pybind11 Documentation

Release 2.2.dev0

Wenzel Jakob

Jul 23, 2017

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 ~4K lines of code and depend on Python (2.7 or 3.x, or PyPy2.7 >= 5.7) 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.

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
- Arbitrary exception types
- Enumerations
- Callbacks
- Iterators and ranges

- Custom operators
- Single and multiple inheritance
- STL data structures
- Iterators and ranges
- 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

Goodies

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

- Python 2.7, 3.x, and PyPy (PyPy2.7 >= 5.7) are supported with an implementation-agnostic interface.
- 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 at least 2 compared to equivalent bindings generated by Boost.Python. A recent pybind11 conversion of PyRosetta, an enormous Boost.Python binding project, reported a binary size reduction of **5.4x** and compile time reduction by **5.8x**.
- 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.

Supported compilers

1. Clang/LLVM (any non-ancient version with C++11 support)
2. GCC 4.8 or newer
3. Microsoft Visual Studio 2015 or newer
4. Intel C++ compiler v15 or newer

CHAPTER 2

Changelog

Starting with version 1.8.0, pybind11 releases use a semantic versioning policy.

v2.2.0 (Not yet released)

- TBD

v2.1.1 (April 7, 2017)

- Fixed minimum version requirement for MSVC 2015u3 #773.

v2.1.0 (March 22, 2017)

- pybind11 now performs function overload resolution in two phases. The first phase only considers exact type matches, while the second allows for implicit conversions to take place. A special `noconvert()` syntax can be used to completely disable implicit conversions for specific arguments. #643, #634, #650.
- Fixed a regression where static properties no longer worked with classes using multiple inheritance. The `py::metaclass` attribute is no longer necessary (and deprecated as of this release) when binding classes with static properties. #679,
- Classes bound using `pybind11` can now use custom metaclasses. #679,
- `py::args` and `py::kwargs` can now be mixed with other positional arguments when binding functions using pybind11. #611.
- Improved support for C++11 unicode string and character types; added extensive documentation regarding pybind11's string conversion behavior. #624, #636, #715.
- pybind11 can now avoid expensive copies when converting Eigen arrays to NumPy arrays (and vice versa). #610.

- The "fast path" in `py::vectorize` now works for any full-size group of C or F-contiguous arrays. The non-fast path is also faster since it no longer performs copies of the input arguments (except when type conversions are necessary). #610.
- Added fast, unchecked access to NumPy arrays via a proxy object. #746.
- Transparent support for class-specific `operator new` and `operator delete` implementations. #755.
- Slimmer and more efficient STL-compatible iterator interface for sequence types. #662.
- Improved custom holder type support. #607.
- `nullptr` to `None` conversion fixed in various builtin type casters. #732.
- `enum_` now exposes its members via a special `__members__` attribute. #666.
- `std::vector` bindings created using `stl_bind.h` can now optionally implement the buffer protocol. #488.
- Automated C++ reference documentation using doxygen and breathe. #598.
- Added minimum compiler version assertions. #727.
- Improved compatibility with C++1z. #677.
- Improved `py::capsule` API. Can be used to implement cleanup callbacks that are involved at module destruction time. #752.
- Various minor improvements and fixes. #595, #588, #589, #603, #619, #648, #695, #720, #723, #729, #724, #742, #753.

v2.0.1 (Jan 4, 2017)

- Fix pointer to reference error in type_caster on MSVC #583.
- Fixed a segmentation in the test suite due to a typo cd7eac.

v2.0.0 (Jan 1, 2017)

- Fixed a reference counting regression affecting types with custom metaclasses (introduced in v2.0.0-rc1). #571.
- Quenched a CMake policy warning. #570.

v2.0.0-rc1 (Dec 23, 2016)

The pybind11 developers are excited to issue a release candidate of pybind11 with a subsequent v2.0.0 release planned in early January next year.

An incredible amount of effort by went into pybind11 over the last ~5 months, leading to a release that is jam-packed with exciting new features and numerous usability improvements. The following list links PRs or individual commits whenever applicable.

Happy Christmas!

- Support for binding C++ class hierarchies that make use of multiple inheritance. #410.

- PyPy support: pybind11 now supports nightly builds of PyPy and will interoperate with the future 5.7 release. No code changes are necessary, everything "just" works as usual. Note that we only target the Python 2.7 branch for now; support for 3.x will be added once its `cpyext` extension support catches up. A few minor features remain unsupported for the time being (notably dynamic attributes in custom types). #527.
- Significant work on the documentation – in particular, the monolitic `advanced.rst` file was restructured into a easier to read hierarchical organization. #448.
- Many NumPy-related improvements:
  1. Object-oriented API to access and modify NumPy `ndarray` instances, replicating much of the corresponding NumPy C API functionality. #402.
  2. NumPy array `dtype` array descriptors are now first-class citizens and are exposed via a new class `py::dtype`.
  3. Structured dtypes can be registered using the `PYBIND11_NUMPY_DTYPE()` macro. Special `array` constructors accepting dtype objects were also added.

     One potential caveat involving this change: format descriptor strings should now be accessed via `format_descriptor::format()` (however, for compatibility purposes, the old syntax `format_descriptor::value` will still work for non-structured data types). #308.
  4. Further improvements to support structured dtypes throughout the system. #472, #474, #459, #453, #452, and #505.
  5. Fast access operators. #497.
  6. Constructors for arrays whose storage is owned by another object. #440.
  7. Added constructors for `array` and `array_t` explicitly accepting shape and strides; if strides are not provided, they are deduced assuming C-contiguity. Also added simplified constructors for 1-dimensional case.
  8. Added buffer/NumPy support for `char[N]` and `std::array<char, N>` types.
  9. Added `memoryview` wrapper type which is constructible from `buffer_info`.
- Eigen: many additional conversions and support for non-contiguous arrays/slices. #427, #315, #316, #312, and #267
- Incompatible changes in `class_<...>::class_()`:
  1. Declarations of types that provide access via the buffer protocol must now include the `py::buffer_protocol()` annotation as an argument to the `class_` constructor.
  2. Declarations of types that require a custom metaclass (i.e. all classes which include static properties via commands such as `def_readwrite_static()`) must now include the `py::metaclass()` annotation as an argument to the `class_` constructor.

     These two changes were necessary to make type definitions in pybind11 future-proof, and to support PyPy via its cpyext mechanism. #527.
  3. This version of pybind11 uses a redesigned mechnism for instantiating trempoline classes that are used to override virtual methods from within Python. This led to the following user-visible syntax change: instead of

     ```
     py::class_<TrampolineClass>("MyClass")
       .alias<MyClass>()
       ....
     ```

     write

```
py::class_<MyClass, TrampolineClass>("MyClass")
  ....
```

Importantly, both the original and the trampoline class are now specified as an arguments (in arbitrary order) to the `py::class_` template, and the `alias<..>()` call is gone. The new scheme has zero overhead in cases when Python doesn't override any functions of the underlying C++ class. rev. 86d825.

- Added `eval` and `eval_file` functions for evaluating expressions and statements from a string or file. rev. 0d3fc3.
- pybind11 can now create types with a modifiable dictionary. #437 and #444.
- Support for translation of arbitrary C++ exceptions to Python counterparts. #296 and #273.
- Report full backtraces through mixed C++/Python code, better reporting for import errors, fixed GIL management in exception processing. #537, #494, rev. e72d95, and rev. 099d6e.
- Support for bit-level operations, comparisons, and serialization of C++ enumerations. #503, #508, #380, #309. #311.
- The `class_` constructor now accepts its template arguments in any order. #385.
- Attribute and item accessors now have a more complete interface which makes it possible to chain attributes as in `obj.attr("a")[key].attr("b").attr("method")(1, 2, 3)`. #425.
- Major redesign of the default and conversion constructors in `pytypes.h`. #464.
- Added built-in support for `std::shared_ptr` holder type. It is no longer necessary to to include a declaration of the form `PYBIND11_DECLARE_HOLDER_TYPE(T, std::shared_ptr<T>)` (though continuing to do so won't cause an error). #454.
- New `py::overload_cast` casting operator to select among multiple possible overloads of a function. An example:

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

This feature only works on C++14-capable compilers. #541.

- C++ types are automatically cast to Python types, e.g. when assigning them as an attribute. For instance, the following is now legal:

```
py::module m = /* ... */
m.attr("constant") = 123;
```

(Previously, a `py::cast` call was necessary to avoid a compilation error.) #551.

- Redesigned `pytest`-based test suite. #321.
- Instance tracking to detect reference leaks in test suite. #324
- pybind11 can now distinguish between multiple different instances that are located at the same memory address, but which have different types. #329.
- Improved logic in `move` return value policy. #510, #297.
- Generalized unpacking API to permit calling Python functions from C++ using notation such as `foo(a1, a2, *args, "ka"_a=1, "kb"_a=2, **kwargs)`. #372.
- `py::print()` function whose behavior matches that of the native Python `print()` function. #372.

- Added `py::dict` keyword constructor:`auto d = dict("number"_a=42, "name"_a="World");`. #372.
- Added `py::str::format()` method and `_s` literal: `py::str s = "1 + 2 = {}"_s.format(3);`. #372.
- Added `py::repr()` function which is equivalent to Python's builtin `repr()`. #333.
- Improved construction and destruction logic for holder types. It is now possible to reference instances with smart pointer holder types without constructing the holder if desired. The `PYBIND11_DECLARE_HOLDER_TYPE` macro now accepts an optional second parameter to indicate whether the holder type uses intrusive reference counting. #533 and #561.
- Mapping a stateless C++ function to Python and back is now "for free" (i.e. no extra indirections or argument conversion overheads). rev. 954b79.
- Bindings for `std::valarray<T>`. #545.
- Improved support for C++17 capable compilers. #562.
- Bindings for `std::optional<t>`. #475, #476, #479, #499, and #501.
- `stl_bind.h`: general improvements and support for `std::map` and `std::unordered_map`. #490, #282, #235.
- The `std::tuple`, `std::pair`, `std::list`, and `std::vector` type casters now accept any Python sequence type as input. rev. 107285.
- Improved CMake Python detection on multi-architecture Linux. #532.
- Infrastructure to selectively disable or enable parts of the automatically generated docstrings. #486.
- `reference` and `reference_internal` are now the default return value properties for static and non-static properties, respectively. #473. (the previous defaults were `automatic`). #473.
- Support for `std::unique_ptr` with non-default deleters or no deleter at all (`py::nodelete`). #384.
- Deprecated `handle::call()` method. The new syntax to call Python functions is simply `handle()`. It can also be invoked explicitly via `handle::operator<X>()`, where `X` is an optional return value policy.
- Print more informative error messages when `make_tuple()` or `cast()` fail. #262.
- Creation of holder types for classes deriving from `std::enable_shared_from_this<>` now also works for `const` values. #260.
- `make_iterator()` improvements for better compatibility with various types (now uses prefix increment operator); it now also accepts iterators with different begin/end types as long as they are equality comparable. #247.
- `arg()` now accepts a wider range of argument types for default values. #244.
- Support `keep_alive` where the nurse object may be `None`. #341.
- Added constructors for `str` and `bytes` from zero-terminated char pointers, and from char pointers and length. Added constructors for `str` from `bytes` and for `bytes` from `str`, which will perform UTF-8 decoding/encoding as required.
- Many other improvements of library internals without user-visible changes

1.8.1 (July 12, 2016)

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

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)

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

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)

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<>`

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

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

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

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

1.0 (October 15, 2015)

- Initial release

CHAPTER 3

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 test cases.

Compiling the test cases

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

```
mkdir build
cd build
cmake ..
make check -j 4
```

The last line will both compile and run the tests.

Windows

On Windows, only **Visual Studio 2015** and newer are supported since pybind11 relies on various C++11 language features that break older versions of Visual Studio.

To compile and run the tests:

```
mkdir build
cd build
cmake ..
cmake --build . --config Release --target check
```

This will create a Visual Studio project, compile and run the target, all from the command line.

Note: If all tests fail, make sure that the Python binary and the testcases are compiled for the same processor type and bitness (i.e. either **i386** or **x86_64**). You can specify **x86_64** as the target architecture for the generated Visual Studio project using `cmake -A x64 ...`.

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 `tests` directory, which exercise all features of pybind11.

Header and namespace conventions

For brevity, all code examples assume that the following two lines are present:

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

namespace py = pybind11;
```

Some features may require additional headers, but those will be specified as needed.

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;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

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

The `PYBIND11_MODULE()` macro creates a function that will be called when an `import` statement is issued from within Python. The module name (`example`) is given as the first macro argument (it should not be in quotes). The second argument (`m`) defines a variable of type *`py::module`* which is the main interface for creating bindings. The method *`module::def()`* generates binding code that exposes the `add()` function to Python.

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

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

```
$ 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
>>>
```

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.

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);
```

Exporting variables

To expose a value from C++, use the `attr` function to register it in a module as shown below. Built-in types and general objects (more on that later) are automatically converted when assigned as attributes, and can be explicitly converted using the function `py::cast`.

```
PYBIND11_MODULE(example, m) {
    m.attr("the_answer") = 42;
    py::object world = py::cast("World");
    m.attr("what") = world;
}
```

These are then accessible from Python:

```
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
```

Supported data types

A large number of data types are supported out of the box and can be used seamlessly as functions arguments, return values or with `py::cast` in general. For a full overview, see the *Type conversions* section.

CHAPTER 4

Object-oriented code

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

`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()`.

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.

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'>
```

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.

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

```
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. Please also see the section on *Static properties* in the advanced part of the documentation.

Dynamic attributes

Native Python classes can pick up new attributes dynamically:

```
>>> class Pet:
...     name = 'Molly'
...
>>> p = Pet()
>>> p.name = 'Charly'  # overwrite existing
>>> p.age = 2  # dynamically add a new attribute
```

By default, classes exported from C++ do not support this and the only writable attributes are the ones explicitly defined using `class_::def_readwrite()` or `class_::def_property()`.

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

Trying to set any other attribute results in an error:

```
>>> p = example.Pet()
>>> p.name = 'Charly'  # OK, attribute defined in C++
>>> p.age = 2  # fail
AttributeError: 'Pet' object has no attribute 'age'
```

To enable dynamic attributes for C++ classes, the `py::dynamic_attr` tag must be added to the `py::class_` constructor:

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

Now everything works as expected:

```
>>> p = example.Pet()
>>> p.name = 'Charly'  # OK, overwrite value in C++
>>> p.age = 2  # OK, dynamically add a new attribute
>>> p.__dict__  # just like a native Python class
{'age': 2}
```

Note that there is a small runtime cost for a class with dynamic attributes. Not only because of the addition of a `__dict__`, but also because of more expensive garbage collection tracking which must be activated to resolve possible circular references. Native Python classes incur this same cost by default, so this is not anything to worry about. By default, pybind11 classes are more efficient than native Python classes. Enabling dynamic attributes just brings them on par.

