

Binding C++ code to Python using Pybind11

Simone Balducci

Alma Mater Studiorum – Università di Bologna

21 October 2024

Content of this lecture

- What are Python/C++ bindings and why we care
- The pybind11 library
- How to define, compile and use a basic binding
- How to bind functions and classes
- How to use STL containers in the binding
- How to handle inheritance and polymorphism
- How to handle the interface between the two languages
- How to use Python code inside C++

Before starting

- To be able to compile and run the codes, you need:
 - gcc 4.8+
 - cmake 3.15+
 - python 3.7+
- You can load them from the list of compilers

```
$ module load compilers/gcc-12.3_sl7  
$ module load compilers/python-3.10.5  
$ module load compilers/cmake-3.27.7
```

- You can automatize this by adding the commands to your `.bashrc`
- get pybind11 with
`git clone https://github.com/pybind/pybind11.git`

Why Python bindings?

- Why do we care about binding C++ and Python together?

Why Python bindings?

- Why do we care about binding C++ and Python together?
- They both have strengths and weaknesses

Why Python bindings?

- Why do we care about binding C++ and Python together?
- They both have strengths and weaknesses
 - C++ is very fast. It's also very verbose and considered difficult to use
 - Python is slow, but also very easy to pick-up and use

Why Python bindings?

- Why do we care about binding C++ and Python together?
- They both have strengths and weaknesses
 - C++ is very fast. It's also very verbose and considered difficult to use
 - Python is slow, but also very easy to pick-up and use
- It's clear how a combination of these two languages could be very useful for a wide variety of applications

What's the end goal?

- We want to generate a Python module
- This module will contain code written in C++
- When we call a function from Python the C++ code will be called
- We can also do the opposite
 - use Python code inside C++

A basic example

- The binding is defined inside the `PYBIND11_MODULE` macro
- Let's try to write a simple C++ function and bind it
- We start by writing the function:

```
1 int add(int a, int b) {  
2     return a + b;  
3 }
```

- Now we can bind it

A basic example (cont.)

```
1 #include <pybind11/pybind11.h>
2
3 int add(int a, int b);
4
5 PYBIND11_MODULE(module_name, handler) {
6     handler.def("add", &add, "Function that adds two ints");
7 }
```

Compiling the module

- Now we compile the module
- We generate a shared library, so we must compile with `-shared` and `-fPIC`

```
$ g++ -O3 -shared -fPIC $(python3 -m pybind11 --includes)  
main.cpp -o module_name$(python3-config --extension-suffix)
```

NOTE: the name of the module in the executable must be identical to the one used inside `PYBIND11_MODULE`

Compiling the module and importing it in Python

- Now we compile the module
- We generate a shared library, so we must compile with `-shared` and `-fPIC`

```
$ g++ -O3 -shared -fPIC $(python3 -m pybind11 --includes)  
main.cpp -o module_name$(python3-config --extension-suffix)
```

- An now we can import it in Python like any library

```
1 import module_name as m  
2  
3 print(m.add(2, 3))
```

```
$ 5
```

Compiling the module using CMake

- Compiling manually is very inconvenient, even using a Makefile
- Most large C++ projects use CMake anyway
- We can compile the binding using CMake (which also makes it more portable)

Compiling the module using CMake (cont.)

- To include pybind11 there are two ways

Compiling the module using CMake (cont.)

- To include pybind11 there are two ways
 - install globally and use `find_package(pybind11 REQUIRED)`

Compiling the module using CMake (cont.)

- To include pybind11 there are two ways
 - install globally and use `find_package(pybind11 REQUIRED)`
 - add it as a submodule and use `add_subdirectory(pybind11)`

Compiling the module using CMake (cont.)

- To include pybind11 there are two ways
 - install globally and use `find_package(pybind11 REQUIRED)`
 - add it as a submodule and use `add_subdirectory(pybind11)`
- Use `find_package` to fetch the interpreter and development components of Python

Compiling the module using CMake (cont.)

- To include pybind11 there are two ways
 - install globally and use `find_package(pybind11 REQUIRED)`
 - add it as a submodule and use `add_subdirectory(pybind11)`
- Use `find_package` to fetch the interpreter and development components of Python
- To compile modules, use the `pybind11_add_module` function

Compiling the module using CMake (cont.)

