Karlsruhe Institute of Technology

Pybind11 Tutorial

October 26, 2022

Tobias Heuer

This page intentionally left blank

Karlsruhe Institute of Technology

Introduction

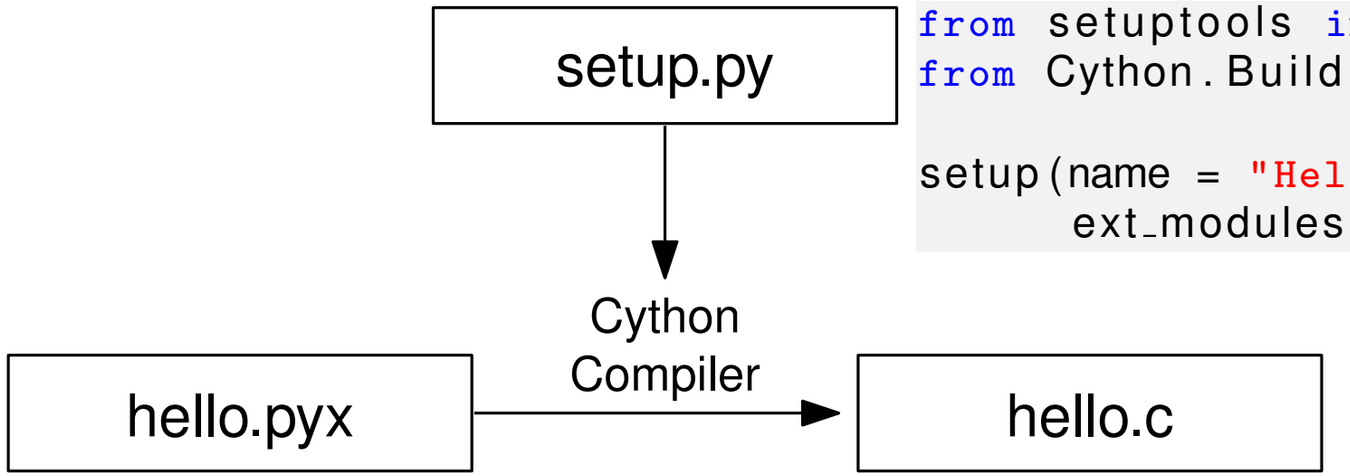
Cython Toolchain

Cython Toolchain

hello.pyx

```
def say_hello():
  print("Hello World")
```

Cython Toolchain



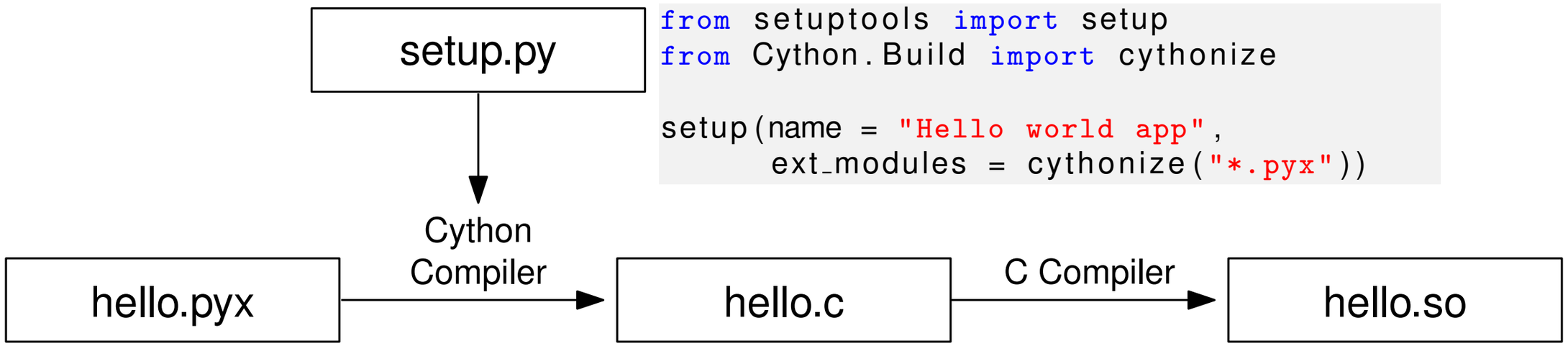

```python
from setuptools import setup
from Cython.Build import cythonize

setup(name = "Hello world app",
      ext_modules = cythonize("*.pyx"))
```