Inheritance and automatic upcasting

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 {
    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 specifies the C++ base class as an extra template parameter of the `class_`:

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

// Method 1: template parameter:
py::class_<Dog, Pet /* <- specify C++ parent type */>(m, "Dog")
```

```
    .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);

// Method 2: pass parent class_ object:
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 equivalent. Afterwards, instances will expose fields and methods of both types:

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

The C++ classes defined above are regular non-polymorphic types with an inheritance relationship. This is reflected in Python:

```
// Return a base pointer to a derived instance
m.def("pet_store", []() { return std::unique_ptr<Pet>(new Dog("Molly")); });
```

```
>>> p = example.pet_store()
>>> type(p)  # `Dog` instance behind `Pet` pointer
Pet          # no pointer upcasting for regular non-polymorphic types
>>> p.bark()
AttributeError: 'Pet' object has no attribute 'bark'
```

The function returned a `Dog` instance, but because it's a non-polymorphic type behind a base pointer, Python only sees a `Pet`. In C++, a type is only considered polymorphic if it has at least one virtual function and pybind11 will automatically recognize this:

```
struct PolymorphicPet {
    virtual ~PolymorphicPet() = default;
};

struct PolymorphicDog : PolymorphicPet {
    std::string bark() const { return "woof!"; }
};

// Same binding code
py::class_<PolymorphicPet>(m, "PolymorphicPet");
py::class_<PolymorphicDog, PolymorphicPet>(m, "PolymorphicDog")
    .def(py::init<>())
    .def("bark", &PolymorphicDog::bark);

// Again, return a base pointer to a derived instance
m.def("pet_store2", []() { return std::unique_ptr<PolymorphicPet>(new PolymorphicDog);
↪ });
```

```
>>> p = example.pet_store2()
>>> type(p)
PolymorphicDog  # automatically upcast
>>> p.bark()
u'woof!'
```

Given a pointer to a polymorphic base, pybind11 performs automatic upcasting to the actual derived type. Note that this goes beyond the usual situation in C++: we don't just get access to the virtual functions of the base, we get the concrete derived type including functions and attributes that the base type may not even be aware of.

See also:

For more information about polymorphic behavior see *Overriding virtual functions in Python*.

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
```

If you have a C++14 compatible compiler[2], you can use an alternative syntax to cast the overloaded function:

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

Here, `py::overload_cast` only requires the parameter types to be specified. The return type and class are deduced. This avoids the additional noise of `void (Pet::*)()` as seen in the raw cast. If a function is overloaded based on constness, the `py::const_` tag should be used:

```
struct Widget {
    int foo(int x, float y);
    int foo(int x, float y) const;
};

py::class_<Widget>(m, "Widget")
   .def("foo_mutable", py::overload_cast<int, float>(&Widget::foo))
   .def("foo_const",   py::overload_cast<int, float>(&Widget::foo, py::const_));
```

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.

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

[2] A compiler which supports the `-std=c++14` flag or Visual Studio 2015 Update 2 and newer.

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
```

The entries defined by the enumeration type are exposed in the `__members__` property:

```
>>> Pet.Kind.__members__
{'Dog': Kind.Dog, 'Cat': Kind.Cat}
```

Note: When the special tag `py::arithmetic()` is specified to the `enum_` constructor, pybind11 creates an enumeration that also supports rudimentary arithmetic and bit-level operations like comparisons, and, or, xor, negation, etc.

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

By default, these are omitted to conserve space.

CHAPTER 5

Build systems

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.

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.

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 the pybind11 project which provides the `pybind11_add_module` function. It will take care of all the details needed to build a Python extension module on any platform.

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

pybind11_add_module

To ease the creation of Python extension modules, pybind11 provides a CMake function with the following signature:

```
pybind11_add_module(<name> [MODULE | SHARED] [EXCLUDE_FROM_ALL]
                    [NO_EXTRAS] [THIN_LTO] source1 [source2 ...])
```

This function behaves very much like CMake's builtin `add_library` (in fact, it's a wrapper function around that command). It will add a library target called `<name>` to be built from the listed source files. In addition, it will take care of all the Python-specific compiler and linker flags as well as the OS- and Python-version-specific file extension. The produced target `<name>` can be further manipulated with regular CMake commands.

`MODULE` or `SHARED` may be given to specify the type of library. If no type is given, `MODULE` is used by default which ensures the creation of a Python-exclusive module. Specifying `SHARED` will create a more traditional dynamic library which can also be linked from elsewhere. `EXCLUDE_FROM_ALL` removes this target from the default build (see CMake docs for details).

Since pybind11 is a template library, `pybind11_add_module` adds compiler flags to ensure high quality code generation without bloat arising from long symbol names and duplication of code in different translation units. The additional flags enable LTO (Link Time Optimization), set default visibility to *hidden* and strip unneeded symbols. See the *FAQ entry* for a more detailed explanation. These optimizations are never applied in `Debug` mode. If `NO_EXTRAS` is given, they will always be disabled, even in `Release` mode. However, this will result in code bloat and is generally not recommended.

As stated above, LTO is enabled by default. Some newer compilers also support different flavors of LTO such as ThinLTO. Setting `THIN_LTO` will cause the function to prefer this flavor if available. The function falls back to regular LTO if `-flto=thin` is not available.

Configuration variables

By default, pybind11 will compile modules with the C++14 standard, if available on the target compiler, falling back to C++11 if C++14 support is not available. Note, however, that this default is subject to change: future pybind11 releases are expected to migrate to newer C++ standards as they become available. To override this, the standard flag can be given explicitly in `PYBIND11_CPP_STANDARD`:

```
# Use just one of these:
# GCC/clang:
set(PYBIND11_CPP_STANDARD -std=c++11)
set(PYBIND11_CPP_STANDARD -std=c++14)
set(PYBIND11_CPP_STANDARD -std=c++1z) # Experimental C++17 support
# MSVC:
set(PYBIND11_CPP_STANDARD /std:c++14)
set(PYBIND11_CPP_STANDARD /std:c++latest) # Enables some MSVC C++17 features

add_subdirectory(pybind11)  # or find_package(pybind11)
```

Note that this and all other configuration variables must be set **before** the call to `add_subdirectory` or `find_package`. The variables can also be set when calling CMake from the command line using the `-D<variable>=<value>` flag.

The target Python version can be selected by setting `PYBIND11_PYTHON_VERSION` or an exact Python installation can be specified with `PYTHON_EXECUTABLE`. For example:

```
cmake -DPYBIND11_PYTHON_VERSION=3.6 ..
# or
cmake -DPYTHON_EXECUTABLE=path/to/python ..
```

find_package vs. add_subdirectory

For CMake-based projects that don't include the pybind11 repository internally, an external installation can be detected through `find_package(pybind11)`. See the Config file docstring for details of relevant CMake variables.

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

find_package(pybind11 REQUIRED)
pybind11_add_module(example example.cpp)
```

Once detected, the aforementioned `pybind11_add_module` can be employed as before. The function usage and configuration variables are identical no matter if pybind11 is added as a subdirectory or found as an installed package. You can refer to the same *[cmake_example]* repository for a full sample project – just swap out `add_subdirectory` for `find_package`.

Advanced: interface library target

When using a version of CMake greater than 3.0, pybind11 can additionally be used as a special *interface library* . The target `pybind11::module` is available with pybind11 headers, Python headers and libraries as needed, and C++ compile definitions attached. This target is suitable for linking to an independently constructed (through `add_library`, not `pybind11_add_module`) target in the consuming project.

```
cmake_minimum_required(VERSION 3.0)
project(example)

find_package(pybind11 REQUIRED)  # or add_subdirectory(pybind11)

add_library(example MODULE main.cpp)
target_link_libraries(example PRIVATE pybind11::module)
set_target_properties(example PROPERTIES PREFIX "${PYTHON_MODULE_PREFIX}"
                                         SUFFIX "${PYTHON_MODULE_EXTENSION}")
```

Warning: Since pybind11 is a metatemplate library, it is crucial that certain compiler flags are provided to ensure high quality code generation. In contrast to the `pybind11_add_module()` command, the CMake interface library only provides the *minimal* set of parameters to ensure that the code using pybind11 compiles, but it does **not** pass these extra compiler flags (i.e. this is up to you).

These include Link Time Optimization (`-flto` on GCC/Clang/ICPC, `/GL` and `/LTCG` on Visual Studio). Default-hidden symbols on GCC/Clang/ICPC (`-fvisibility=hidden`) and .OBJ files with many sections on Visual Studio (`/bigobj`). The *FAQ* contains an explanation on why these are needed.

Embedding the Python interpreter

In addition to extension modules, pybind11 also supports embedding Python into a C++ executable or library. In CMake, simply link with the `pybind11::embed` target. It provides everything needed to get the interpreter run-

ning. The Python headers and libraries are attached to the target. Unlike `pybind11::module`, there is no need to manually set any additional properties here. For more information about usage in C++, see *Embedding the interpreter*.

```
cmake_minimum_required(VERSION 3.0)
project(example)

find_package(pybind11 REQUIRED)  # or add_subdirectory(pybind11)

add_executable(example main.cpp)
target_link_libraries(example PRIVATE pybind11::embed)
```

Generating binding code automatically

The `Binder` project is a tool for automatic generation of pybind11 binding code by introspecting existing C++ codebases using LLVM/Clang. See the *[binder]* documentation for details.

CHAPTER 6

Functions

Before proceeding with this section, make sure that you are already familiar with the basics of binding functions and classes, as explained in *First steps* and *Object-oriented code*. The following guide is applicable to both free and member functions, i.e. *methods* in Python.

Return value policies

Python and C++ use fundamentally 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 policies are tricky, and it's very important to get them right. Just to illustrate what can go wrong, consider the following simple example:

```
/* Function declaration */
Data *get_data() { return _data; /* (pointer to a static data structure) */ }
...

/* Binding code */
m.def("get_data", &get_data); // <-- KABOOM, will cause crash when called from Python
```

What's going on here? When `get_data()` is called from Python, the return value (a native C++ type) must be wrapped to turn it into a usable Python type. In this case, the default return value policy (`return_value_policy::automatic`) causes pybind11 to assume ownership of the static `_data` instance.

When Python's garbage collector eventually deletes the Python wrapper, pybind11 will also attempt to delete the C++ instance (via `operator delete()`) due to the implied ownership. At this point, the entire application will come crashing down, though errors could also be more subtle and involve silent data corruption.

In the above example, the policy `return_value_policy::reference` should have been specified so that the global data instance is only *referenced* without any implied transfer of ownership, i.e.:

```
m.def("get_data", &get_data, return_value_policy::reference);
```

On the other hand, this is not the right policy for many other situations, where ignoring ownership could lead to resource leaks. As a developer using pybind11, it's important to be familiar with the different return value policies, including which situation calls for which one of them. The following table provides an overview of available policies:

Return value policy	Description
`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, or when the data was not dynamically allocated.
`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`	Indicates that the lifetime of the return value is tied to the lifetime of a parent object, namely the implicit `this`, or `self` argument of the called method or property. Internally, this policy works just like `return_value_policy::reference` but additionally applies a `keep_alive<0, 1>` *call policy* (described in the next section) that prevents the parent object from being garbage collected as long as the return value is referenced by Python. This is the default policy for property getters created via `def_property`, `def_readwrite`, etc.
`return_value_policy::automatic`	**Default policy.** This policy falls back to the policy `return_value_policy::take_ownership` when the return value is a pointer. Otherwise, it uses `return_value_policy::move` or `return_value_policy::copy` for rvalue and lvalue references, respectively. See above 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. This is the default conversion policy for function arguments when calling Python functions manually from C++ code (i.e. via handle::operator()). You probably won't need to use this.

Return value policies can also be applied to properties:

```
class_<MyClass>(m, "MyClass")
    .def_property("data", &MyClass::getData, &MyClass::setData,
                  py::return_value_policy::copy);
```

Technically, the code above applies the policy to both the getter and the setter function, however, the setter doesn't really care about *return* value policies which makes this a convenient terse syntax. Alternatively, targeted arguments can be passed through the `cpp_function` constructor:

```
class_<MyClass>(m, "MyClass")
    .def_property("data"
        py::cpp_function(&MyClass::getData, py::return_value_policy::copy),
        py::cpp_function(&MyClass::setData)
    );
```

Warning: Code with invalid return value 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: One important aspect of the above policies is that they only apply to instances which pybind11 has *not* seen before, in which case the policy clarifies essential questions about the return value's lifetime and ownership. When pybind11 knows the instance already (as identified by its type and address in memory), it will return the existing Python object wrapper rather than creating a new copy.

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 *Custom 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.

Additional call policies

In addition to the above return value policies, further *call policies* can be specified to indicate dependencies between parameters or ensure a certain state for the function call.

Keep alive

In general, this policy is required when the C++ object is any kind of container and another object is being added to the container. `keep_alive<Nurse, Patient>` 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 `1` refers to the implicit `this` pointer, while regular

arguments begin at index `2`. Arbitrarily many call policies can be specified. When a `Nurse` with value `None` is detected at runtime, the call policy does nothing.

This feature internally relies on the ability to create a *weak reference* to the nurse object, which is permitted by all classes exposed via pybind11. When the nurse object does not support weak references, an exception will be thrown.

Consider the following example: here, the binding code for a list append operation ties the lifetime of the newly added element to the underlying container:

```
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.

Call guard

The `call_guard<T>` policy allows any scope guard type `T` to be placed around the function call. For example, this definition:

```
m.def("foo", foo, py::call_guard<T>());
```

is equivalent to the following pseudocode:

```
m.def("foo", [](args...) {
    T scope_guard;
    return foo(args...); // forwarded arguments
});
```

The only requirement is that `T` is default-constructible, but otherwise any scope guard will work. This is very useful in combination with `gil_scoped_release`. See *Global Interpreter Lock (GIL)*.

Multiple guards can also be specified as `py::call_guard<T1, T2, T3...>`. The constructor order is left to right and destruction happens in reverse.

See also:

The file `tests/test_call_policies.cpp` contains a complete example that demonstrates using *`keep_alive`* and *`call_guard`* in more detail.

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=" << std::string(py::str(item.first)) << ", "
                  << "value=" << std::string(py::str(item.second)) << std::endl;
}
```

It can be exported:

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

And used in Python as usual:

```
>>> print_dict({'foo': 123, 'bar': 'hello'})
key=foo, value=123
key=bar, value=hello
```

For more information on using Python objects in C++, see *Python C++ interface*.

Accepting *args and **kwargs

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);
```

The class `py::args` derives from `py::tuple` and `py::kwargs` derives from `py::dict`.

You may also use just one or the other, and may combine these with other arguments as long as the `py::args` and `py::kwargs` arguments are the last arguments accepted by the function.

Please refer to the other examples for details on how to iterate over these, and on how to cast their entries into C++ objects. A demonstration is also available in `tests/test_kwargs_and_defaults.cpp`.

Note: When combining *args or **kwargs with *Keyword arguments* you should *not* include `py::arg` tags for the `py::args` and `py::kwargs` arguments.

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_v` notation:

```
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg_v("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);
```

Non-converting arguments

Certain argument types may support conversion from one type to another. Some examples of conversions are:

- *Implicit conversions* declared using `py::implicitly_convertible<A,B>()`
- Calling a method accepting a double with an integer argument
- Calling a `std::complex<float>` argument with a non-complex python type (for example, with a float). (Requires the optional `pybind11/complex.h` header).
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout. (Requires the optional `pybind11/eigen.h` header).

