cppyy Documentation

Release 1.4.0

Wim Lavrijsen

Mar 23, 2019

Getting Started

cppyy is an automatic Python-C++ bindings generator, for calling C++ from Python and Python from C++, designed for large scale programs in high performance computing that use modern C++. Design and performance are described in this PyHPC paper, albeit that the CPython/cppyy performance has been vastly improved since.

cppyy is based on Cling, the C++ interpreter, to match Python's dynamism and interactivity. Consider this session, showing dynamic, interactive, mixing of C++ and Python features (more examples are in the tutorial):

```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)
>>> cppyy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppyy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
>>> class PyMyClass(MyClass):
...     def add_int(self, i):  # python side override (CPython only)
...         return self.m_data + 2*i
...
>>> cppyy.cppdef("int callback(MyClass* m, int i) { return m->add_int(i); }")
True
>>> cppyy.gbl.callback(m, 2)             # calls C++ add_int
15
>>> cppyy.gbl.callback(PyMyClass(1), 2)  # calls Python-side override
5
>>>
```

With a modern C++ compiler having its back, cppyy is future-proof. Consider the following session using `boost::any`, a capsule-type that allows for heterogeneous containers in C++. The Boost library is well known for its no holds barred use of modern C++ and heavy use of templates:

```
>>> import cppyy
>>> cppyy.include('boost/any.hpp')
>>> from cppyy.gbl import std, boost
>>> val = boost.any()                    # the capsule
>>> val.__assign__(std.vector[int]())    # assign it a std::vector<int>
<cppyy.gbl.boost.any object at 0xf6a8a0>
>>> val.type() == cppyy.typeid(std.vector[int])    # verify type
True
>>> extract = boost.any_cast[int](std.move(val))   # wrong cast
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: int boost::any_cast(boost::any&& operand) =>
```

(continues on next page)

(continued from previous page)

```
    boost::bad_any_cast: failed conversion using boost::any_cast (C++ exception)
>>> extract = boost.any_cast[std.vector[int]](val) # correct cast
>>> type(extract) is std.vector[int]
True
>>> extract += xrange(100)
>>> len(extract)
100
>>> val.__assign__(std.move(extract))     # move forced
<cppyy.gbl.boost.any object at 0xf6a8a0>
>>> len(extract)                          # now empty (or invalid)
0
>>> extract = boost.any_cast[std.vector[int]](val)
>>> list(extract)
[0, 1, 2, 3, 4, 5, 6, ..., 97, 98, 99]
>>>
```

And yes, there is no reason to use Boost from Python (in fact, this example calls out for *pythonizations*), but it shows that cppyy seamlessly supports many advanced C++ features.

cppyy is available for both CPython (v2 and v3) and PyPy, reaching C++-like performance with the latter. It makes judicious use of precompiled headers, dynamic loading, and lazy instantiation, to support C++ programs consisting of millions of lines of code and many thousands of classes. cppyy minimizes dependencies to allow its use in distributed, heterogeneous, development environments.

CHAPTER 1

Installation

The cppyy module and its dependencies are available through PyPI for both CPython (2 and 3) and PyPy (5.9.0 and later). The cleanest/easiest way to install cppyy is using virtualenv and pip:

```
$ virtualenv WORK
$ source WORK/bin/activate
(WORK) $ python -m pip install cppyy
```

The use of virtualenv prevents pollution of any system directories and allows you to wipe out the full installation simply by removing the virtualenv created directory:

```
$ rm -rf WORK
```

If you use anaconda, it is recommended to use an anaconda python and (modern) c++ compiler, rather than the system ones. For example:

```
$ conda create -n WORK
$ conda activate WORK
(WORK) $ conda install python
(WORK) $ conda install -c conda-forge cxx-compiler
(WORK) [x86_64-conda_cos6-linux-gnu] $ python -m pip install cppyy
```

Wheels for the backend are available for GNU/Linux, MacOS-X, and MS Windows (support for MS Windows is in beta, 32b Windows support in vastly better shape than 64b Windows). The Linux wheels are built on manylinux, but with gcc 5.5, not the 4.8.2 that ships with manylinux, since cppyy exposes C++ APIs. Using 4.8.2 would have meant that any software using cppyy would have to be (re)compiled for the older gcc ABI, which the odds don't favor. Note that building cppyy with 4.8.2 (and requiring the old ABI) works fine, but would only support C++11.

The CPyCppyy and cppyy packages can not produce wheels as they must be build locally in order to match the local compiler and system files and CPU features (e.g. AVX). Use the CC and CXX envars to set any local compiler if you do not want to use the system compiler.

The C++17 standard is the default for Mac and Linux wheels; it is C++14 for Windows (compiler limitation). When building from source, the highest version among 17, 14, and 11 that your native compiler supports will be chosen. You can control the standard selection by setting the STDCXX envar to '17', '14', or '11' (for Linux, the backend does

not need to be recompiled). When building from source, build-time only dependencies are `cmake` (for general build), `python` (obviously, but also for LLVM), and a modern C++ compiler (one that supports at least C++11).

Compilation of the backend from source, which contains a customized version of Clang/LLVM, can take a long time, so by default the setup script will use all cores (x2 if hyperthreading is enabled). To change that behavior, set the MAKE_NPROCS environment variable to the desired number of processes to use. To see progress while waiting, use `--verbose`:

```
$ STDCXX=17 MAKE_NPROCS=32 pip install --verbose cppyy
```

The bdist_wheel of the backend is reused by pip for all versions of CPython and PyPy, thus the long compilation is needed only once for all different versions of Python on the same machine. Unless you build on the manylinux1 docker images, wheels for `cppyy-backend` and for `CPyCppyy` are disabled, because `setuptools` (as used by `pip`) does not properly resolve dependencies for wheels. You will see a harmless "error" message to that effect fly by in the verbose output.