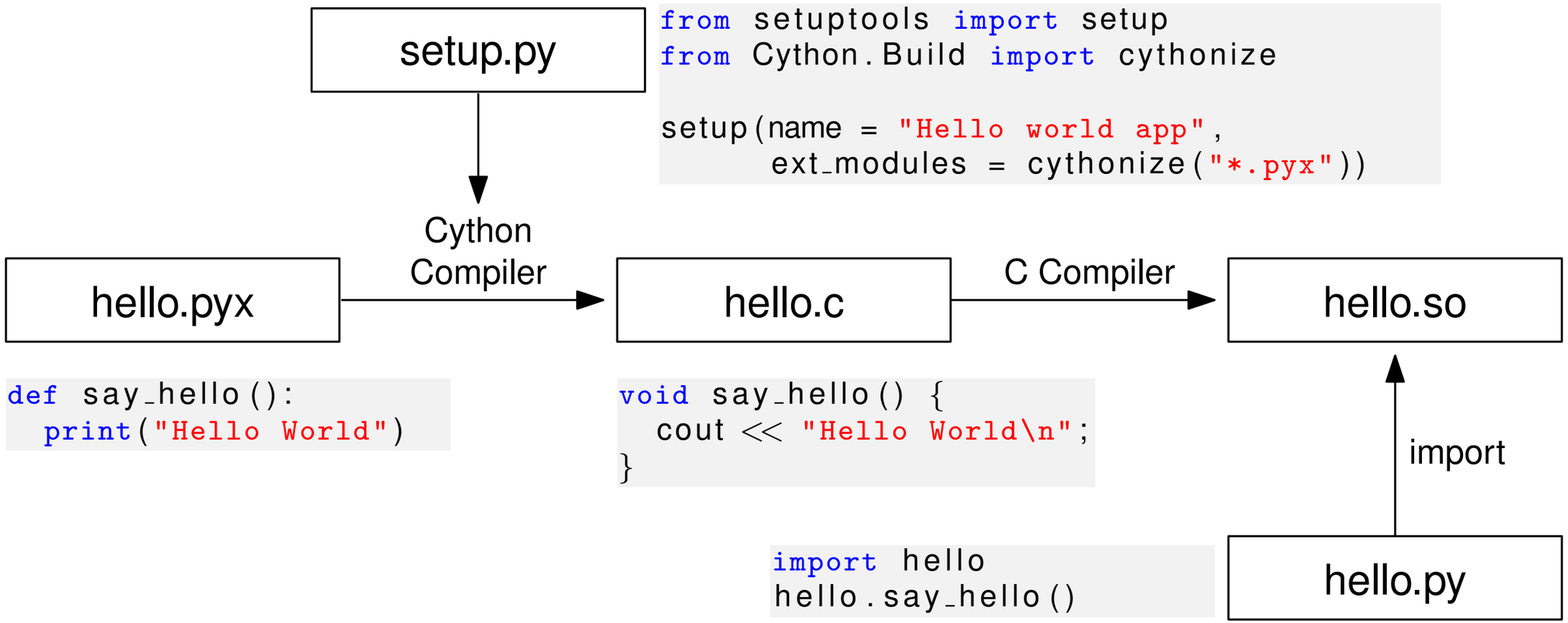
```python
def say_hello():
  print("Hello World")
```

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```

Cython Toolchain



Cython Toolchain



Pybind11 Toolchain

Pybind11 Toolchain

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```

hello.h

Pybind11 Toolchain

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```

hello.h

module.cpp

```cpp
#include "hello.h"

PYBIND11_MODULE(hello, m) {
  m.def("say_hello", &say_hello);
}
```

Pybind11 Toolchain

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```

hello.h

Creates a function called when an `import` statement is issued from Python

Module Name

Member of type `py::module_` which is the main interface for creating bindings

```cpp
#include "hello.h"

PYBIND11_MODULE(hello, m) {
  m.def("say_hello", &say_hello);
}
```

