xeus-cling

Johan Mabille, Loic Gouarin and Sylvain Corlay

Feb 03, 2023

INSTALLATION

xeus-cling is a Jupyter kernel for C++ based on the C++ interpreter cling and the native implementation of the Jupyter protocol xeus.

CHAPTER

ONE

LICENSING

We use a shared copyright model that enables all contributors to maintain the copyright on their contributions.

This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

1.1 Installation

1.1.1 Using the conda-forge package

A package for xeus-cling is available for the mamba (or conda) package manager.

```
mamba install -c conda-forge xeus-cling
```

1.1.2 From source with cmake

You can also install `xeus-cling` from source with cmake. This requires that you have all the dependencies installed in the same prefix.

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
make install
```

On Windows platforms, from the source directory:

```
mkdir build
cd build
cmake -G "NMake Makefiles" -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
nmake
nmake install
```

1.2 Installing the Kernel Spec

When installing xeus-cling in a given installation prefix, the corresponding Jupyter kernelspecs are installed in the same environment and are automatically picked up by Jupyter if it is installed in the same prefix.

However, if Jupyter is installed in a different location, it will not pick up the new kernels. The xeus-cling kernels (for C++11, C++14 and C++17 respectively) can be registered with the following commands:

```
jupyter kernelspec install PREFIX/share/jupyter/xcpp11 --sys-prefix
jupyter kernelspec install PREFIX/share/jupyter/xcpp14 --sys-prefix
jupyter kernelspec install PREFIX/share/jupyter/xcpp17 --sys-prefix
```

For more information on the `jupyter kernelspec` command, please consult the `jupyter_client` documentation.

1.3 Build options

1.3.1 Build flags

You can specify additional build flags that will be used by `xeus-cling` to compile the code in the notebook. To do so, you need to edit the kernelspec file (usually `share/jupyter/kernels/xcppSTD/kernel.json`, where `STD` is the version of the cpp standard) and add the build flags in the `argv` array.

For instance, if you want to pass the `-pthread -lpthread` flags to `xeus-cling` and compile C++17 code, the C++17 kernelpec file becomes:

```
{
    "display_name": "C++17",
    "argv": [
        "/home/yoyo/miniconda3/envs/xwidgets/bin/xcpp",
        "-f",
        "{connection_file}",
        "-std=c++17",
        "-pthread",
        "lpthread"
    ],
    "language": "C++17"
}
```

1.3.2 Using third-party libraries

When building a binary, you usually specify the include directories and the library path of third-party libraries in the build tool. The library will be loaded upon binary execution.

`xeus-cling` is slightly different, it allows you to specify both include directories and library path, however you need to load the library explicitly. This is done with special pragma commands that you can use in a code cell in a Jupyter Notebook:

- `#pragma cling add_include_path("inc_directory")`
- `#pragma cling add_library_path("lib_directory")`
- `#pragma cling load("libname")`

1.4 Magic commands

Magics are special commands for the kernel that are not part of the C++ programming language.

There are defined with the symbol `%` for a line magic and `%%` for a cell magic.

A few magics are available in xeus-cling. In the future, user-defined magics will also be enabled.

1.4.1 %%executable

Dump the code from all entered cells into an executable binary. The content of the cell is used for the body of the *main* function.

```
%%executable filename [-- linker options]
```

- Example

```
In [1]:
1 #include <iostream>

In [2]:
1 int square(int x) { return x * x; }

In [3]:
1 %%executable square.x
2 std::cout << square(4) << std::endl;

Writing executable to square.x

In [4]:
1 !./square.x

16
```

- Optional arguments:

You can use the following options which will be passed to the linker and will influence code generation:

-fsanitize	enable instrumentation with ThreadSanitizer
-g	enable debug information in the executable

1.4.2 %%file

This magic command copies the content of the cell in a file named *filename*.