On Windows, some temporary path names may be too long, causing the build to fail. To resolve this issue, set the `TMP` and `TEMP` envars to something short, before building. For example:

```
> set TMP=C:\TMP
> set TEMP=C:\TMP
```

If you use the `--user` option to pip and use pip directly on the command line, make sure that the PATH envar points to the bin directory that will contain the installed entry points during the installation, as the build process needs them. You may also need to install `wheel` first, if you have an older version of pip and/or do not use virtualenv (which installs wheel by default). Example:

```
$ python -m pip install wheel --user
$ PATH=$HOME/.local/bin:$PATH python -m pip install cppyy --user
```

PyPy 5.7 and 5.8 have a built-in module `cppyy`. You can still install the `cppyy` package, but the built-in module takes precedence. To use `cppyy`, first import a compatibility module:

```
$ pypy
[PyPy 5.8.0 with GCC 5.4.0] on linux2
>>>> import cppyy_compat, cppyy
>>>>
```

You will have to set `LD_LIBRARY_PATH` appropriately if you get an `EnvironmentError` (it will indicate the needed directory).

Note that your python interpreter (whether CPython or `pypy-c`) may not have been linked by the C++ compiler. This can lead to problems during loading of C++ libraries and program shutdown. In that case, re-linking is highly recommended.

Older versions of PyPy (5.6.0 and earlier) have a built-in `cppyy` based on Reflex, which is less feature-rich and no longer supported. However, both the *distribution tools* and user-facing Python codes are very backwards compatible.

1.1 Precompiled Header

For performance reasons (reduced memory and CPU usage), a precompiled header (PCH) of the system and compiler header files will be installed or, failing that, generated on startup. Obviously, this PCH is not portable and should not be part of a wheel.

Some compiler features, such as AVX, OpenMP, fast math, etc. need to be active during compilation of the PCH, as they depend both on the compiler flag and system headers (for intrinsics, or API calls). You can control compiler flags

through the EXTRA_CLING_ARGS envar and thus what is active in the PCH. In principle, you can also change the C++ language standard by setting the appropriate flag on EXTRA_CLING_ARGS and rebuilding the PCH.

If you want multiple PCHs living side-by-side, you can generate them yourself (note that the given path must be absolute):

```
>>> import cppyy_backend.loader as l
>>> l.set_cling_compile_options(True)          # adds defaults to EXTRA_CLING_ARGS
>>> install_path = '/full/path/to/target/location/for/PCH'
>>> l.ensure_precompiled_header(install_path)
```

You can then select the appropriate PCH with the CLING_STANDARD_PCH envar:

```
$ export CLING_STANDARD_PCH=/full/path/to/target/location/for/PCH/allDict.cxx.pch
```

CHAPTER 2

Changelog

For convenience, this changelog keeps tracks of changes with version numbers of the main cppyy package, but many of the actual changes are in the lower level packages, which have their own releases. See *packages*, for details on the package structure. PyPy support lags CPython support.

2.1 MASTER : 1.4.5

- allow templated free functions to be attached as methods to classes

2.2 2019-03-20 : 1.4.4

- Support for 'using' of namespaces
- Improved support for alias templates
- Faster template lookup
- Have rootcling/genreflex respect compile-time flags (except for –std if overridden by CLING_EXTRA_FLAGS)
- Utility to build dictionarys on Windows (32/64)
- Name mangling fixes in Cling for JITed global/static variables on Windows
- Several pointer truncation fixes for 64b Windows

2.3 2019-03-10 : 1.4.3

- Cross-inheritance from abstract C++ base classes
- Preserve 'const' when overriding virtual functions
- Support for by-ref (using ctypes) for function callbacks

- Identity of nested typedef'd classes matches actual
- Expose function pointer variables as std::function's
- More descriptive printout of global functions
- Ensure that standard pch is up-to-date and that it is removed on uninstall
- Remove standard pch from wheels on all platforms
- Add -cxxflags option to rootcling
- Install clang resource directory on Windows

CHAPTER 3

Basic Types

C++ has a far richer set of builtin types than Python. Most Python code can remain relatively agnostic to that, and `cppyy` provides automatic conversions as appropriate. On the other hand, Python builtin types such as lists and maps are far richer than any builtin types in C++. These are mapped to their Standard Template Library equivalents instead.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

3.1 *Builtins*

Most builtin data types map onto the expected equivalent Python types, with the caveats that there may be size differences, different precision or rounding. For example, a C++ `float` is returned as a Python `float`, which is in fact a C++ `double`. If sizes allow, conversions are automatic. For example, a C++ `unsigned int` becomes a Python `long`, but unsigned-ness is still honored:

```
>>> type(cppyy.gbl.gUint)
<type 'long'>
>>> cppyy.gbl.gUint = -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot convert negative integer to unsigned
>>>
```

3.2 *Arrays*

Builtin arrays are supported through arrays from module `array` (or any other builtin-type array that implements the Python buffer interface, such as numpy arrays) and a low-level view type from `cppyy` for returns and variable access

(that implements the buffer interface as well). Out-of-bounds checking is limited to those cases where the size is known at compile time. Example:

```
>>> from cppyy.gbl import Concrete
>>> from array import array
>>> c = Concrete()
>>> c.array_method(array('d', [1., 2., 3., 4.]), 4)
1 2 3 4
>>> c.m_data[4] # static size is 4, so out of bounds
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: buffer index out of range
>>>
```

3.3 *Pointers*

When the C++ code takes a pointer or reference type to a specific builtin type (such as an `unsigned int` for example), then types need to match exactly. `cppyy` supports the types provided by the standard modules `ctypes` and `array` for those cases. Example of using a reference to builtin:

```
>>> from ctypes import c_uint
>>> u = c_uint(0)
>>> c.uint_ref_assign(u, 42)
>>> u.value
42
>>>
```

For objects, a pointer to an object and an object are represented the same way, with the necessary (de)referencing applied automatically. Pointer variables are also bound by reference, so that updates on either the C++ or Python side are reflected on the other side as well.