This behaviour is sometimes undesirable: the binding code may prefer to raise an error rather than convert the argument. This behaviour can be obtained through `py::arg` by calling the `.noconvert()` method of the `py::arg` object, such as:

```
m.def("floats_only", [](double f) { return 0.5 * f; }, py::arg("f").noconvert());
m.def("floats_preferred", [](double f) { return 0.5 * f; }, py::arg("f"));
```

Attempting the call the second function (the one without `.noconvert()`) with an integer will succeed, but attempting to call the `.noconvert()` version will fail with a `TypeError`:

```
>>> floats_preferred(4)
2.0
>>> floats_only(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: floats_only(): incompatible function arguments. The following argument␣
↪types are supported:
    1. (f: float) -> float
```

```
Invoked with: 4
```

You may, of course, combine this with the _a shorthand notation (see *Keyword arguments*) and/or *Default arguments*. It is also permitted to omit the argument name by using the `py::arg()` constructor without an argument name, i.e. by specifying `py::arg().noconvert()`.

Note: When specifying `py::arg` options it is necessary to provide the same number of options as the bound function has arguments. Thus if you want to enable no-convert behaviour for just one of several arguments, you will need to specify a `py::arg()` annotation for each argument with the no-convert argument modified to `py::arg().noconvert()`.

Allow/Prohibiting None arguments

When a C++ type registered with `py::class_` is passed as an argument to a function taking the instance as pointer or shared holder (e.g. `shared_ptr` or a custom, copyable holder as described in *Custom smart pointers*), pybind allows `None` to be passed from Python which results in calling the C++ function with `nullptr` (or an empty holder) for the argument.

To explicitly enable or disable this behaviour, using the `.none` method of the `py::arg` object:

```
py::class_<Dog>(m, "Dog").def(py::init<>());
py::class_<Cat>(m, "Cat").def(py::init<>());
m.def("bark", [](Dog *dog) -> std::string {
    if (dog) return "woof!"; /* Called with a Dog instance */
    else return "(no dog)"; /* Called with None, d == nullptr */
}, py::arg("dog").none(true));
m.def("meow", [](Cat *cat) -> std::string {
    // Can't be called with None argument
    return "meow";
}, py::arg("cat").none(false));
```

With the above, the Python call `bark(None)` will return the string `"(no dog)"`, while attempting to call `meow(None)` will raise a `TypeError`:

```
>>> from animals import Dog, Cat, bark, meow
>>> bark(Dog())
'woof!'
>>> meow(Cat())
'meow'
>>> bark(None)
'(no dog)'
>>> meow(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: meow(): incompatible function arguments. The following argument types are␣
↪supported:
    1. (cat: animals.Cat) -> str

Invoked with: None
```

The default behaviour when the tag is unspecified is to allow `None`.

Overload resolution order

When a function or method with multiple overloads is called from Python, pybind11 determines which overload to call in two passes. The first pass attempts to call each overload without allowing argument conversion (as if every argument had been specified as `py::arg().noconvert()` as decribed above).

If no overload succeeds in the no-conversion first pass, a second pass is attempted in which argument conversion is allowed (except where prohibited via an explicit `py::arg().noconvert()` attribute in the function definition).

If the second pass also fails a `TypeError` is raised.

Within each pass, overloads are tried in the order they were registered with pybind11.

What this means in practice is that pybind11 will prefer any overload that does not require conversion of arguments to an overload that does, but otherwise prefers earlier-defined overloads to later-defined ones.

Note: pybind11 does *not* further prioritize based on the number/pattern of overloaded arguments. That is, pybind11 does not prioritize a function requiring one conversion over one requiring three, but only prioritizes overloads requiring no conversion at all to overloads that require conversion of at least one argument.

CHAPTER 7

Classes

This section presents advanced binding code for classes and it is assumed that you are already familiar with the basics from *Object-oriented code*.

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) override {
        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_MODULE(example, m) {
    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);
}
```

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) override {
        PYBIND11_OVERLOAD_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function in C++ (must match Python name) */
            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 between the *Parent class* and *Name of the function* slots, which defines the name of function in Python. This is required 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_MODULE(example, m) {
    py::class_<Animal, PyAnimal /* <--- trampoline*/> animal(m, "Animal");
    animal
        .def(py::init<>())
        .def("go", &Animal::go);

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

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

Importantly, pybind11 is made aware of the trampoline helper class by specifying it as an extra template argument to `class_`. (This can also be combined with other template arguments such as a custom holder type; the order of template types does not matter). Following this, we are able to define a constructor as usual.

Bindings should be made against the actual class, not the trampoline helper class.

```
py::class_<Animal, PyAnimal /* <--- trampoline*/> animal(m, "Animal");
    animal
        .def(py::init<>())
        .def("go", &PyAnimal::go); /* <--- THIS IS WRONG, use &Animal::go */
```

Note, however, that the above is sufficient for allowing python classes to extend `Animal`, but not `Dog`: see ref:`virtual_and_inheritance` for the necessary steps required to providing proper overload support for inherited classes.

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):
...     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.

Note: When the overridden type returns a reference or pointer to a type that pybind11 converts from Python (for example, numeric values, std::string, and other built-in value-converting types), there are some limitations to be aware of:

- because in these cases there is no C++ variable to reference (the value is stored in the referenced Python variable), pybind11 provides one in the PYBIND11_OVERLOAD macros (when needed) with static storage duration. Note that this means that invoking the overloaded method on *any* instance will change the referenced value stored in *all* instances of that type.
- Attempts to modify a non-const reference will not have the desired effect: it will change only the static cache variable, but this change will not propagate to underlying Python instance, and the change will be replaced the next time the overload is invoked.

See also:

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

Combining virtual functions and inheritance

When combining virtual methods with inheritance, you need to be sure to provide an override for each method for which you want to allow overrides from derived python classes. For example, suppose we extend the above `Animal`/`Dog` example as follows:

```
class Animal {
public:
    virtual std::string go(int n_times) = 0;
    virtual std::string name() { return "unknown"; }
};
class Dog : public Animal {
```

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

then the trampoline class for `Animal` must, as described in the previous section, override `go()` and `name()`, but in order to allow python code to inherit properly from `Dog`, we also need a trampoline class for `Dog` that overrides both the added `bark()` method *and* the `go()` and `name()` methods inherited from `Animal` (even though `Dog` doesn't directly override the `name()` method):

```
class PyAnimal : public Animal {
public:
    using Animal::Animal; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERLOAD_PURE(std::string, Animal,
↪ go, n_times); }
    std::string name() override { PYBIND11_OVERLOAD(std::string, Animal, name, ); }
};
class PyDog : public Dog {
public:
    using Dog::Dog; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERLOAD_PURE(std::string, Dog,␣
↪go, n_times); }
    std::string name() override { PYBIND11_OVERLOAD(std::string, Dog, name, ); }
    std::string bark() override { PYBIND11_OVERLOAD(std::string, Dog, bark, ); }
};
```

Note: Note the trailing commas in the `PYBIND11_OVERLOAD` calls to `name()` and `bark()`. These are needed to portably implement a trampoline for a function that does not take any arguments. For functions that take a nonzero number of arguments, the trailing comma must be omitted.

A registered class derived from a pybind11-registered class with virtual methods requires a similar trampoline class, *even if* it doesn't explicitly declare or override any virtual methods itself:

```
class Husky : public Dog {};
class PyHusky : public Husky {
public:
    using Husky::Husky; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERLOAD_PURE(std::string, Husky,␣
↪go, n_times); }
    std::string name() override { PYBIND11_OVERLOAD(std::string, Husky, name, ); }
    std::string bark() override { PYBIND11_OVERLOAD(std::string, Husky, bark, ); }
};
```

There is, however, a technique that can be used to avoid this duplication (which can be especially helpful for a base class with several virtual methods). The technique involves using template trampoline classes, as follows:

```
template <class AnimalBase = Animal> class PyAnimal : public AnimalBase {
public:
    using AnimalBase::AnimalBase; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERLOAD_PURE(std::string,␣
↪AnimalBase, go, n_times); }
```

```
    std::string name() override { PYBIND11_OVERLOAD(std::string, AnimalBase, name, );␣
↪}
};
template <class DogBase = Dog> class PyDog : public PyAnimal<DogBase> {
public:
    using PyAnimal<DogBase>::PyAnimal; // Inherit constructors
    // Override PyAnimal's pure virtual go() with a non-pure one:
    std::string go(int n_times) override { PYBIND11_OVERLOAD(std::string, DogBase, go,
↪ n_times); }
    std::string bark() override { PYBIND11_OVERLOAD(std::string, DogBase, bark, ); }
};
```

This technique has the advantage of requiring just one trampoline method to be declared per virtual method and pure virtual method override. It does, however, require the compiler to generate at least as many methods (and possibly more, if both pure virtual and overridden pure virtual methods are exposed, as above).

The classes are then registered with pybind11 using:

```
py::class_<Animal, PyAnimal<>> animal(m, "Animal");
py::class_<Dog, PyDog<>> dog(m, "Dog");
py::class_<Husky, PyDog<Husky>> husky(m, "Husky");
// ... add animal, dog, husky definitions
```

Note that `Husky` did not require a dedicated trampoline template class at all, since it neither declares any new virtual methods nor provides any pure virtual method implementations.

With either the repeated-virtuals or templated trampoline methods in place, you can now create a python class that inherits from `Dog`:

```
class ShihTzu(Dog):
    def bark(self):
        return "yip!"
```

See also:

See the file `tests/test_virtual_functions.cpp` for complete examples using both the duplication and templated trampoline approaches.

Extended trampoline class functionality

The trampoline classes described in the previous sections are, by default, only initialized when needed. More specifically, they are initialized when a python class actually inherits from a registered type (instead of merely creating an instance of the registered type), or when a registered constructor is only valid for the trampoline class but not the registered class. This is primarily for performance reasons: when the trampoline class is not needed for anything except virtual method dispatching, not initializing the trampoline class improves performance by avoiding needing to do a run-time check to see if the inheriting python instance has an overloaded method.

Sometimes, however, it is useful to always initialize a trampoline class as an intermediate class that does more than just handle virtual method dispatching. For example, such a class might perform extra class initialization, extra destruction operations, and might define new members and methods to enable a more python-like interface to a class.

In order to tell pybind11 that it should *always* initialize the trampoline class when creating new instances of a type, the class constructors should be declared using `py::init_alias<Args, ...>()` instead of the usual `py::init<Args, ...>()`. This forces construction via the trampoline class, ensuring member initialization and (eventual) destruction.

See also:

See the file `tests/test_virtual_functions.cpp` for complete examples showing both normal and forced trampoline instantiation.

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.

Non-public destructors

If a class has a private or protected destructor (as might e.g. be the case in a singleton pattern), a compile error will occur when creating bindings via pybind11. The underlying issue is that the `std::unique_ptr` holder type that is responsible for managing the lifetime of instances will reference the destructor even if no deallocations ever take place. In order to expose classes with private or protected destructors, it is possible to override the holder type via a holder type argument to `class_`. Pybind11 provides a helper class `py::nodelete` that disables any destructor invocations. In this case, it is crucial that instances are deallocated on the C++ side to avoid memory leaks.

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

class MyClass {
private:
    ~MyClass() { }
};

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

py::class_<MyClass, std::unique_ptr<MyClass, py::nodelete>>(m, "MyClass")
    .def(py::init<>())
```

Implicit 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>();
```

Note: Implicit conversions from `A` to `B` only work when `B` is a custom data type that is exposed to Python via pybind11.

Static properties

The section on *Instance and static fields* discussed the creation of instance properties that are implemented in terms of C++ getters and setters.

Static properties can also be created in a similar way to expose getters and setters of static class attributes. Note that the implicit `self` argument also exists in this case and is used to pass the Python `type` subclass instance. This parameter will often not be needed by the C++ side, and the following example illustrates how to instantiate a lambda getter function that ignores it:

```
py::class_<Foo>(m, "Foo")
    .def_property_readonly_static("foo", [](py::object /* self */) { return Foo(); });
```

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); }
    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_MODULE(example, m) {
    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(py::self * float())
        .def("__repr__", &Vector2::toString);
}
```

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;
}, py::is_operator())
```

This can be useful for exposing additional operators that don't exist on the C++ side, or to perform other types of customization. The `py::is_operator` flag marker is needed to inform pybind11 that this is an operator, which returns `NotImplemented` when invoked with incompatible arguments rather than throwing a type error.

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

See also:

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

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[1] looks as follows:

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

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

Multiple Inheritance

pybind11 can create bindings for types that derive from multiple base types (aka. *multiple inheritance*). To do so, specify all bases in the template arguments of the `class_` declaration:

```
py::class_<MyType, BaseType1, BaseType2, BaseType3>(m, "MyType")
   ...
```

The base types can be specified in arbitrary order, and they can even be interspersed with alias types and holder types (discussed earlier in this document)—pybind11 will automatically find out which is which. The only requirement is that the first template argument is the type to be declared.

It is also permitted to inherit multiply from exported C++ classes in Python, as well as inheriting from multiple Python and/or pybind-exported classes.

There is one caveat regarding the implementation of this feature:

When only one base type is specified for a C++ type that actually has multiple bases, pybind11 will assume that it does not participate in multiple inheritance, which can lead to undefined behavior. In such cases, add the tag `multiple_inheritance` to the class constructor:

```
py::class_<MyType, BaseType2>(m, "MyType", py::multiple_inheritance());
```

The tag is redundant and does not need to be specified when multiple base types are listed.

CHAPTER 8

Exceptions

Built-in exception translation

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::key_error`	`KeyError` (used to indicate out of bounds accesses in `__getitem__`, `__setitem__` in dict-like objects, etc.)
`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.

Registering custom translators

If the default exception conversion policy described above is insufficient, pybind11 also provides support for registering custom exception translators. To register a simple exception conversion that translates a C++ exception into a new Python exception using the C++ exception's `what()` method, a helper function is available:

```
py::register_exception<CppExp>(module, "PyExp");
```

This call creates a Python exception class with the name `PyExp` in the given module and automatically converts any encountered exceptions of type `CppExp` into Python exceptions of type `PyExp`.

When more advanced exception translation is needed, the function `py::register_exception_translator(translator)` can be used to register functions that can translate arbitrary exception types (and which may include additional logic to do so). The function takes a stateless callable (e.g. a function pointer or a lambda function without captured variables) with the call signature `void(std::exception_ptr)`.

When a C++ exception is thrown, the registered exception translators are tried in reverse order of registration (i.e. the last registered translator gets the first shot at handling the exception).

Inside the translator, `std::rethrow_exception` should be used within a try block to re-throw the exception. One or more catch clauses to catch the appropriate exceptions should then be used with each clause using `PyErr_SetString` to set a Python exception or `ex(string)` to set the python exception to a custom exception type (see below).

To declare a custom Python exception type, declare a `py::exception` variable and use this in the associated exception translator (note: it is often useful to make this a static declaration when using it inside a lambda expression without requiring capturing).

The following example demonstrates this for a hypothetical exception classes `MyCustomException` and `OtherException`: the first is translated to a custom python exception `MyCustomError`, while the second is translated to a standard python RuntimeError:

```
static py::exception<MyCustomException> exc(m, "MyCustomError");
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    } catch (const MyCustomException &e) {
        exc(e.what());
    } catch (const OtherException &e) {
        PyErr_SetString(PyExc_RuntimeError, e.what());
    }
});
```

Multiple exceptions can be handled by a single translator, as shown in the example above. If the exception is not caught by the current translator, the previously registered one gets a chance.

If none of the registered exception translators is able to handle the exception, it is handled by the default converter as described in the previous section.

See also:

The file `tests/test_exceptions.cpp` contains examples of various custom exception translators and custom exception types.

Note: You must call either `PyErr_SetString` or a custom exception's call operator (`exc(string)`) for every exception caught in a custom exception translator. Failure to do so will cause Python to crash with `SystemError: error return without exception set`.

Exceptions that you do not plan to handle should simply not be caught, or may be explicity (re-)thrown to delegate it to the other, previously-declared existing exception translators.

CHAPTER 9

Smart pointers

std::unique_ptr

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).

std::shared_ptr

The binding generator for classes, `class_`, can be passed a template type that denotes a special *holder* type that is used to manage references to the object. If no such holder type template argument is given, the default for a type named `Type` 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.

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_MODULE(example, m) {
    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);
}
```

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> { };
```

Custom smart pointers

pybind11 supports `std::unique_ptr` and `std::shared_ptr` right out of the box. For any other custom smart pointer, transparent conversions can be enabled using a macro invocation similar to the following. It must be declared at the top namespace level before any binding code:

```
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>);
```

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.

The macro also accepts a third optional boolean parameter that is set to false by default. Specify

```
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>, true);
```

if `SmartPtr<T>` can always be initialized from a `T*` pointer without the risk of inconsistencies (such as multiple independent `SmartPtr` instances believing that they are the sole owner of the `T*` pointer). A common situation where `true` should be passed is when the `T` instances use *intrusive* reference counting.

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

By default, pybind11 assumes that your custom smart pointer has a standard interface, i.e. provides a `.get()` member function to access the underlying raw pointer. If this is not the case, pybind11's `holder_helper` must be specialized:

```
// Always needed for custom holder types
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>);

// Only needed if the type's `.get()` goes by another name
namespace pybind11 { namespace detail {
    template <typename T>
    struct holder_helper<SmartPtr<T>> { // <-- specialization
        static const T *get(const SmartPtr<T> &p) { return p.getPointer(); }
    };
}}
```

The above specialization informs pybind11 that the custom `SmartPtr` class provides `.get()` functionality via `.getPointer()`.

See also:

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

CHAPTER 10

Type conversions

Apart from enabling cross-language function calls, a fundamental problem that a binding tool like pybind11 must address is to provide access to native Python types in C++ and vice versa. There are three fundamentally different ways to do this—which approach is preferable for a particular type depends on the situation at hand.

1. Use a native C++ type everywhere. In this case, the type must be wrapped using pybind11-generated bindings so that Python can interact with it.
2. Use a native Python type everywhere. It will need to be wrapped so that C++ functions can interact with it.
3. Use a native C++ type on the C++ side and a native Python type on the Python side. pybind11 refers to this as a *type conversion*.

   Type conversions are the most "natural" option in the sense that native (non-wrapped) types are used everywhere. The main downside is that a copy of the data must be made on every Python C++ transition: this is needed since the C++ and Python versions of the same type generally won't have the same memory layout.

   pybind11 can perform many kinds of conversions automatically. An overview is provided in the table "*List of all builtin conversions*".

The following subsections discuss the differences between these options in more detail. The main focus in this section is on type conversions, which represent the last case of the above list.

Overview

1. Native type in C++, wrapper in Python

Exposing a custom C++ type using `py::class_` was covered in detail in the *Object-oriented code* section. There, the underlying data structure is always the original C++ class while the `py::class_` wrapper provides a Python interface. Internally, when an object like this is sent from C++ to Python, pybind11 will just add the outer wrapper layer over the native C++ object. Getting it back from Python is just a matter of peeling off the wrapper.

2. Wrapper in C++, native type in Python

This is the exact opposite situation. Now, we have a type which is native to Python, like a `tuple` or a `list`. One way to get this data into C++ is with the `py::object` family of wrappers. These are explained in more detail in the *Python types* section. We'll just give a quick example here:

```
void print_list(py::list my_list) {
    for (auto item : my_list)
        std::cout << item << " ";
}
```

```
>>> print_list([1, 2, 3])
1 2 3
```

The Python `list` is not converted in any way – it's just wrapped in a C++ `py::list` class. At its core it's still a Python object. Copying a `py::list` will do the usual reference-counting like in Python. Returning the object to Python will just remove the thin wrapper.

3. Converting between native C++ and Python types

In the previous two cases we had a native type in one language and a wrapper in the other. Now, we have native types on both sides and we convert between them.

```
void print_vector(const std::vector<int> &v) {
    for (auto item : v)
        std::cout << item << "\n";
}
```

```
>>> print_vector([1, 2, 3])
1 2 3
```

In this case, pybind11 will construct a new `std::vector<int>` and copy each element from the Python `list`. The newly constructed object will be passed to `print_vector`. The same thing happens in the other direction: a new `list` is made to match the value returned from C++.

Lots of these conversions are supported out of the box, as shown in the table below. They are very convenient, but keep in mind that these conversions are fundamentally based on copying data. This is perfectly fine for small immutable types but it may become quite expensive for large data structures. This can be avoided by overriding the automatic conversion with a custom wrapper (i.e. the above-mentioned approach 1). This requires some manual effort and more details are available in the *Making opaque types* section.

List of all builtin conversions

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`

Continued on next page

Table 10.1 – continued from previous page

Data type	Description	Header file
`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`
`char16_t`	UTF-16 character literal	`pybind11/pybind11.h`
`char32_t`	UTF-32 character literal	`pybind11/pybind11.h`
`wchar_t`	Wide character literal	`pybind11/pybind11.h`
`const char *`	UTF-8 string literal	`pybind11/pybind11.h`
`const char16_t *`	UTF-16 string literal	`pybind11/pybind11.h`
`const char32_t *`	UTF-32 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::u16string`	STL dynamic UTF-16 string	`pybind11/pybind11.h`
`std::u32string`	STL dynamic UTF-32 string	`pybind11/pybind11.h`
`std::wstring`	STL dynamic wide string	`pybind11/pybind11.h`
`std::string_view`, `std::u16string_view`, etc.	STL C++17 string views	`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::valarray<T>`	STL value 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::optional<T>`	STL optional type (C++17)	`pybind11/stl.h`
`std::experimental::optional<T>`	STL optional type (exp.)	`pybind11/stl.h`
`std::variant<...>`	Type-safe union (C++17)	`pybind11/stl.h`
`std::function<...>`	STL polymorphic function	`pybind11/functional.h`
`std::chrono::duration<...>`	STL time duration	`pybind11/chrono.h`
`std::chrono::time_point<...>`	STL date/time	`pybind11/chrono.h`
`Eigen::Matrix<...>`	Eigen: dense matrix	`pybind11/eigen.h`
`Eigen::Map<...>`	Eigen: mapped memory	`pybind11/eigen.h`
`Eigen::SparseMatrix<...>`	Eigen: sparse matrix	`pybind11/eigen.h`

Strings, bytes and Unicode conversions

Note: This section discusses string handling in terms of Python 3 strings. For Python 2.7, replace all occurrences of `str` with `unicode` and `bytes` with `str`. Python 2.7 users may find it best to use `from __future__ import unicode_literals` to avoid unintentionally using `str` instead of `unicode`.

Passing Python strings to C++

When a Python `str` is passed from Python to a C++ function that accepts `std::string` or `char *` as arguments, pybind11 will encode the Python string to UTF-8. All Python `str` can be encoded in UTF-8, so this operation does not fail.

The C++ language is encoding agnostic. It is the responsibility of the programmer to track encodings. It's often easiest to simply use UTF-8 everywhere.

```
m.def("utf8_test",
    [](const std::string &s) {
        cout << "utf-8 is icing on the cake.\n";
        cout << s;
    }
);
m.def("utf8_charptr",
    [](const char *s) {
        cout << "My favorite food is\n";
        cout << s;
    }
);
```

```
>>> utf8_test('')
utf-8 is icing on the cake.


>>> utf8_charptr('')
My favorite food is
```

Note: Some terminal emulators do not support UTF-8 or emoji fonts and may not display the example above correctly.

The results are the same whether the C++ function accepts arguments by value or reference, and whether or not `const` is used.

Passing bytes to C++

A Python `bytes` object will be passed to C++ functions that accept `std::string` or `char*` *without* conversion.

Returning C++ strings to Python

When a C++ function returns a `std::string` or `char*` to a Python caller, **pybind11 will assume that the string is valid UTF-8** and will decode it to a native Python `str`, using the same API as Python uses to perform `bytes.decode('utf-8')`. If this implicit conversion fails, pybind11 will raise a `UnicodeDecodeError`.

```
m.def("std_string_return",
    []() {
        return std::string("This string needs to be UTF-8 encoded");
    }
);
```

```
>>> isinstance(example.std_string_return(), str)
True
```

Because UTF-8 is inclusive of pure ASCII, there is never any issue with returning a pure ASCII string to Python. If there is any possibility that the string is not pure ASCII, it is necessary to ensure the encoding is valid UTF-8.

Warning: Implicit conversion assumes that a returned `char *` is null-terminated. If there is no null terminator a buffer overrun will occur.

Explicit conversions

If some C++ code constructs a `std::string` that is not a UTF-8 string, one can perform a explicit conversion and return a `py::str` object. Explicit conversion has the same overhead as implicit conversion.

```
// This uses the Python C API to convert Latin-1 to Unicode
m.def("str_output",
    [] () {
        std::string s = "Send your r\xe9sum\xe9 to Alice in HR"; // Latin-1
        py::str py_s = PyUnicode_DecodeLatin1(s.data(), s.length());
        return py_s;
    }
);
```

```
>>> str_output()
'Send your résumé to Alice in HR'
```

The Python C API provides several built-in codecs.

One could also use a third party encoding library such as libiconv to transcode to UTF-8.

Return C++ strings without conversion

If the data in a C++ `std::string` does not represent text and should be returned to Python as `bytes`, then one can return the data as a `py::bytes` object.

```
m.def("return_bytes",
    [] () {
        std::string s("\xba\xd0\xba\xd0");  // Not valid UTF-8
        return py::bytes(s);  // Return the data without transcoding
    }
);
```

```
>>> example.return_bytes()
b'\xba\xd0\xba\xd0'
```

Note the asymmetry: pybind11 will convert `bytes` to `std::string` without encoding, but cannot convert `std::string` back to `bytes` implicitly.

```
m.def("asymmetry",
    [](std::string s) {  // Accepts str or bytes from Python
        return s;  // Looks harmless, but implicitly converts to str
    }
);
```

```
>>> isinstance(example.asymmetry(b"have some bytes"), str)
True
```

```
>>> example.asymmetry(b"\xba\xd0\xba\xd0")  # invalid utf-8 as bytes
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xba in position 0: invalid start␣
↪byte
```

Wide character strings

When a Python `str` is passed to a C++ function expecting `std::wstring`, `wchar_t*`, `std::u16string` or `std::u32string`, the `str` will be encoded to UTF-16 or UTF-32 depending on how the C++ compiler implements each type, in the platform's native endianness. When strings of these types are returned, they are assumed to contain valid UTF-16 or UTF-32, and will be decoded to Python `str`.

```
#define UNICODE
#include <windows.h>

m.def("set_window_text",
    [](HWND hwnd, std::wstring s) {
        // Call SetWindowText with null-terminated UTF-16 string
        ::SetWindowText(hwnd, s.c_str());
    }
);
m.def("get_window_text",
    [](HWND hwnd) {
        const int buffer_size = ::GetWindowTextLength(hwnd) + 1;
        auto buffer = std::make_unique< wchar_t[] >(buffer_size);

        ::GetWindowText(hwnd, buffer.data(), buffer_size);

        std::wstring text(buffer.get());

        // wstring will be converted to Python str
        return text;
    }
);
```

Warning: Wide character strings may not work as described on Python 2.7 or Python 3.3 compiled with `--enable-unicode=ucs2`.

Strings in multibyte encodings such as Shift-JIS must transcoded to a UTF-8/16/32 before being returned to Python.

Character literals

C++ functions that accept character literals as input will receive the first character of a Python `str` as their input. If the string is longer than one Unicode character, trailing characters will be ignored.

When a character literal is returned from C++ (such as a `char` or a `wchar_t`), it will be converted to a `str` that represents the single character.

```
m.def("pass_char", [](char c) { return c; });
m.def("pass_wchar", [](wchar_t w) { return w; });
```

```
>>> example.pass_char('A')
'A'
```

While C++ will cast integers to character types (`char c = 0x65;`), pybind11 does not convert Python integers to characters implicitly. The Python function `chr()` can be used to convert integers to characters.

```
>>> example.pass_char(0x65)
TypeError

>>> example.pass_char(chr(0x65))
'A'
```

If the desire is to work with an 8-bit integer, use `int8_t` or `uint8_t` as the argument type.

Grapheme clusters

A single grapheme may be represented by two or more Unicode characters. For example 'é' is usually represented as U+00E9 but can also be expressed as the combining character sequence U+0065 U+0301 (that is, the letter 'e' followed by a combining acute accent). The combining character will be lost if the two-character sequence is passed as an argument, even though it renders as a single grapheme.

```
>>> example.pass_wchar('é')
'é'

>>> combining_e_acute = 'e' + '\u0301'

>>> combining_e_acute
'e'

>>> combining_e_acute == 'é'
False

>>> example.pass_wchar(combining_e_acute)
'e'
```

Normalizing combining characters before passing the character literal to C++ may resolve *some* of these issues:

```
>>> example.pass_wchar(unicodedata.normalize('NFC', combining_e_acute))
'é'
```

In some languages (Thai for example), there are graphemes that cannot be expressed as a single Unicode code point, so there is no way to capture them in a C++ character type.

C++17 string views

C++17 string views are automatically supported when compiling in C++17 mode. They follow the same rules for encoding and decoding as the corresponding STL string type (for example, a `std::u16string_view` argument will be passed UTF-16-encoded data, and a returned `std::string_view` will be decoded as UTF-8).

References

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

- C++ - Using STL Strings at Win32 API Boundaries

STL containers

Automatic conversion

When including the additional header file `pybind11/stl.h`, conversions between `std::vector<>`/`std::list<>`/`std::array<>`, `std::set<>`/`std::unordered_set<>`, and `std::map<>`/`std::unordered_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.

The major downside of these implicit conversions is that containers must be converted (i.e. copied) on every Python->C++ and C++->Python transition, which can have implications on the program semantics and performance. Please read the next sections for more details and alternative approaches that avoid this.

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

See also:

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

C++17 library containers

The `pybind11/stl.h` header also includes support for `std::optional<>` and `std::variant<>`. These require a C++17 compiler and standard library. In C++14 mode, `std::experimental::optional<>` is supported if available.

Various versions of these containers also exist for C++11 (e.g. in Boost). pybind11 provides an easy way to specialize the `type_caster` for such types:

```
// `boost::optional` as an example -- can be any `std::optional`-like container
namespace pybind11 { namespace detail {
    template <typename T>
    struct type_caster<boost::optional<T>> : optional_caster<boost::optional<T>> {};
}}
```

The above should be placed in a header file and included in all translation units where automatic conversion is needed. Similarly, a specialization can be provided for custom variant types:

```
// `boost::variant` as an example -- can be any `std::variant`-like container
namespace pybind11 { namespace detail {
    template <typename... Ts>
    struct type_caster<boost::variant<Ts...>> : variant_caster<boost::variant<Ts...>>
↪{};

    // Specifies the function used to visit the variant -- `apply_visitor` instead of␣
↪`visit`
    template <>
    struct visit_helper<boost::variant> {
        template <typename... Args>
        static auto call(Args &&...args)
```

```
            -> decltype(boost::apply_visitor(std::forward<Args>(args)...)) {
            return boost::apply_visitor(std::forward<Args>(args)...);
        }
    };
}} // namespace pybind11::detail
```

The `visit_helper` specialization is not required if your `name::variant` provides a `name::visit()` function. For any other function name, the specialization must be included to tell pybind11 how to visit the variant.

Making opaque types

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;
};

/* ... 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` operation 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]
```

Finally, the involved copy operations can be costly when dealing with very large lists. To deal with all 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 (and outside of any namespaces), since it 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, e.g.:

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

The ability to expose STL containers as native Python objects is a fairly common request, hence pybind11 also provides an optional header file named `pybind11/stl_bind.h` that does exactly this. The mapped containers try to match the behavior of their native Python counterparts as much as possible.

