child");

    py::class_<Parent, std::shared_ptr<Parent>>(m, "Parent")
       .def(py::init<>())
       .def("get_child", &Parent::get_child);

    return m.ptr();
}
```

The following Python code will cause undefined behavior (and likely a segmentation fault).

```
from example import Parent
print(Parent().get_child())
```

The problem is that `Parent::get_child()` returns a pointer to an instance of `Child`, but the fact that this instance is already managed by `std::shared_ptr<...>` is lost when passing raw pointers. In this case, pybind11 will create a second independent `std::shared_ptr<...>` that also claims ownership of the pointer. In the end, the object will be freed **twice** since these shared pointers have no way of knowing about each other.

There are two ways to resolve this issue:

1. For types that are managed by a smart pointer class, never use raw pointers in function arguments or return values. In other words: always consistently wrap pointers into their designated holder types (such as `std::shared_ptr<...>`). In this case, the signature of `get_child()` should be modified as follows:

```
std::shared_ptr<Child> get_child() { return child; }
```

2. Adjust the definition of `Child` by specifying `std::enable_shared_from_this<T>` (see cppreference for details) as a base class. This adds a small bit of information to `Child` that allows pybind11 to realize that there is already an existing `std::shared_ptr<...>` and communicate with it. In this case, the declaration of `Child` should look as follows:

```
class Child : public std::enable_shared_from_this<Child> { };
```

Please take a look at the *General notes regarding convenience macros* before using this feature.

See also:

The file `example/example8.cpp` contains a complete example that demonstrates how to work with custom reference-counting holder types in more detail.

4.14 Custom constructors

The syntax for binding constructors was previously introduced, but it only works when a constructor with the given parameters actually exists on the C++ side. To extend this to more general cases, let's take a look at what actually happens under the hood: the following statement

```
py::class_<Example>(m, "Example")
    .def(py::init<int>());
```

is short hand notation for

```
py::class_<Example>(m, "Example")
    .def("__init__",
        [](Example &instance, int arg) {
            new (&instance) Example(arg);
        }
    );
```

In other words, `init()` creates an anonymous function that invokes an in-place constructor. Memory allocation etc. is already take care of beforehand within pybind11.

4.15 Catching and throwing exceptions

When C++ code invoked from Python throws an `std::exception`, it is automatically converted into a Python `Exception`. pybind11 defines multiple special exception classes that will map to different types of Python exceptions:

C++ exception type	Python exception type
`std::exception`	`RuntimeError`
`std::bad_alloc`	`MemoryError`
`std::domain_error`	`ValueError`
`std::invalid_argument`	`ValueError`
`std::length_error`	`ValueError`
`std::out_of_range`	`ValueError`
`std::range_error`	`ValueError`
`pybind11::stop_iteration`	`StopIteration` (used to implement custom iterators)
`pybind11::index_error`	`IndexError` (used to indicate out of bounds accesses in `__getitem__`, `__setitem__`, etc.)
`pybind11::value_error`	`ValueError` (used to indicate wrong value passed in `container.remove(...)`
`pybind11::error_already_set`	Indicates that the Python exception flag has already been initialized

When a Python function invoked from C++ throws an exception, it is converted into a C++ exception of type `error_already_set` whose string payload contains a textual summary.

There is also a special exception `cast_error` that is thrown by *`handle::call()`* when the input arguments cannot be converted to Python objects.

4.16 Treating STL data structures as opaque objects

pybind11 heavily relies on a template matching mechanism to convert parameters and return values that are constructed from STL data types such as vectors, linked lists, hash tables, etc. This even works in a recursive manner, for instance to deal with lists of hash maps of pairs of elementary and custom types, etc.

However, a fundamental limitation of this approach is that internal conversions between Python and C++ types involve a copy operation that prevents pass-by-reference semantics. What does this mean?

Suppose we bind the following function

```
void append_1(std::vector<int> &v) {
   v.push_back(1);
}
```

and call it from Python, the following happens:

```
>>> v = [5, 6]
>>> append_1(v)
>>> print(v)
[5, 6]
```

As you can see, when passing STL data structures by reference, modifications are not propagated back the Python side. A similar situation arises when exposing STL data structures using the `def_readwrite` or `def_readonly` functions:

```
/* ... definition ... */

class MyClass {
    std::vector<int> contents;
```

(continues on next page)

(continued from previous page)

```
};

/* ... binding code ... */

py::class_<MyClass>(m, "MyClass")
    .def(py::init<>)
    .def_readwrite("contents", &MyClass::contents);
```

In this case, properties can be read and written in their entirety. However, an `append` operaton involving such a list type has no effect:

```
>>> m = MyClass()
>>> m.contents = [5, 6]
>>> print(m.contents)
[5, 6]
>>> m.contents.append(7)
>>> print(m.contents)
[5, 6]
```

To deal with both of the above situations, pybind11 provides a macro named `PYBIND11_MAKE_OPAQUE(T)` that disables the template-based conversion machinery of types, thus rendering them *opaque*. The contents of opaque objects are never inspected or extracted, hence they can be passed by reference. For instance, to turn `std::vector<int>` into an opaque type, add the declaration

```
PYBIND11_MAKE_OPAQUE(std::vector<int>);
```

before any binding code (e.g. invocations to `class_::def()`, etc.). This macro must be specified at the top level, since instantiates a partial template overload. If your binding code consists of multiple compilation units, it must be present in every file preceding any usage of `std::vector<int>`. Opaque types must also have a corresponding `class_` declaration to associate them with a name in Python, and to define a set of available operations:

```
py::class_<std::vector<int>>(m, "IntVector")
    .def(py::init<>())
    .def("clear", &std::vector<int>::clear)
    .def("pop_back", &std::vector<int>::pop_back)
    .def("__len__", [](const std::vector<int> &v) { return v.size(); })
    .def("__iter__", [](std::vector<int> &v) {
       return py::make_iterator(v.begin(), v.end());
    }, py::keep_alive<0, 1>()) /* Keep vector alive while iterator is used */
    // ....
```

Please take a look at the *General notes regarding convenience macros* before using this feature.

See also:

The file `example/example14.cpp` contains a complete example that demonstrates how to create and expose opaque types using pybind11 in more detail.

4.17 Transparent conversion of dense and sparse Eigen data types

Eigen[1] is C++ header-based library for dense and sparse linear algebra. Due to its popularity and widespread adoption, pybind11 provides transparent conversion support between Eigen and Scientific Python linear algebra data types.

[1] http://eigen.tuxfamily.org

Specifically, when including the optional header file `pybind11/eigen.h`, pybind11 will automatically and transparently convert

1. Static and dynamic Eigen dense vectors and matrices to instances of `numpy.ndarray` (and vice versa).

1. Eigen sparse vectors and matrices to instances of `scipy.sparse.csr_matrix`/`scipy.sparse.csc_matrix` (and vice versa).

This makes it possible to bind most kinds of functions that rely on these types. One major caveat are functions that take Eigen matrices *by reference* and modify them somehow, in which case the information won't be propagated to the caller.

