Integrating C++ with Other Languages

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Autumn 2020

Scripting Support in Your Applications

- you want to support plugins or user scripts:
  - e.g., scripts in a game engine for game logic,
  - macros for CAD software,
  - plugins for a chat client.
- technically you could:
  - write the code in C++
  - compile them in shared library
  - load the library at runtime

Scripting Support in Your Applications

- you want to support plugins or user scripts:
  - e.g., scripts in a game engine for game logic,
  - macros for CAD software,
  - plugins for a chat client.
- technically you could:
  - write the code in C++
  - compile them in shared library
  - load the library at runtime
- simple for you, annoying for the user (recompilation & reloading)
  - make sense only when user code is in a hot path
- better solution: **embed a scripting engine in your application**
  - Lua
  - JavaScript
  - Python
  - ~~design custom language and write interpreter~~ (usually a bad idea)
  - ...

C++ Does Not Fit All Use Cases

- C++ allows you to write high-performing code
- handling GUI and IO can be tedious in C++[1]

[1] The Discord Bot That Nearly Killed Me

C++ Does Not Fit All Use Cases

- C++ allows you to write high-performing code
- handling GUI and IO can be tedious in C++[1]
- write a library for the core functionality in C++
- expose bindings for other languages
  - e.g., machine learning library (Tensor Flow: core in C++, Python bindings for "basic users"),
  - write a micro service in C++, use Python's Flask for handling the HTTP server,
  - write business logic in C++, use C# for GUI,
  - write a kick-ass C++ library, use Python & Jupyter for demonstration & benchmarking.

[1] The Discord Bot That Nearly Killed Me

You Need a C Interface

- you need to extend existing C software and you prefer to write C++
- you write a kick-ass C++ library, but your potential users still write plain C code
- C interface is pretty simple to handle
  - most of existing languages provide a way to load C library and call functions
  - your C++ $\rightarrow$ C interface $\rightarrow$ target language bindings

This Lecture

- an overview of the common pitfalls
- calling your C++ code in C
- binging C++ to Python via PyBind11 (in-depth + exercise)
- scripting support in C++
  - Javascript
  - Lua (in-depth + exercise)

Common Pitfalls

- memory management
  - most modern languages use garbage collector
  - C++ does not (RAII)
  - **you have to be careful about shared objects/memory management**
- type conversion
  - built-in types might not match
  - conversion of user-defined types
  - potential bottle neck
- language features (how to map them?)
  - C++ templates do not map to other languages
  - C++ does not support named arguments
  - exception handling
  - ...

Calling C++ in a C Code

C and C++ are more-less binary compatible.

- you can call C++ functions in C
  - problem: name mangling
  - you can disable name mangling via `extern "C"`
  - e.g., `extern "C" void foo(int arg)`
  - cannot call template functions (you can wrap them inside an ordinary function)
- problem: shared header files
  - header files are processed by C compiler → no C++ constructions allowed (e.g., only structs without member functions)
  - preprocessor work-around: `#ifdef __cplusplus`
  - workaround: pass opaque `void *` instead of objects to functions
- uncaught exceptions in C code lead to undefined behavior unless C code is compiled with `-fexceptions`

Binding C++ to Python: ctypes and CFFI

There are multiple ways to create the bindings. The first option is: ctypes or CFFI

- at Python runtime load shared library, call C-functions
- we do not recommended it

```
from ctypes import cdll, c_float
lib = cdll.LoadLibrary('./simple.so')
lib.square.argtypes = (c_float,)
lib.square.restype = c_float
lib.square(2.0)
```

Binding C++ to Python: CPython

- the way the Python interpreter is implemented
- write C code, use `#include <Python>`
- specify module and function by specially named symbols

```
static PyObject *method_foo(PyObject *self,
                            PyObject *args) {
    //something
}
static PyMethodDef MymoduleMethods[] = {
    {"foo", method_foor, METH_VARARGS, "docstring"},
    {NULL, NULL, 0, NULL}};
static struct PyModuleDef mymodulemodule = {
    PyModuleDef_HEAD_INIT, "foo", "docstring",
    -1, MymoduleMethods};
```

Tedious to write; solution: use binding tools

Binding C++ to Python: SWIG

- "Simplified Wrapper and Interface Generator"
- automatic, old and mature solution
- supports multiple languages
- put special comments into your headers to define modules
- "all or nothing" – low control on what and how is exported
- is/used to be de-facto standard

Binding C++ to Python: CPPYY

- very new project
- based on Clang & LLVM
- supports JIT
  - can support C++ templates (instantiate in runtime)
- interesting project
- we do not recommend it as universal solution, but might map to some situations well

Binding C++ to Python: PyBind11

Our recommended solution.

- pure C++11 solution (no external tools required),
- integrates well with CMake
- fine-grained control over the exports
  - you have the ability to make the interface more "Pythonic"
- seamless type cooperation between C++ and Python
  - specify type casting
  - predefined casts for standard library (containers, `std::function`, `std::string`)
- well designed Python objects management (behaves like smart pointers)
- can also embed Python interpreter inside your code:
  - Python can be used as scripting language
  - you can use Python libraries in C++ code

PyBind11: First Steps

- read documentation
- install it
- setup CMake project and specify your module, e.g.:

  ```
  pybind11_add_module(example exampleSources.cpp)
  ```
- Note: You have to invoke Python with env variable `PYTHONPATH` pointing to the compiled module directory
- e.g., `PYTHONPATH=path_to_your_build_directory python`

PyBind11: Hello Math!