3.4 *Enums*

Both named and anonymous enums are supported. The type of an enum is implementation dependent and may even be different for different enums on the same compiler. Typically, however, the types are `int` or `unsigned int`, which translates to Python's `int` or `long` on Python2 or class `int` on Python3:

```
>>> from cppyy.gbl import kApple, kBanana, kCitrus
>>> cppyy.gbl.kApple
78
>>>
```

CHAPTER 4

Classes

Both Python and C++ support object-oriented code through classes and thus it is logical to expose C++ classes as Python ones, including the full inheritance hierarchy.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

4.1 *Basics*

All bound C++ code starts off from the global C++ namespace, represented in Python by `gbl`. This namespace, as any other namespace, is treated as a module after it has been loaded. Thus, we can import C++ classes that live underneath it:

```
>>> from cppyy.gbl import Concrete
>>> Concrete
<class cppyy.gbl.Concrete at 0x2058e30>
>>>
```

Placing classes in the same structure as imposed by C++ guarantees identity, even if multiple Python modules bind the same class. There is, however, no necessity to expose that structure to end-users: when developing a Python package that exposes C++ classes through `cppyy`, consider `cppyy.gbl` an "internal" module, and expose the classes in any structure you see fit. The C++ names will continue to follow the C++ structure, however, as is needed for e.g. pickling:

```
>>> from cppyy.gbl import Namespace
>>> Concrete == Namespace.Concrete
False
>>> n = Namespace.Concrete.NestedClass()
>>> type(n)
```

(continues on next page)

(continued from previous page)

```
<class cppyy.gbl.Namespace.Concrete.NestedClass at 0x22114c0>
>>> type(n).__name__
NestedClass
>>> type(n).__module__
cppyy.gbl.Namespace.Concrete
>>> type(n).__cppname__
Namespace::Concrete::NestedClass
>>>
```

4.2 *Inheritance*

The output of help shows the inheritance hierarchy, constructors, public methods, and public data. For example, `Concrete` inherits from `Abstract` and it has a constructor that takes an `int` argument, with a default value of 42. Consider:

```
>>> from cppyy.gbl import Abstract
>>> issubclass(Concrete, Abstract)
True
>>> a = Abstract()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: cannot instantiate abstract class 'Abstract'
>>> c = Concrete()
>>> isinstance(c, Concrete)
True
>>> isinstance(c, Abstract)
True
>>> d = Concrete(13)
>>>
```

Just like in C++, interface classes that define pure virtual methods, such as `Abstract` does, can not be instantiated, but their concrete implementations can. As the output of `help` showed, the `Concrete` constructor takes an integer argument, that by default is 42.

4.3 *Cross-inheritance*

Python classes that derive from C++ classes can override virtual methods as long as those methods are declared on class instantiation (adding methods to the Python class after the fact will not provide overrides on the C++ side, only on the Python side). Example:

```
>>> from cppyy.gbl import Abstract, call_abstract_method
>>> class PyConcrete(Abstract):
...     def abstract_method(self):
...         print("Hello, Python World!\n")
...     def concrete_method(self):
...         pass
...
>>> pc = PyConcrete()
>>> call_abstract_method(pc)
Hello, Python World!
>>>
```

Note that it is not necessary to provide a constructor (`__init__`), but if you do, you *must* call the base class constructor through the `super` mechanism.

4.4 Typedefs

Typedefs are simple python references to the actual classes to which they refer.

```
>>> from cppyy.gbl import Concrete_t
>>> Concrete is Concrete_t
True
>>>
```

4.5 Data members

The `Concrete` instances have a public data member `m_int` that is treated as a Python property, albeit a typed one:

```
>>> c.m_int, d.m_int
(42, 13)
>>> c.m_int = 3.14   # a float does not fit in an int
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: int/long conversion expects an integer object
>>> c.m_int = int(3.14)
>>> c.m_int, d.m_int
(3, 13)
>>>
```

Note that private and protected data members are not accessible and C++ const-ness is respected:

```
>>> c.m_const_int = 71    # declared 'const int' in class definition
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: assignment to const data not allowed
>>>
```

Static C++ data members act like Python class-level data members. They are also represented by property objects and both read and write access behave as expected:

```
>>> Concrete.s_int       # access through class
321
>>> c.s_int = 123        # access through instance
>>> Concrete.s_int
123
```

4.6 Methods

C++ methods are represented as Python ones: these are first-class objects and can be bound to an instance. If a method is virtual in C++, the proper concrete method is called, whether or not the concrete class is bound. Similarly, if all classes are bound, the normal Python rules apply:

```
>>> c.abstract_method()
called Concrete::abstract_method
>>> c.concrete_method()
called Concrete::concrete_method
>>> m = c.abstract_method
>>> m()
called Concrete::abstract_method
>>>
```

4.7 *Templates*

Templated classes are instantiated using square brackets. (For backwards compatibility reasons, parentheses work as well.) The instantiation of a templated class yields a class, which can then be used to create instances.