- To include pybind11 there are two ways
 - install globally and use `find_package(pybind11 REQUIRED)`
 - add it as a submodule and use `add_subdirectory(pybind11)`
- Use `find_package` to fetch the interpreter and development components of Python
- To compile modules, use the `pybind11_add_module` function

```
1 set(CMAKE_CXX_STANDARD 20)
2 set(CMAKE_CXX_STANDARD_REQUIRED ON)
3 set(CMAKE_CXX_EXTENSIONS OFF)
4
5 find_package(Python 3.10 COMPONENTS Interpreter Development REQUIRED)
6 add_subdirectory(pybind11)
7
8 pybind11_add_module(module_name main.cpp)
```

Binding a class

- We can bind a class using `pybind11::class_`

```
1 #include <pybind11/pybind11.h>
2
3 namespace py = pybind11;
4
5 class MyClass;
6
7 PYBIND11_MODULE(Module, handler) {
8     py::class_<MyClass>(handler, "MyClass")
9
10 }
```

Binding a class

- We can bind a class using `pybind11::class_`
- we then bind the class methods using the `def` function

```
1 #include <pybind11/pybind11.h>
2
3 namespace py = pybind11;
4
5 class MyClass;
6
7 PYBIND11_MODULE(Module, handler) {
8     py::class_<MyClass>(handler, "MyClass")
9         .def(py::init<>())
10
11 }
```

Binding a class

- We can bind a class using `pybind11::class_`
- we then bind the class methods using the `def` function
- for the constructor(s) we use the `pybind11::init` function

```
1 #include <pybind11/pybind11.h>
2
3 namespace py = pybind11;
4
5 class MyClass;
6
7 PYBIND11_MODULE(Module, handler) {
8     py::class_<MyClass>(handler, "MyClass")
9         .def(py::init<>())
10        .def("method1", &MyClass::method1)
11        .def("method2", &MyClass::method2);
12 }
```

Binding a class: multiple constructors

- If the class has multiple constructors, just repeat the declaration

```
1 class MyClass {  
2 public:  
3     MyClass();  
4     MyClass(int, int);  
5     MyClass(std::vector<int>);  
6 };  
7  
8
```

Binding a class: multiple constructors

- If the class has multiple constructors, just repeat the declaration
- Put the parameters inside the angle brackets

```
1 class MyClass {
2 public:
3     MyClass();
4     MyClass(int, int);
5     MyClass(std::vector<int>);
6 };
7
8 PYBIND11_MODULE(Module, handler) {
9     py::class_<MyClass>(handler, "MyClass")
10        .def(py::init<>())
11        .def(py::init<int, int>())
12        .def(py::init<std::vector<int>>())
13 }
```

Binding a class: overloaded methods (cont.)

Example:

```
1 int add(int i, int j);  
2 float add(float i, float j);
```

Binding a class: overloaded methods (cont.)

Example:

```
1 int add(int i, int j);  
2 float add(float i, float j);
```

Binding overloads with **manual casting**

```
1 PYBIND11_MODULE(Add, m) {  
2     m.def("add", (int (*)(int, int)) &add);  
3     m.def("add", (float (*)(float, float)) &add);  
4 }
```

Binding a class: overloaded methods (cont.)

Example:

```
1 int add(int i, int j);
2 float add(float i, float j);
```

Binding overloads with **manual casting**

```
1 PYBIND11_MODULE(Add, m) {
2     m.def("add", (int (*)(int, int)) &add);
3     m.def("add", (float (*)(float, float)) &add);
4 }
```

Binding overloads with **overload cast**

```
1 PYBIND11_MODULE(Add, m) {
2     m.def("add", py::overload_cast<int, int>(&add));
3     m.def("add", py::overload_cast<float, float>(&add));
4 }
```

Return value policies

There are several ways for Python to treat the return values of bound functions:

- `return_value_policy::take_ownership` → default for **pointers**
- `return_value_policy::copy` → default for **lvalue references**
- `return_value_policy::move` → default for **rvalue references**
- `return_value_policy::reference`
- `return_value_policy::reference_internal`
- `return_value_policy::automatic`
- `return_value_policy::automatic_reference`

Keyworded arguments

- When we bind a function we can also specify the names of the arguments

Keyworded arguments

- When we bind a function we can also specify the names of the arguments
- Use the `py::arg` function for this

Keyworded arguments

- When we bind a function we can also specify the names of the arguments
- Use the `py::arg` function for this
 - Allows keyworded arguments from Python

Keyworded arguments

- When we bind a function we can also specify the names of the arguments
- Use the `py::arg` function for this
 - Allows keyworded arguments from Python
 - Provides better function signature in Docstring

Keyworded arguments

- When we bind a function we can also specify the names of the arguments
- Use the `py::arg` function for this
 - Allows keyworded arguments from Python
 - Provides better function signature in Docstring
 - Can specify the default value of the argument

