# Interfacing C++ and Python with Boost.Python (https://flanusse.net/interfacing-c++-with-python.html)

**Date** Sat 01 April 2017 **Tags** tutorial (https://flanusse.net/tag/tutorial.html) / python (https://flanusse.net/tag/python.html) / c++ (https://flanusse.net/tag/c.html)

A large number of methods exist to interface Python and C/C++, the choice of a particular method mostly depend on the size and complexity of the codes you are trying to interface. From simple to more complex, I would first recommend ctypes (https://docs.python.org/2/library/ctypes.html) for quick and dirty interfacing with a handful of C functions. For wrapping a couple of C++ classes, the best approach is probably the popular SWIG (http://www.swig.org/) for "Simple Wrapper Interface Generator". Now, if you want to interface a whole C++ library, one of the most powerful options is Boost.Python (http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/index.html) , which is the solution I'm introducing in this tutorial. You can find a comparison of the relative benefits of SWIG vs Boost.Python on this LSST wiki (https://dev.lsstcorp.org/trac/wiki/SwigVsBoostPython). I should also mention pybind11 (https://pybind11.readthedocs.io/en/master/), a more recent and actively developed alternative to Boost.Python.

In this tutorial we will cover the following topics:

- Manually wrapping a simple C++ class using Boost.Python
- Compilation of Boost.Python project with CMake
- Automatic generation of Boost.Python wrapper using Py++

The source for all the examples presented here can be found on GitHub (https://github.com/EiffL/Tutorials/tree/master/PythonC%2B%2B)

## Interfacing Hello World

This first part of the tutorial demonstrates how to write a simple Boost.Python wrapper for a C++ class, and, maybe more importantly, how to get everything to compile and link properly.

### Simple C++ class

To get started, let's consider a very basic C++ class, that we will store in Bonjour.hpp (https://github.com/EiffL/Tutorials/blob/master/PythonC%2B%2B/Bonjour.hpp) :

```
#include <iostream>
#include <string>

class Bonjour
{
    // Private attribute
    std::string m_msg;
public:
    // Constructor
    Bonjour(std::string msg):m_msg(msg) { }

    // Methods
    void greet() { std::cout << m_msg << std::endl; }

    // Getter/Setter functions for the attribute
    void set_msg(std::string msg) { this->m_msg = msg; }
    std::string get_msg() const { return m_msg; }
};
```

## Boost.Python interface

Our goal will be to use Boost.Python to include this class in a module, that we will call `pylib`, which will be directly importable from Python. To do so, Boost.Python provides a C++ API which allows us to declare the classes and functions we wish to export to Python. These declarations are made in a .cpp interface file, which we will call pylib.cpp (https://github.com/EiffL/Tutorials/blob/master/PythonC%2B%2B/pylib.cpp) :

```
#include <boost/python.hpp>
#include "Bonjour.hpp"

using namespace boost::python;

BOOST_PYTHON_MODULE(pylib)
{
    class_< Bonjour >("Bonjour", init<std::string>())
      .def("greet", &Bonjour::greet)
      .add_property("msg", &Bonjour::get_msg, &Bonjour::set_msg);
}
```

Let's decrypt what's happening here. The `BOOST_PYTHON_MODULE` macro declares a Python module that will be called pylib. We can then add classes or functions to this module by adding the proper declarations between the parenthesis.

To add a class, we create a new `class_<>` object. The template of this object is the class that we want to export, in our case `class_<Bonjour>` will wrap our C++ class into Python. The first argument of the `class_<>()` constructor is the name that we want to use for this class in Python, here, we use the same name as in C++, "Bonjour", but it doesn't have to be. The second argument is used to define which C++ constructor to use. Here we only have one C++ constructor, but a C++ class can have many, which will differ by their prototypes. To identify which one to export, we use the `init<>` object which template corresponds to the prototype of the constructor, here a single argument of type `std::string`.

Declaring a class this way will just create an empty Python class, we then need to declare the methods and attributes that we wish to export from C++. To add a method, we use the `def()` function, which takes as a first argument the name of the method in Python, and as a second argument, a reference to the actual C++ method. Finally, we add a property to our Python class using `add_property()`, so that we can query and edit the content of the greeting message. The first argument of this function is the name of the property in Python, the next arguments are references to getter and setter functions in our C++ implementation.

See this page (http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/tutorial/tutorial/exposing.html) for more information on how to export classes.