Templated classes need not pre-exist in the bound code, just their declaration needs to be available. This is true for e.g. all of STL:

```
>>> cppyy.gbl.std.vector                # template metatype
<cppyy.Template 'std::vector' object at 0x7fffed2674d0>
>>> cppyy.gbl.std.vector(int)           # instantiates template -> class
<class cppyy.gbl.std.vector<int> at 0x1532190>
cppyy.gbl.std.vector[int]()             # instantiates class -> object
<cppyy.gbl.std.vector<int> object at 0x2341ec0>
>>>
```

The template arguments may be actual types or their names as a string, whichever is more convenient. Thus, the following are equivalent:

```
>>> from cppyy.gbl.std import vector
>>> type1 = vector[Concrete]
>>> type2 = vector['Concrete']
>>> type1 == type2
True
>>>
```

CHAPTER 5

Functions

C++ functions are first-class objects in Python and can be used wherever Python functions can be used, including for dynamically constructing classes.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

5.1 *Free functions*

All bound C++ code starts off from the global C++ namespace, represented in Python by `gbl`. This namespace, as any other namespace, is treated as a module after it has been loaded. Thus, we can directly import C++ functions that live underneath it.

```
>>> from cppyy.gbl import global_function, Namespace
>>> global_function == Namespace.global_function
False
>>>
```

C++ supports overloading, whereas Python supports "duck typing", so C++ overloads have to be selected dynamically:

```
>>> global_function(1.)         # selects 'double' overload
2.718281828459045
>>> global_function(1)          # selects 'int' overload
42
>>>
```

C++ does a static dispatch at compile time based on the argument types. The dispatch is a selection among overloads (incl. templates) visible at that point in the translation unit. Bound C++ in Python does a dynamic dispatch: it considers all overloads visible _globally_ at that point in the execution. Because the dispatch is fundamentally different (albeit

in line with the expectation of the respective languages), differences can occur. Especially if overloads live in different header files and are only an implicit conversion apart.

If the overload selection fails in a specific case, the `__overload__` function can be called directly with a signature:

```
>>> global_function.__overload__('double')(1)    # int implicitly converted
2.718281828459045
>>>
```

5.2 *args and **kwds

C++ default arguments work as expected, but python keywords are not (yet) supported. (It is technically possible to support keywords, but for the C++ interface, the formal argument names have no meaning and are not considered part of the API, hence it is not a good idea to use keywords.) Example:

```
>>> from cppyy.gbl import Concrete
>>> c = Concrete()        # uses default argument
>>> c.m_int
42
>>> c = Concrete(13)      # uses provided argument
>>> c.m_int
13
>>> args = (27,)
>>> c = Concrete(*args)   # argument pack
>>> c.m_int
27
>>>
```

5.3 Callbacks

Python callables (functions/lambdas/instances) can be passed to C++ through function pointers and/or `std::function`. This involves creation of a temporary wrapper, which has the same life time as the Python callable it wraps, so the callable needs to be kept alive on the Python side if the C++ side stores the callback. Example:

```
>>> from cppyy.gbl import call_int_int
>>> print(call_int_int.__doc__)
int ::call_int_int(int(*)(int,int) f, int i1, int i2)
>>> def add(a, b):
...     return a+b
...
>>> call_int_int(add, 3, 7)
7
>>> call_int_int(lambda x, y: x*y, 3, 7)
21
>>>
```

CHAPTER 6

Type Conversions

Most type conversions are done implicitly, e.g. between Python `str` and C++ `std::string` and `const char*`, but low-level APIs exist to perform explicit conversions.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

6.1 *Casting*

Object pointer returns from functions provide the most derived class known (i.e. exposed in header files) in the hierarchy of the object being returned. This is important to preserve object identity as well as to make casting, a pure C++ feature after all, superfluous. Example:

```
>>> from cppyy.gbl import Abstract, Concrete
>>> c = Concrete()
>>> Concrete.show_autocast.__doc__
'Abstract* Concrete::show_autocast()'
>>> d = c.show_autocast()
>>> type(d)
<class '__main__.Concrete'>
>>>
```

As a consequence, if your C++ classes should only be used through their interfaces, then no bindings should be provided to the concrete classes (e.g. by excluding them using a *selection file*). Otherwise, more functionality will be available in Python than in C++.

Sometimes, however, full control over a cast is needed. For example, if the instance is bound by another tool or even a 3rd party, hand-written, extension library. Assuming the object supports the `PyCapsule` or `CObject` abstraction,

then a C++-style reinterpret_cast (i.e. without implicitly taking offsets into account), can be done by taking and rebinding the address of an object:

```
>>> from cppyy import addressof, bind_object
>>> e = bind_object(addressof(d), Abstract)
>>> type(e)
<class '__main__.Abstract'>
>>>
```

6.2 *Operators*

If conversion operators are defined in the C++ class and a Python equivalent exists (i.e. all builtin integer and floating point types, as well as `bool`), then these will map onto those Python conversions. Note that `char*` is mapped onto `__str__`. Example:

```
>>> from cppyy.gbl import Concrete
>>> print(Concrete())
Hello operator const char*!
>>>
```

C++ code can overload conversion operators by providing methods in a class or global functions. Special care needs to be taken for the latter: first, make sure that they are actually available in some header file. Second, make sure that headers are loaded in the desired order. I.e. that these global overloads are available before use.

CHAPTER 7

Python

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

7.1 *PyObject*

Arguments and return types of `PyObject*` can be used, and passed on to CPython API calls (or through `cpyext` in PyPy).

