Wrapping LSST C++ with pybind11

Russell Owen

Documentation

- LSST pybind11 Tutorial DMTN-026: https://dmtn-026.lsst.io
- pybind11 Documentation: pybind11.readthedocs.io/en/master/
- LSST pybind11 Style Guide DMTN-024: https://dmtn-024.lsst.io

Boilerplate: one .cc file per .h file

```
#include “pybind11/pybind11.h”
//#include “pybind11/stl.h” // uncomment if using STL

#include “lsst/your/code.h”

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

namespace lsst { namespace your { namespace code {

namespace {

// your helper functions here

}  // namespace lsst::your::code::<anonymous>

PYBIND11_PLUGIN(_example) {
  py::module mod("_example", "Example wrapper");

  // your wrapper code here

}

}}}  // namespace lsst::your::code
```

Simple Function

C++11:

```
int sfunc(int arg1, bool arg2=true);
```

pybind11:

```
mod.def("sfunc", &sfunc, "arg1"_a, "arg2"_a=true);
```

Overloaded Functions

C++11:

```
int olfunc(int arg1, bool arg2=true);
int olfunc(double & arg1);
```

pybind11:

```
mod.def("olfunc", (int (*)(int, bool)) &olfunc,
        "arg1"_a, "arg2"_a=true);
mod.def("olfunc" (int (*)(double &)) &olfunc,
        "arg1"_a);
```

Overloaded Functions

C++11:

```
int olfunc(int arg1, bool arg2=true);
int olfunc(double & arg1);
```

pybind11:

```
mod.def("olfunc", (int (*)(int, bool)) &olfunc,
        "arg1"_a, "arg2"_a=true);
mod.def("olfunc" (int (*)(double &)) &olfunc,
        "arg1"_a);
```

Overloaded Functions

C++11:

```
int olfunc(int arg1, bool arg2=true);
int olfunc(double & arg1);
```

pybind11:

```
mod.def("olfunc", (int (*)(int, bool)) &olfunc,
        "arg1"_a, "arg2"_a=true);
mod.def("olfunc" (int (*)(double &)) &olfunc,
        "arg1"_a);
```

Simple Class

C++11:

```
class Foo: public Base { …
  explicit Foo(int arg1, int arg2);
  explicit Foo(double splot=3.15);
  int munge(int data);
  static void strew(bool evenly);
```

pybind11:

```
py::class_<Foo, shared_ptr<Foo>, Base> cls(mod, "Foo");

// constructors
cls.def(py::init<int, int>(), "arg1"_a, "arg2"_a);
cls.def(py::init<double>(), "splot"_a=3.15);

// methods
cls.def("munge", &Foo::munge, "data"_a);
cls.def_static(“strew”, &Foo::strew, "evenly"_a);
```

Simple Class

C++11:

```
class Foo: public Base { …
  explicit Foo(int arg1, int arg2);
  explicit Foo(double splot=3.15);
  int munge(int data);
  static void strew(bool evenly);
```

pybind11:

if needed

```
py::class_<Foo, shared_ptr<Foo>, Base> cls(mod, "Foo");

// constructors
cls.def(py::init<int, int>(), "arg1"_a, "arg2"_a);
cls.def(py::init<double>(), "splot"_a=3.15);

// methods
cls.def("munge", &Foo::munge, "data"_a);
cls.def_static(“strew”, &Foo::strew, "evenly"_a);
```

Simple Class

C++11:

```
class Foo: public Base { …
  explicit Foo(int arg1, int arg2);
  explicit Foo(double splot=3.15);
  int munge(int data);
  static void strew(bool evenly);
```

pybind11:

```
py::class_<Foo, shared_ptr<Foo>, Base> cls(mod, "Foo");

// constructors
cls.def(py::init<int, int>(), "arg1"_a, "arg2"_a);
cls.def(py::init<double>(), "splot"_a=3.15);

// methods
cls.def("munge", &Foo::munge, "data"_a);
cls.def_static(“strew”, &Foo::strew, "evenly"_a);
```

Simple Class

C++11:

```
class Foo: public Base { …
  explicit Foo(int arg1, int arg2);
  explicit Foo(double splot=3.15);
  int munge(int data);
  static void strew(bool evenly);
```

pybind11:

```
py::class_<Foo, shared_ptr<Foo>, Base> cls(mod, "Foo");

// constructors
cls.def(py::init<int, int>(), "arg1"_a, "arg2"_a);
cls.def(py::init<double>(), "splot"_a=3.15);

// methods
cls.def("munge", &Foo::munge, "data"_a);
cls.def_static(“strew”, &Foo::strew, "evenly"_a);
```