The following example showcases usage of `pybind11/stl_bind.h`:

```
// Don't forget this
#include <pybind11/stl_bind.h>

PYBIND11_MAKE_OPAQUE(std::vector<int>);
PYBIND11_MAKE_OPAQUE(std::map<std::string, double>);

// ...

// later in binding code:
py::bind_vector<std::vector<int>>(m, "VectorInt");
py::bind_map<std::map<std::string, double>>(m, "MapStringDouble");
```

Please take a look at the *General notes regarding convenience macros* before using the `PYBIND11_MAKE_OPAQUE` macro.

See also:

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

The file `tests/test_stl_binders.cpp` shows how to use the convenience STL container wrappers.

Functional

The following features must be enabled by including `pybind11/functional.h`.

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;
    };
}
```

This example demonstrates using python named parameters in C++ callbacks which requires using `py::cpp_function` as a wrapper. Usage is similar to defining methods of classes:

```
py::cpp_function func_cpp() {
    return py::cpp_function([](int i) { return i+1; },
       py::arg("number"));
}
```

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

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

PYBIND11_MODULE(example, m) {
    m.def("func_arg", &func_arg);
    m.def("func_ret", &func_ret);
    m.def("func_cpp", &func_cpp);
}
```

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
>>> plus_1 = func_cpp()
```

```
>>> plus_1(number=43)
44L
```

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. Naturally, this translation increases the computational cost of each function call somewhat. A problematic situation can arise when a function is copied back and forth between Python and C++ many times in a row, in which case the underlying wrappers will accumulate correspondingly. The resulting long sequence of C++ -> Python -> C++ -> ... roundtrips can significantly decrease performance.

There is one exception: pybind11 detects case where a stateless function (i.e. a function pointer or a lambda function without captured variables) is passed as an argument to another C++ function exposed in Python. In this case, there is no overhead. Pybind11 will extract the underlying C++ function pointer from the wrapped function to sidestep a potential C++ -> Python -> C++ roundtrip. This is demonstrated in `tests/test_callbacks.cpp`.

Note: This functionality is very useful when generating bindings for callbacks in C++ libraries (e.g. GUI libraries, asynchronous networking libraries, etc.).

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

Chrono

When including the additional header file `pybind11/chrono.h` conversions from C++11 chrono datatypes to python datetime objects are automatically enabled. This header also enables conversions of python floats (often from sources such as `time.monotonic()`, `time.perf_counter()` and `time.process_time()`) into durations.

An overview of clocks in C++11

A point of confusion when using these conversions is the differences between clocks provided in C++11. There are three clock types defined by the C++11 standard and users can define their own if needed. Each of these clocks have different properties and when converting to and from python will give different results.

The first clock defined by the standard is `std::chrono::system_clock`. This clock measures the current date and time. However, this clock changes with to updates to the operating system time. For example, if your time is synchronised with a time server this clock will change. This makes this clock a poor choice for timing purposes but good for measuring the wall time.

The second clock defined in the standard is `std::chrono::steady_clock`. This clock ticks at a steady rate and is never adjusted. This makes it excellent for timing purposes, however the value in this clock does not correspond to the current date and time. Often this clock will be the amount of time your system has been on, although it does not have to be. This clock will never be the same clock as the system clock as the system clock can change but steady clocks cannot.

The third clock defined in the standard is `std::chrono::high_resolution_clock`. This clock is the clock that has the highest resolution out of the clocks in the system. It is normally a typedef to either the system clock or the steady clock but can be its own independent clock. This is important as when using these conversions as the types you get in python for this clock might be different depending on the system. If it is a typedef of the system clock, python will get datetime objects, but if it is a different clock they will be timedelta objects.

Provided conversions

C++ to Python

- **std::chrono::system_clock::time_point** → **datetime.datetime** System clock times are converted to python datetime instances. They are in the local timezone, but do not have any timezone information attached to them (they are naive datetime objects).
- **std::chrono::duration** → **datetime.timedelta** Durations are converted to timedeltas, any precision in the duration greater than microseconds is lost by rounding towards zero.
- **std::chrono::[other_clocks]::time_point** → **datetime.timedelta** Any clock time that is not the system clock is converted to a time delta. This timedelta measures the time from the clocks epoch to now.

Python to C++

- **datetime.datetime** → **std::chrono::system_clock::time_point** Date/time objects are converted into system clock timepoints. Any timezone information is ignored and the type is treated as a naive object.
- **datetime.timedelta** → **std::chrono::duration** Time delta are converted into durations with microsecond precision.
- **datetime.timedelta** → **std::chrono::[other_clocks]::time_point** Time deltas that are converted into clock timepoints are treated as the amount of time from the start of the clocks epoch.
- **float** → **std::chrono::duration** Floats that are passed to C++ as durations be interpreted as a number of seconds. These will be converted to the duration using `duration_cast` from the float.
- **float** → **std::chrono::[other_clocks]::time_point** Floats that are passed to C++ as time points will be interpreted as the number of seconds from the start of the clocks epoch.

Eigen

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

To enable the built-in Eigen support you must include the optional header file `pybind11/eigen.h`.

Pass-by-value

When binding a function with ordinary Eigen dense object arguments (for example, `Eigen::MatrixXd`), pybind11 will accept any input value that is already (or convertible to) a `numpy.ndarray` with dimensions compatible with the Eigen type, copy its values into a temporary Eigen variable of the appropriate type, then call the function with this temporary variable.

Sparse matrices are similarly copied to or from `scipy.sparse.csr_matrix`/`scipy.sparse.csc_matrix` objects.

Pass-by-reference

One major limitation of the above is that every data conversion implicitly involves a copy, which can be both expensive (for large matrices) and disallows binding functions that change their (Matrix) arguments. Pybind11 allows you to work around this by using Eigen's `Eigen::Ref<MatrixType>` class much as you would when writing a function taking a generic type in Eigen itself (subject to some limitations discussed below).

When calling a bound function accepting a `Eigen::Ref<const MatrixType>` type, pybind11 will attempt to avoid copying by using an `Eigen::Map` object that maps into the source `numpy.ndarray` data: this requires both that the data types are the same (e.g. `dtype='float64'` and `MatrixType::Scalar` is `double`); and that the storage is layout compatible. The latter limitation is discussed in detail in the section below, and requires careful consideration: by default, numpy matrices and eigen matrices are *not* storage compatible.

If the numpy matrix cannot be used as is (either because its types differ, e.g. passing an array of integers to an Eigen paramater requiring doubles, or because the storage is incompatible), pybind11 makes a temporary copy and passes the copy instead.

When a bound function parameter is instead `Eigen::Ref<MatrixType>` (note the lack of `const`), pybind11 will only allow the function to be called if it can be mapped *and* if the numpy array is writeable (that is `a.flags.writeable` is true). Any access (including modification) made to the passed variable will be transparently carried out directly on the `numpy.ndarray`.

This means you can can write code such as the following and have it work as expected:

```
void scale_by_2(Eigen::Ref<Eigen::VectorXd> m) {
    v *= 2;
}
```

Note, however, that you will likely run into limitations due to numpy and Eigen's difference default storage order for data; see the below section on *Storage orders* for details on how to bind code that won't run into such limitations.

Note: Passing by reference is not supported for sparse types.

Returning values to Python

When returning an ordinary dense Eigen matrix type to numpy (e.g. `Eigen::MatrixXd` or `Eigen::RowVectorXf`) pybind11 keeps the matrix and returns a numpy array that directly references the Eigen matrix: no copy of the data is performed. The numpy array will have `array.flags.owndata` set to `False` to indicate that it does not own the data, and the lifetime of the stored Eigen matrix will be tied to the returned `array`.

If you bind a function with a non-reference, `const` return type (e.g. `const Eigen::MatrixXd`), the same thing happens except that pybind11 also sets the numpy array's `writeable` flag to false.

If you return an lvalue reference or pointer, the usual pybind11 rules apply, as dictated by the binding function's return value policy (see the documentation on *Return value policies* for full details). That means, without an explicit return value policy, lvalue references will be copied and pointers will be managed by pybind11. In order to avoid copying, you should explictly specify an appropriate return value policy, as in the following example:

```
class MyClass {
    Eigen::MatrixXd big_mat = Eigen::MatrixXd::Zero(10000, 10000);
public:
    Eigen::MatrixXd &getMatrix() { return big_mat; }
    const Eigen::MatrixXd &viewMatrix() { return big_mat; }
};

// Later, in binding code:
```

```
py::class_<MyClass>(m, "MyClass")
    .def(py::init<>())
    .def("copy_matrix", &MyClass::getMatrix) // Makes a copy!
    .def("get_matrix", &MyClass::getMatrix, py::return_value_policy::reference_
↪internal)
    .def("view_matrix", &MyClass::viewMatrix, py::return_value_policy::reference_
↪internal)
    ;
```

```
a = MyClass()
m = a.get_matrix()    # flags.writeable = True,  flags.owndata = False
v = a.view_matrix()   # flags.writeable = False, flags.owndata = False
c = a.copy_matrix()   # flags.writeable = True,  flags.owndata = True
# m[5,6] and v[5,6] refer to the same element, c[5,6] does not.
```

Note in this example that `py::return_value_policy::reference_internal` is used to tie the life of the MyClass object to the life of the returned arrays.

You may also return an `Eigen::Ref`, `Eigen::Map` or other map-like Eigen object (for example, the return value of `matrix.block()` and related methods) that map into a dense Eigen type. When doing so, the default behaviour of pybind11 is to simply reference the returned data: you must take care to ensure that this data remains valid! You may ask pybind11 to explicitly *copy* such a return value by using the `py::return_value_policy::copy` policy when binding the function. You may also use `py::return_value_policy::reference_internal` or a `py::keep_alive` to ensure the data stays valid as long as the returned numpy array does.

When returning such a reference of map, pybind11 additionally respects the readonly-status of the returned value, marking the numpy array as non-writeable if the reference or map was itself read-only.

Note: Sparse types are always copied when returned.

Storage orders

Passing arguments via `Eigen::Ref` has some limitations that you must be aware of in order to effectively pass matrices by reference. First and foremost is that the default `Eigen::Ref<MatrixType>` class requires contiguous storage along columns (for column-major types, the default in Eigen) or rows if `MatrixType` is specifically an `Eigen::RowMajor` storage type. The former, Eigen's default, is incompatible with `numpy`'s default row-major storage, and so you will not be able to pass numpy arrays to Eigen by reference without making one of two changes.

(Note that this does not apply to vectors (or column or row matrices): for such types the "row-major" and "column-major" distinction is meaningless).

The first approach is to change the use of `Eigen::Ref<MatrixType>` to the more general `Eigen::Ref<MatrixType, 0, Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic>>` (or similar type with a fully dynamic stride type in the third template argument). Since this is a rather cumbersome type, pybind11 provides a `py::EigenDRef<MatrixType>` type alias for your convenience (along with EigenDMap for the equivalent Map, and EigenDStride for just the stride type).

This type allows Eigen to map into any arbitrary storage order. This is not the default in Eigen for performance reasons: contiguous storage allows vectorization that cannot be done when storage is not known to be contiguous at compile time. The default `Eigen::Ref` stride type allows non-contiguous storage along the outer dimension (that is, the rows of a column-major matrix or columns of a row-major matrix), but not along the inner dimension.

This type, however, has the added benefit of also being able to map numpy array slices. For example, the following (contrived) example uses Eigen with a numpy slice to multiply by 2 all coefficients that are both on even rows (0, 2, 4, ...) and in columns 2, 5, or 8:

```
m.def("scale", [](py::EigenDRef<Eigen::MatrixXd> m, double c) { m *= c; });
```

```
# a = np.array(...)
scale_by_2(myarray[0::2, 2:9:3])
```

The second approach to avoid copying is more intrusive: rearranging the underlying data types to not run into the non-contiguous storage problem in the first place. In particular, that means using matrices with `Eigen::RowMajor` storage, where appropriate, such as:

```
using RowMatrixXd = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic,␣
↪Eigen::RowMajor>;
// Use RowMatrixXd instead of MatrixXd
```

Now bound functions accepting `Eigen::Ref<RowMatrixXd>` arguments will be callable with numpy's (default) arrays without involving a copying.

You can, alternatively, change the storage order that numpy arrays use by adding the `order='F'` option when creating an array:

```
myarray = np.array(source, order='F')
```

Such an object will be passable to a bound function accepting an `Eigen::Ref<MatrixXd>` (or similar column-major Eigen type).

One major caveat with this approach, however, is that it is not entirely as easy as simply flipping all Eigen or numpy usage from one to the other: some operations may alter the storage order of a numpy array. For example, `a2 = array.transpose()` results in `a2` being a view of `array` that references the same data, but in the opposite storage order!

While this approach allows fully optimized vectorized calculations in Eigen, it cannot be used with array slices, unlike the first approach.

When *returning* a matrix to Python (either a regular matrix, a reference via `Eigen::Ref<>`, or a map/block into a matrix), no special storage consideration is required: the created numpy array will have the required stride that allows numpy to properly interpret the array, whatever its storage order.

Failing rather than copying

The default behaviour when binding `Eigen::Ref<const MatrixType>` eigen references is to copy matrix values when passed a numpy array that does not conform to the element type of `MatrixType` or does not have a compatible stride layout. If you want to explicitly avoid copying in such a case, you should bind arguments using the `py::arg().noconvert()` annotation (as described in the *Non-converting arguments* documentation).

The following example shows an example of arguments that don't allow data copying to take place:

```
// The method and function to be bound:
class MyClass {
    // ...
    double some_method(const Eigen::Ref<const MatrixXd> &matrix) { /* ... */ }
};
float some_function(const Eigen::Ref<const MatrixXf> &big,
                    const Eigen::Ref<const MatrixXf> &small) {
    // ...
}

// The associated binding code:
using namespace pybind11::literals; // for "arg"_a
```