```
/* The Python bindings of this function won't replicate
   the intended effect of modifying the function argument */
void scale_by_2(Eigen::Vector3f &v) {
   v *= 2;
}
```

To see why this is, refer to the section on *Treating STL data structures as opaque objects* (although that section specifically covers STL data types, the underlying issue is the same). The next two sections discuss an efficient alternative for exposing the underlying native Eigen types as opaque objects in a way that still integrates with NumPy and SciPy.

See also:

The file `example/eigen.cpp` contains a complete example that shows how to pass Eigen sparse and dense data types in more detail.

4.18 Buffer protocol

Python supports an extremely general and convenient approach for exchanging data between plugin libraries. Types can expose a buffer view[2], which provides fast direct access to the raw internal data representation. Suppose we want to bind the following simplistic Matrix class:

```
class Matrix {
public:
    Matrix(size_t rows, size_t cols) : m_rows(rows), m_cols(cols) {
        m_data = new float[rows*cols];
    }
    float *data() { return m_data; }
    size_t rows() const { return m_rows; }
    size_t cols() const { return m_cols; }
private:
    size_t m_rows, m_cols;
    float *m_data;
};
```

The following binding code exposes the `Matrix` contents as a buffer object, making it possible to cast Matrices into NumPy arrays. It is even possible to completely avoid copy operations with Python expressions like `np.array(matrix_instance, copy = False)`.

```
py::class_<Matrix>(m, "Matrix")
   .def_buffer([](Matrix &m) -> py::buffer_info {
        return py::buffer_info(
            m.data(),                               /* Pointer to buffer */
```

(continues on next page)

[2] http://docs.python.org/3/c-api/buffer.html

(continued from previous page)

```
            sizeof(float),                          /* Size of one scalar */
            py::format_descriptor<float>::value, /* Python struct-style format␣
↪descriptor */
            2,                                      /* Number of dimensions */
            { m.rows(), m.cols() },                 /* Buffer dimensions */
            { sizeof(float) * m.rows(),             /* Strides (in bytes) for each index␣
↪*/
              sizeof(float) }
        );
    });
```

The snippet above binds a lambda function, which can create `py::buffer_info` description records on demand describing a given matrix. The contents of `py::buffer_info` mirror the Python buffer protocol specification.

```
struct buffer_info {
    void *ptr;
    size_t itemsize;
    std::string format;
    int ndim;
    std::vector<size_t> shape;
    std::vector<size_t> strides;
};
```

To create a C++ function that can take a Python buffer object as an argument, simply use the type `py::buffer` as one of its arguments. Buffers can exist in a great variety of configurations, hence some safety checks are usually necessary in the function body. Below, you can see an basic example on how to define a custom constructor for the Eigen double precision matrix (`Eigen::MatrixXd`) type, which supports initialization from compatible buffer objects (e.g. a NumPy matrix).

```
/* Bind MatrixXd (or some other Eigen type) to Python */
typedef Eigen::MatrixXd Matrix;

typedef Matrix::Scalar Scalar;
constexpr bool rowMajor = Matrix::Flags & Eigen::RowMajorBit;

py::class_<Matrix>(m, "Matrix")
    .def("__init__", [](Matrix &m, py::buffer b) {
        typedef Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic> Strides;

        /* Request a buffer descriptor from Python */
        py::buffer_info info = b.request();

        /* Some sanity checks ... */
        if (info.format != py::format_descriptor<Scalar>::value)
            throw std::runtime_error("Incompatible format: expected a double array!");

        if (info.ndim != 2)
            throw std::runtime_error("Incompatible buffer dimension!");

        auto strides = Strides(
            info.strides[rowMajor ? 0 : 1] / sizeof(Scalar),
            info.strides[rowMajor ? 1 : 0] / sizeof(Scalar));

        auto map = Eigen::Map<Matrix, 0, Strides>(
            static_cast<Scalar *>(info.ptr), info.shape[0], info.shape[1], strides);
```

(continues on next page)

(continued from previous page)

```
        new (&m) Matrix(map);
    });
```

For reference, the `def_buffer()` call for this Eigen data type should look as follows:

```
.def_buffer([](Matrix &m) -> py::buffer_info {
    return py::buffer_info(
        m.data(),                /* Pointer to buffer */
        sizeof(Scalar),          /* Size of one scalar */
        /* Python struct-style format descriptor */
        py::format_descriptor<Scalar>::value,
        /* Number of dimensions */
        2,
        /* Buffer dimensions */
        { (size_t) m.rows(),
          (size_t) m.cols() },
        /* Strides (in bytes) for each index */
        { sizeof(Scalar) * (rowMajor ? m.cols() : 1),
          sizeof(Scalar) * (rowMajor ? 1 : m.rows()) }
    );
 })
```

For a much easier approach of binding Eigen types (although with some limitations), refer to the section on *Transparent conversion of dense and sparse Eigen data types*.

See also:

The file `example/example7.cpp` contains a complete example that demonstrates using the buffer protocol with pybind11 in more detail.

4.19 NumPy support

By exchanging `py::buffer` with `py::array` in the above snippet, we can restrict the function so that it only accepts NumPy arrays (rather than any type of Python object satisfying the buffer protocol).

In many situations, we want to define a function which only accepts a NumPy array of a certain data type. This is possible via the `py::array_t<T>` template. For instance, the following function requires the argument to be a NumPy array containing double precision values.

```
void f(py::array_t<double> array);
```

When it is invoked with a different type (e.g. an integer or a list of integers), the binding code will attempt to cast the input into a NumPy array of the requested type. Note that this feature requires the :file:`pybind11/numpy.h` header to be included.

Data in NumPy arrays is not guaranteed to packed in a dense manner; furthermore, entries can be separated by arbitrary column and row strides. Sometimes, it can be useful to require a function to only accept dense arrays using either the C (row-major) or Fortran (column-major) ordering. This can be accomplished via a second template argument with values `py::array::c_style` or `py::array::f_style`.

```
void f(py::array_t<double, py::array::c_style | py::array::forcecast> array);
```

The `py::array::forcecast` argument is the default value of the second template paramenter, and it ensures that non-conforming arguments are converted into an array satisfying the specified requirements instead of trying the next function overload.

4.20 Vectorizing functions

Suppose we want to bind a function with the following signature to Python so that it can process arbitrary NumPy array arguments (vectors, matrices, general N-D arrays) in addition to its normal arguments:

```
double my_func(int x, float y, double z);
```

After including the `pybind11/numpy.h` header, this is extremely simple:

```
m.def("vectorized_func", py::vectorize(my_func));
```

Invoking the function like below causes 4 calls to be made to `my_func` with each of the array elements. The significant advantage of this compared to solutions like `numpy.vectorize()` is that the loop over the elements runs entirely on the C++ side and can be crunched down into a tight, optimized loop by the compiler. The result is returned as a NumPy array of type `numpy.dtype.float64`.