Keyworded arguments

- When we bind a function we can also specify the names of the arguments
- Use the `py::arg` function for this
 - Allows keyworded arguments from Python
 - Provides better function signature in Docstring
 - Can specify the default value of the argument

```
1 // Binding module
2 m.def("add", py::overload_cast<int, int>(&add),
3       py::arg("x"), py::arg("y"));
4 m.def("add", py::overload_cast<float, float>(&add),
5       py::arg("x")=0.f, py::arg("y")=0.f);
```

Binding inheriting classes

- If I have one class inheriting from another, I can specify this relation when declaring the class

Binding inheriting classes

- If I have one class inheriting from another, I can specify this relation when declaring the class
- We can either pass it as a template parameter to `py::class_<` or pass it as a parameter to the object

```
1 PYBIND11_MODULE(Module, m) {  
2     py::class_<Base>(m, "Base")  
3         .def(...);  
4     py::class_<Derived, Base>(m, "Derived")  
5         .def(...);  
6 }
```

Binding inheriting classes (cont.)

```
1 struct Base {
2     void foo() {
3         std::cout << "Method from the Base class\n";
4     }
5 };
6
7 struct Derived : public Base {
8     void bar() {
9         std::cout << "Method from the Derived class\n";
10    }
11 };
```

Binding inheriting classes (cont.)

```
1 from Module import Derived
2
3 d = Derived()
4 d.foo()
5 d.bar()
```

```
$ python3 inheritance.py
Method from the Base class
Method from the Derived class
```

Binding polymorphic types

- In C++, classes like the ones just defined are not polymorphic
 - they don't provide any virtual methods

Binding polymorphic types

- In C++, classes like the ones just defined are not polymorphic
 - they don't provide any virtual methods
- This can be seen also from Python

```
1 PYBIND11_MODULE(Module, m) {  
2     m.def("make_derived", [](){  
3         return std::unique_ptr<Base>(new Derived());  
4     });  
5 }
```

Binding polymorphic types

- In C++, classes like the ones just defined are not polymorphic
 - they don't provide any virtual methods
- This can be seen also from Python

```
1 PYBIND11_MODULE(Module, m) {  
2     m.def("make_derived", [](){  
3         return std::unique_ptr<Base>(new Derived());  
4     });  
5 }
```

- If we call the function above from Python we see:

```
1 p = make_derived()  
2 print(type(p)) # Output: <class 'Module.Base'>
```

Binding polymorphic types (cont.)

Let's re-define the Base and Derived classes in a polymorphic way

Binding polymorphic types (cont.)

Let's re-define the Base and Derived classes in a polymorphic way

```
1 struct Base {
2     virtual ~Base() = default;
3     virtual void foo() = 0;
4 };
5
6 struct Derived : public Base {
7     Derived() = default;
8     void foo() override {
9         std::cout << "Implemented foo\n";
10    }
11};
```

Binding polymorphic types (cont.)

Let's re-define the Base and Derived classes in a polymorphic way

```
1 struct Base {
2     virtual ~Base() = default;
3     virtual void foo() = 0;
4 };
5
6 struct Derived : public Base {
7     Derived() = default;
8     void foo() override {
9         std::cout << "Implemented foo\n";
10    }
11};
```

If we call the `make_derived` function from Python like before:

```
1 p = make_derived()
2 print(type(p)) # Output: <class 'Module.Derived'>
```

Overriding from Python

- In some cases it might be useful to override a class virtual method from Python

Overriding from Python

- In some cases it might be useful to override a class virtual method from Python
- Doing so requires to call the Parent class constructor

```
1 from Module import AbstractBase
2
3 def PyDerived(AbstractBase):
4     def __init__(self):
5         AbstractBase.__init__()
```

Overriding from Python

- In some cases it might be useful to override a class virtual method from Python
- Doing so requires to call the Parent class constructor

```
1 from Module import AbstractBase
2
3 def PyDerived(AbstractBase):
4     def __init__(self):
5         AbstractBase.__init__()
```

- If the Parent is an abstract class, this is a problem
 - No constructor to be called

- We can solve this using **trampoline classes**

Trampoline classes

- We can solve this using **trampoline classes**
- Take an abstract class with a virtual method

```
1 class AbstractBase {  
2 public:  
3     virtual ~AbstractBase = default;  
4     virtual void method(int) = 0;  
5 };
```

Trampoline classes (cont.)

- Define a class inheriting from the abstract base

```
1 struct TrampolineAbstract : public AbstractBase {
2
};
```

Trampoline classes (cont.)

- Define a class inheriting from the abstract base
- Inherit the constructor

```
1 struct TrampolineAbstract : public AbstractBase {  
2     using AbstractBase::AbstractBase;  
3  
4  
  
};
```

Trampoline classes (cont.)