## Compilation with CMake

Now that the interface file is written, we need to compile it. This compilation process involves in particular linking to the Python and Boost.Python libraries, which can be very painful (especially on MacOS X). To make the compilation as painless as possible, I will be using CMake to automatically identify the default C++ compiler, Python interpreter and download and build the required Boost modules.

We first need to install CMake, on an Ubuntu machine:

```
$ sudo apt-get install cmake
```

or whatever package manager you have on your system. If running on a cluster, you might have to install it locally from source, follow these instructions (https://cmake.org/install/).

Now, to compile our module we need to write a CMakeLists.txt (https://github.com/EiffL/Tutorials/blob/master/PythonC%2B%2B/CMakeLists.txt) file, which specifies the libraries required for compilation and defines the libraries and/or executables to build :

```
cmake_minimum_required(VERSION 2.8)
include(ExternalProject)

# Adding customized cmake module for building boost
list(APPEND CMAKE_MODULE_PATH  "${CMAKE_SOURCE_DIR}/cmake/Modules/")

project(tutorial)

  # Find default python libraries and interpreter
  find_package(PythonInterp REQUIRED)
  find_package(PythonLibs REQUIRED)
  include(BuildBoost) # Custom module

  include_directories(${Boost_INCLUDE_DIR} ${PYTHON_INCLUDE_DIRS})
  link_directories(${Boost_LIBRARY_DIR})

  # Build and link the pylib module
  add_library(pylib SHARED pylib.cpp)
  target_link_libraries(pylib ${Boost_LIBRARIES} ${PYTHON_LIBRARIES})
  add_dependencies(mylib Boost)

  # Tweaks the name of the library to match what Python expects
  set_target_properties(pylib PROPERTIES SUFFIX .so)
  set_target_properties(pylib PROPERTIES PREFIX "")
```

For this file to work, you need to copy locally my custom CMake module (https://github.com/EiffL/Tutorials/blob/master/PythonC%2B%2B/cmake/Modules/BuildBoost.cmake) for Boost, and put it in a *cmake/Modules* folder in the current directory. This file instructs CMake to compile our interface `pylib.cpp` into a library which will be named `pylib.so`. Note that you do not need to have Boost installed on your system, it will be automatically downloaded and compiled by CMake to match your specific setup.

Final step, compiling:

```
$ mkdir build   # Creates a folder for CMake to do its thing
$ cd build
$ cmake ..      # Runs CMake configuration and produces a MakeFile
$ make pylib    # Builds the library
```

And that's it, this should generate a file called pylib.so, which is a Python module directly loadable from the interpreter:

```
In [1]: from pylib import Bonjour

In [2]: b = Bonjour("Hello World")

In [3]: b.greet()
Hello World

In [4]: b.msg = "Bonjour tout le monde"

In [5]: b.greet()
Bonjour tout le monde

In [6]: b.msg
Out[6]: 'Bonjour tout le monde'
```

So, that's the basic idea of how to write and interface and compile it with CMake. Check the Boost.Python documentation (http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/index.html) for more details.

## Automated interface generation with Py++

Something that may already be apparent in the simple example considered above is that to wrap an entire C++ library in Boost.Python, someone has to declare every single method and property of every single C++ class, which can be a lot of work. Luckily, that someone doesn't have to be a Human. Instead one can rely on automated code generators, which will parse your C++ files and automatically create the corresponding Boost.Python declarations.

The official interface generator for Boost.Python used to be Pyste (http://www.boost.org/doc/libs/1_60_0/libs/python/pyste/), unfortunately it is no longer maintained and has been removed from current versions of Boost. It has now been superseded by a different project called Py++ (http://pyplusplus.readthedocs.io/en/latest/).

A word of warning here, installing and using Py++ in 2017 is a non trivial endeavor, most of the documentation and resources you can find online are out of date. The official documentation points to a stale branch of the project, from 2008. Luckily, the development of Py++ is not completely dead, an up to date version is being maintained as part of the Open Motion Planning Library OMPL (http://ompl.kavrakilab.org/) .

### Installing requirements

Py++ relies on CastXML (https://github.com/CastXML/CastXML) (formerly GCC-XML) a tool that parses C++ header files into an XML tree, and on pygccxml (http://pygccxml.readthedocs.io/en/develop/) to process the output XML files.