```
>>> x = np.array([[1, 3],[5, 7]])
>>> y = np.array([[2, 4],[6, 8]])
>>> z = 3
>>> result = vectorized_func(x, y, z)
```

The scalar argument `z` is transparently replicated 4 times. The input arrays `x` and `y` are automatically converted into the right types (they are of type `numpy.dtype.int64` but need to be `numpy.dtype.int32` and `numpy.dtype.float32`, respectively)

Sometimes we might want to explicitly exclude an argument from the vectorization because it makes little sense to wrap it in a NumPy array. For instance, suppose the function signature was

```
double my_func(int x, float y, my_custom_type *z);
```

This can be done with a stateful Lambda closure:

```
// Vectorize a lambda function with a capture object (e.g. to exclude some arguments␣
↪from the vectorization)
m.def("vectorized_func",
    [](py::array_t<int> x, py::array_t<float> y, my_custom_type *z) {
        auto stateful_closure = [z](int x, float y) { return my_func(x, y, z); };
        return py::vectorize(stateful_closure)(x, y);
    }
);
```

In cases where the computation is too complicated to be reduced to `vectorize`, it will be necessary to create and access the buffer contents manually. The following snippet contains a complete example that shows how this works (the code is somewhat contrived, since it could have been done more simply using `vectorize`).

```
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

namespace py = pybind11;

py::array_t<double> add_arrays(py::array_t<double> input1, py::array_t<double>␣
↪input2) {
    auto buf1 = input1.request(), buf2 = input2.request();

    if (buf1.ndim != 1 || buf2.ndim != 1)
        throw std::runtime_error("Number of dimensions must be one");
```

(continues on next page)

(continued from previous page)

```
    if (buf1.shape[0] != buf2.shape[0])
        throw std::runtime_error("Input shapes must match");

    auto result = py::array(py::buffer_info(
        nullptr,            /* Pointer to data (nullptr -> ask NumPy to allocate!) */
        sizeof(double),     /* Size of one item */
        py::format_descriptor<double>::value(), /* Buffer format */
        buf1.ndim,          /* How many dimensions? */
        { buf1.shape[0] },  /* Number of elements for each dimension */
        { sizeof(double) }  /* Strides for each dimension */
    ));

    auto buf3 = result.request();

    double *ptr1 = (double *) buf1.ptr,
           *ptr2 = (double *) buf2.ptr,
           *ptr3 = (double *) buf3.ptr;

    for (size_t idx = 0; idx < buf1.shape[0]; idx++)
        ptr3[idx] = ptr1[idx] + ptr2[idx];

    return result;
}

PYBIND11_PLUGIN(test) {
    py::module m("test");
    m.def("add_arrays", &add_arrays, "Add two NumPy arrays");
    return m.ptr();
}
```

See also:

The file `example/example10.cpp` contains a complete example that demonstrates using `vectorize()` in more detail.

4.21 Functions taking Python objects as arguments

pybind11 exposes all major Python types using thin C++ wrapper classes. These wrapper classes can also be used as parameters of functions in bindings, which makes it possible to directly work with native Python types on the C++ side. For instance, the following statement iterates over a Python `dict`:

```
void print_dict(py::dict dict) {
    /* Easily interact with Python types */
    for (auto item : dict)
        std::cout << "key=" << item.first << ", "
                  << "value=" << item.second << std::endl;
}
```

Available types include *handle*, *object*, `bool_`, `int_`, `float_`, `str`, `bytes`, `tuple`, `list`, `dict`, `slice`, `none`, `capsule`, `iterable`, `iterator`, `function`, `buffer`, `array`, and `array_t`.

In this kind of mixed code, it is often necessary to convert arbitrary C++ types to Python, which can be done using `cast()`:

```
MyClass *cls = ..;
py::object obj = py::cast(cls);
```

The reverse direction uses the following syntax:

```
py::object obj = ...;
MyClass *cls = obj.cast<MyClass *>();
```

When conversion fails, both directions throw the exception `cast_error`. It is also possible to call python functions via `operator()`.

```
py::function f = <...>;
py::object result_py = f(1234, "hello", some_instance);
MyClass &result = result_py.cast<MyClass>();
```

The special `f(*args)` and `f(*args, **kwargs)` syntax is also supported to supply arbitrary argument and keyword lists, although these cannot be mixed with other parameters.

```
py::function f = <...>;
py::tuple args = py::make_tuple(1234);
py::dict kwargs;
kwargs["y"] = py::cast(5678);
py::object result = f(*args, **kwargs);
```

See also:

The file `example/example2.cpp` contains a complete example that demonstrates passing native Python types in more detail. The file `example/example11.cpp` discusses usage of `args` and `kwargs`.

4.22 Default arguments revisited

The section on *Default arguments* previously discussed basic usage of default arguments using pybind11. One noteworthy aspect of their implementation is that default arguments are converted to Python objects right at declaration time. Consider the following example:

```
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg("arg") = SomeType(123));
```

In this case, pybind11 must already be set up to deal with values of the type `SomeType` (via a prior instantiation of `py::class_<SomeType>`), or an exception will be thrown.

Another aspect worth highlighting is that the "preview" of the default argument in the function signature is generated using the object's `__repr__` method. If not available, the signature may not be very helpful, e.g.:

```
FUNCTIONS
...
|  myFunction(...)
|      Signature : (MyClass, arg : SomeType = <SomeType object at 0x101b7b080>) ->␣
↪NoneType
...
```

The first way of addressing this is by defining `SomeType.__repr__`. Alternatively, it is possible to specify the human-readable preview of the default argument manually using the `arg_t` notation:

```
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg_t<SomeType>("arg", SomeType(123), "SomeType(123)"));
```

Sometimes it may be necessary to pass a null pointer value as a default argument. In this case, remember to cast it to the underlying type in question, like so:

```
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg("arg") = (SomeType *) nullptr);
```

4.23 Binding functions that accept arbitrary numbers of arguments and keywords arguments

Python provides a useful mechanism to define functions that accept arbitrary numbers of arguments and keyword arguments:

```
def generic(*args, **kwargs):
    # .. do something with args and kwargs
```

Such functions can also be created using pybind11:

```
void generic(py::args args, py::kwargs kwargs) {
    /// .. do something with args
    if (kwargs)
        /// .. do something with kwargs
}

/// Binding code
m.def("generic", &generic);
```

(See `example/example11.cpp`). The class `py::args` derives from `py::list` and `py::kwargs` derives from `py::dict` Note that the `kwargs` argument is invalid if no keyword arguments were actually provided. Please refer to the other examples for details on how to iterate over these, and on how to cast their entries into C++ objects.

4.24 Partitioning code over multiple extension modules

It's straightforward to split binding code over multiple extension modules, while referencing types that are declared elsewhere. Everything "just" works without any special precautions. One exception to this rule occurs when extending a type declared in another extension module. Recall the basic example from Section *Inheritance*.