```
py::class_<MyClass>(m, "MyClass")
    // ... other class definitions
    .def("some_method", &MyClass::some_method, py::arg().nocopy());

m.def("some_function", &some_function,
    "big"_a.nocopy(), // <- Don't allow copying for this arg
    "small"_a         // <- This one can be copied if needed
);
```

With the above binding code, attempting to call the the `some_method(m)` method on a `MyClass` object, or attempting to call `some_function(m, m2)` will raise a `RuntimeError` rather than making a temporary copy of the array. It will, however, allow the `m2` argument to be copied into a temporary if necessary.

Note that explicitly specifying `.noconvert()` is not required for *mutable* Eigen references (e.g. `Eigen::Ref<MatrixXd>` without `const` on the `MatrixXd`): mutable references will never be called with a temporary copy.

Vectors versus column/row matrices

Eigen and numpy have fundamentally different notions of a vector. In Eigen, a vector is simply a matrix with the number of columns or rows set to 1 at compile time (for a column vector or row vector, respectively). Numpy, in contast, has comparable 2-dimensional 1xN and Nx1 arrays, but *also* has 1-dimensional arrays of size N.

When passing a 2-dimensional 1xN or Nx1 array to Eigen, the Eigen type must have matching dimensions: That is, you cannot pass a 2-dimensional Nx1 numpy array to an Eigen value expecting a row vector, or a 1xN numpy array as a column vector argument.

On the other hand, pybind11 allows you to pass 1-dimensional arrays of length N as Eigen parameters. If the Eigen type can hold a column vector of length N it will be passed as such a column vector. If not, but the Eigen type constraints will accept a row vector, it will be passed as a row vector. (The column vector takes precendence when both are supported, for example, when passing a 1D numpy array to a MatrixXd argument). Note that the type need not be expicitly a vector: it is permitted to pass a 1D numpy array of size 5 to an Eigen `Matrix<double, Dynamic, 5>`: you would end up with a 1x5 Eigen matrix. Passing the same to an `Eigen::MatrixXd` would result in a 5x1 Eigen matrix.

When returning an eigen vector to numpy, the conversion is ambiguous: a row vector of length 4 could be returned as either a 1D array of length 4, or as a 2D array of size 1x4. When encoutering such a situation, pybind11 compromises by considering the returned Eigen type: if it is a compile-time vector–that is, the type has either the number of rows or columns set to 1 at compile time–pybind11 converts to a 1D numpy array when returning the value. For instances that are a vector only at run-time (e.g. `MatrixXd`, `Matrix<float, Dynamic, 4>`), pybind11 returns the vector as a 2D array to numpy. If this isn't want you want, you can use `array.reshape(...)` to get a view of the same data in the desired dimensions.

See also:

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

Custom type casters

In very rare cases, applications may require custom type casters that cannot be expressed using the abstractions provided by pybind11, thus requiring raw Python C API calls. This is fairly advanced usage and should only be pursued by experts who are familiar with the intricacies of Python reference counting.

The following snippets demonstrate how this works for a very simple `inty` type that that should be convertible from Python types that provide a `__int__(self)` method.

```
struct inty { long long_value; };

void print(inty s) {
    std::cout << s.long_value << std::endl;
}
```

The following Python snippet demonstrates the intended usage from the Python side:

```
class A:
    def __int__(self):
        return 123

from example import print
print(A())
```

To register the necessary conversion routines, it is necessary to add a partial overload to the `pybind11::detail::type_caster<T>` template. Although this is an implementation detail, adding partial overloads to this type is explicitly allowed.

```
namespace pybind11 { namespace detail {
    template <> struct type_caster<inty> {
    public:
        /**
         * This macro establishes the name 'inty' in
         * function signatures and declares a local variable
         * 'value' of type inty
         */
        PYBIND11_TYPE_CASTER(inty, _("inty"));

        /**
         * Conversion part 1 (Python->C++): convert a PyObject into a inty
         * instance or return false upon failure. The second argument
         * indicates whether implicit conversions should be applied.
         */
        bool load(handle src, bool) {
            /* Extract PyObject from handle */
            PyObject *source = src.ptr();
            /* Try converting into a Python integer value */
            PyObject *tmp = PyNumber_Long(source);
            if (!tmp)
                return false;
            /* Now try to convert into a C++ int */
            value.long_value = PyLong_AsLong(tmp);
            Py_DECREF(tmp);
            /* Ensure return code was OK (to avoid out-of-range errors etc) */
            return !(value.long_value == -1 && !PyErr_Occurred());
        }

        /**
         * Conversion part 2 (C++ -> Python): convert an inty instance into
         * a Python object. The second and third arguments are used to
         * indicate the return value policy and parent object (for
         * ``return_value_policy::reference_internal``) and are generally
         * ignored by implicit casters.
         */
```

```
        static handle cast(inty src, return_value_policy /* policy */, handle /*␣
↪parent */) {
            return PyLong_FromLong(src.long_value);
        }
    };
}} // namespace pybind11::detail
```

Warning: When using custom type casters, it's important to declare them consistently in every compilation unit of the Python extension module. Otherwise, undefined behavior can ensue.

CHAPTER 11

Python C++ interface

pybind11 exposes Python types and functions using thin C++ wrappers, which makes it possible to conveniently call Python code from C++ without resorting to Python's C API.

Python types

Available wrappers

All major Python types are available as thin C++ wrapper classes. These can also be used as function parameters – see *Python objects as arguments*.

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

Casting back and forth

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

Calling Python functions

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>();
```

Keyword arguments are also supported. In Python, there is the usual call syntax:

```
def f(number, say, to):
    ...  # function code

f(1234, say="hello", to=some_instance)  # keyword call in Python
```

In C++, the same call can be made using:

```
using namespace pybind11::literals; // to bring in the `_a` literal
f(1234, "say"_a="hello", "to"_a=some_instance); // keyword call in C++
```

Unpacking of `*args` and `**kwargs` is also possible and can be mixed with other arguments:

```
// * unpacking
py::tuple args = py::make_tuple(1234, "hello", some_instance);
f(*args);

// ** unpacking
py::dict kwargs = py::dict("number"_a=1234, "say"_a="hello", "to"_a=some_instance);
f(**kwargs);

// mixed keywords, * and ** unpacking
py::tuple args = py::make_tuple(1234);
py::dict kwargs = py::dict("to"_a=some_instance);
f(*args, "say"_a="hello", **kwargs);
```

Generalized unpacking according to PEP448 is also supported:

```
py::dict kwargs1 = py::dict("number"_a=1234);
py::dict kwargs2 = py::dict("to"_a=some_instance);
f(**kwargs1, "say"_a="hello", **kwargs2);
```

See also:

The file `tests/test_pytypes.cpp` contains a complete example that demonstrates passing native Python types in more detail. The file `tests/test_callbacks.cpp` presents a few examples of calling Python functions from C++, including keywords arguments and unpacking.

NumPy

Buffer protocol

Python supports an extremely general and convenient approach for exchanging data between plugin libraries. Types can expose a buffer view[1], which provides fast direct access to the raw internal data representation. Suppose we want to bind the following simplistic Matrix class:

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

```
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", py::buffer_protocol())
   .def_buffer([](Matrix &m) -> py::buffer_info {
        return py::buffer_info(
            m.data(),                                /* Pointer to buffer */
            sizeof(float),                           /* Size of one scalar */
            py::format_descriptor<float>::format(), /* 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) }
        );
    });
```

Supporting the buffer protocol in a new type involves specifying the special `py::buffer_protocol()` tag in the `py::class_` constructor and calling the `def_buffer()` method with a lambda function that creates a `py::buffer_info` description record on demand describing a given matrix instance. The contents of `py::buffer_info` mirror the Python buffer protocol specification.

```
struct buffer_info {
    void *ptr;
    ssize_t itemsize;
    std::string format;
    ssize_t ndim;
    std::vector<ssize_t> shape;
    std::vector<ssize_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", py::buffer_protocol())
    .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>::format())
            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] / (py::ssize_t)sizeof(Scalar),
            info.strides[rowMajor ? 1 : 0] / (py::ssize_t)sizeof(Scalar));

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

        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 */
        py::format_descriptor<Scalar>::format(), /* Python struct-style format␣
↪descriptor */
        2,                                       /* Number of dimensions */
        { m.rows(), m.cols() },                  /* Buffer dimensions */
        { sizeof(Scalar) * (rowMajor ? m.cols() : 1),
          sizeof(Scalar) * (rowMajor ? 1 : m.rows()) }
                                                 /* Strides (in bytes) for each index␣
↪*/
    );
 })
```

For a much easier approach of binding Eigen types (although with some limitations), refer to the section on *Eigen*.

See also:

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

Arrays

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 `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 parameter, and it ensures that non-conforming arguments are converted into an array satisfying the specified requirements instead of trying the next function overload.

Structured types

In order for `py::array_t` to work with structured (record) types, we first need to register the memory layout of the type. This can be done via `PYBIND11_NUMPY_DTYPE` macro, called in the plugin definition code, which expects the type followed by field names:

```
struct A {
    int x;
    double y;
};

struct B {
    int z;
    A a;
};

// ...
PYBIND11_MODULE(test, m) {
    // ...

    PYBIND11_NUMPY_DTYPE(A, x, y);
    PYBIND11_NUMPY_DTYPE(B, z, a);
    /* now both A and B can be used as template arguments to py::array_t */
}
```

The structure should consist of fundamental arithmetic types, `std::complex`, previously registered substructures, and arrays of any of the above. Both C++ arrays and `std::array` are supported. While there is a static assertion to prevent many types of unsupported structures, it is still the user's responsibility to use only "plain" structures that can be safely manipulated as raw memory without violating invariants.

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).

Note: Only arithmetic, complex, and POD types passed by value or by `const &` reference are vectorized; all other arguments are passed through as-is. Functions taking rvalue reference arguments cannot be vectorized.

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");

    if (buf1.size != buf2.size)
        throw std::runtime_error("Input shapes must match");

    /* No pointer is passed, so NumPy will allocate the buffer */
    auto result = py::array_t<double>(buf1.size);

    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_MODULE(test, m) {
```

```
    m.def("add_arrays", &add_arrays, "Add two NumPy arrays");
}
```

See also:

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

Direct access

For performance reasons, particularly when dealing with very large arrays, it is often desirable to directly access array elements without internal checking of dimensions and bounds on every access when indices are known to be already valid. To avoid such checks, the `array` class and `array_t<T>` template class offer an unchecked proxy object that can be used for this unchecked access through the `unchecked<N>` and `mutable_unchecked<N>` methods, where `N` gives the required dimensionality of the array:

```
m.def("sum_3d", [](py::array_t<double> x) {
    auto r = x.unchecked<3>(); // x must have ndim = 3; can be non-writeable
    double sum = 0;
    for (ssize_t i = 0; i < r.shape(0); i++)
        for (ssize_t j = 0; j < r.shape(1); j++)
            for (ssize_t k = 0; k < r.shape(2); k++)
                sum += r(i, j, k);
    return sum;
});
m.def("increment_3d", [](py::array_t<double> x) {
    auto r = x.mutable_unchecked<3>(); // Will throw if ndim != 3 or flags.writeable␣
↪is false
    for (ssize_t i = 0; i < r.shape(0); i++)
        for (ssize_t j = 0; j < r.shape(1); j++)
            for (ssize_t k = 0; k < r.shape(2); k++)
                r(i, j, k) += 1.0;
}, py::arg().noconvert());
```

To obtain the proxy from an `array` object, you must specify both the data type and number of dimensions as template arguments, such as `auto r = myarray.mutable_unchecked<float, 2>()`.

If the number of dimensions is not known at compile time, you can omit the dimensions template parameter (i.e. calling `arr_t.unchecked()` or `arr.unchecked<T>()`. This will give you a proxy object that works in the same way, but results in less optimizable code and thus a small efficiency loss in tight loops.

Note that the returned proxy object directly references the array's data, and only reads its shape, strides, and writeable flag when constructed. You must take care to ensure that the referenced array is not destroyed or reshaped for the duration of the returned object, typically by limiting the scope of the returned instance.

The returned proxy object supports some of the same methods as `py::array` so that it can be used as a drop-in replacement for some existing, index-checked uses of `py::array`:

- `r.ndim()` returns the number of dimensions
- `r.data(1, 2, ...)` and `r.mutable_data(1, 2, ...)`` returns a pointer to the `const T` or `T` data, respectively, at the given indices. The latter is only available to proxies obtained via `a.mutable_unchecked()`.
- `itemsize()` returns the size of an item in bytes, i.e. `sizeof(T)`.
- `ndim()` returns the number of dimensions.
- `shape(n)` returns the size of dimension `n`

- `size()` returns the total number of elements (i.e. the product of the shapes).
- `nbytes()` returns the number of bytes used by the referenced elements (i.e. `itemsize()` times `size()`).

See also:

The file `tests/test_numpy_array.cpp` contains additional examples demonstrating the use of this feature.

Utilities

Using Python's print function in C++

The usual way to write output in C++ is using `std::cout` while in Python one would use `print`. Since these methods use different buffers, mixing them can lead to output order issues. To resolve this, pybind11 modules can use the `py::print()` function which writes to Python's `sys.stdout` for consistency.

Python's `print` function is replicated in the C++ API including optional keyword arguments `sep`, `end`, `file`, `flush`. Everything works as expected in Python:

```
py::print(1, 2.0, "three"); // 1 2.0 three
py::print(1, 2.0, "three", "sep"_a="-"); // 1-2.0-three

auto args = py::make_tuple("unpacked", true);
py::print("->", *args, "end"_a="<-"); // -> unpacked True <-
```

Evaluating Python expressions from strings and files

pybind11 provides the `eval`, `exec` and `eval_file` functions to evaluate Python expressions and statements. The following example illustrates how they can be used.

```
// At beginning of file
#include <pybind11/eval.h>

...

// Evaluate in scope of main module
py::object scope = py::module::import("__main__").attr("__dict__");

// Evaluate an isolated expression
int result = py::eval("my_variable + 10", scope).cast<int>();

// Evaluate a sequence of statements
py::exec(
    "print('Hello')\n"
    "print('world!');",
    scope);

// Evaluate the statements in an separate Python file on disk
py::eval_file("script.py", scope);
```

C++11 raw string literals are also supported and quite handy for this purpose. The only requirement is that the first statement must be on a new line following the raw string delimiter `R"(`, ensuring all lines have common leading indent:

```
py::exec(R"(
    x = get_answer()
    if x == 42:
        print('Hello World!')
    else:
        print('Bye!')
    )", scope
);
```

Note: `eval` and `eval_file` accept a template parameter that describes how the string/file should be interpreted. Possible choices include `eval_expr` (isolated expression), `eval_single_statement` (a single statement, return value is always `none`), and `eval_statements` (sequence of statements, return value is always `none`). `eval` defaults to `eval_expr`, `eval_file` defaults to `eval_statements` and `exec` is just a shortcut for `eval<eval_statements>`.

CHAPTER 12

Embedding the interpreter

While pybind11 is mainly focused on extending Python using C++, it's also possible to do the reverse: embed the Python interpreter into a C++ program. All of the other documentation pages still apply here, so refer to them for general pybind11 usage. This section will cover a few extra things required for embedding.

Getting started

A basic executable with an embedded interpreter can be created with just a few lines of CMake and the `pybind11::embed` target, as shown below. For more information, see *Build systems*.