Pybind11 Toolchain

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```

hello.h

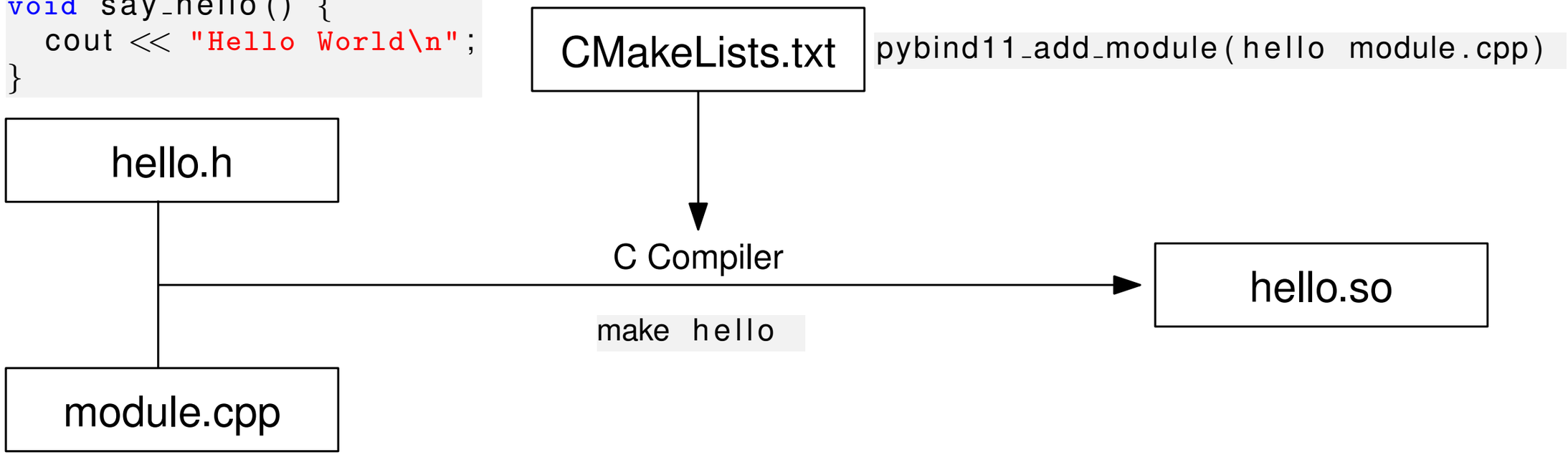
module.cpp

```cpp
#include "hello.h"

PYBIND11_MODULE(hello, m) {
  m.def("say_hello", &say_hello);
}
```

Pybind11 Toolchain

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```




```cpp
#include "hello.h"

PYBIND11_MODULE(hello, m) {
  m.def("say_hello", &say_hello);
}
```

Pybind11 Toolchain

```cpp
void say_hello() {
  cout << "Hello World\n";
}
```

CMakeLists.txt

```
pybind11_add_module(hello module.cpp)
```

hello.h

C Compiler

```
make hello
```

hello.so

module.cpp

```cpp
#include "hello.h"

PYBIND11_MODULE(hello, m) {
  m.def("say_hello", &say_hello);
}
```

import

```python
import hello
hello.say_hello()
```

hello.py

Karlsruhe Institute of Technology

Get Started

Pybind11 – Get Started

vector_int.h

```
class VectorInt {

 public:
  VectorInt();

  void push_back(const int val);
  int get(const size_t idx) const;
  size_t size() const;
  bool is_empty() const;

 private:
  std::vector<int> _vec;
};
```

Pybind11 – Get Started

Karlsruhe Institute of Technology

vector_int.h

```
class VectorInt {

 public:
  VectorInt();

  void push_back(const int val);
  int get(const size_t idx) const;
  size_t size() const;
  bool is_empty() const;

 private:
  std::vector<int> _vec;
};
```

module.cpp

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

namespace py = pybind11;

PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
     .def(py::init<>())
     .def("push_back", &VectorInt::push_back)
     .def("get", &VectorInt::get)
     .def("size", &VectorInt::size)
     .def("is_empty", &VectorInt::is_empty);
}
```

Pybind11 – Get Started

vector_int.h

```
class VectorInt {

 public:
  VectorInt();

  void push_back(const int val);
  int get(const size_t idx) const;
  size_t size() const;
  bool is_empty() const;
};
```

Parameters and return values are automatically inferred using template metaprogramming

module.cpp

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

namespace py = pybind11;

PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    .def(py::init<>())
    .def("push_back", &VectorInt::push_back)
    .def("get", &VectorInt::get)
    .def("size", &VectorInt::size)
    .def("is_empty", &VectorInt::is_empty);
}
```