- Define a class inheriting from the abstract base
- Inherit the constructor
- Override the method with
PYBIND11_OVERRIDE/PYBIND11_OVERRIDE_PURE

```
1 struct TrampolineAbstract : public AbstractBase {
2     using AbstractBase::AbstractBase;
3
4     void method() override {
5         PYBIND11_OVERRIDE_PURE(
6             void,
7             AbstractBase,
8             method,
9             int
10        );
11    }
12};
```

Using the trampoline class

- Only need to change two lines in the binding

```
1 PYBIND11_MODULE(Module, m) {
2     py::class_<AbstractBase, TrampolineAbstract>(m, "AbstractBase")
3         .def(py::init<>())
4         ...
5 }
```

Using the trampoline class

- Only need to change two lines in the binding

```
1 PYBIND11_MODULE(Module, m) {
2     py::class_<AbstractBase, TrampolineAbstract>(m, "AbstractBase")
3         .def(py::init<>())
4         ...
5 }
```

- Then we can derive the class from Python

```
1 class PyDerived(AbstractBase):
2     def __init__(self):
3         AbstractBase.__init__()
4     def method(self, int):
5         ...
```

Example: Vector3 class

```
1 struct Vector3 {
2     float x, y, z;
3
4     Vector3() : x{0.}, y{0.}, z{0.} {}
5     Vector3(float x, float y, float z) : x{x}, y{y}, z{z} {}
6
7     Vector3 operator+(const Vector3D &other) const;
8     Vector3& operator+=(const Vector3D &other);
9     Vector3 operator*(float scalar) const;
10    friend Vector3 operator*(float scalar, const Vector3& vector);
11    Vector3& operator*=(float scalar);
12    Vector3 operator/(float scalar) const;
13    Vector3& operator/=(float scalar);
14};
```

How to bind operators

Operators can be bound in two ways:

How to bind operators

Operators can be bound in two ways:

- Manually by binding the method and specifying the Python operator
 - marking the method with the `py::is_operator` flag

```
1 .def("__add__", [](const Object& self, const Object& other){
2     return self + other;
3 }, py::is_operator())
```

How to bind operators

Operators can be bound in two ways:

- Manually by binding the method and specifying the Python operator
 - marking the method with the `py::is_operator` flag

```
1 .def("__add__", [](const Object& self, const Object& other){
2     return self + other;
3 }, py::is_operator())
```

- Using the `py::self` notation
 - included in the `operators.h` header

```
1 .def(py::self + py::self)
```

Binding the operators of Vector3

The binding of the operators using the `py::self` notation looks like this

```
1 PYBIND11_MODULE(Vector, m) {  
2     py::class_<Vector3>(m, "Vector3")  
3         .def(py::init<int, int, int>())  
4         .def(py::self + py::self)  
5         .def(py::self += py::self)  
6         .def(py::self * float())  
7         .def(float() * py::self)  
8         .def(py::self *= float())  
9         .def(py::self / float())  
10        .def(py::self /= float())  
11 }
```

Binding the operators of Vector3 (cont.)

The manual binding is still very useful because it allows us to bind operators not defined in the C++ class

Binding the operators of Vector3 (cont.)

The manual binding is still very useful because it allows us to bind operators not defined in the C++ class

```
1 .def("__len__", [] (const Vector3D<int>& v) { return 3; })
```

```
2
```

Binding the operators of Vector3 (cont.)

The manual binding is still very useful because it allows us to bind operators not defined in the C++ class

```
1 .def("__len__", [](const Vector3D<int>& v) { return 3; })
2 .def("__str__", [](const Vector3D<int>& v) -> std::string {
3     return "[" + std::to_string(v.x) + ", "
4         + std::to_string(v.y) + ", "
5         + std::to_string(v.z) + "];"
6     })
7
```

Binding the operators of Vector3 (cont.)

The manual binding is still very useful because it allows us to bind operators not defined in the C++ class

```
1 .def("__len__", [](const Vector3D<int>& v) { return 3; })
2 .def("__str__", [](const Vector3D<int>& v) -> std::string {
3     return "[" + std::to_string(v.x) + ", "
4         + std::to_string(v.y) + ", "
5         + std::to_string(v.z) + "]"
6     })
7 .def("__mul__",
8     [](const Vector3D<int>& v, const Vector3D<int>& w) -> int {
9         return v.x * w.x + v.y * w.y + v.z * w.z;
10    },
11    py::is_operator())
12
```

Binding the operators of Vector3 (cont.)