```
%%file [-a] filename
```

- Example

```
In [1]:
%%file tmp.txt
Demo of magic command

Writing tmp.txt

In [2]:
%%file -a tmp.txt
append at the end

Appending to tmp.txt

In [3]:
!cat tmp.txt

Demo of magic command
append at the end

In [ ]:
```

- Optional argument:

-a	append the content to the file.

1.4.3 %timeit

Measure the execution time execution for a line statement (*%timeit*) or for a block of statements (*%%timeit*)

- Usage in line mode

```
%timeit [-n<N> -r<R> -p<P>] statement
```

- Usage in cell mode

```
%%timeit [-n<N> -r<R> -p<P>]
statements
```

- Example

```
In [1]: #include <xtensor/xtensor.hpp>

In [2]: auto x = xt::linspace<double>(1.0, 10.0, 100);

In [3]: %timeit xt::eval(xt::sin(x));
        118 us +- 2.68 us per loop (mean +- std. dev. of 7 runs 10000 loops each)

In [4]: %timeit -n 10 -r 1 -p 6 xt::eval(xt::sin(x));
        147.225 us +- 0 ns per loop (mean +- std. dev. of 1 run, 10 loops each)

In [5]: %%timeit
        auto y = xt::linspace<double>(1.0, 10.0, 100);
        xt::eval(xt::sin(y)*xt::cos(x));
        266 us +- 3.07 us per loop (mean +- std. dev. of 7 runs 1000 loops each)

In [ ]:
```

- Optional arguments:

-n	execute the given statement <N> times in a loop. If this value is not given, a fitting value is chosen.
-r	repeat the loop iteration <R> times and take the best result. Default: 7
-p	use a precision of <P> digits to display the timing result. Default: 3

1.5 Displaying rich content

The Jupyter rich display system allows displaying rich content in the Jupyter notebook and other frontend.

This is achieved by sending mime bundles to the front-end containing various representations of the data that the frontend may use.

A mime bundle may contain multiple alternative representations of the same object for example

- a `text/html` representation for the notebook and other web frontends.
- a `text/plain` representation for the console.

Besides plain text and html, other mime type can be used such as `image/png` or even custom mime type for which a renderer is available in the front-end.

1.5.1 Default plain text representation

By default, xeus-cling provides a plain text representation for any object.

In the case of a basic type such as `double` or `int`, the value will be displayed.

For sequences (exposing an iterator pair `begin` / `end`), the content of the sequence is also displayed.

Finally, for more conplex types, the address of the object is displayed.

1.5.2 Providing custom mime representations for user-defined types

For a user-defined class `myns::foo`, you can easily provide a mime representation taylored to your needs such as a styled `html` table including the values of various attributes.

This can be achieved by simply overloading the function

```
nl::json mime_bundle_repr(const foo&);
```

in the same namespace `myns` as `foo`.

The rich display mechanism of `xeus-cling` will pick up this function through argument-dependent-lookup (ADL) and make use of it upon display.

Example: `image/png` representation of an image class

In this example, the `im::image` class holds a buffer read from a file. The `mime_bundle_repr` overload defined in the same namespace simply forwards the buffer to the frontend.

```
#include <string>
#include <fstream>

#include "xtl/xbase64.hpp"
#include "nlohmann/json.hpp"

namespace nl = nlohmann;

namespace im
{
    struct image
    {
        inline image(const std::string& filename)
        {
            std::ifstream fin(filename, std::ios::binary);
            m_buffer << fin.rdbuf();
        }

        std::stringstream m_buffer;
    };