```
py::class_<Pet> pet(m, "Pet");
pet.def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

py::class_<Dog>(m, "Dog", pet /* <- specify parent */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Suppose now that `Pet` bindings are defined in a module named `basic`, whereas the `Dog` bindings are defined somewhere else. The challenge is of course that the variable `pet` is not available anymore though it is needed to indicate the inheritance relationship to the constructor of `class_<Dog>`. However, it can be acquired as follows:

```
py::object pet = (py::object) py::module::import("basic").attr("Pet");

py::class_<Dog>(m, "Dog", pet)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Alternatively, we can rely on the `base` tag, which performs an automated lookup of the corresponding Python type. However, this also requires invoking the `import` function once to ensure that the pybind11 binding code of the module `basic` has been executed.

```
py::module::import("basic");

py::class_<Dog>(m, "Dog", py::base<Pet>())
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

Naturally, both methods will fail when there are cyclic dependencies.

Note that compiling code which has its default symbol visibility set to *hidden* (e.g. via the command line flag `-fvisibility=hidden` on GCC/Clang) can interfere with the ability to access types defined in another extension module. Workarounds include changing the global symbol visibility (not recommended, because it will lead unnecessarily large binaries) or manually exporting types that are accessed by multiple extension modules:

```
#ifdef _WIN32
#  define EXPORT_TYPE __declspec(dllexport)
#else
#  define EXPORT_TYPE __attribute__ ((visibility("default")))
#endif

class EXPORT_TYPE Dog : public Animal {
    ...
};
```

4.25 Pickling support

Python's `pickle` module provides a powerful facility to serialize and de-serialize a Python object graph into a binary data stream. To pickle and unpickle C++ classes using pybind11, two additional functions must be provided. Suppose the class in question has the following signature:

```
class Pickleable {
public:
    Pickleable(const std::string &value) : m_value(value) { }
    const std::string &value() const { return m_value; }

    void setExtra(int extra) { m_extra = extra; }
    int extra() const { return m_extra; }
private:
    std::string m_value;
    int m_extra = 0;
};
```

The binding code including the requisite `__setstate__` and `__getstate__` methods[3] looks as follows:

[3] http://docs.python.org/3/library/pickle.html#pickling-class-instances

```
py::class_<Pickleable>(m, "Pickleable")
    .def(py::init<std::string>())
    .def("value", &Pickleable::value)
    .def("extra", &Pickleable::extra)
    .def("setExtra", &Pickleable::setExtra)
    .def("__getstate__", [](const Pickleable &p) {
        /* Return a tuple that fully encodes the state of the object */
        return py::make_tuple(p.value(), p.extra());
    })
    .def("__setstate__", [](Pickleable &p, py::tuple t) {
        if (t.size() != 2)
            throw std::runtime_error("Invalid state!");

        /* Invoke the in-place constructor. Note that this is needed even
           when the object just has a trivial default constructor */
        new (&p) Pickleable(t[0].cast<std::string>());

        /* Assign any additional state */
        p.setExtra(t[1].cast<int>());
    });
```

An instance can now be pickled as follows:

```
try:
    import cPickle as pickle  # Use cPickle on Python 2.7
except ImportError:
    import pickle

p = Pickleable("test_value")
p.setExtra(15)
data = pickle.dumps(p, 2)
```

Note that only the cPickle module is supported on Python 2.7. The second argument to `dumps` is also crucial: it selects the pickle protocol version 2, since the older version 1 is not supported. Newer versions are also fine—for instance, specify `-1` to always use the latest available version. Beware: failure to follow these instructions will cause important pybind11 memory allocation routines to be skipped during unpickling, which will likely lead to memory corruption and/or segmentation faults.

See also:

The file `example/example15.cpp` contains a complete example that demonstrates how to pickle and unpickle types using pybind11 in more detail.

4.26 Generating documentation using Sphinx

Sphinx[4] has the ability to inspect the signatures and documentation strings in pybind11-based extension modules to automatically generate beautiful documentation in a variety formats. The python_example repository[5] contains a simple example repository which uses this approach.

There are two potential gotchas when using this approach: first, make sure that the resulting strings do not contain any `TAB` characters, which break the docstring parsing routines. You may want to use C++11 raw string literals, which are convenient for multi-line comments. Conveniently, any excess indentation will be automatically be removed by Sphinx. However, for this to work, it is important that all lines are indented consistently, i.e.:

[4] http://www.sphinx-doc.org
[5] http://github.com/pybind/python_example

```
// ok
m.def("foo", &foo, R"mydelimiter(
    The foo function

    Parameters
    ----------
)mydelimiter");

// *not ok*
m.def("foo", &foo, R"mydelimiter(The foo function

    Parameters
    ----------
)mydelimiter");
```

CHAPTER 5

Build systems

5.1 Building with setuptools

For projects on PyPI, building with setuptools is the way to go. Sylvain Corlay has kindly provided an example project which shows how to set up everything, including automatic generation of documentation using Sphinx. Please refer to the [python_example] repository.

5.2 Building with cppimport

> cppimport is a small Python import hook that determines whether there is a C++ source file whose name matches the requested module. If there is, the file is compiled as a Python extension using pybind11 and placed in the same folder as the C++ source file. Python is then able to find the module and load it.

5.3 Building with CMake

For C++ codebases that have an existing CMake-based build system, a Python extension module can be created with just a few lines of code:

```
cmake_minimum_required(VERSION 2.8.12)
project(example)

add_subdirectory(pybind11)
pybind11_add_module(example example.cpp)
```

This assumes that the pybind11 repository is located in a subdirectory named `pybind11` and that the code is located in a file named `example.cpp`. The CMake command `add_subdirectory` will import a function with the signature `pybind11_add_module(<name> source1 [source2 ...])`. It will take care of all the details needed to build a Python extension module on any platform.

The target Python version can be selected by setting the `PYBIND11_PYTHON_VERSION` variable before adding the pybind11 subdirectory. Alternatively, an exact Python installation can be specified by setting `PYTHON_EXECUTABLE`.

A working sample project, including a way to invoke CMake from `setup.py` for PyPI integration, can be found in the [cmake_example] repository.

CHAPTER 6

Benchmark

The following is the result of a synthetic benchmark comparing both compilation time and module size of pybind11 against Boost.Python.

6.1 Setup

A python script (see the `docs/benchmark.py` file) was used to generate a set of files with dummy classes whose count increases for each successive benchmark (between 1 and 2048 classes in powers of two). Each class has four methods with a randomly generated signature with a return value and four arguments. (There was no particular reason for this setup other than the desire to generate many unique function signatures whose count could be controlled in a simple way.)