The manual binding is still very useful because it allows us to bind operators not defined in the C++ class

```
1 .def("__len__", [](const Vector3D<int>& v) { return 3; })
2 .def("__str__", [](const Vector3D<int>& v) -> std::string {
3     return "[" + std::to_string(v.x) + ", "
4         + std::to_string(v.y) + ", "
5         + std::to_string(v.z) + "]"
6     })
7 .def("__mul__",
8     [](const Vector3D<int>& v, const Vector3D<int>& w) -> int {
9         return v.x * w.x + v.y * w.y + v.z * w.z;
10    },
11    py::is_operator())
12 .def("__getitem__",
13     [](const Vector3D<int>& v, int i) -> int { return v[i]; })
14 .def("__setitem__",
15     [](Vector3D<int>& v, int i, int val) -> void { v[i] = val; });
```

Handling code with templates

- In C++, template parameters have to be specified at compile time

Handling code with templates

- In C++, template parameters have to be specified at compile time
- This means that when binding template functions or classes, we must specialize the templates

Handling code with templates

- In C++, template parameters have to be specified at compile time
- This means that when binding template functions or classes, we must specialize the templates

```
1 template <typename T>
2 void foo(T x);
3
4 PYBIND11_MODULE(Module, m) {
5     m.def("foo", &foo);
```

Handling code with templates

- In C++, template parameters have to be specified at compile time
- This means that when binding template functions or classes, we must specialize the templates

```
1 template <typename T>
2 void foo(T x);
3
4 PYBIND11_MODULE(Module, m) {
5     m.def("foo", &foo);           // Error!
6     m.def("foo", &foo<int>);
```

Handling code with templates

- In C++, template parameters have to be specified at compile time
- This means that when binding template functions or classes, we must specialize the templates

```
1 template <typename T>
2 void foo(T x);
3
4 PYBIND11_MODULE(Module, m) {
5     m.def("foo", &foo);           // Error!
6     m.def("foo", &foo<int>);     // Ok
7 }
```

There are three ways to interface C++ and Python data structures

There are three ways to interface C++ and Python data structures

- 1 Use C++ containers in Python

There are three ways to interface C++ and Python data structures

- 1 Use C++ containers in Python
- 2 Use Python containers in C++

There are three ways to interface C++ and Python data structures

- 1 Use C++ containers in Python
- 2 Use Python containers in C++
- 3 Type conversions between C++ and Python types

There are three ways to interface C++ and Python data structures

- 1 Use C++ containers in Python
- 2 Use Python containers in C++
- 3 Type conversions between C++ and Python types

The third one is the simplest, but often not the most efficient

→ **the data is copied**

Opaque C++ data structures

- pybind11 provides type conversion from most STL containers
- These conversions are used by default at the C++/Python interface
- We can prevent this by marking a type as **opaque**, which disables the conversion
- Then we have to bind the container as if it was a user-defined class
 - we can bind manually or use the `py::bind_vector` utility defined in the `stl_bind.h` header

Making vector opaque

```
1 void append_42(std::vector<int>&);
2
3 PYBIND11_MAKE_OPAQUE(std::vector<int>);
4
5 PYBIND11_MODULE(Vector, m) {
6     py::class_<std::vector<int>>(m, "VectorInt")
7         .def(py::init<>())
8         .def("push_back", py::overload_cast<const int>&(&std::vector<int>::push_back))
9         .def("__len__", [](const std::vector<int>& v) { return v.size(); })
10        .def("__str__", [](const std::vector<int>& v) -> std::string {
11            std::string res = "[";
12            std::for_each(v.begin(), v.end()-1, [&res](auto x) -> void {
13                res += std::to_string(x) + ", ";
14            });
15            res += std::to_string(v.back()) + "]";
16            return res;
17        });
18        .def("__iter__", [](std::vector<int>& v) {
19            return py::make_iterator(v.begin(), v.end());
20        }, py::keep_alive<0, 1>())
21    m.def("append_42", &append_42);
22 }
```

Making vector opaque (cont.)

If now we try to call the method from Python:

```
1 from Vector import VectorInt
2 from Vector import append_42
3
4 v = VectorInt()
5 v.push_back(1)
6 print(v)
7 append_42(v)
8 print(v)
```

Making vector opaque (cont.)