7.2 *Doc Strings*

The documentation string of a method or function contains the C++ arguments and return types of all overloads of that name, as applicable. Example:

```
>>> from cppyy.gbl import Concrete
>>> print Concrete.array_method.__doc__
void Concrete::array_method(int* ad, int size)
void Concrete::array_method(double* ad, int size)
>>>
```

7.3 *Help*

Bound C++ class is first-class Python and can thus be inspected like any Python objects can. For example, we can ask for `help()`:

```
>>> help(Concrete)
Help on class Concrete in module gbl:

class Concrete(Abstract)
 |  Method resolution order:
 |      Concrete
 |      Abstract
 |      CPPInstance
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __assign__(self, const Concrete&)
 |      Concrete& Concrete::operator=(const Concrete&)
 |
 |  __init__(self, *args)
 |      Concrete::Concrete(int n = 42)
 |      Concrete::Concrete(const Concrete&)
 |
 etc. ....
```

CHAPTER 8

Miscellaneous

8.1 File features.h

```
#include <cmath>
#include <iostream>
#include <vector>


//-----
unsigned int gUint = 0;

//-----
class Abstract {
public:
    virtual ~Abstract() {}
    virtual void abstract_method() = 0;
    virtual void concrete_method() = 0;
};

void Abstract::concrete_method() {
    std::cout << "called Abstract::concrete_method" << std::endl;
}

//-----
class Concrete : Abstract {
public:
    Concrete(int n=42) : m_int(n), m_const_int(17) {}
    ~Concrete() {}

    virtual void abstract_method() {
        std::cout << "called Concrete::abstract_method" << std::endl;
    }

    virtual void concrete_method() {
```

(continues on next page)

(continued from previous page)

```
        std::cout << "called Concrete::concrete_method" << std::endl;
    }

    void array_method(int* ad, int size) {
        for (int i=0; i < size; ++i)
            std::cout << ad[i] << ' ';
        std::cout << std::endl;
    }

    void array_method(double* ad, int size) {
        for (int i=0; i < size; ++i)
            std::cout << ad[i] << ' ';
        std::cout << std::endl;
    }

    void uint_ref_assign(unsigned int& target, unsigned int value) {
        target = value;
    }

    Abstract* show_autocast() {
        return this;
    }

    operator const char*() {
        return "Hello operator const char*!";
    }

public:
    double m_data[4];
    int m_int;
    const int m_const_int;

    static int s_int;
};

typedef Concrete Concrete_t;

int Concrete::s_int = 321;

void call_abstract_method(Abstract* a) {
    a->abstract_method();
}

//-----
int global_function(int) {
    return 42;
}

double global_function(double) {
    return std::exp(1);
}

int call_int_int(int (*f)(int, int), int i1, int i2) {
    return f(i1, i2);
}

//-----
```

(continues on next page)

(continued from previous page)

```
namespace Namespace {

    class Concrete {
    public:
        class NestedClass {
        public:
            std::vector<int> m_v;
        };

    };

    int global_function(int i) {
        return 2*::global_function(i);
    }

    double global_function(double d) {
        return 2*::global_function(d);
    }

} // namespace Namespace

//-----
enum EFruit {kApple=78, kBanana=29, kCitrus=34};
```

This is a collection of a few more features listed that do not have a proper place yet in the rest of the documentation.

The C++ code used for the examples below can be found *here*, and it is assumed that that code is loaded at the start of any session. Download it, save it under the name `features.h`, and load it:

```
>>> import cppyy
>>> cppyy.include('features.h')
>>>
```

8.2 Odds and ends

- **memory**: C++ instances created by calling their constructor from python are owned by python. You can check/change the ownership with the __python_owns__ flag that every bound instance carries. Example:

```
>>> from cppyy.gbl import Concrete
>>> c = Concrete()
>>> c.__python_owns__          # True: object created in Python
True
>>>
```

- **namespaces**: Are represented as python classes. Namespaces are more open-ended than classes, so sometimes initial access may result in updates as data and functions are looked up and constructed lazily. Thus the result of `dir()` on a namespace shows the classes available, even if they may not have been created yet. It does not show classes that could potentially be loaded by the class loader. Once created, namespaces are registered as modules, to allow importing from them. Namespace currently do not work with the class loader. Fixing these bootstrap problems is on the TODO list. The global namespace is `cppyy.gbl`.
- **NULL**: Is represented as `cppyy.nullptr`. In C++11, the keyword `nullptr` is used to represent `NULL`. For clarity of intent, it is recommended to use this instead of `None` (or the integer `0`, which can serve in some cases), as `None` is better understood as `void` in C++.

- **static methods**: Are represented as python's `staticmethod` objects and can be called both from the class as well as from instances.
- **templated functions**: Automatically participate in overloading and are used in the same way as other global functions.
- **templated methods**: For now, require an explicit selection of the template parameters. This will be changed to allow them to participate in overloads as expected.
- **unary operators**: Are supported if a python equivalent exists, and if the operator is defined in the C++ class.

CHAPTER 9

Pythonizations

Automatic bindings generation mostly gets the job done, but unless a C++ library was designed with expressiveness and interactivity in mind, using it will feel stilted. Thus, if you are not the end-user of a set of bindings, it is beneficial to implement *pythonizations*. Some of these are already provided by default, e.g. for STL containers. Consider the following code, iterating over an STL map, using naked bindings (i.e. "the C++ way"):

```
>>> from cppyy.gbl import std
>>> m = std.map[int, int]()
>>> for i in range(10):
...     m[i] = i*2
...
>>> b = m.begin()
>>> while b != m.end():
...     print(b.__deref__().second, end=' ')
...     b.__preinc__()
...
0 2 4 6 8 10 12 14 16 18
>>>
```

Yes, that is perfectly functional, but it is also very clunky. Contrast this to the (automatic) pythonization:

```
>>> for key, value in m:
...    print(value, end=' ')
...
0 2 4 6 8 10 12 14 16 18
>>>
```

Such a pythonization can be written completely in python using the bound C++ methods, with no intermediate language necessary. Since it is written on abstract features, there is also only one such pythonization that works for all STL map instantiations.

9.1 Installing callbacks

CHAPTER 10

Dictionaries

Loading code directly into Cling is fine for interactive work and small scripts, but large scale applications should take advantage of pre-packaging code, linking in libraries, and describing other dependencies. The necessary tools are installed as part of the backend.