Here is an example of the binding code for one class:

```
...
class cl034 {
public:
    cl279 *fn_000(cl084 *, cl057 *, cl065 *, cl042 *);
    cl025 *fn_001(cl098 *, cl262 *, cl414 *, cl121 *);
    cl085 *fn_002(cl445 *, cl297 *, cl145 *, cl421 *);
    cl470 *fn_003(cl200 *, cl323 *, cl332 *, cl492 *);
};
...

PYBIND11_PLUGIN(example) {
    py::module m("example");
    ...
    py::class_<cl034>(m, "cl034")
        .def("fn_000", &cl034::fn_000)
        .def("fn_001", &cl034::fn_001)
        .def("fn_002", &cl034::fn_002)
        .def("fn_003", &cl034::fn_003)
    ...
```

(continues on next page)

(continued from previous page)

```
    return m.ptr();
}
```

The Boost.Python version looks almost identical except that a return value policy had to be specified as an argument to `def()`. For both libraries, compilation was done with

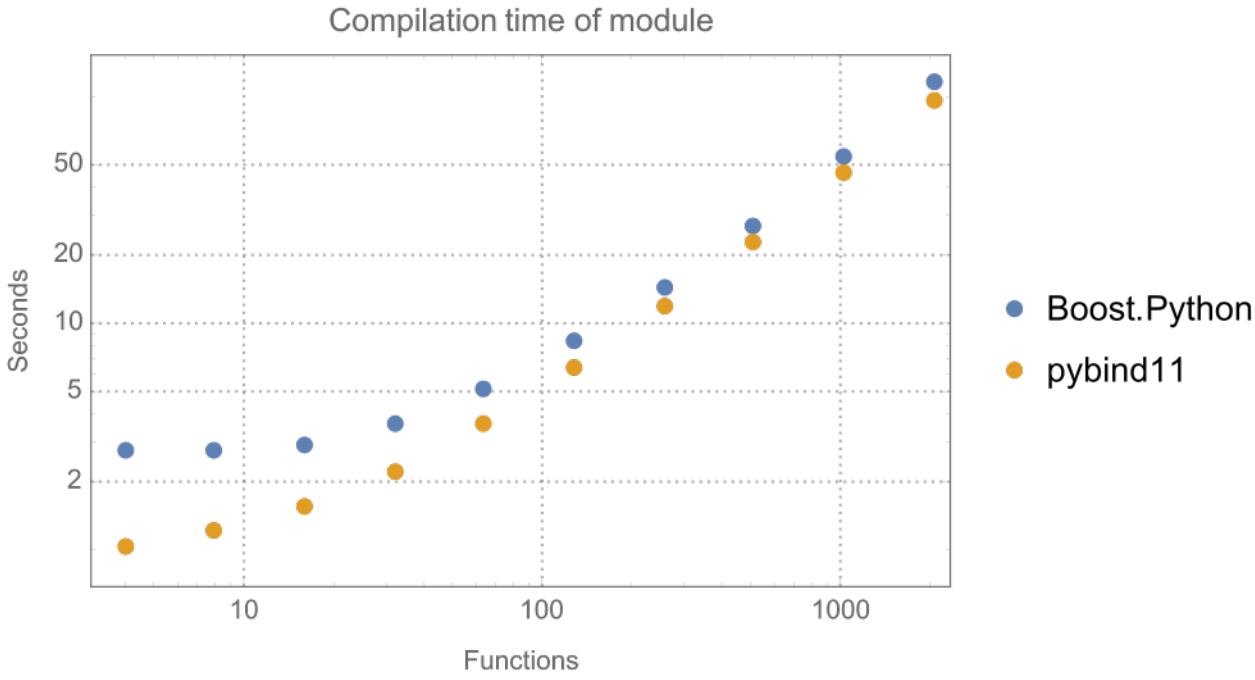
```
Apple LLVM version 7.0.2 (clang-700.1.81)
```

and the following compilation flags

```
g++ -Os -shared -rdynamic -undefined dynamic_lookup -fvisibility=hidden -std=c++14
```

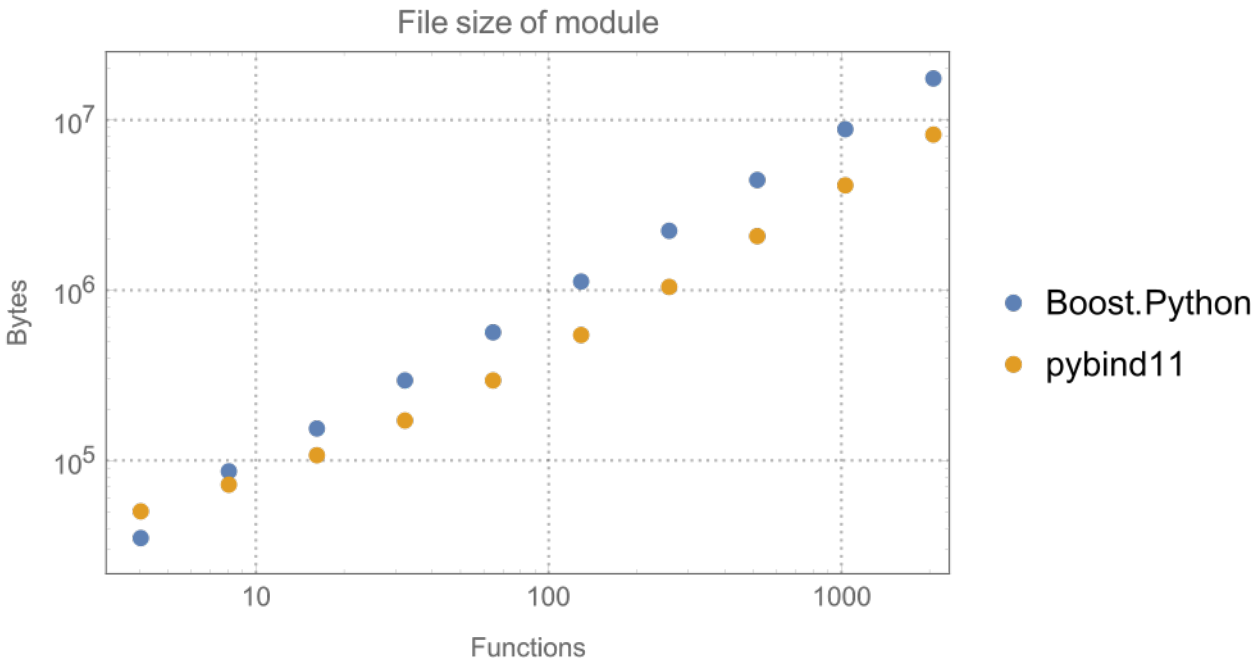
6.2 Compilation time

The following log-log plot shows how the compilation time grows for an increasing number of class and function declarations. pybind11 includes many fewer headers, which initially leads to shorter compilation times, but the performance is ultimately fairly similar (pybind11 is 19.8 seconds faster for the largest largest file with 2048 classes and a total of 8192 methods – a modest **1.2x** speedup relative to Boost.Python, which required 116.35 seconds).



6.3 Module size

Differences between the two libraries become much more pronounced when considering the file size of the generated Python plugin: for the largest file, the binary generated by Boost.Python required 16.8 MiB, which was **2.17 times / 9.1 megabytes** larger than the output generated by pybind11. For very small inputs, Boost.Python has an edge in the plot below – however, note that it stores many definitions in an external library, whose size was not included here, hence the comparison is slightly shifted in Boost.Python's favor.