Pybind11 – Get Started

Karlsruhe Institute of Technology

vector_int.h

```
class VectorInt {

 public:
  VectorInt();

  void push_back(const int val);
  int get(const size_t idx) const;
  size_t size() const;
  bool is_empty() const;

 private:
  std::vector<int> _vec;
};
```

module.cpp

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

namespace py = pybind11;

PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
     .def(py::init<>())
     .def("push_back", &VectorInt::push_back)
     .def("get", &VectorInt::get)
     .def("size", &VectorInt::size)
     .def("is_empty", &VectorInt::is_empty);
}
```

Pybind11 – Get Started

Karlsruhe Institute of Technology

test.py

```
from vecint import PyVectorInt

vec = PyVectorInt()
print("Empty?="  + str(vec.is_empty()))

for i in range(10):
  vec.push_back(i)

for i in range(10):
  print("vec[" + str(i) + "]="
    + str(vec.get(i)))

print("Size="  + str(vec.size()))
print("Empty?="  + str(vec.is_empty()))
```

module.cpp

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

namespace py = pybind11;

PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    .def(py::init<>())
    .def("push_back", &VectorInt::push_back)
    .def("get", &VectorInt::get)
    .def("size", &VectorInt::size)
    .def("is_empty", &VectorInt::is_empty);
}
```

Pybind11 – Get Started

Karlsruhe Institute of Technology

test.py

```
from vecint import PyVectorInt

vec = PyVectorInt()
print("Empty?="  + str(vec.is_empty()))

for i in range(10):
  vec.push_back(i)

for i in range(10):
  print("vec[" + str(i) + "]="
    + str(vec.get(i)))

print("Size="  + str(vec.size()))
print("Empty?="  + str(vec.is_empty()))
```

Output

```
Empty?=True
vec[0]=0
vec[1]=1
vec[2]=2
vec[3]=3
vec[4]=4
vec[5]=5
vec[6]=6
vec[7]=7
vec[8]=8
vec[9]=9
Size=10
Empty?=False
```

Operator Overloading

vector_int.h

```
class VectorInt {
  using iterator =
    typename std::vector<int>::iterator;

 public:
  ...
  iterator begin();
  iterator end();
  int get(const size_t idx) const;
  size_t size() const;
  std::string to_string() const;
  ...
};
```

Operator Overloading

vector_int.h

```
class VectorInt {
  using iterator =
    typename std::vector<int>::iterator;

 public:
  ...
  iterator begin();
  iterator end();
  int get(const size_t idx) const;
  size_t size() const;
  std::string to_string() const;
  ...
};
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("__repr__", &VectorInt::to_string)
    .def("__str__", &VectorInt::to_string)
    .def("__getitem__", &VectorInt::get)
    .def("__len__", &VectorInt::size)
    .def("__iter__", [](VectorInt& vec) {
       return py::make_iterator(
         vec.begin(), vec.end());
     }, py::keep_alive<0,1>())
}
```

Operator Overloading

vector_int.h

```
class VectorInt {
  using iterator =
    typename std::vector<int>::iterator;

 public:
  ...
  iterator begin();
  iterator end();
  int get(const size_t idx) const;
  size_t size() const;
  std::string to_string() const;
  ...
};
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("__repr__", &VectorInt::to_string)
    .def("__str__", &VectorInt::to_string)
    .def("__getitem__", &VectorInt::get)
    .def("__len__", &VectorInt::size)
    .def("__iter__", [](VectorInt& vec) {
       return py::make_iterator(
         vec.begin(), vec.end());
     }, py::keep_alive<0,1>())
}
```

keeps object alive while iterator exists

Operator Overloading

Karlsruhe Institute of Technology

Output

```
>> print(vec) # __repr__
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> print(str(vec)) # __str__
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> print(vec[1]) # __getitem__
1
>> print(len(vec)) # __len__
10
>> for elem in vec: # __iter__
    print(elem, end=' ')
0 1 2 3 4 5 6 7 8 9
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("__repr__", &VectorInt::to_string)
    .def("__str__", &VectorInt::to_string)
    .def("__getitem__", &VectorInt::get)
    .def("__len__", &VectorInt::size)
    .def("__iter__", [](VectorInt& vec) {
       return py::make_iterator(
         vec.begin(), vec.end());
     }, py::keep_alive<0,1>())
}
```