    nl::json mime_bundle_repr(const image& i)
    {
        auto bundle = nl::json::object();
        bundle["image/png"] = xtl::base64encode(i.m_buffer.str());
```

(continues on next page)

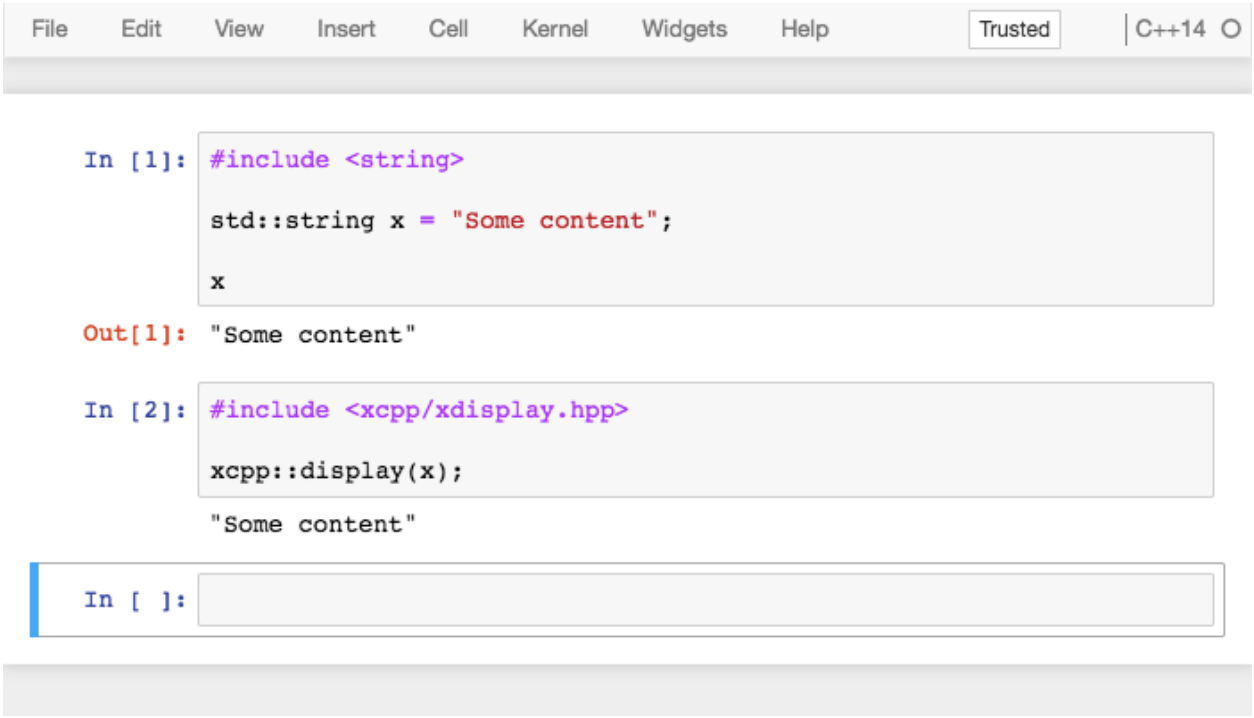
(continued from previous page)