If now we try to call the method from Python:

```
1 from Vector import VectorInt
2 from Vector import append_42
3
4 v = VectorInt()
5 v.push_back(1)
6 print(v)
7 append_42(v)
8 print(v)
```

```
$ python3 append.py
[1]
[1, 42]
```

- We can expose our bound classes as buffers

Buffer protocols

- We can expose our bound classes as buffers
- This allows them to be cast into numpy arrays without any copy of data

- We can expose our bound classes as buffers
- This allows them to be cast into numpy arrays without any copy of data
- We can mark a function to accept generic buffers with the `py::buffer` type

Buffer protocols

- We can expose our bound classes as buffers
- This allows them to be cast into numpy arrays without any copy of data
- We can mark a function to accept generic buffers with the `py::buffer` type

```
1 struct buffer_info {
2     void *ptr;           // buffer pointer
3     py::ssize_t itemsize; // (bytes) size of a scalar
4     std::string format;  // format descriptor
5     py::ssize_t ndim;    // number of dimensions
6     std::vector<py::ssize_t> shape; // shape of the buffer
7     std::vector<py::ssize_t> strides; // strides of the buffer
8 };
```

Exposing class as buffer

- Add the `py::buffer_protocol()` tag in the class binding declaration

```
1 py::class_<Vector>(m, "Vector", py::buffer_protocol())
```

```
2
```

Exposing class as buffer

- Add the `py::buffer_protocol()` tag in the class binding declaration
- Call the `def_buffer` method with a function returning a `py::buffer_info` describing an instance of the class

```
1 py::class_<Vector>(m, "Vector", py::buffer_protocol())
2   .def_buffer([](Vector& v) -> py::buffer_info {
3       return py::buffer_info(
4           v.data(),
5           sizeof(float),
6           py::format_descriptor<float>::format(),
7           1,
8           { m.size() },
9           { sizeof(float) }
10      );
11  });
```

Using numpy arrays in C++

- pybind11 provides the `array` and `array_t<T>` classes that specialize the generic buffer to numpy arrays
- We access the content buffer with the `request` method
- The buffer then contains all the information needed to use the array

```
1 #include <pybind11/numpy.h>
2
3 void foo(py::array_t<float> arr) {
4     auto buf = arr.request();
5     float* p = static_cast<float*>(buf.ptr);
6
7
```

Using numpy arrays in C++

- pybind11 provides the `array` and `array_t<T>` classes that specialize the generic buffer to numpy arrays
- We access the content buffer with the `request` method
- The buffer then contains all the information needed to use the array

```
1 #include <pybind11/numpy.h>
2
3 void foo(py::array_t<float> arr) {
4     auto buf = arr.request();
5     float* p = static_cast<float*>(buf.ptr);
6
7     std::span<float> data{p, buf.size};
8 }
```

Using numpy arrays in C++

- pybind11 provides the `array` and `array_t<T>` classes that specialize the generic buffer to numpy arrays
- We access the content buffer with the `request` method
- The buffer then contains all the information needed to use the array

```
1 #include <pybind11/numpy.h>
2
3 void foo(py::array_t<float> arr) {
4     auto buf = arr.request();
5     float* p = static_cast<float*>(buf.ptr);
6
7     std::span<float> data{p, buf.size};
8     const auto sum = std::ranges::accumulate(data, 0.f);
9     return sum;
10 }
```

Using numpy arrays in C++ (cont.)

- In order to prevent copies, you must be careful with the data type

Using numpy arrays in C++ (cont.)

- In order to prevent copies, you must be careful with the data type
- If the data type between the Python and C++ arrays is not the same, there is a data conversion
 - The data is copied

Using numpy arrays in C++ (cont.)

- In order to prevent copies, you must be careful with the data type
- If the data type between the Python and C++ arrays is not the same, there is a data conversion
 - The data is copied
- For example:

```
1 from module import foo
2 import numpy as np
3
4 arr = np.array([1, 2, 3], dtype=np.float64) # [1, 2, 3]
5 append_42(arr) # [1, 2, 3] (still)
```

Using numpy arrays in C++ (cont.)

- In order to prevent copies, you must be careful with the data type
- If the data type between the Python and C++ arrays is not the same, there is a data conversion
 - The data is copied
- For example:

```
1 from module import foo
2 import numpy as np
3
4 arr = np.array([1, 2, 3], dtype=np.float64) # [1, 2, 3]
5 append_42(arr) # [1, 2, 3] (still)
```

- The data is copied into a new array, so our changes are not applied to the original one

Request, unchecked, mutable unchecked

- The `request` method does bound checking on every access to the array

- The `request` method does bound checking on every access to the array
- If we need to optimize performance and if we can be safe that the indices don't overflow, we can use the more efficient
 - `unchecked<N>`
 - `mutable_unchecked<N>`