```
cmake_minimum_required(VERSION 3.0)
project(example)

find_package(pybind11 REQUIRED)  # or `add_subdirectory(pybind11)`

add_executable(example main.cpp)
target_link_libraries(example PRIVATE pybind11::embed)
```

The essential structure of the `main.cpp` file looks like this:

```
#include <pybind11/embed.h> // everything needed for embedding
namespace py = pybind11;

int main() {
    py::scoped_interpreter guard{}; // start the interpreter and keep it alive

    py::print("Hello, World!"); // use the Python API
}
```

The interpreter must be initialized before using any Python API, which includes all the functions and classes in pybind11. The RAII guard class *scoped_interpreter* takes care of the interpreter lifetime. After the guard is destroyed, the interpreter shuts down and clears its memory. No Python functions can be called after this.

Executing Python code

There are a few different ways to run Python code. One option is to use `eval`, `exec` or `eval_file`, as explained in *Evaluating Python expressions from strings and files*. Here is a quick example in the context of an executable with an embedded interpreter:

```
#include <pybind11/embed.h>
namespace py = pybind11;

int main() {
    py::scoped_interpreter guard{};

    py::exec(R"(
        kwargs = dict(name="World", number=42)
        message = "Hello, {name}! The answer is {number}".format(**kwargs)
        print(message)
    )");
}
```

Alternatively, similar results can be achieved using pybind11's API (see *Python C++ interface* for more details).

```
#include <pybind11/embed.h>
namespace py = pybind11;
using namespace py::literals;

int main() {
    py::scoped_interpreter guard{};

    auto kwargs = py::dict("name"_a="World", "number"_a=42);
    auto message = "Hello, {name}! The answer is {number}"_s.format(**kwargs);
    py::print(message);
}
```

The two approaches can also be combined:

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

namespace py = pybind11;
using namespace py::literals;

int main() {
    py::scoped_interpreter guard{};

    auto locals = py::dict("name"_a="World", "number"_a=42);
    py::exec(R"(
        message = "Hello, {name}! The answer is {number}".format(**locals())
    )", py::globals(), locals);

    auto message = locals["message"].cast<std::string>();
    std::cout << message;
}
```

Importing modules

Python modules can be imported using *module::import()*:

```
py::module sys = py::module::import("sys");
py::print(sys.attr("path"));
```

For convenience, the current working directory is included in `sys.path` when embedding the interpreter. This makes it easy to import local Python files:

```
"""calc.py located in the working directory"""

def add(i, j):
    return i + j
```

```
py::module calc = py::module::import("calc");
py::object result = calc.attr("add")(1, 2);
int n = result.cast<int>();
assert(n == 3);
```

Adding embedded modules

Embedded binary modules can be added using the `PYBIND11_EMBEDDED_MODULE` macro. Note that the definition must be placed at global scope. They can be imported like any other module.

```
#include <pybind11/embed.h>
namespace py = pybind11;

PYBIND11_EMBEDDED_MODULE(fast_calc, m) {
    // `m` is a `py::module` which is used to bind functions and classes
    m.def("add", [](int i, int j) {
        return i + j;
    });
}

int main() {
    py::scoped_interpreter guard{};

    auto fast_calc = py::module::import("fast_calc");
    auto result = fast_calc.attr("add")(1, 2).cast<int>();
    assert(result == 3);
}
```

Unlike extension modules where only a single binary module can be created, on the embedded side an unlimited number of modules can be added using multiple `PYBIND11_EMBEDDED_MODULE` definitions (as long as they have unique names).

These modules are added to Python's list of builtins, so they can also be imported in pure Python files loaded by the interpreter. Everything interacts naturally:

```
"""py_module.py located in the working directory"""
import cpp_module

a = cpp_module.a
b = a + 1
```

```
#include <pybind11/embed.h>
namespace py = pybind11;

PYBIND11_EMBEDDED_MODULE(cpp_module, m) {
    m.attr("a") = 1
}

int main() {
    py::scoped_interpreter guard{};

    auto py_module = py::module::import("py_module");

    auto locals = py::dict("fmt"_a="{} + {} = {}", **py_module.attr("__dict__"));
    assert(locals["a"].cast<int>() == 1);
    assert(locals["b"].cast<int>() == 2);

    py::exec(R"(
        c = a + b
        message = fmt.format(a, b, c)
    )", py::globals(), locals);

    assert(locals["c"].cast<int>() == 3);
    assert(locals["message"].cast<std::string>() == "1 + 2 = 3");
}
```

Interpreter lifetime

The Python interpreter shuts down when *scoped_interpreter* is destroyed. After this, creating a new instance will restart the interpreter. Alternatively, the *initialize_interpreter()* / *finalize_interpreter()* pair of functions can be used to directly set the state at any time.

Modules created with pybind11 can be safely re-initialized after the interpreter has been restarted. However, this may not apply to third-party extension modules. The issue is that Python itself cannot completely unload extension modules and there are several caveats with regard to interpreter restarting. In short, not all memory may be freed, either due to Python reference cycles or user-created global data. All the details can be found in the CPython documentation.

> **Warning:** Creating two concurrent *scoped_interpreter* guards is a fatal error. So is calling *initialize_interpreter()* for a second time after the interpreter has already been initialized.
>
> Do not use the raw CPython API functions `Py_Initialize` and `Py_Finalize` as these do not properly handle the lifetime of pybind11's internal data.

Sub-interpreter support

Creating multiple copies of *scoped_interpreter* is not possible because it represents the main Python interpreter. Sub-interpreters are something different and they do permit the existence of multiple interpreters. This is an advanced feature of the CPython API and should be handled with care. pybind11 does not currently offer a C++ interface for sub-interpreters, so refer to the CPython documentation for all the details regarding this feature.

We'll just mention a couple of caveats the sub-interpreters support in pybind11:

1. Sub-interpreters will not receive independent copies of embedded modules. Instead, these are shared and modifications in one interpreter may be reflected in another.
2. Managing multiple threads, multiple interpreters and the GIL can be challenging and there are several caveats here, even within the pure CPython API (please refer to the Python docs for details). As for pybind11, keep in mind that `gil_scoped_release` and `gil_scoped_acquire` do not take sub-interpreters into account.

CHAPTER 13

Miscellaneous

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 beginning of the next parameter. Use a `typedef` to bind the template to another name and use it in the macro to avoid this problem.

Global Interpreter Lock (GIL)

When calling a C++ function from Python, the GIL is always held. 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 *Overriding virtual functions in Python* 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_MODULE(example, m) {
    py::class_<Animal, PyAnimal> animal(m, "Animal");
    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);
    });
}
```

The `call_go` wrapper can also be simplified using the *`call_guard`* policy (see *Additional call policies*) which yields the same result:

```
m.def("call_go", &call_go, py::call_guard<py::gil_scoped_release>());
```

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

Please refer to the supplemental example for details.

See also:

The file `tests/test_sequences_and_iterators.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.

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 and automatic upcasting*.

```
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, you can specify the base class as a template parameter option to `class_`, which performs an automated lookup of the corresponding Python type. Like the above code, 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, Pet>(m, "Dog")
    .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 {
    ...
};
```

Note also that it is possible (although would rarely be required) to share arbitrary C++ objects between extension modules at runtime. Internal library data is shared between modules using capsule machinery[1] which can be also utilized for storing, modifying and accessing user-defined data. Note that an extension module will "see" other extensions' data if and only if they were built with the same pybind11 version. Consider the following example:

```
auto data = (MyData *) py::get_shared_data("mydata");
if (!data)
    data = (MyData *) py::set_shared_data("mydata", new MyData(42));
```

If the above snippet was used in several separately compiled extension modules, the first one to be imported would create a `MyData` instance and associate a `"mydata"` key with a pointer to it. Extensions that are imported later would be then able to access the data behind the same pointer.

Module Destructors

pybind11 does not provide an explicit mechanism to invoke cleanup code at module destruction time. In rare cases where such functionality is required, it is possible to emulate it using Python capsules with a destruction callback.

```
auto cleanup_callback = []() {
    // perform cleanup here -- this function is called with the GIL held
```

[1] https://docs.python.org/3/extending/extending.html#using-capsules

```
};

m.add_object("_cleanup", py::capsule(cleanup_callback));
```

Generating documentation using Sphinx

Sphinx[2] 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[3] 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.:

```
// 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");
```

By default, pybind11 automatically generates and prepends a signature to the docstring of a function registered with `module::def()` and `class_::def()`. Sometimes this behavior is not desirable, because you want to provide your own signature or remove the docstring completely to exclude the function from the Sphinx documentation. The class `options` allows you to selectively suppress auto-generated signatures:

```
PYBIND11_MODULE(example, m) {
    py::options options;
    options.disable_function_signatures();

    m.def("add", [](int a, int b) { return a + b; }, "A function which adds two␣
↪numbers");
}
```

Note that changes to the settings affect only function bindings created during the lifetime of the `options` instance. When it goes out of scope at the end of the module's init function, the default settings are restored to prevent unwanted side effects.

[2] http://www.sphinx-doc.org
[3] http://github.com/pybind/python_example

CHAPTER 14

Frequently asked questions

“ImportError: dynamic module does not define init function”

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).

“Symbol not found: `__Py_ZeroStruct` / `_PyInstanceMethod_Type`“

See the first answer.

“SystemError: dynamic module not initialized properly”

See the first answer.

The Python interpreter immediately crashes when importing my module

See the first answer.

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> .
```

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); });
```

How can I reduce the build time?

It's good practice to split binding code over multiple files, as in the following example:

`example.cpp`:

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

PYBIND11_MODULE(example, m) {
    init_ex1(m);
    init_ex2(m);
    /* ... */
}
```

`ex1.cpp`:

```
void init_ex1(py::module &m) {
    m.def("add", [](int a, int b) { return a + b; });
}
```

ex2.cpp:

```
void init_ex1(py::module &m) {
    m.def("sub", [](int a, int b) { return a - b; });
}
```

python:

```
>>> import example
>>> example.add(1, 2)
3
>>> example.sub(1, 1)
0
```

As shown above, the various `init_ex` functions should be contained in separate files that can be compiled independently from one another, and then linked together into the same final shared object. 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 definition is changed, only a subset of the binding code will generally need to be recompiled.

"recursive template instantiation exceeded maximum depth of 256"

If you receive an error about excessive recursive template evaluation, try specifying a larger value, e.g. `-ftemplate-depth=1024` on GCC/Clang. The culprit is generally the generation of function signatures at compile time using C++14 template metaprogramming.

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 instructions 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
|
|      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.

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.

CHAPTER 15

Benchmark

The following is the result of a synthetic benchmark comparing both compilation time and module size of pybind11 against Boost.Python. A detailed report about a Boost.Python to pybind11 conversion of a real project is available here:[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_MODULE(example, m) {
    ...
    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)
```

[1] http://graylab.jhu.edu/RosettaCon2016/PyRosetta-4.pdf

```
    ...
}
```

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

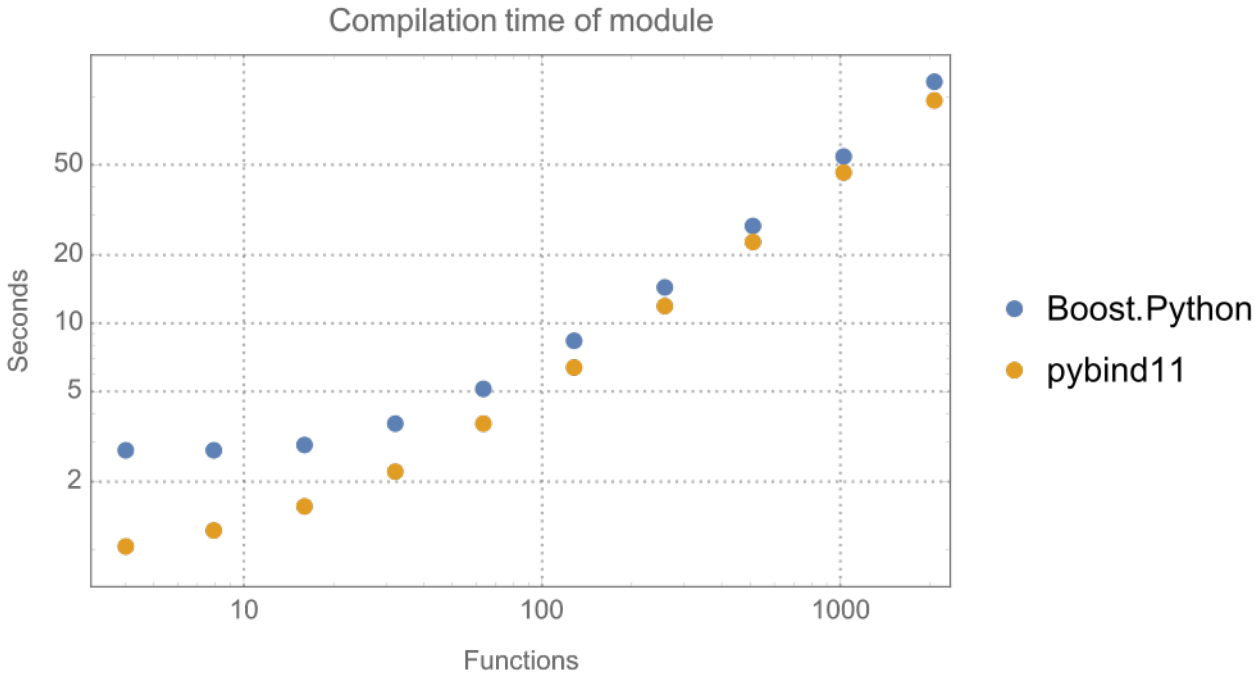
```
Apple LLVM version 7.0.2 (clang-700.1.81)
```

and the following compilation flags

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

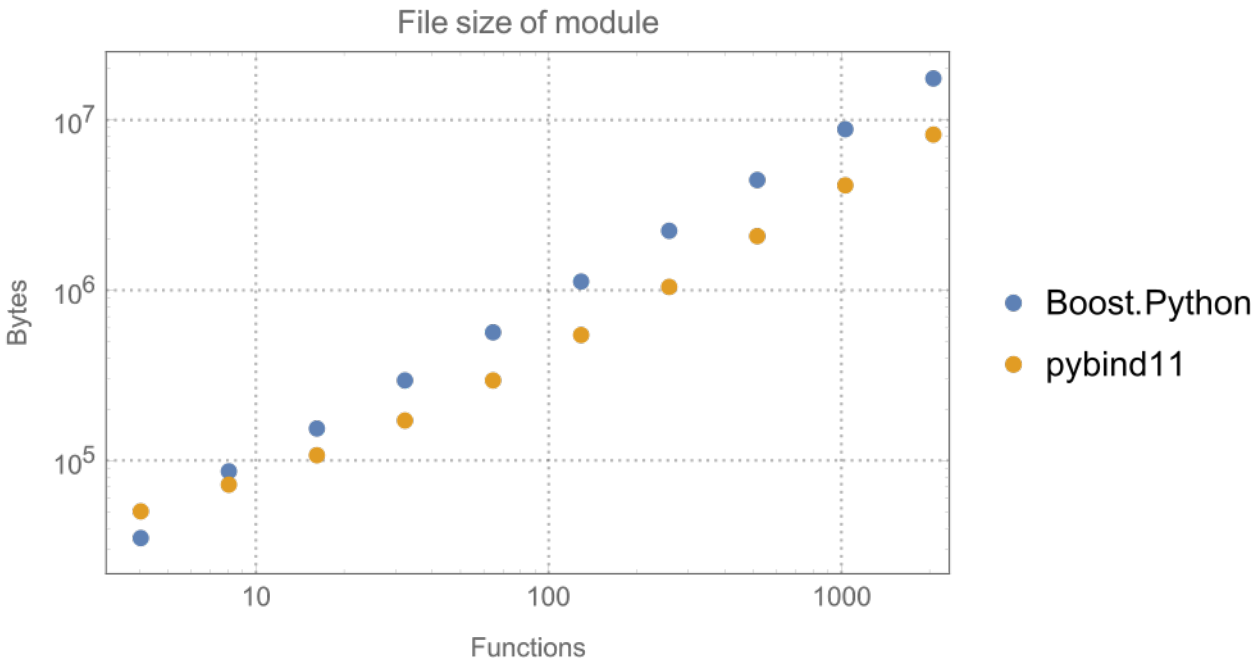
Compilation time

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).



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 16

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.
- The NumPy interface `pybind11::array` greatly simplifies accessing numerical data from C++ (and vice versa), but it's not a full-blown array class like `Eigen::Array` or `boost.multi_array`.

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.

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 17

Reference

Macros

PYBIND11_MODULE(name, variable)

This macro creates the entry point that will be invoked when the Python interpreter imports an extension module. The module name is given as the fist argument and it should not be in quotes. The second macro argument defines a variable of type `py::module` which can be used to initialize the module.