```
        return bundle;
    }
}
```

1.5.3 Displaying content in the frontend

The first way to display an object in the front-end is to omit the last semicolon of a code cell. When doing so, the last expression will be displayed.

Another way of achieving this, is to include the `xcpp::display` function and passing the object to display. `xcpp::display` is defined in the <`xcpp/xdisplay.hpp`> header.



Note: A subtle distinction between using `xcpp::display` and omitting the last semicolon is that the latter results in a cell *output* including a prompt number, while the former will only show the rich front-end representation.

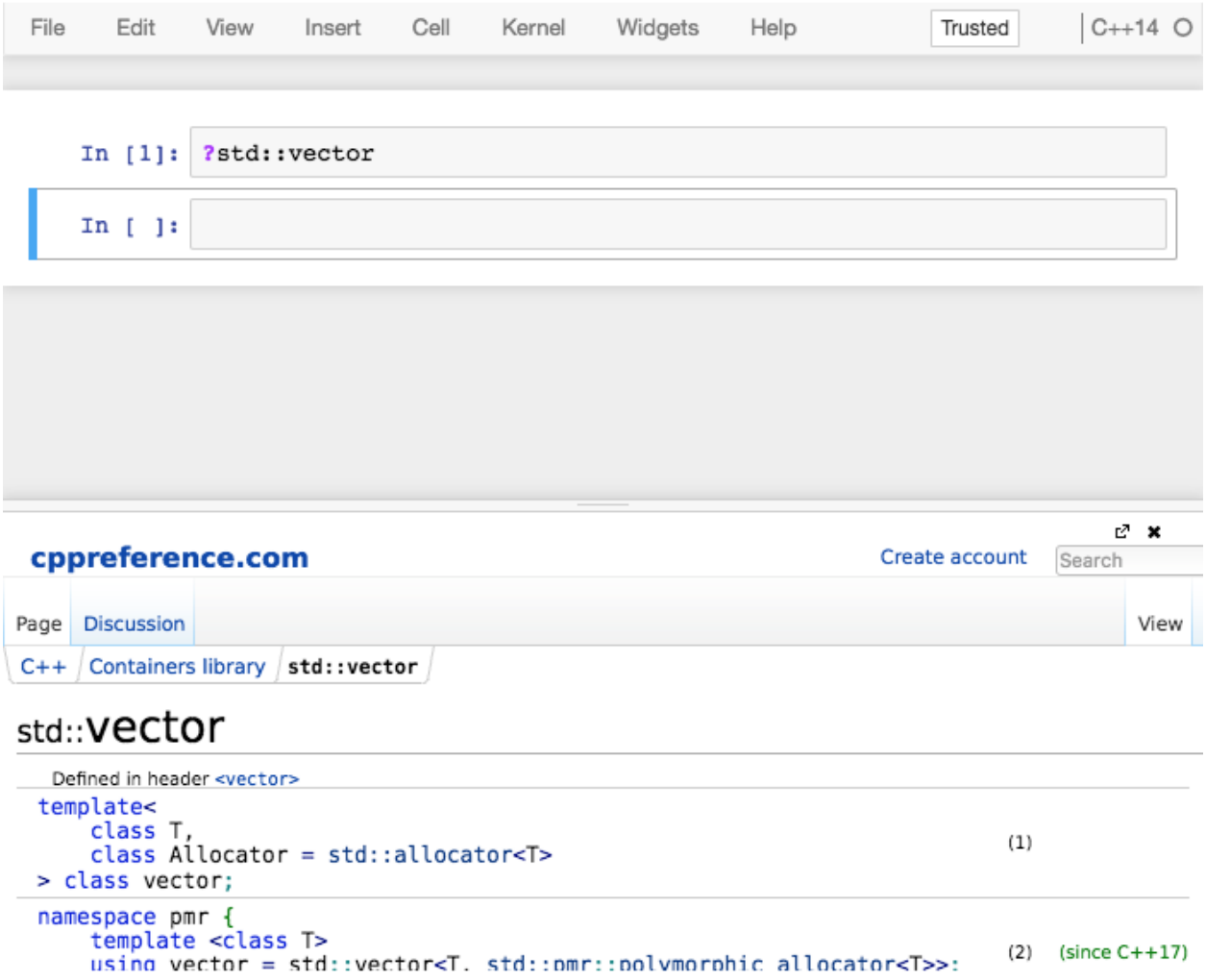
This behavior is consistent to the Python kernel implementation where `1` results in an output while `print(1)` result in a display message.

1.6 Inline documentation

1.6.1 The standard library

The `xeus-cling` kernel allows users to access help on functions and classes of the standard library.

In a code cell, typing `?std::vector` will simply display the help page on vector from the cppreference website.



1.6.2 Enabling the quick-help feature for third-party libraries

The quick help feature can be enabled for other libraries. To do so, a doxygen tag file for your library must be placed under the `xeus-cling` "data" directory of the installation prefix, namely

```
PREFIX/share/xeus-cling/tagfiles
```

For `xeus-cling` to be able to make use of that information, a JSON configuration file must be placed under the `xeus-cling` *configuration* directory of the installation prefix, namely

```
PREFIX/etc/xeus-cling/tags.d
```

Note: For more information on how to generate tag files for a doxygen documentation, check the relevant section of the doxygen documentation.

The format for the JSON configuration file is the following

```
{
    "url": "Base URL for the documentation",
    "tagfile": "Name of the doxygen tagfile"
}
```

For example the JSON configuration file for the documentation of the standard library is

```
{
    "url": "https://en.cppreference.com/w/",
    "tagfile": "cppreference-doxygen-web.tag.xml"
}
```

Note: We recommend that you only use the `https` protocol for the URL. Indeed, when the notebook is served over `https`, content from unsecure sources will not be rendered.

1.6.3 The case of breathe and sphinx documentation

Another popular documentation system is the combination of doxygen and sphinx, thanks for the breathe package, which generates sphinx documentation using the XML output of doxygen.

The xhale Python package can be used to convert the sphinx inventory files produced breathe into doxygen tag files.