Attributes and Enum Types

vector_int.h

```
enum Access {
  READONLY,
  READWRITE
};

class VectorInt {
 ...
 public:
  VectorInt() :
    _access(Access::READWRITE) { }
  ...
  Access _access;
};
```

Attributes and Enum Types

vector_int.h

```
enum Access {
  READONLY,
  READWRITE
};

class VectorInt {
 ...
 public:
  VectorInt() :
    _access(Access::READWRITE) { }
  ...
  Access _access;
};
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::enum_<Access>(m, "Access")
    .value("READONLY", Access::READONLY)
    .value("READWRITE", Access::READWRITE);

  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def_readwrite("access",
      &VectorInt::_access)
}
```

Attributes and Enum Types

Output

```
>> print(vec.access)
Access.READWRITE

>> vec.access = Access.READONLY
>> print(vec.access)
Access.READONLY
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::enum_<Access>(m, "Access")
    .value("READONLY", Access::READONLY)
    .value("READWRITE", Access::READWRITE);

  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def_readwrite("access",
      &VectorInt::_access)
}
```

Dynamic Attributes

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(
    m, "PyVectorInt", py::dynamic_attr())
  ...
    .def_property_readonly(
      "length", &VectorInt::size)
}
```

Dynamic Attributes

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(
    m, "PyVectorInt", py::dynamic_attr())
  ...
    .def_property_readonly(
      "length", &VectorInt::size)
}
```

enables dynamic attributes

Dynamic Attributes

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(
    m, "PyVectorInt", py::dynamic_attr())
  ...
    .def_property_readonly(
      "length", &VectorInt::size)
}
```

enables dynamic attributes

Binds attribute `length` to `VectorInt::size()`

It is also possible to add attributes with getter and setter:

```
    .def_property("name",
      &Foo::getName,
      &Foo::setName)
```

Dynamic Attributes

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(
    m, "PyVectorInt", py::dynamic_attr())
  ...
    .def_property_readonly(
      "length", &VectorInt::size)
}
```

Output

```
>> vec = PyVectorInt()
>> vec.push_back(1)
>> print(vec.length)
1

>> vec.name = "Vector of Integers"
>> print(vec.__dict__)
{'name': 'Vector of Integers'}
```

Function Overloading

vector_int.h

```
class VectorInt {
 ...
 public:
  ...
  // Inserts an element at the end
  void push_back(const int elem);
  // Inserts an element n times at the end
  void push_back(const int elem, size_t n);
};
```

Function Overloading

vector_int.h

```
class VectorInt {
 ...
 public:
  ...
  // Inserts an element at the end
  void push_back(const int elem);
  // Inserts an element n times at the end
  void push_back(const int elem, size_t n);
};
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("push_back", static_cast<void
      (VectorInt::*)(const int)>(
        &VectorInt::push_back))

    .def("push_back", static_cast<void
      (VectorInt::*)(const int, size_t)>(
        &VectorInt::push_back))
}
```

Function Overloading

vector_int.h

```
class VectorInt {
 ...
 public:
  ...
  // Inserts an element at the end
  void push_back(const int elem);
  // In
  void
};
```

C++14 Feature:

```
.def("push_back",
  py::overload_cast<const int>(
    &VectorInt::push_back))
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("push_back", static_cast<void
      (VectorInt::*)(const int)>(
        &VectorInt::push_back))

    .def("push_back", static_cast<void
      (VectorInt::*)(const int, size_t)>(
        &VectorInt::push_back))
}
```

Function Overloading

Karlsruhe Institute of Technology

Output

```
>> vec.push_back(41)
>> vec.push_back(42, 3)
>> print(vec)
[41, 42, 42, 42]
```

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("push_back", static_cast<void
      (VectorInt::*)(const int)>(
        &VectorInt::push_back))

    .def("push_back", static_cast<void
      (VectorInt::*)(const int, size_t)>(
        &VectorInt::push_back))
}
```

Binding Lambda Functions

Karlsruhe Institute of Technology

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
  ...
    .def("push_back",
      [](VectorInt& vec,
         const int elem,
         const size_t n) {
        for ( size_t i = 0; i < n; ++i ) {
          vec.push_back(elem);
        }
      })
}
```