- The `request` method does bound checking on every access to the array
- If we need to optimize performance and if we can be safe that the indices don't overflow, we can use the more efficient
 - `unchecked<N>`
 - `mutable_unchecked<N>`

where `N` represents the dimensionality of the array

- The `request` method does bound checking on every access to the array
- If we need to optimize performance and if we can be safe that the indices don't overflow, we can use the more efficient
 - `unchecked<N>`
 - `mutable_unchecked<N>`

where N represents the dimensionality of the array

- As the name suggests, they provide unchecked access to the data

Using Python inside C++ code

- While we usually want to bind C++ code to Python, the opposite is also possible

Using Python inside C++ code

- While we usually want to bind C++ code to Python, the opposite is also possible
- We can embed the Python interpreter inside a C++ object

Using Python inside C++ code

- While we usually want to bind C++ code to Python, the opposite is also possible
- We can embed the Python interpreter inside a C++ object
- This allows us to:

Using Python inside C++ code

- While we usually want to bind C++ code to Python, the opposite is also possible
- We can embed the Python interpreter inside a C++ object
- This allows us to:
 - execute Python code

Using Python inside C++ code

- While we usually want to bind C++ code to Python, the opposite is also possible
- We can embed the Python interpreter inside a C++ object
- This allows us to:
 - execute Python code
 - import and use Python libraries

Embedding the Python interpreter

- The interpreter's lifetime is managed by the RAII object `py::scoped_interpreter`

Embedding the Python interpreter

- The interpreter's lifetime is managed by the RAII object `py::scoped_interpreter`
- Included in the `embed.h` header

Embedding the Python interpreter

- The interpreter's lifetime is managed by the RAII object `py::scoped_interpreter`
- Included in the `embed.h` header
- Python code can be executed with the `py::exec` function

Embedding the Python interpreter

- The interpreter's lifetime is managed by the RAII object `py::scoped_interpreter`
- Included in the `embed.h` header
- Python code can be executed with the `py::exec` function
- We can use Python libraries by:
 - importing them with `py::module_::import`

Embedding the Python interpreter

- The interpreter's lifetime is managed by the RAII object `py::scoped_interpreter`
- Included in the `embed.h` header
- Python code can be executed with the `py::exec` function
- We can use Python libraries by:
 - importing them with `py::module_::import`
 - accessing attributes with `py::module_::attr`

Embedding the Python interpreter

- The interpreter's lifetime is managed by the RAII object `py::scoped_interpreter`
- Included in the `embed.h` header
- Python code can be executed with the `py::exec` function
- We can use Python libraries by:
 - importing them with `py::module_::import`
 - accessing attributes with `py::module_::attr`
- As an example, let's try to plot some functions using `matplotlib`

Example: plotting in C++ with `module_::import`

By importing the library the code looks like this:

```
1 #include <pybind11/embed.h>
2 #include <pybind11/stl_bind.h>
3
4 int main() {
5     py::scoped_interpreter guard{};
6     py::module_ plt = py::module_::import("matplotlib.pyplot");
7     std::vector<double> x(10), y(10);
8     std::iota(x.begin(), x.end(), 0.);
9     std::transform(x.begin(), x.begin(), y.begin(), [](double x) {
10         return x * x;
11     });
12     py::bind_vector<std::vector<double>>(plt, "VectorDouble");
13     plt.attr("plot")(x, y);
14     plt.attr("show")();
15 }
```

Example: plotting in C++ with `py::exec`

By executing Python code directly with `py::exec` the code looks like this:

```
1 #include <pybind11/embed.h>
2
3 int main() {
4     py::scoped_interpreter guard{};
5     py::exec(R"(
6         import matplotlib.pyplot as plt
7
8         x = range(10)
9         y = [i * i for i in x]
10        plt.plot(x, y)
11        plt.show()
12    )");
13 }
```

Embedding modules

- We can define embedded modules
 - Python modules that can be imported in the same cpp file that defines it

```
1 #include <pybind11/embed.h>
2 namespace py = pybind11;
3
4 PYBIND11_EMBEDDED_MODULE(Module, m) {
5     m.def("parallel_add", [](py::array_t<float> arr) { ... });
6 }
7
8 int main() {
9     py::scoped_interpreter guard{};
10    py::module_ m = py::module_::import("Module");
11    // ...
12 }
```

Summary

We learned to:

We learned to:

- bind C++ classes and functions to Python using pybind11

We learned to:

- bind C++ classes and functions to Python using pybind11
- handle inheritance and override from Python

We learned to:

- bind C++ classes and functions to Python using pybind11
- handle inheritance and override from Python
- overload Python operators

We learned to:

- bind C++ classes and functions to Python using pybind11
- handle inheritance and override from Python
- overload Python operators
- define buffer protocols for objects

We learned to:

- bind C++ classes and functions to Python using pybind11
- handle inheritance and override from Python
- overload Python operators
- define buffer protocols for objects
- use Python types in C++

We learned to:

- bind C++ classes and functions to Python using pybind11
- handle inheritance and override from Python
- overload Python operators
- define buffer protocols for objects
- use Python types in C++
- embed the Python interpreter in a C++ executable

Thanks for the attention

Binding protected methods

- We might want to bind methods marked as protected

Binding protected methods

- We might want to bind methods marked as protected
- It can be done with the following pattern:

```
1 class A {
2 protected:
3     void foo() { std::cout << "I'm a protected method\n";}
4 };
5
6 class Publicist : public A {
7 public:
8     using A::foo;
9 };
10
11 PYBIND11_MODULE(Module, m) {
12     py::class_<A>(m, "A")
13         .def("foo", &Publicist::foo);
14 }
```

Backup

Backup: Binding overloaded operators

- Consider a class representing an object for which arithmetic operations are defined

Backup: Binding overloaded operators

- Consider a class representing an object for which arithmetic operations are defined
- We want to implement the arithmetic operators in Python
 - `__add__`, `__sub__`, `__mul__`, `__neg__`, ...

Backup: Binding overloaded operators

- Consider a class representing an object for which arithmetic operations are defined
- We want to implement the arithmetic operators in Python
 - `__add__`, `__sub__`, `__mul__`, `__neg__`, ...
- As an example, let's implement a 3D Vector class and bind its operators

Backup: Managing references with smart pointers

- When we bind a class we can define a special **holder type** than will manage the references to the object

Backup: Managing references with smart pointers

- When we bind a class we can define a special **holder type** than will manage the references to the object
- By default `std::unique_ptr<...>` is used

Backup: Managing references with smart pointers

- When we bind a class we can define a special **holder type** than will manage the references to the object
- By default `std::unique_ptr<...>` is used
- We can use `std::shared_ptr<...>`

```
1 class Agent {
2     private:
3         std::string m_name;
4         Agent(const std::string& name) : m_name(name) {}
5     public:
6         static std::shared_ptr<Agent> create(const std::string& name) {
7             return std::make_shared<Agent>(name);
8         }
9         void say_hello() const {
10             std::cout << "Hello, " << m_name << "!\n";
11         }
12 };
13
14 PYBIND11_MODULE(Shared, m) {
15     py::class_<Agent, std::shared_ptr<Agent>>(m, "Agent")
16         .def_static("create", &Agent::create)
17         .def("say_hello", &Agent::say_hello);
18 }
```

The problem with `shared_ptr` holders

Now consider an example:

```
1 class Data {};  
2  
3 class Holder {  
4     private:  
5         std::shared_ptr<Data> m_data;  
6     public:  
7         Holder() : m_data(std::make_shared<Data>()) {}  
8         Data* getptr() { return m_data.get(); }  
9 };
```

- Python will see the pointer from `getptr` and wrap it inside a shared pointer
- This will likely cause undefined behaviour

- The **Curiously Recurring Template Pattern** is a template pattern in C++

```
1 template <typename T>
2 class Base;
3
4 class Derived : Base<Derived>;
```

C RTP and `std::enable_shared_from_this`

- The **Curiously Recurring Template Pattern** is a template pattern in C++

```
1 template <typename T>
2 class Base;
3
4 class Derived : Base<Derived>;
```

- A useful application of this is the `std::enable_shared_from_this` template

C RTP and `std::enable_shared_from_this`

- The **Curiously Recurring Template Pattern** is a template pattern in C++

```
1 template <typename T>
2 class Base;
3
4 class Derived : Base<Derived>;
```

- A useful application of this is the `std::enable_shared_from_this` template
- If a class inherits from it, the `shared_from_this` method is defined

C RTP and `std::enable_shared_from_this`

- The **Curiously Recurring Template Pattern** is a template pattern in C++

```
1 template <typename T>
2 class Base;
3
4 class Derived : Base<Derived>;
```

- A useful application of this is the `std::enable_shared_from_this` template
- If a class inherits from it, the `shared_from_this` method is defined
- It allows to create new shared pointers holding a weak reference to the object

C RTP and `std::enable_shared_from_this`

- The **Curiously Recurring Template Pattern** is a template pattern in C++

```
1 template <typename T>
2 class Base;
3
4 class Derived : Base<Derived>;
```

- A useful application of this is the `std::enable_shared_from_this` template
- If a class inherits from it, the `shared_from_this` method is defined
- It allows to create new shared pointers holding a weak reference to the object
 - It's safer than copying the original pointer
 - It's more efficient if we need to do a lot of copies