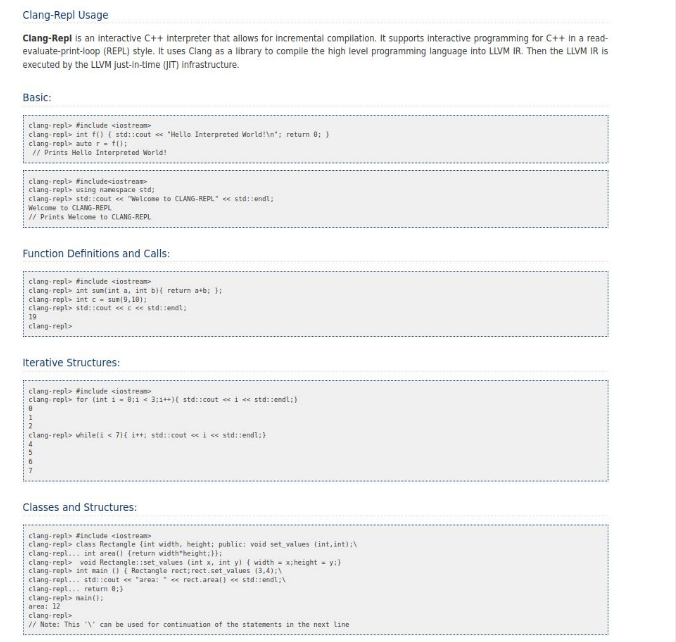
GSoC 2023
Program @ LLVM

Student:
Krishna Narayanan

Mentors:
Vassil Vassilev, David Lange

Tutorial Development with clang-repl and xeus-clang-repl

Incremental Compilation (Clang-Repl)



Clang-Repl Usage

Clang-Repl is an interactive C++ interpreter that allows for incremental compilation. It supports interactive programming for C++ in a read-evaluate-print-loop (REPL) style. It uses Clang as a library to compile the high level programming language into LLVM IR. Then the LLVM IR is executed by the LLVM just-in-time (JIT) infrastructure.

Basic:

```
clang-repl> #include <iostream>
clang-repl> int f() { std::cout << "Hello Interpreted World!\n"; return 0; }
clang-repl> auto r = f();
 // Prints Hello Interpreted World!
```

```
clang-repl> #include<iostream>
clang-repl> using namespace std;
clang-repl> std::cout << "Welcome to CLANG-REPL" << std::endl;
Welcome to CLANG-REPL
// Prints Welcome to CLANG-REPL
```

Function Definitions and Calls:

```
clang-repl> #include <iostream>
clang-repl> int sum(int a, int b){ return a+b; };
clang-repl> int c = sum(9,10);
clang-repl> std::cout << c << std::endl;
19
clang-repl>
```

Iterative Structures:

```
clang-repl> #include <iostream>
clang-repl> for (int i = 0;i < 3;i++){ std::cout << i << std::endl;}
0
1
2
clang-repl> while(i < 7){ i++; std::cout << i << std::endl;}
4
5
6
7
```

Classes and Structures:

```
clang-repl> #include <iostream>
clang-repl> class Rectangle {int width, height; public: void set_values (int,int);\
clang-repl... int area() {return width*height;}};
clang-repl>  void Rectangle::set_values (int x, int y) { width = x;height = y;}
clang-repl> int main () { Rectangle rect;rect.set_values (3,4);\
clang-repl... std::cout << "area: " << rect.area() << std::endl;\
clang-repl... return 0;}
clang-repl> main();
area: 12
clang-repl>
// Note: This '\' can be used for continuation of the statements in the next line
```

Using Dynamic Library:

```
clang-repl> %lib print.so
clang-repl> #include"print.hpp"
clang-repl> print(9);
9
```

Generation of dynamic library

```
// print.cpp
#include <iostream>
#include "print.hpp"

void print(int a)
{
   std::cout << a << std::endl;
}

// print.hpp
void print (int a);

// Commands
clang++-17  -c -o print.o print.cpp
clang-17 -shared print.o -o print.so
```

Comments:

```
clang-repl> // Comments in Clang-Repl
clang-repl> /* Comments in Clang-Repl */
```