10.1 Dictionary generation

A "reflection dictionary" makes it simple to combine the necessary headers and libraries into a single package for use and distribution. The relevant headers are read by a tool called genreflex which generates C++ files that are to be compiled into a shared library. That library can further be linked with any relevant project libraries that contain the implementation of the functionality declared in the headers. For example, given a file called `project_header.h` and an implementation residing in `libproject.so`, the following will generate a `libProjectDict.so` reflection dictionary:

```
$ genreflex project_header.h
$ g++ -std=c++11 -fPIC -rdynamic -O2 -shared `genreflex --cppflags` project_header_
↪rflx.cpp -o libProjectDict.so -L$PROJECTHOME/lib -lproject
```

Instead of loading the header text into Cling, you can now load the dictionary:

```
>>> import cppyy
>>> cppyy.load_reflection_info('libProjectDict.so')
<CPPLibrary object at 0xb6fd7c4c>
>>> from cppyy.gbl import SomeClassFromProject
>>>
```

and use the C++ entities from the header as before.

10.2 Automatic class loader

Explicitly loading dictionaries is fine if this is hidden under the hood of a Python package and thus simply done on import. Otherwise, the automatic class loader is more convenient, as it allows direct use without having to manually find and load dictionaries.

The class loader utilizes so-called rootmap files, which by convention should live alongside the dictionaries in places reachable by LD_LIBRARY_PATH. These are simple text files, which map C++ entities (such as classes) to the dictionaries and other libraries that need to be loaded for their use.

The `genreflex` tool can produce rootmap files automatically. For example:

```
$ genreflex project_header.h --rootmap=libProjectDict.rootmap --rootmap-
↪lib=libProjectDict.so
$ g++ -std=c++11 -fPIC -rdynamic -O2 -shared `genreflex --cppflags` project_header_
↪rflx.cpp -o libProjectDict.so -L$CPPYYHOME/lib -lCling -L$PROJECTHOME/lib -lproject
```

where the first option (`--rootmap`) specifies the output file name, and the second option (`--rootmap-lib`) the name of the reflection library. It is necessary to provide that name explicitly, since it is only in the separate linking step where these names are fixed (if the second option is not given, the library is assumed to be libproject_header.so).

With the rootmap file in place, the above example can be rerun without explicit loading of the reflection info library:

```
>>> import cppyy
>>> from cppyy.gbl import SomeClassFromProject
>>>
```

10.3 Selection files

Sometimes it is necessary to restrict or expand what genreflex will pick up from the header files. For example, to add or remove standard classes or to hide implementation details. This is where selection files come in. These are XML specifications that allow exact or pattern matching to classes, functions, etc. See `genreflex --help` for a detailed specification and add `--selection=project_selection.xml` to the `genreflex` command line.

With the aid of a selection file, a large project can be easily managed: simply `#include` all relevant headers into a single header file that is handed to `genreflex`.

CHAPTER 11

Bindings Generation

Binding developers have two levels of access to cling via `cppyy`:

- A simple command line interface.
- Automated generation of an end-user bindings package from a CMake-based project build.

Both are installed as part of the `cppyy-backend` component since they are not needed by users of the bindings.

11.1 Command Line Interface

11.1.1 rootcling

This provides basic access to `cling`:

```
$ rootcling
Usage: rootcling [-v][-v0-4] [-f] [out.cxx] [opts] file1.h[+][-][!] file2.h[+][-][!] .
↪..[LinkDef.h]
For more extensive help type: /usr/local/lib/python2.7/dist-packages/cppyy_backend/
↪bin/rootcling -h
```

The basic mode of operation is to process the header files ('fileN.h') according to certain #pragmas in the LinkDef.h file in order to generate bindings accessible in Python under the 'cppyy.gbl' namespace.

The output is

- A .cpp file (which, when compiled to a shared library)
- A .rootmap file
- A .pcm file

which are used at runtime by `cling` to expose the semantics expressed by the header files to Python. Nominally, the compiled .cpp provides low-level Python access to the library API defined by the header files, while `cling` uses the other files to provide the rich features it supports. Thus, the shipping form of the bindings contains:

- A shared library (which must be compiled from the .cpp)
- A .rootmap file
- A .pcm file

11.1.2 cling-config

This is a small utility whose main purpose is to provide access to the as-installed configuration of other components. For-example:

```
$ cling-config --help
Usage: cling-config [--cflags] [--cppflags] [--cmake]
$ cling-config --cmake
/usr/local/lib/python2.7/dist-packages/cppyy_backend/cmake
```

11.1.3 cppyy-generator

This is a clang-based utility program which takes a set of C++ header files and generate a JSON output file describing the objects found in them. This output is intended to support more convenient access to a set of cppyy-supported bindings:

```
$ cppyy-generator --help
usage: cppyy-generator [-h] [-v] [--flags FLAGS] [--libclang LIBCLANG]
                       output sources [sources ...]
...
```

See the `Cmake interface` for details.

11.2 CMake interface

The bindings generated by rootcling, are 'raw' in the sense that:

- The .cpp file be compiled. The required compilation steps are platform-dependent.
- The bindings are not packaged for distribution. Typically, users expect to have a pip-compatible package.
- The binding are in the 'cppyy.gbl' namespace. This is an inconvenience at best for users who might expect C++ code from KF5::Config to appear in Python via "import KF5.Config".
- The bindings are loaded lazily, which limits the discoverability of the content of the bindings.
- `cppyy` supports customisation of the bindings via 'Pythonization' but there is no automated way to load them.

These issues are addressed by the CMake support. This is a blend of Python packaging and CMake where CMake provides:

- Platform-independent scripting of the creation of a Python 'wheel' package for the bindings.
- An facility for CMake-based projects to automate the entire bindings generation process, including basic automated tests.

11.2.1 Python packaging of bindings

Modern Python packaging usage is based on the 'wheel'. This is places the onus on the creation of binary artefacts in the package on the distributor. In this case, this includes the platform-dependent steps necessary to compile the .cpp file.