File size of module
Bytes
10^7
10^6
10^5
10
100
1000
Functions
Boost.Python
pybind11

CHAPTER 7

Limitations

pybind11 strives to be a general solution to binding generation, but it also has certain limitations:

- pybind11 casts away `const`-ness in function arguments and return values. This is in line with the Python language, which has no concept of `const` values. This means that some additional care is needed to avoid bugs that would be caught by the type checker in a traditional C++ program.
- Multiple inheritance relationships on the C++ side cannot be mapped to Python.

Both of these features could be implemented but would lead to a significant increase in complexity. I've decided to draw the line here to keep this project simple and compact. Users who absolutely require these features are encouraged to fork pybind11.

CHAPTER 8

Frequently asked questions

8.1 "ImportError: dynamic module does not define init function"

1. Make sure that the name specified in `pybind::module` and `PYBIND11_PLUGIN` is consistent and identical to the filename of the extension library. The latter should not contain any extra prefixes (e.g. `test.so` instead of `libtest.so`).
2. If the above did not fix your issue, then you are likely using an incompatible version of Python (for instance, the extension library was compiled against Python 2, while the interpreter is running on top of some version of Python 3, or vice versa)

8.2 "Symbol not found: `___Py_ZeroStruct` / `_PyInstanceMethod_Type`"

See item 2 of the first answer.

8.3 The Python interpreter immediately crashes when importing my module

See item 2 of the first answer.

8.4 CMake doesn't detect the right Python version

The CMake-based build system will try to automatically detect the installed version of Python and link against that. When this fails, or when there are multiple versions of Python and it finds the wrong one, delete `CMakeCache.txt` and then invoke CMake as follows:

```
cmake -DPYTHON_EXECUTABLE:FILEPATH=<path-to-python-executable> .
```

8.5 Limitations involving reference arguments

In C++, it's fairly common to pass arguments using mutable references or mutable pointers, which allows both read and write access to the value supplied by the caller. This is sometimes done for efficiency reasons, or to realize functions that have multiple return values. Here are two very basic examples:

```
void increment(int &i) { i++; }
void increment_ptr(int *i) { (*i)++; }
```

In Python, all arguments are passed by reference, so there is no general issue in binding such code from Python.

However, certain basic Python types (like `str`, `int`, `bool`, `float`, etc.) are **immutable**. This means that the following attempt to port the function to Python doesn't have the same effect on the value provided by the caller – in fact, it does nothing at all.

```
def increment(i):
    i += 1 # nope..
```

pybind11 is also affected by such language-level conventions, which means that binding `increment` or `increment_ptr` will also create Python functions that don't modify their arguments.

Although inconvenient, one workaround is to encapsulate the immutable types in a custom type that does allow modifications.

An other alternative involves binding a small wrapper lambda function that returns a tuple with all output arguments (see the remainder of the documentation for examples on binding lambda functions). An example:

```
int foo(int &i) { i++; return 123; }
```

and the binding code

```
m.def("foo", [](int i) { int rv = foo(i); return std::make_tuple(rv, i); });
```

8.6 How can I reduce the build time?

It's good practice to split binding code over multiple files, as is done in the included file `example/example.cpp`.

```
void init_ex1(py::module &);
void init_ex2(py::module &);
/* ... */

PYBIND11_PLUGIN(example) {
    py::module m("example", "pybind example plugin");

    init_ex1(m);
    init_ex2(m);

    /* ... */

    return m.ptr();
}
```

The various `init_ex` functions should be contained in separate files that can be compiled independently from another. Following this approach will

1. reduce memory requirements per compilation unit.
2. enable parallel builds (if desired).
3. allow for faster incremental builds. For instance, when a single class definiton is changed, only a subset of the binding code will generally need to be recompiled.

8.7 How can I create smaller binaries?

To do its job, pybind11 extensively relies on a programming technique known as *template metaprogramming*, which is a way of performing computation at compile time using type information. Template metaprogamming usually instantiates code involving significant numbers of deeply nested types that are either completely removed or reduced to just a few instrutions during the compiler's optimization phase. However, due to the nested nature of these types, the resulting symbol names in the compiled extension library can be extremely long. For instance, the included test suite contains the following symbol:

```
__ZN8pybind1112cpp_functionC1Iv8Example2JRNSt3__16vectorINS3_12basic_stringIwNS3_
↪11char_traitsIwEENS3_9allocatorIwEEEENS8_ISA_EEEEEJNS_4nameENS_7siblingENS_9is_
↪methodEA28_cEEEMT0_FT_DpT1_EDpRKT2_
```

which is the mangled form of the following function type:

```
pybind11::cpp_function::cpp_function<void, Example2, std::__1::vector<std::__1::basic_
↪string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t> >,␣
↪std::__1::allocator<std::__1::basic_string<wchar_t, std::__1::char_traits<wchar_t>,␣
↪std::__1::allocator<wchar_t> > > >&, pybind11::name, pybind11::sibling,␣
↪pybind11::is_method, char [28]>(void (Example2::*)(std::__1::vector<std::__1::basic_
↪string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t> >,␣
↪std::__1::allocator<std::__1::basic_string<wchar_t, std::__1::char_traits<wchar_t>,␣
↪std::__1::allocator<wchar_t> > > >&), pybind11::name const&, pybind11::sibling␣
↪const&, pybind11::is_method const&, char const (&) [28])
```

The memory needed to store just the mangled name of this function (196 bytes) is larger than the actual piece of code (111 bytes) it represents! On the other hand, it's silly to even give this function a name – after all, it's just a tiny cog in a bigger piece of machinery that is not exposed to the outside world. So we'll generally only want to export symbols for those functions which are actually called from the outside.

This can be achieved by specifying the parameter `-fvisibility=hidden` to GCC and Clang, which sets the default symbol visibility to *hidden*. It's best to do this only for release builds, since the symbol names can be helpful in debugging sessions. On Visual Studio, symbols are already hidden by default, so nothing needs to be done there. Needless to say, this has a tremendous impact on the final binary size of the resulting extension library.

Another aspect that can require a fair bit of code are function signature descriptions. pybind11 automatically generates human-readable function signatures for docstrings, e.g.:

```
|  __init__(...)
|      __init__(*args, **kwargs)
|      Overloaded function.
|
|      1. __init__(example.Example1) -> NoneType
|
|      Docstring for overload #1 goes here
|
```

(continues on next page)

(continued from previous page)

```
|      2. __init__(example.Example1, int) -> NoneType
|
|      Docstring for overload #2 goes here
|
|      3. __init__(example.Example1, example.Example1) -> NoneType
|
|      Docstring for overload #3 goes here
```

In C++11 mode, these are generated at run time using string concatenation, which can amount to 10-20% of the size of the resulting binary. If you can, enable C++14 language features (using `-std=c++14` for GCC/Clang), in which case signatures are efficiently pre-generated at compile time. Unfortunately, Visual Studio's C++14 support (`constexpr`) is not good enough as of April 2016, so it always uses the more expensive run-time approach.