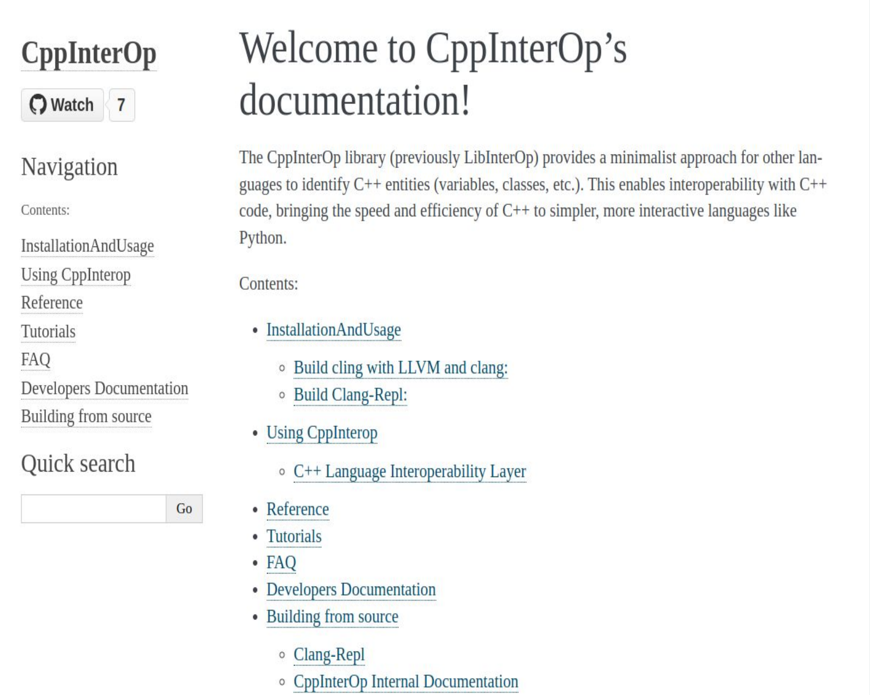
Closure or Termination:

```
clang-repl>%quit
```

Just like Clang, Clang-Repl can be integrated in existing applications as a library (using the clangInterpreter library). This turns your C++ compiler into a service that can incrementally consume and execute code. The **Compiler as A Service** (**CaaS**) concept helps support advanced use cases such as template instantiations on demand and automatic language interoperability. It also helps static languages such as C/C++ become apt for data science.

https://clang.llvm.org/docs/ClangRepl.html

Add CppInterOp Documentation



CppInterOp

Navigation

Contents:

- InstallationAndUsage
- Using CppInterop
- Reference
- Tutorials
- FAQ
- Developers Documentation
- Building from source

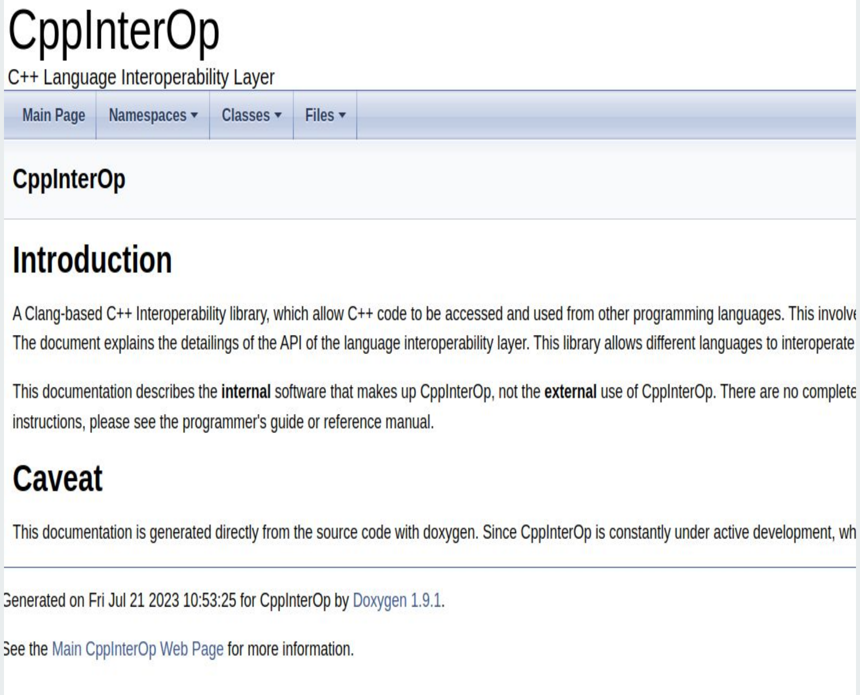
Quick search

Welcome to CppInterOp's documentation!