The generated package also takes advantage of the __init__.py load-time mechanism to enhance the bindings:

- The bindings are rehosted in a "native" namespace so that C++ code from KF5::Config appears in Python via "import KF5.Config".
- (TBD) Load Pythonizations.

Both of these need/can use the output of the cppyy-generator (included in the package) as well as other runtime support included in `cppyy`.

11.2.2 CMake usage

The CMake usage is via two modules:

- FindLibClang.cmake provides some bootstrap support needed to locate clang. This is provided mostly as a temporary measure; hopefully upstream support will allow this to be eliminated in due course.
- FindCppyy.cmake provides the interface described further here.

Details of the usage of these modules is within the modules themselves, but here is a summary of the usage. `FindLibClang.cmake` sets the following variables:

```
LibClang_FOUND             - True if libclang is found.
LibClang_LIBRARY           - Clang library to link against.
LibClang_VERSION           - Version number as a string (e.g. "3.9").
LibClang_PYTHON_EXECUTABLE - Compatible python version.
```

`FindCppyy.cmake` sets the following variables:

```
Cppyy_FOUND - set to true if Cppyy is found
Cppyy_DIR - the directory where Cppyy is installed
Cppyy_EXECUTABLE - the path to the Cppyy executable
Cppyy_INCLUDE_DIRS - Where to find the ROOT header files.
Cppyy_VERSION - the version number of the Cppyy backend.
```

and also defines the following functions:

```
cppyy_add_bindings - Generate a set of bindings from a set of header files.
cppyy_find_pips - Return a list of available pip programs.
```

cppyy_add_bindings

Generate a set of bindings from a set of header files. Somewhat like CMake's add_library(), the output is a compiler target. In addition ancilliary files are also generated to allow a complete set of bindings to be compiled, packaged and installed:

```
cppyy_add_bindings(
    pkg
    pkg_version
    author
    author_email
```

(continues on next page)

(continued from previous page)

```
    [URL url]
    [LICENSE license]
    [LANGUAGE_STANDARD std]
    [LINKDEFS linkdef...]
    [IMPORTS pcm...]
    [GENERATE_OPTIONS option...]
    [COMPILE_OPTIONS option...]
    [INCLUDE_DIRS dir...]
    [LINK_LIBRARIES library...]
    [H_DIRS H_DIRSectory]
    H_FILES h_file...)
```

The bindings are based on https://cppyy.readthedocs.io/en/latest/, and can be used as per the documentation provided via the cppyy.cgl namespace. First add the directory of the <pkg>.rootmap file to the LD_LIBRARY_PATH environment variable, then "import cppyy; from cppyy.gbl import <some-C++-entity>".

Alternatively, use "import <pkg>". This convenience wrapper supports "discovery" of the available C++ entities using, for example Python 3's command line completion support.

The bindings are complete with a setup.py, supporting Wheel-based packaging, and a test.py supporting pytest/nosetest sanity test of the bindings.

The bindings are generated/built/packaged using 3 environments:

- One compatible with the header files being bound. This is used to generate the generic C++ binding code (and some ancilliary files) using a modified C++ compiler. The needed options must be compatible with the normal build environment of the header files.
- One to compile the generated, generic C++ binding code using a standard C++ compiler. The resulting library code is "universal" in that it is compatible with both Python2 and Python3.
- One to package the library and ancilliary files into standard Python2/3 wheel format. The packaging is done using native Python tooling.

Arguments and options	Description
pkg	The name of the package to generate. This can be either of the form "simplename" (e.g. "Akonadi"), or of the form "namespace.simplename" (e.g. "KF5.Akonadi").
pkg_version	The version of the package.
author	The name of the library author.
author_email	The email address of the library author.
URL url	The home page for the library. Default is "https://pypi.python.org/pypi/<pkg>".
LICENSE license	The license, default is "LGPL 2.0".
LANGUAGE_STANDARD std	The version of C++ in use, "14" by default.
IMPORTS pcm	Files which contain previously-generated bindings which pkg depends on.
GENERATE_OPTIONS optio	Options which are to be passed into the rootcling command. For example, bindings which depend on Qt may need "-D__PIC__;-Wno-macro-redefined" as per https://sft.its.cern.ch/jira/browse/ROOT-8719.
LINKDEFS def	Files or lines which contain extra #pragma content for the linkdef.h file used by rootcling. See https://root.cern.ch/root/html/guides/users-guide/AddingaClass.html#the-linkdef.h-file. In lines, literal semi-colons must be escaped: ";".
EXTRA_CODES code	Files which contain extra code needed by the bindings. Customisation is by routines named "c13n_<something>"; each such routine is passed the module for <pkg>: :: code-block python **def c13n_doit(pkg_module):** print(pkg_module.__dict__) The files and individual routines within files are processed in alphabetical order.
EXTRA_HEADERS hdr	Files which contain extra headers needed by the bindings.
EXTRA_PYTHONS py	Files which contain extra Python code needed by the bindings.
COMPILE_OPTIONS option	Options which are to be passed into the compile/link command.
INCLUDE_DIRS dir	Include directories.
LINK_LIBRARIES library	Libraries to link against.
H_DIRS directory	Base directories for H_FILES.
H_FILES h_file	Header files for which to generate bindings in pkg. Absolute filenames, or filenames relative to H_DIRS. All definitions found directly in these files will contribute to the bindings. (NOTE: This means that if "forwarding headers" are present, the real "legacy" headers must be specified as H_FILES). All header files which contribute to a given C++ namespace should be grouped into a single pkg to ensure a 1-to-1 mapping with the implementing Python class.