Binding Lambda Functions

Karlsruhe Institute of Technology

module.cpp

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

PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
  ...
    .def("doForAllElements",
      [](VectorInt& vec,
         std::function<void(int)>& f) {
           for ( int elem : vec ) {
             f(elem);
           }
         })
```

Binding Lambda Functions

Karlsruhe Institute of Technology

module.cpp

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

PYBIND11_MODULE(vecint, m) {
  py::class_<VectorInt>(m, "PyVectorInt")
  ...
    .def("doForAllElements",
      [](VectorInt& vec,
         std::function<void(int)>& f) {
           for ( int elem : vec ) {
             f(elem);
           }
         })
```

Output

```
>> print(vec)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>> l = []
>> vec.doForAllElements(
     lambda x : l.append(x * x))
>> print(l)
[0, 1, 4, 9, 16, 25,
 36, 49, 64, 81]
```

Documentation

Karlsruhe Institute of Technology

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  m.doc() = "A vector storing integers";
   py::class_<VectorInt>(m, "PyVectorInt")
      ...
      .def("get", &VectorInt::get,
        "Returns the number at position i",
        py::arg("i") = 0)
      ...
}
```

Documentation

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  m.doc() = "A vector storing integers";
   py::class_<VectorInt>(m, "PyVectorInt")
      ...
      .def("get", &VectorInt::get,
        "Returns the number at position i",
        py::arg("i") = 0)
      ...
}
```

Named parameter with default argument

```
vec.get() # Returns element at position 0
vec.get(i = 5)
```

Documentation

Karlsruhe Institute of Technology

module.cpp

```
PYBIND11_MODULE(vecint, m) {
  m.doc() = "A vector storing integers";
  py::class_<VectorInt>(m, "PyVectorInt")
    ...
    .def("get", &VectorInt::get,
      "Returns the number at position i",
      py::arg("i") = 0)
    ...
}
```

Named parameter with default argument

```
vec.get() # Returns element at position 0
vec.get(i = 5)
```

```
>> help(vecint)

Help on module vecint:

NAME
    vecint - A vector storing integers
...
get(...)
  get(self: vecint.PyVectorInt, i: int = 0)
  -> int

  Returns the number at position i
...
```

Karlsruhe Institute of Technology

Advanced Topics

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data);
```

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data);
```

Can lead to hard-to-debug non-determinism and segmentation faults

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data);
```

Can lead to hard-to-debug non-determinism and segmentation faults

$\Rightarrow$ Per default, Python **takes ownership** of pointer return values and garbage collection can eventually delete the Python wrapper, which will delete the pointer.

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data, py::return_value_policy::reference);
```

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data, py::return_value_policy::reference);
```

- `return_value_policy::reference`: Reference an existing object, but do not take ownership

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data, py::return_value_policy::reference);
```

- `return_value_policy::reference`: Reference an existing object, but do not take ownership
- `return_value_policy::take_ownership`: Reference an existing object and take ownership

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data, py::return_value_policy::reference);
```

- `return_value_policy::reference`: Reference an existing object, but do not take ownership
- `return_value_policy::take_ownership`: Reference an existing object and take ownership
- `return_value_policy::copy`: Create new copy of the returned object
- `return_value_policy::move`: Uses `std::move`. The new object is then owned by Python

Memory Management

```cpp
// Function in our VectorInt implementation
int* data() { return _vec.data(); }

// Pybind Binding
m.def("data", &VectorInt::data, py::return_value_policy::reference);
```

- `return_value_policy::reference`: Reference an existing object, but do not take ownership
- `return_value_policy::take_ownership`: Reference an existing object and take ownership
- `return_value_policy::copy`: Create new copy of the returned object
- `return_value_policy::move`: Uses `std::move`. The new object is then owned by Python
- `return_value_policy::automatic` (**default**)
  - `return_value_policy::take_ownership` when return value is a pointer
  - Otherwise, `return_value_policy::move` for rvalues and `return_value_policy::copy` for lvalues

Custom Constructors

C++

```
class Point {
 public:
  Point();
  Point(int x, int y);
  // Factory function
  static Point create(int x, int y);
}
```

Custom Constructors

Karlsruhe Institute of Technology

C++

```cpp
class Point {
 public:
  Point();
  Point(int x, int y);
  // Factory function
  static Point create(int x, int y);
}
```

Pybind