```
PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example module";

    // Add bindings here
    m.def("foo", []() {
        return "Hello, World!";
    });
}
```

Convenience classes for arbitrary Python types

Common member functions

template <typename *Derived*>
class **object_api**

A mixin class which adds common functions to *handle*, *object* and various accessors. The only requirement for `Derived` is to implement `PyObject *Derived::ptr() const`.

Inherits from pyobject_tag

Public Functions

iterator **begin**() **const**

Return an iterator equivalent to calling `iter()` in Python. The object must be a collection which supports the iteration protocol.

iterator **end**() **const**

Return a sentinel which ends iteration.

item_accessor **operator[]**(*handle key*) **const**

Return an internal functor to invoke the object's sequence protocol. Casting the returned `detail::item_accessor` instance to a *handle* or *object* subclass causes a corresponding call to `__getitem__`. Assigning a *handle* or *object* subclass causes a call to `__setitem__`.

item_accessor **operator[]**(**const** char **key*) **const**

See above (the only difference is that they key is provided as a string literal)

obj_attr_accessor **attr**(*handle key*) **const**

Return an internal functor to access the object's attributes. Casting the returned `detail::obj_attr_accessor` instance to a *handle* or *object* subclass causes a corresponding call to `getattr`. Assigning a *handle* or *object* subclass causes a call to `setattr`.

str_attr_accessor **attr**(**const** char **key*) **const**

See above (the only difference is that they key is provided as a string literal)

args_proxy **operator***() **const**

Matches * unpacking in Python, e.g. to unpack arguments out of a `tuple` or `list` for a function call. Applying another * to the result yields ** unpacking, e.g. to unpack a dict as function keyword arguments. See *Calling Python functions*.

template <**typename** T>
bool **contains**(T &&*item*) **const**

Check if the given item is contained within this object, i.e. `item in obj`.

template <return_value_policy *policy* = return_value_policy::automatic_reference, typename... *Args*>
object **operator()**(Args&&... *args*) **const**

Assuming the Python object is a function or implements the `__call__` protocol, `operator()` 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.

bool **is**(*object_api* **const** &*other*) **const**

Equivalent to `obj is other` in Python.

bool **is_none**() **const**

Equivalent to `obj is None` in Python.

str_attr_accessor **doc**() **const**

Get or set the object's docstring, i.e. `obj.__doc__`.

int **ref_count**() **const**

Return the object's current reference count.

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

Without reference counting

class handle

Holds a reference to a Python object (no reference counting)

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.

Inherits from detail::object_api< handle >

Subclassed by args_proxy, kwargs_proxy, *object*

Public Functions

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

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

PyObject ***ptr**() **const**
Return the underlying `PyObject *` pointer.

const *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* &**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.

template <typename T>
T **cast**() **const**

Attempt to cast the Python object into the given C++ type. A `cast_error` will be throw upon failure.

operator bool() **const**
Return `true` when the `handle` wraps a valid Python object.

bool **operator==**(**const** *handle* &*h*) **const**

Deprecated: Check that the underlying pointers are the same. Equivalent to `obj1 is obj2` in Python.

With reference counting

class object

Holds a reference to a Python object (with reference counting)

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.

Inherits from *handle*

Subclassed by *bool_*, *buffer*, *bytes*, *capsule*, *dict*, dtype, exception< type >, *float_*, *function*, generic_type, *int_*, *iterable*, *iterator*, *list*, *memoryview*, *module*, *none*, *sequence*, *set*, *slice*, *str*, *tuple*, *weakref*

Public Functions

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

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

~object ()
Destructor; automatically calls *handle::dec_ref()*

handle **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.

template <typename T>
T **reinterpret_borrow** (*handle* *h*)

Declare that a *handle* or `PyObject *` is a certain type and borrow the reference. The target type `T` must be *object* or one of its derived classes. The function doesn't do any conversions or checks. It's up to the user to make sure that the target type is correct.

```
PyObject *p = PyList_GetItem(obj, index);
py::object o = reinterpret_borrow<py::object>(p);
// or
py::tuple t = reinterpret_borrow<py::tuple>(p); // <-- `p` must be already be a
↪`tuple`
```

template <typename T>
T **reinterpret_steal** (*handle* *h*)

Like *reinterpret_borrow()*, but steals the reference.

```
PyObject *p = PyObject_Str(obj);
py::str s = reinterpret_steal<py::str>(p); // <-- `p` must be already be
↪a `str`
```

Convenience classes for specific Python types

class module
Wrapper for Python extension modules.

Inherits from *object*

Public Functions

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

template <typename Func, typename... *Extra*>
module &**def** (**const** char *_name__, Func &&_f_, **const** 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 def or class_*.

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.

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

Public Static Functions

static *module* **import** (**const** char *_name_)
Import and return a module or throws *error_already_set*.

group **pytypes**

Functions

template <typename Unsigned>
Unsigned **as_unsigned** (PyObject *_o_)

bytes (**const** pybind11::str &_s_)

str (**const** *bytes* &_b_)

class iterator
#include <pytypes.h>

Wraps a Python iterator so that it can also be used as a C++ input iterator

Caveat: copying an iterator does not (and cannot) clone the internal state of the Python iterable. This also applies to the post-increment operator. This iterator should only be used to retrieve the current value using `operator*()`.

Inherits from *object*

Public Static Functions

static *iterator* **sentinel** ()

The value which marks the end of the iteration. `it == iterator::sentinel()` is equivalent to catching `StopIteration` in Python.

```
void foo(py::iterator it) {
    while (it != py::iterator::sentinel()) {
       // use `*it`
       ++it;
    }
}
```

class iterable
Inherits from *object*

class str
Inherits from *object*

Public Functions

str(*handle h*)

Return a string representation of the object. This is analogous to the `str()` function in Python.

class bytes
Inherits from *object*

class none
Inherits from *object*

class bool_
Inherits from *object*

class int_
Inherits from *object*

class float_
Inherits from *object*

class weakref
Inherits from *object*

class slice
Inherits from *object*

class capsule
Inherits from *object*

class tuple
Inherits from *object*

Subclassed by *args*

class dict
Inherits from *object*

Subclassed by *kwargs*

class sequence
Inherits from *object*

class list
Inherits from *object*

class args
Inherits from *tuple*

class `kwargs`
Inherits from *dict*

class `set`
Inherits from *object*

class `function`
Inherits from *object*

Subclassed by cpp_function

class `buffer`
Inherits from *object*

Subclassed by array

class `memoryview`
Inherits from *object*

Passing extra arguments to `def` or `class_`

group `annotations`

struct `is_method`
#include <*attr.h*> Annotation for methods.

struct `is_operator`
#include <*attr.h*> Annotation for operators.

struct `scope`
#include <*attr.h*> Annotation for parent scope.

struct `doc`
#include <*attr.h*> Annotation for documentation.

struct `name`
#include <*attr.h*> Annotation for function names.

struct `sibling`
#include <*attr.h*> Annotation indicating that a function is an overload associated with a given "sibling".
template <**typename** T>
struct `base`
#include <*attr.h*> Annotation indicating that a class derives from another given type.
template <size_t *Nurse*, size_t *Patient*>
struct `keep_alive`
#include <*attr.h*> Keep patient alive while nurse lives.

struct `multiple_inheritance`
#include <*attr.h*> Annotation indicating that a class is involved in a multiple inheritance relationship.

struct `dynamic_attr`
#include <*attr.h*> Annotation which enables dynamic attributes, i.e. adds `__dict__` to a class.

struct `buffer_protocol`
#include <*attr.h*> Annotation which enables the buffer protocol for a type.

struct `metaclass`
#include <*attr.h*> Annotation which requests that a special metaclass is created for a type.

Public Functions

metaclass (*handle value*)
Override pybind11's default metaclass.

struct arithmetic
#include <*attr.h*> Annotation to mark enums as an arithmetic type.

template <typename... *Ts*>
struct call_guard
#include <*attr.h*>

A call policy which places one or more guard variables (`Ts...`) around the function call.

For example, this definition:

```
m.def("foo", foo, py::call_guard<T>());
```

is equivalent to the following pseudocode:

```
m.def("foo", [](args...) {
    T scope_guard;
    return foo(args...); // forwarded arguments
});
```

struct arg
#include <*cast.h*> Annotation for arguments

Subclassed by *arg_v*

Public Functions

constexpr arg (**const** char **name* = nullptr)
Constructs an argument with the name of the argument; if null or omitted, this is a positional argument.

template <typename T>
arg_v **operator=** (T &&*value*) **const**
Assign a value to this argument.

arg &**noconvert** (bool *flag* = true)
Indicate that the type should not be converted in the type caster.

arg &**none** (bool *flag* = true)
Indicates that the argument should/shouldn't allow None (e.g. for nullable pointer args)

Public Members

const char ***name**
If non-null, this is a named kwargs argument.

bool **flag_noconvert**
If set, do not allow conversion (requires a supporting type caster!)

bool **flag_none**
If set (the default), allow None to be passed to this argument.

struct `arg_v`

#include <cast.h> Annotation for arguments with values

Inherits from *arg*

Public Functions

template <typename T>
`arg_v`(**const** char **name*, T &&*x*, **const** char **descr* = nullptr)
Direct construction with name, default, and description.

template <typename T>
`arg_v`(**const** *arg* &*base*, T &&*x*, **const** char **descr* = nullptr)
Called internally when invoking `py::arg("a") = value`

arg_v &**`noconvert`**(bool *flag* = true)
Same as *`arg::noconvert()`*, but returns *this as *arg_v*&, not arg&.

arg_v &**`none`**(bool *flag* = true)
Same as `arg::nonone()`, but returns *this as *arg_v*&, not arg&.

Public Members

object **`value`**
The default value.

const char ***`descr`**
The (optional) description of the default value.

std::string **`type`**
The C++ type name of the default value (only available when compiled in debug mode)

Embedding the interpreter

`PYBIND11_EMBEDDED_MODULE`(name, variable)
Add a new module to the table of builtins for the interpreter. Must be defined in global scope. The first macro parameter is the name of the module (without quotes). The second parameter is the variable which will be used as the interface to add functions and classes to the module.

```
PYBIND11_EMBEDDED_MODULE(example, m) {
    // ... initialize functions and classes here
    m.def("foo", []() {
        return "Hello, World!";
    });
}
```

void **`initialize_interpreter`**(bool *init_signal_handlers* = true)

Initialize the Python interpreter. No other pybind11 or CPython API functions can be called before this is done; with the exception of `PYBIND11_EMBEDDED_MODULE`. The optional parameter can be used to skip the registration of signal handlers (see the Python documentation for details). Calling this function again after the interpreter has already been initialized is a fatal error.

void **finalize_interpreter**()

Shut down the Python interpreter. No pybind11 or CPython API functions can be called after this. In addition, pybind11 objects must not outlive the interpreter:

```
{ // BAD
    py::initialize_interpreter();
    auto hello = py::str("Hello, World!");
    py::finalize_interpreter();
} // <-- BOOM, hello's destructor is called after interpreter shutdown

{ // GOOD
    py::initialize_interpreter();
    { // scoped
        auto hello = py::str("Hello, World!");
    } // <-- OK, hello is cleaned up properly
    py::finalize_interpreter();
}

{ // BETTER
    py::scoped_interpreter guard{};
    auto hello = py::str("Hello, World!");
}
```

Warning: The interpreter can be restarted by calling *initialize_interpreter()* again. Modules created using pybind11 can be safely re-initialized. However, Python itself cannot completely unload binary extension modules and there are several caveats with regard to interpreter restarting. All the details can be found in the CPython documentation. In short, not all interpreter memory may be freed, either due to reference cycles or user-created global data.

class scoped_interpreter

Scope guard version of *initialize_interpreter()* and *finalize_interpreter()*. This a move-only guard and only a single instance can exist.

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

int main() {
    py::scoped_interpreter guard{};
    py::print(Hello, World!);
} // <-- interpreter shutdown
```

Python build-in functions

group **python_builtins**

Unless stated otherwise, the following C++ functions behave the same as their Python counterparts.

Functions

dict **globals**()

Return a dictionary representing the global variables in the current execution frame, or `__main__.__dict__` if there is no frame (usually when the interpreter is embedded).

template <typename T, detail::enable_if_t< std::is_base_of< *object*, T >::value, int > = 0>
bool **isinstance**(*handle obj*)

Return true if `obj` is an instance of `T`. Type `T` must be a subclass of *`object`* or a class which was exposed to Python as `py::class_<T>`.

bool **isinstance**(*handle obj*, *handle type*)
Return true if `obj` is an instance of the `type`.

bool **hasattr**(*handle obj*, *handle name*)

bool **hasattr**(*handle obj*, **const** char **name*)

object **getattr**(*handle obj*, *handle name*)

object **getattr**(*handle obj*, **const** char **name*)

object **getattr**(*handle obj*, *handle name*, *handle default_*)

object **getattr**(*handle obj*, **const** char **name*, *handle default_*)

void **setattr**(*handle obj*, *handle name*, *handle value*)

void **setattr**(*handle obj*, **const** char **name*, *handle value*)

size_t **len**(*handle h*)

str **repr**(*handle h*)

iterator **iter**(*handle obj*)

Exceptions

class error_already_set
Fetch and hold an error which was already set in Python.

Inherits from runtime_error

Public Functions

void **restore**()
Give the error back to Python.

void **clear**()
Clear the held Python error state (the C++ `what()` message remains intact)

bool **matches**(PyObject **ex*) **const**
Check if the trapped exception matches a given Python exception class.

class builtin_exception
C++ bindings of builtin Python exceptions.

Inherits from runtime_error

Subclassed by cast_error, index_error, key_error, reference_cast_error, stop_iteration, type_error, value_error

Public Functions

virtual void **set_error**() **const** = 0
Set the error using the Python C API.

Literals

namespace **literals**

Functions

constexpr *arg* **operator""_a**(**const** char *name, size_t)

String literal version of *arg*

str **operator""_s**(**const** char *s, size_t *size*)

String literal version of *str*

Bibliography

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

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

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

[binder] http://cppbinder.readthedocs.io/en/latest/about.html

Index