Simple Class

C++11:

```
class Foo: public Base { …
  explicit Foo(int arg1, int arg2);
  explicit Foo(double splot=3.15);
  int munge(int data);
  static void strew(bool evenly);
```

pybind11:

```
py::class_<Foo, shared_ptr<Foo>, Base> cls(mod, "Foo");

// constructors
cls.def(py::init<int, int>(), "arg1"_a, "arg2"_a);
cls.def(py::init<double>(), "splot"_a=3.15);

// methods
cls.def("munge", &Foo::munge, "data"_a);
cls.def_static(“strew”, &Foo::strew, "evenly"_a);
```

Simple Class

C++11:

```
class Foo: public Base { …
  explicit Foo(int arg1, int arg2);
  explicit Foo(double splot=3.15);
  int munge(int data);
  static void strew(bool evenly);
```

pybind11:

```
py::class_<Foo, shared_ptr<Foo>, Base> cls(mod, "Foo");

// constructors
cls.def(py::init<int, int>(), "arg1"_a, "arg2"_a);
cls.def(py::init<double>(), "splot"_a=3.15);

// methods
cls.def("munge", &Foo::munge, "data"_a);
cls.def_static(“strew”, &Foo::strew, "evenly"_a);
```

C++ Operators

Like methods, but use **py::is_operator()** to get Python-like type checking:

```
cls.def("__eq__", &Foo::operator==, py::is_operator());
cls.def("__ne__", &Foo::operator!=, py::is_operator());
```

C++11 Lambdas

Use C++11 lambdas to add behavior:

```
cls.def("__repr__", [](Foo const & self) {
  std::ostringstream os;
  os << self;
  return os.str();
});
```

You can also add methods to wrapped classes in Python

Helper Functions

Use helper functions:

- to wrap templated classes
- when one header file defines many classes

Put helper functions in an anonymous namespace

Templated Class 1/2

C++11:

```
template <typename T>
class Foo: { …
  explicit Foo(bool flag);
  int munge(T data);
```

pybind11: write a helper function

```
template <typename T>
void declareFoo(py::module & mod, string const & suffix) {
  py::class_<Foo<T>> cls(mod, ("Foo" + suffix).c_str());
  cls.def(py::init<bool>(), "flag"_a);
  cls.def("munge", &Foo<T>::munge, "data"_a);
}
```

Templated Class 2/2

Put the helper in an anonymous namespace and call it:

```
namespace lsst { namespace your { namespace code {

namespace {

template <typename T>
void declareFoo(py::module & mod, string const & suffix) {
 …
}

}  // namespace lsst::your::code::<anonymous>

PYBIND11_PLUGIN(_example) {
  py::module mod("_example", "Example wrapper");

  declareFoo<double>(mod, "D");
```

Enums

C++11:

```
class La { …
  enum Mode {FAST, SLOW};
```

pybind11:

```
py::class_<La> cls(mod, "La");

py::enum_<La::Mode>(cls, "Mode")
  .value("FAST", La::Mode::FAST)
  .value("SLOW", La::Mode::SLOW)
  .export_values();
```

Notes:

- pybind11 enums are not ints
- `py::arithmetic()` is a useful option for bit field enums

Fields and Properties

Fields:

```
def_readwrite("name", &Foo::name);
def_readonly("constname", &Foo::constname);
```

Accessors as Properties:

```
cls.def_property("data", &Foo::getData, &Foo::setData);
cls.def_property_readonly(“flag”, &Foo::getFlag);
```

Static Fields and Properties:

Append **_static**, e.g.: `cls.def_readwrite_static`

Overloaded Methods

C++:

```
class Foo { …
  void hey(double arg) const;
  void hey(int arg1, bool flag);
  static void gack(bool ok);
  static void gack(int x, int y);
```

pybind11:

```
// Use (Class::*) for a normal method
cls.def("hey", (void (Foo::*)(double) const) &Foo::hey);
cls.def("hey", (void (Foo::*)(int, bool)) &Foo::hey);

// Use (*) for a static method
cls.def_static(“gack", (void (*)(bool)) &Foo::gack);
cls.def_static(“gack", (void (*)(int, int)) &Foo::gack);
```

Return Value Policies

When C++ returns a reference or pointer:

```
class Foo {…
  double & getValue(int index);
  Bar * getBar(std::string const & name);
```

you must specify: copy, live reference or…?

```
mod.def("getValue", &Foo::getValue, "index"_a,
        py::return_value_policy::copy);
mod.def("getBar", &Foo::getBar, “name”_a,
        py::return_value_policy::reference_internal);
```