CastXML is available as a package at least in Ubuntu 16.04+ and MacPorts, to install:

```
$ sudo apt-get install castxml
```

Pygccxml can be installed through PyPI using pip:

```
$ pip install --user pygccxml
```

Finally, a working version of Py++ itself can be installed from the OMPL bitbucket repository (as of April 2017):

```
$ pip install --user https://bitbucket.org/ompl/pyplusplus/get/1.7.0.zip
```

## Writing a Py++ script

Py++ is a Python library which you can use in a Python script to parse the source files of your C++ project and produce the corresponding interface. Here is an example of such a script, that will automatically create the interface for our C++ class in a new module `pylib_auto` (to differentiate with the previous one). We will call this script pylib_generator.py (https://github.com/EiffL/Tutorials/blob/master/PythonC%2B%2B/pylib_generator.py) :

```
#!/usr/bin/python
from pygccxml import parser
from pyplusplus import module_builder

# Configurations that you may have to change on your system
generator_path = "/usr/bin/castxml"
generator_name = "castxml"
compiler = "gnu"
compiler_path = "/usr/bin/gcc"

# Create configuration for CastXML
xml_generator_config = parser.xml_generator_configuration_t(
                                    xml_generator_path=generator_path,
                                    xml_generator=generator_name,
                                    compiler=compiler,
                                    compiler_path=compiler_path)

# List of all the C++ header of our library
header_collection = ["Bonjour.hpp"]

# Parses the source files and creates a module_builder object
builder = module_builder.module_builder_t(
                        header_collection,
                        xml_generator_path=generator_path,
                        xml_generator_config=xml_generator_config)

# Automatically detect properties and associated getters/setters
builder.classes().add_properties(exclude_accessors=True)

# Define a name for the module
builder.build_code_creator(module_name="pylib_auto")

# Writes the C++ interface file
builder.write_module('pylib_auto.cpp')
```

This script should be mostly self-explanatory, the most important part is the creation of the `module_builder` which takes a list of C++ header files to parse. See the Py++ documentation (http://pyplusplus.readthedocs.io/en/latest/) for more indications on how to fine-tune the process (which can be necessary for more complex projects).

We can now parse our source code by running this script:

```
$ python pylib_generator.py
```

If everything works as expected, this script should create a new interface file pylib_auto.cpp (https://github.com/EiffL/Tutorials/blob/master/PythonC%2B%2B/pylib_auto.cpp) . To compile this new module, let's just add a new section at the end of our CMakeLists.txt:

```
# Build and link the pylib_auto module
add_library(pylib_auto SHARED pylib_auto.cpp)
target_link_libraries(pylib_auto ${Boost_LIBRARIES} ${PYTHON_LIBRARIES})
add_dependencies(pylib_auto Boost)

# Tweaks the name of the library to match what Python expects
set_target_properties(pylib_auto PROPERTIES SUFFIX .so)
set_target_properties(pylib_auto PROPERTIES PREFIX "")
```

Let's update the CMake configuration and compile:

```
$ cd build
$ cmake ..
$ make pylib_auto
```

We can now test our newly created module:

```
In [1]: from pylib_auto import Bonjour

In [2]: b = Bonjour("Hello World")

In [3]: b.greet()
Hello World

In [4]: b.msg = "Bonjour tout le monde"

In [5]: b.greet()
Bonjour tout le monde

In [6]: b.msg
Out[6]: 'Bonjour tout le monde'
```

And that's it, we have created an interface for our code without manually specifying anything, and it behaves just the same.

## Going Further

The goal of this tutorial was to illustrate the basics of Boost.Python, which is only scratching the surface. Unfortunately, the documentation surrounding Boost.Python is notoriously sparse, but here are some links you might find useful:

- Official Boost.Python documentation (http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/index.html)
- Official Py++ documentation (http://pyplusplus.readthedocs.io/en/latest/)

Oh, and did I mention that Boost.Python now has official support for interfacing with Numpy, through Boost.Numpy ?

- Boost.Numpy documentation (http://www.boost.org/doc/libs/1_63_0/libs/python/doc/html/numpy/index.html) (part of Boost.Python as of 1.63)

© 2017 François Lanusse · Powered by pelican-bootstrap3 (https://github.com/getpelican/pelican-themes/tree/master/pelican-bootstrap3), Pelican (http://docs.getpelican.com/), Bootstrap (http://getbootstrap.com)

Back to top

(https://creativecommons.org/licenses/by-nc/4.0/deed.en) Content licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (https://creativecommons.org/licenses/by-nc/4.0/deed.en), except where indicated otherwise.