Returns via PARENT_SCOPE variables:

```
target              The CMake target used to build.
setup_py            The setup.py script used to build or install pkg.
```

Examples:

```
find_package(Qt5Core NO_MODULE)
find_package(KF5KDcraw NO_MODULE)
get_target_property(_H_DIRS KF5::KDcraw INTERFACE_INCLUDE_DIRECTORIES)
get_target_property(_LINK_LIBRARIES KF5::KDcraw INTERFACE_LINK_LIBRARIES)
set(_LINK_LIBRARIES KF5::KDcraw ${_LINK_LIBRARIES})
include(${KF5KDcraw_DIR}/KF5KDcrawConfigVersion.cmake)

cppyy_add_bindings(
    "KDCRAW" "${PACKAGE_VERSION}" "Shaheed" "srhaque@theiet.org"
    LANGUAGE_STANDARD "14"
    LINKDEFS "../linkdef_overrides.h"
    GENERATE_OPTIONS "-D__PIC__;-Wno-macro-redefined"
    INCLUDE_DIRS ${Qt5Core_INCLUDE_DIRS}
    LINK_LIBRARIES ${_LINK_LIBRARIES}
    H_DIRS ${_H_DIRS}
    H_FILES "dcrawinfocontainer.h;kdcraw.h;rawdecodingsettings.h;rawfiles.h")
```

There is a fuller example of embedding the use of cppyy_add_bindings for a large set of bindings:

```
https://cgit.kde.org/pykde5.git/plain/KF5/CMakeLists.txt?h=include_qt_binding
```

cppyy_find_pips

Return a list of available pip programs.

CHAPTER 12

Packages

12.1 Cppyy

The `cppyy` module is a frontend (see *Package Structure*), and most of the code is elsewhere. However, it does contain the docs for all of the modules, which are built using Sphinx: http://www.sphinx-doc.org/en/stable/ and published to http://cppyy.readthedocs.io/en/latest/index.html using a webhook. To create the docs:

```
$ pip install sphinx_rtd_theme
Collecting sphinx_rtd_theme
...
Successfully installed sphinx-rtd-theme-0.2.4
$ cd docs
$ make html
```

The Python code in this module supports:

- Interfacing to the correct backend for CPython or PyPy.
- Pythonizations (TBD)

12.2 Cppyy-backend

The `cppyy-backend` module contains two areas:

- A patched copy of cling
- Wrapper code

12.3 Package structure

There are four PyPA packages involved in a full installation, with the following structure:

```
              (A) _cppyy (PyPy)
            /                    \
(1) cppyy                          (3) cling-backend -- (4) cppyy-cling
            \                    /
              (2) CPyCppyy (CPython)
```

The user-facing package is always `cppyy` (1). It is used to select the other (versioned) required packages, based on the python interpreter for which it is being installed.

Below (1) follows a bifurcation based on interpreter. This is needed for functionality and performance: for CPython, there is the CPyCppyy package (2). It is written in C++, makes use of the Python C-API, and installs as a Python extension module. For PyPy, there is the builtin module `_cppyy` (A). This is not a PyPA package. It is written in RPython as it needs access to low-level pointers, JIT hints, and the `_cffi_backend` backend module (itself builtin).

Shared again across interpreters is the backend, which is split in a small wrapper (3) and a large package that contains Cling/LLVM (4). The former is still under development and expected to be updated frequently. It is small enough to download and build very quickly. The latter, however, takes a long time to build, but since it is very stable, splitting it off allows the creation of binary wheels that need updating only infrequently (expected about twice a year).

All code is publicly available; see the *section on repositories*.

CHAPTER 13

Repositories

The `cppyy` module is a frontend that requires an intermediate (Python interpreter dependent) layer, and a backend (see *Package Structure*). Because of this layering and because it leverages several existing packages through reuse, the relevant codes are contained across a number of repositories.

- Frontend, cppyy: https://bitbucket.org/wlav/cppyy
- CPython (v2/v3) intermediate: https://bitbucket.org/wlav/cpycppyy
- PyPy intermediate (module _cppyy): https://bitbucket.org/pypy/pypy/
- Backend, cppyy: https://bitbucket.org/wlav/cppyy-backend

The backend repo contains both the cppyy-cling (under "cling") and cppyy-backend (under "clingwrapper") packages.

13.1 Building

Except for cppyy-cling, the structure in the repositories follows a normal PyPA package and they are thus are ready to build with setuptools: simply checkout the package and either run `python setup.py`, or use `pip`. It is highly recommended to follow the dependency chain when manually upgrading packages individually (i.e. cppyy-cling, cppyy-backend, CPyCppyy if on CPython, and then finally cppyy).

As an example, to upgrade CPyCppyy to the latest version in the repository, do:

```
$ git clone https://bitbucket.org/wlav/CPyCppyy.git
$ pip install ./CPyCppyy --upgrade
```

Installation of the cppyy package works the same way (just replace "CPyCppyy" with "cppyy"). Please see the pip documentation for more options, such as developer mode.

For the clingwrapper part of the backend (package "cppyy-backend"), which lives in a subdirectory in the cppyy-backend repository, do:

```
$ git clone https://bitbucket.org/wlav/cppyy-backend.git
$ pip install ./cppyy-backend/clingwrapper --upgrade
```

Finally, the cppyy-cling package (subdirectory "cling") requires sources being pulled in from upstream, and thus takes a few extra steps:

```
$ git clone https://bitbucket.org/wlav/cppyy-backend.git
$ cd cppyy-backend/cling
$ python setup.py egg_info
$ python create_src_directory.py
$ pip install . --upgrade
```

The egg_info command is needed for `create_src_directory.py` to find the right version. It in turn downloads the proper release from upstream, trims and patches it, and installs the result in the "src" directory. When done, the structure of cppyy-cling looks again like a PyPA package and can be used as expected.

CHAPTER 14

Comments and bugs

Please report bugs or requests for improvement on the issue tracker.