The CppInterOp library (previously LibInterOp) provides a minimalist approach for other languages to identify C++ entities (variables, classes, etc.). This enables interoperability with C++ code, bringing the speed and efficiency of C++ to simpler, more interactive languages like Python.

Contents:

- InstallationAndUsage
  - Build cling with LLVM and clang:
  - Build Clang-Repl:
- Using CppInterop
  - C++ Language Interoperability Layer
- Reference
- Tutorials
- FAQ
- Developers Documentation
- Building from source
  - Clang-Repl
  - CppInterOp Internal Documentation



CppInterOp

C++ Language Interoperability Layer

CppInterOp

Introduction

A Clang-based C++ Interoperability library, which allow C++ code to be accessed and used from other programming languages. This involve
The document explains the detailings of the API of the language interoperability layer. This library allows different languages to interoperate

This documentation describes the **internal** software that makes up CppInterOp, not the **external** use of CppInterOp. There are no complete
instructions, please see the programmer's guide or reference manual.

Caveat

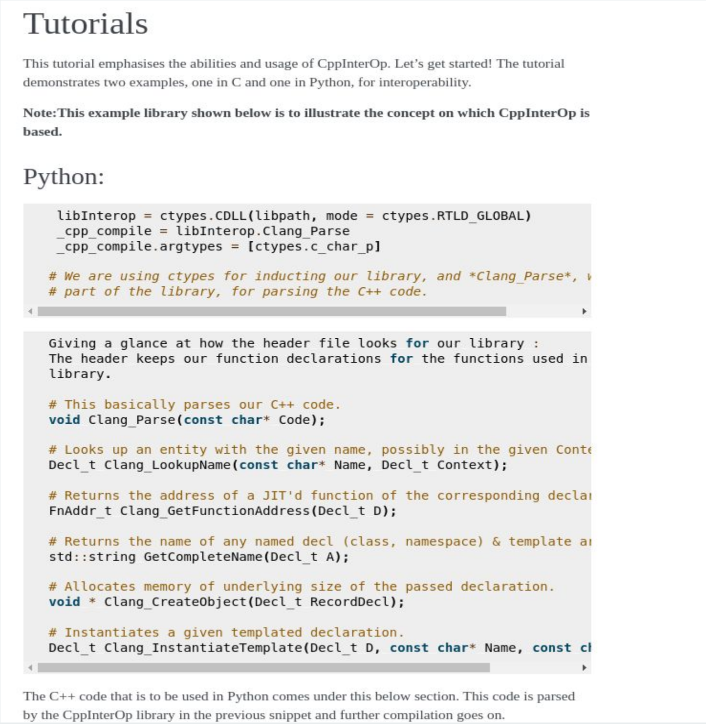
This documentation is generated directly from the source code with doxygen. Since CppInterOp is constantly under active development, wh

Generated on Fri Jul 21 2023 10:53:25 for CppInterOp by Doxygen 1.9.1.

See the Main CppInterOp Web Page for more information.

https://cppinterop.readthedocs.io/en/latest/index.html

CppInterOp Tutorials



Tutorials

This tutorial emphasises the abilities and usage of CppInterOp. Let's get started! The tutorial demonstrates two examples, one in C and one in Python, for interoperability.

Note:This example library shown below is to illustrate the concept on which CppInterOp is based.

Python:

```
 libInterop = ctypes.CDLL(libpath, mode = ctypes.RTLD_GLOBAL)
 _cpp_compile = libInterop.Clang_Parse
 _cpp_compile.argtypes = [ctypes.c_char_p]

# We are using ctypes for inducting our library, and *Clang_Parse*,
# part of the library, for parsing the C++ code.
```

```
Giving a glance at how the header file looks for our library :
The header keeps our function declarations for the functions used in
library.

# This basically parses our C++ code.
void Clang_Parse(const char* Code);

# Looks up an entity with the given name, possibly in the given Conte
Decl_t Clang_LookupName(const char* Name, Decl_t Context);

# Returns the address of a JIT'd function of the corresponding decla
FnAddr_t Clang_GetFunctionAddress(Decl_t D);

# Returns the name of any named decl (class, namespace) & template a
std::string GetCompleteName(Decl_t A);

# Allocates memory of underlying size of the passed declaration.
void * Clang_CreateObject(Decl_t RecordDecl);

# Instantiates a given templated declaration.
Decl_t Clang_InstantiateTemplate(Decl_t D, const char* Name, const c
```

The C++ code that is to be used in Python comes under this below section. This code is parsed by the CppInterOp library in the previous snippet and further compilation goes on.