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

int add(int a, int b) { return a + b; }

PYBIND11_MODULE(example, m) {
    m.def("add", &add, "Add two integers");
    // Or you can use lambda
    m.def("subtract", [](int a, int b){ return a - b },
        "Subtract two integers");
}
```

Compile it and use it:

```python
from example import add, subtract

print(add(41, 1))
print(subtract(43, 1))
```

PyBind11: Argument Checking is For Free

```
>>> from example import add
>>> help(add)
add(arg0: int, arg1: int) -> int

Add two integers

>>> add('foo', 'bar')
TypeError: add(): Incompatible function arguments.
The following argument types are supported:
  1. add(arg0: int, arg1: int) -> int

Invoked with: 'foo', 'bar'
```

PyBind11: Structures & Classes

Consider a simple struct:

```
struct Cat {
    std::string name;
    int age;

    Cat(std::string name, int age = -1):
        name(std::move(name)), age(age) {}

    bool hasKnownAge() const { return age >= 0; }
    bool operator==( const Cat& o ) { /* omitted */ }
};
```

PyBind11: Structures & Classes

```
PYBIND11_MODULE(animals, m) {
    py::class_<Cat>(m, "Cat")
        // Constructors
        .def(py::init<std::string, int>())
        .def(py::init<std::string>())
        // Attributes
        .def_readonly("name", &Cat::name)
        .def_readwrite("age", &Cat::age)
        // Methods
        .def("hasKnownAge", &Cat::hasKnownAge)
        // Methods as properties
        .def_property_readonly("hasKnownAgeProp",
            &Cat::hasKnownAge),
        // Equality operator
        .def(py::self == py::self); // __eq__
};
```

PyBind11: Function Overloads

```
std::string g( int ) { return "int"; }
std::string g( float ) { return "float"; }

PYBIND11_MODULE(example, m) {
    m.def("f", [](int x){ return "int"; });
    m.def("f", [](float x){ return "float"; });
    // The C++ function must be unique
    // To distinguish overloads, use py::overload_cast
    m.def("g", py::overload_cast<int>(g));
    m.def("g", py::overload_cast<float>(g));
};
```

PyBind11: Named Arguments

```
m.def("greet", [](const std::string& name, int times){
        for (int i = 0; i != times; i++)
            py::print("Hello " + name + ".")
    },
    // Docstring
    "Greet",
    // Argument definition
    py::arg("name"),
    py::arg("times") = 1); // Note the default value
```

PyBind11: args and kwargs

```
m.def("count_args", [](py::args a, py::kwargs kw) {
        py::print(a.size(), "args, ", kw.size(), " kwargs");
    });
```

PyBind11: Type conversion

- quite broad topic, see documentation
- already prepared conversions for
  - scalars,
  - `std::string`, `const char*`,
  - tuples, pairs,
  - containers (`std::vector`, `std::map`, ...),
  - `std::function` (accepts any Python function),
  - chrono,
  - `std::optional`

PyBind11: Python Native Types

- `py::object` (internal refcounting, owning)
- `py::handle` (no refcounting, non-owning)
- `py::module`, `py::function`
- `py::int_`, `py::float_`,
- `py::str`,
- `py::list`, `py::dict`, `py::slice`
- `...`

PyBind11: Return Value Policy

C++ uses different resource management compared to Python

- when a C++ function invoked from Python returns non-trivial value:
  - should the Python side keep track of the value and free it, or
  - will the C++ side take care of it?
- return value policy
  - `take_ownership` (Python handles lifetime)
  - `copy` (Python will make its own copy)
  - `move` (into Python's ownership)
  - `reference` (an existing object)
  - `automatic` (default one, see documentation for details)
  - several others, rather specialized

PyBind11: Templates

Templated functions and classes cannot be bind to a Python name – only to concrete instantiations.

```cpp
template < typename T >
class MyContainer { /* omitted */ };

using MyIntContainer = MyContainer<int>;
using MyPyContainer = MyContainer<py::object>;

PYBIND11_MODULE(example, m) {
    py::class_<MyContainer>(m, "Container"); // Invalid
    py::class_<MyIntContainer>(m, "IntContainer"); // Valid
    py::class_<MyPyContainer>(m, "Container"); // Valid,
        // can keep any Python object
};
```

PyBind11: Caveats

- binary compatibility
  - Python binary interface is incompatible between minor versions (e.g, 3.6 vs. 3.7)
  - module compiled with, e.g., Python 3.6 cannot be loaded in 3.7
  - makes distribution of precompiled packages painful
- can be little verbose when you need to export nearly all code
  - . . . but you can create nice Python interfaces!

Javascript Interpreter in C++

- nowadays, Google's V8 Javascript Engine is the best solution:
  - highly optimized, JIT support
  - well maintained (part of Chromium core, powering Node.js)
  - you can attach debuggers into the engine and step the user scripts
- usage of V8 is not covered by this lecture
- good starting points:
  - Official embedding guide
  - ruby0x1/v8-tutorials (older, however nice collection of examples)