8.8 Working with ancient Visual Studio 2009 builds on Windows

The official Windows distributions of Python are compiled using truly ancient versions of Visual Studio that lack good C++11 support. Some users implicitly assume that it would be impossible to load a plugin built with Visual Studio 2015 into a Python distribution that was compiled using Visual Studio 2009. However, no such issue exists: it's perfectly legitimate to interface DLLs that are built with different compilers and/or C libraries. Common gotchas to watch out for involve not `free()`-ing memory region that that were `malloc()`-ed in another shared library, using data structures with incompatible ABIs, and so on. pybind11 is very careful not to make these types of mistakes.

Warning: Please be advised that the reference documentation discussing pybind11 internals is currently incomplete. Please refer to the previous sections and the pybind11 header files for the nitty gritty details.

CHAPTER 9

Reference

9.1 Macros

PYBIND11_PLUGIN(**const** char *_name_)

This macro creates the entry point that will be invoked when the Python interpreter imports a plugin library. Please create a *module* in the function body and return the pointer to its underlying Python object at the end.

```
PYBIND11_PLUGIN(example) {
    pybind11::module m("example", "pybind11 example plugin");
    /// Set up bindings here
    return m.ptr();
}
```

9.2 Convenience classes for arbitrary Python types

9.2.1 Without reference counting

class handle

The *handle* class is a thin wrapper around an arbitrary Python object (i.e. a `PyObject *` in Python's C API). It does not perform any automatic reference counting and merely provides a basic C++ interface to various Python API functions.

See also:

The *object* class inherits from *handle* and adds automatic reference counting features.

handle::**handle**()

The default constructor creates a handle with a `nullptr`-valued pointer.

handle::**handle**(**const** *handle*&)

Copy constructor

handle::**handle**(PyObject *)
Creates a *handle* from the given raw Python object pointer.

PyObject **handle*::**ptr**() **const**
Return the `PyObject *` underlying a *handle*.

const *handle* &*handle*::**inc_ref**() **const**
Manually increase the reference count of the Python object. Usually, it is preferable to use the *object* class which derives from *handle* and calls this function automatically. Returns a reference to itself.

const *handle* &*handle*::**dec_ref**() **const**
Manually decrease the reference count of the Python object. Usually, it is preferable to use the *object* class which derives from *handle* and calls this function automatically. Returns a reference to itself.

void *handle*::**ref_count**() **const**
Return the object's current reference count

handle *handle*::**get_type**() **const**
Return a handle to the Python type object underlying the instance

template<typename **T**>
T *handle*::**cast**() **const**
Attempt to cast the Python object into the given C++ type. A `cast_error` will be throw upon failure.

template<typename ...**Args**>
object *handle*::**call**(*Args*&&... *args*) **const**
Assuming the Python object is a function or implements the `__call__` protocol, `call()` invokes the underlying function, passing an arbitrary set of parameters. The result is returned as a *object* and may need to be converted back into a Python object using *handle::cast()*.

When some of the arguments cannot be converted to Python objects, the function will throw a `cast_error` exception. When the Python function call fails, a `error_already_set` exception is thrown.

9.2.2 With reference counting

class **object** : **public** *handle*
Like *handle*, the object class is a thin wrapper around an arbitrary Python object (i.e. a `PyObject *` in Python's C API). In contrast to *handle*, it optionally increases the object's reference count upon construction, and it *always* decreases the reference count when the *object* instance goes out of scope and is destructed. When using *object* instances consistently, it is much easier to get reference counting right at the first attempt.

object::**object**(**const** *object* &*o*)
Copy constructor; always increases the reference count

object::**object**(**const** *handle* &*h*, bool *borrowed*)
Creates a *object* from the given *handle*. The reference count is only increased if the `borrowed` parameter is set to `true`.

object::**object**(PyObject **ptr*, bool *borrowed*)
Creates a *object* from the given raw Python object pointer. The reference count is only increased if the `borrowed` parameter is set to `true`.

object::**object**(*object* &&*other*)
Move constructor; steals the object from `other` and preserves its reference count.

handle *object*::**release**()
Resets the internal pointer to `nullptr` without without decreasing the object's reference count. The function returns a raw handle to the original Python object.

object::**~object**()
Destructor, which automatically calls *handle::dec_ref()*.

9.3 Convenience classes for specific Python types

class module : **public** *object*

module::**module**(**const** char *_name_, **const** char *_doc_ = nullptr)
Create a new top-level Python module with the given name and docstring

module *module*::**def_submodule**(**const** char *_name_, **const** char *_doc_ = nullptr)
Create and return a new Python submodule with the given name and docstring. This also works recursively, i.e.

```
pybind11::module m("example", "pybind11 example plugin");
pybind11::module m2 = m.def_submodule("sub", "A submodule of 'example'");
pybind11::module m3 = m2.def_submodule("subsub", "A submodule of 'example.sub'");
```

template<typename **Func**, typename ...**Extra**>
module &*module*::**def**(**const** char *_name_, *Func* &&_f_, *Extra*&&... *extra*)
Create Python binding for a new function within the module scope. `Func` can be a plain C++ function, a function pointer, or a lambda function. For details on the `Extra&& ... extra` argument, see section *Passing extra arguments to the def function*.

9.4 Passing extra arguments to the def function

class arg

arg::**arg**(**const** char *_name_)

template<typename **T**>
arg_t<*T*> *arg*::**operator=**(**const** *T* &_value_)

template<typename **T**>
class arg_t<*T*> : **public** *arg*
Represents a named argument with a default value

class sibling
Used to specify a handle to an existing sibling function; used internally to implement function overloading in *module::def()* and `class_::def()`.

sibling::**sibling**(*handle* *handle*)

doc::**doc**(**const** char *_value_)
Create a new docstring with the specified value

name::**name**(**const** char *_value_)
Used to specify the function name

CHAPTER 10

Changelog

Starting with version 1.8, pybind11 releases use a [semantic versioning](http://semver.org) policy.

10.1 Breaking changes queued for v2.0.0 (Not yet released)

- Redesigned virtual call mechanism and user-facing syntax (see https://github.com/pybind/pybind11/commit/86d825f3302701d81414ddd3d38bcd09433076bc)
- Remove `handle.call()` method

10.2 1.8.1 (July 12, 2016)

- Fixed a rare but potentially very severe issue when the garbage collector ran during pybind11 type creation.