```cpp
py::class_<Point>(m, "Point")
  // Default constructor
  .def(py::init())
  // Point(int x, int y)
  .def(py::init<int,int>())
  // Factory function
  .def(py::init(&Point::create));
```

Template Classes and Functions

Bad News

```cpp
template<typename T>
class Vector {
  T& get(const size_t idx);
}

// Does not work :(
py::class_<Vector>(m, "Vector")
  .def("get", &Vector::get);
```

Template Classes and Functions

Karlsruhe Institute of Technology

Bad News

```cpp
template<typename T>
class Vector {
  T& get(const size_t idx);
}

// Does not work :(
py::class_<Vector>(m, "Vector")
  .def("get", &Vector::get);
```

Template parameters must be instantiated

```cpp
py::class_<Vector<int>>(m, "VectorInt")
  .def("get", &Vector<int>::get);

py::class_<Vector<double>>(m, "VectorDouble")
  .def("get", &Vector<double>::get);
```

Template Classes and Functions

Karlsruhe Institute of Technology

Bad News

```cpp
template<typename T>
class Vector {
  T& get(const size_t idx);
}

// Does not work :(
py::class_<Vector>(m, "Vector")
  .def("get", &Vector::get);
```

Good News

```cpp
template<typename T>
void foo(T& bar)

// Treated as overloaded functions
m.def("foo", &foo<int>);
m.def("foo", &foo<double>)
```

Template parameters must be instantiated

```cpp
py::class_<Vector<int>>(m, "VectorInt")
  .def("get", &Vector<int>::get);

py::class_<Vector<double>>(m, "VectorDouble")
  .def("get", &Vector<double>::get);
```

STL Containers

C++

```
m.def("someFunction",
   [](std::vector<int>& vec) {
      ...
   });
```

Python

```
l = [0, 1, 2, 3, 4, 5]
someFunction(l)
```

STL Containers

Karlsruhe Institute of Technology

C++

```
m.def("someFunction",
   [](std::vector<int>& vec) {
      ...
   });
```

Python

```
l = [0, 1, 2, 3, 4, 5]
someFunction(l)
```

- Conversion between STL and Python containers are automatically enabled (when including `pybind11/stl.h`) $\Rightarrow$ can be **expensive**

STL Containers

Karlsruhe Institute of Technology

C++

```
m.def("someFunction",
  [](std::vector<int>& vec) {
    ...
  });
```

Python

```
l = [0, 1, 2, 3, 4, 5]
someFunction(l)
```

- Conversion between STL and Python containers are automatically enabled (when including `pybind11/stl.h`) $\Rightarrow$ can be **expensive**
- There exists thin C++ wrapper classes for all major Python types

```
m.def("someFunction",
  [](py::list& list) {
    ...
  });
```

- For a list of all available types see `https://pybind11.readthedocs.io/en/stable/advanced/pycpp/object.html`

Some Fancy Shit At The End

Some Fancy Shit At The End

Embedding the Interpreter

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

namespace py = pybind11;

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

  // Run Python code in C++
  py::exec("print('Hello World')");
```

Some Fancy Shit At The End

Karlsruhe Institute of Technology

Embedding the Interpreter

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

namespace py = pybind11;

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

  // Run Python code in C++
  py::exec("print('Hello World')");

  // Import python module and list content of current directory
  py::module os = py::module::import("os");
  py::object result = os.attr("listdir")(".");
  for ( auto& res : result ) {
    std::cout << res.cast<std::string>() << std::endl;
  }
}
```

Some Fancy Shit At The End

Karlsruhe Institute of Technology

Embedding the Interpreter

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

namespace py = pybind11;

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

  // Run Python code in C++
  py::exec("print('Hello World')");

  // Import python module and list content of current directory
  py::module os = py::module::import("os");
  py::object result = os.attr("listdir")(".");
  for ( auto& res : result ) {
    std::cout << res.cast<std::string>() << std::endl;
  }
}
```

Running Python Code in C++ is also possible :)

Additional Resources

- pybind11 Documentation
  `https://pybind11.readthedocs.io/en/stable/`
- Mt-KaHyPar Python Interface
  `https://github.com/kahypar/mt-kahypar/tree/master/python`
- Slides and example code of this talk can be found here
  `https://github.com/kittobi1992/pybind11-tutorial`