10.3 1.8.0 (June 14, 2016)

- Redesigned CMake build system which exports a convenient `pybind11_add_module` function to parent projects.
- `std::vector<>` type bindings analogous to Boost.Python's `indexing_suite`
- Transparent conversion of sparse and dense Eigen matrices and vectors (`eigen.h`)
- Added an `ExtraFlags` template argument to the NumPy `array_t<>` wrapper to disable an enforced cast that may lose precision, e.g. to create overloads for different precisions and complex vs real-valued matrices.
- Prevent implicit conversion of floating point values to integral types in function arguments
- Fixed incorrect default return value policy for functions returning a shared pointer
- Don't allow registering a type via `class_` twice

- Don't allow casting a `None` value into a C++ lvalue reference
- Fixed a crash in `enum_::operator==` that was triggered by the `help()` command
- Improved detection of whether or not custom C++ types can be copy/move-constructed
- Extended `str` type to also work with `bytes` instances
- Added a `"name"_a` user defined string literal that is equivalent to `py::arg("name")`.
- When specifying function arguments via `py::arg`, the test that verifies the number of arguments now runs at compile time.
- Added `[[noreturn]]` attribute to `pybind11_fail()` to quench some compiler warnings
- List function arguments in exception text when the dispatch code cannot find a matching overload
- Added `PYBIND11_OVERLOAD_NAME` and `PYBIND11_OVERLOAD_PURE_NAME` macros which can be used to override virtual methods whose name differs in C++ and Python (e.g. `__call__` and `operator()`)
- Various minor `iterator` and `make_iterator()` improvements
- Transparently support `__bool__` on Python 2.x and Python 3.x
- Fixed issue with destructor of unpickled object not being called
- Minor CMake build system improvements on Windows
- New `pybind11::args` and `pybind11::kwargs` types to create functions which take an arbitrary number of arguments and keyword arguments
- New syntax to call a Python function from C++ using `*args` and `*kwargs`
- The functions `def_property_*` now correctly process docstring arguments (these formerly caused a segmentation fault)
- Many `mkdoc.py` improvements (enumerations, template arguments, `DOC()` macro accepts more arguments)
- Cygwin support
- Documentation improvements (pickling support, `keep_alive`, macro usage)

10.4 1.7 (April 30, 2016)

- Added a new `move` return value policy that triggers C++11 move semantics. The automatic return value policy falls back to this case whenever a rvalue reference is encountered
- Significantly more general GIL state routines that are used instead of Python's troublesome `PyGILState_Ensure` and `PyGILState_Release` API
- Redesign of opaque types that drastically simplifies their usage
- Extended ability to pass values of type `[const] void *`
- `keep_alive` fix: don't fail when there is no patient
- `functional.h`: acquire the GIL before calling a Python function
- Added Python RAII type wrappers `none` and `iterable`
- Added `*args` and `*kwargs` pass-through parameters to `pybind11.get_include()` function
- Iterator improvements and fixes
- Documentation on return value policies and opaque types improved

10.5 1.6 (April 30, 2016)

- Skipped due to upload to PyPI gone wrong and inability to recover (https://github.com/pypa/packaging-problems/issues/74)

10.6 1.5 (April 21, 2016)

- For polymorphic types, use RTTI to try to return the closest type registered with pybind11
- Pickling support for serializing and unserializing C++ instances to a byte stream in Python
- Added a convenience routine `make_iterator()` which turns a range indicated by a pair of C++ iterators into a iterable Python object
- Added `len()` and a variadic `make_tuple()` function
- Addressed a rare issue that could confuse the current virtual function dispatcher and another that could lead to crashes in multi-threaded applications
- Added a `get_include()` function to the Python module that returns the path of the directory containing the installed pybind11 header files
- Documentation improvements: import issues, symbol visibility, pickling, limitations
- Added casting support for `std::reference_wrapper<>`

10.7 1.4 (April 7, 2016)

- Transparent type conversion for `std::wstring` and `wchar_t`
- Allow passing `nullptr`-valued strings
- Transparent passing of `void *` pointers using capsules
- Transparent support for returning values wrapped in `std::unique_ptr<>`
- Improved docstring generation for compatibility with Sphinx
- Nicer debug error message when default parameter construction fails
- Support for "opaque" types that bypass the transparent conversion layer for STL containers
- Redesigned type casting interface to avoid ambiguities that could occasionally cause compiler errors
- Redesigned property implementation; fixes crashes due to an unfortunate default return value policy
- Anaconda package generation support

10.8 1.3 (March 8, 2016)

- Added support for the Intel C++ compiler (v15+)
- Added support for the STL unordered set/map data structures
- Added support for the STL linked list data structure
- NumPy-style broadcasting support in `pybind11::vectorize`

- pybind11 now displays more verbose error messages when `arg::operator=()` fails
- pybind11 internal data structures now live in a version-dependent namespace to avoid ABI issues
- Many, many bugfixes involving corner cases and advanced usage

10.9 1.2 (February 7, 2016)

- Optional: efficient generation of function signatures at compile time using C++14
- Switched to a simpler and more general way of dealing with function default arguments. Unused keyword arguments in function calls are now detected and cause errors as expected
- New `keep_alive` call policy analogous to Boost.Python's `with_custodian_and_ward`
- New `pybind11::base<>` attribute to indicate a subclass relationship
- Improved interface for RAII type wrappers in `pytypes.h`
- Use RAII type wrappers consistently within pybind11 itself. This fixes various potential refcount leaks when exceptions occur
- Added new `bytes` RAII type wrapper (maps to `string` in Python 2.7)
- Made handle and related RAII classes const correct, using them more consistently everywhere now
- Got rid of the ugly `__pybind11__` attributes on the Python side—they are now stored in a C++ hash table that is not visible in Python
- Fixed refcount leaks involving NumPy arrays and bound functions
- Vastly improved handling of shared/smart pointers
- Removed an unnecessary copy operation in `pybind11::vectorize`
- Fixed naming clashes when both pybind11 and NumPy headers are included
- Added conversions for additional exception types
- Documentation improvements (using multiple extension modules, smart pointers, other minor clarifications)
- unified infrastructure for parsing variadic arguments in `class_` and cpp_function
- Fixed license text (was: ZLIB, should have been: 3-clause BSD)
- Python 3.2 compatibility
- Fixed remaining issues when accessing types in another plugin module
- Added enum comparison and casting methods
- Improved SFINAE-based detection of whether types are copy-constructible
- Eliminated many warnings about unused variables and the use of `offsetof()`
- Support for `std::array<>` conversions

10.10 1.1 (December 7, 2015)

- Documentation improvements (GIL, wrapping functions, casting, fixed many typos)
- Generalized conversion of integer types
- Improved support for casting function objects

- Improved support for `std::shared_ptr<>` conversions
- Initial support for `std::set<>` conversions
- Fixed type resolution issue for types defined in a separate plugin module
- Cmake build system improvements
- Factored out generic functionality to non-templated code (smaller code size)
- Added a code size / compile time benchmark vs Boost.Python
- Added an appveyor CI script

10.11 1.0 (October 15, 2015)

- Initial release

Bibliography

[python_example] https://github.com/pybind/python_example

[cppimport] https://github.com/tbenthompson/cppimport

[cmake_example] https://github.com/pybind/cmake_example

Index