C:

Include **p3-ex4-lib.h**, which contains the declarations for the functions used in this code. The detailed summary of header comes in the latter part.

The variable Code is given as a C-style string, it contains the C++ code to be parsed. It has two classes, class *A* and a templated class *B* with a member function callme.

```
const char* Code = "void* operator new(__SIZE_TYPE__, void* __p)
                   "extern \"C\" int printf(const char*,...);"
                   "class A {};"
                   "\n #include <typeinfo> \n"
                   "class B {"
                   "public:"
                   "  template<typename T>"
                   "  void callme(T) {"
                   "    printf(\" Instantiated with [%s] \\n \", typeid(
                   " }"
                   "};";
```

The main() begins with the call to **Clang_Parse** from interop library responsible for parsing the provided C++ code.

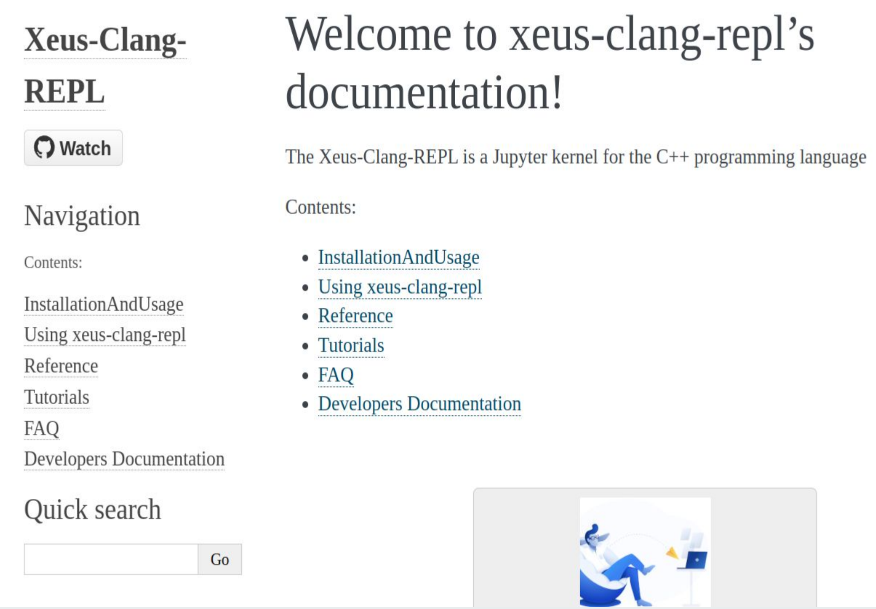
Next there a number of initializations, **Instantiation** is initialized to zero, it will be used to store the instantiated template. The **InstantiationArgs** is initialized to "A", it will be used as the argument when instantiating the template. *T* is initialized to zero, used to store the declaration of the type "T" used for instantiation.

```
Decl_t Instantiation = 0;
const char * InstantiationArgs = "A";
Decl_t TemplatedClass = Clang_LookupName("B", /*Context=*/0);
Decl_t T = 0;
```

This snippet checks command-line arguments were provided by the argc arguments. We take the first argument (*argv[1]*), parse it, then take the second argument (*argv[2]*) using **Clang_LookupName**, and reassigns **InstantiationArgs** to the third argument (*argv[3]*). In the else case, we decide to go with the "A".

The code proceeds to instantiate the template *B::callme* with the given type, using the **Clang_InstantiateTemplate** function from the library. The instantiated template is stored in the **Instantiation**.

Add xeus-clang-repl Documentation



Xeus-Clang-REPL

Navigation

Contents:

- InstallationAndUsage
- Using xeus-clang-repl
- Reference
- Tutorials
- FAQ
- Developers Documentation

Quick search

Welcome to xeus-clang-repl's documentation!

The Xeus-Clang-REPL is a Jupyter kernel for the C++ programming language

Contents:

- InstallationAndUsage
- Using xeus-clang-repl
- Reference
- Tutorials
- FAQ
- Developers Documentation

Xeus-Clang-REPL

C++ usage in Jupyter Notebooks

Xeus-Clang-Repl

Introduction

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

This documentation describes the **internal** software that makes up Xeus-Clang-Repl, not the **external** use of Xeus-Clang-Repl. There are no compl

Clang-Repl, only the APIs that make up the software. For usage instructions, please see the programmer's guide or reference manual.

Caveat

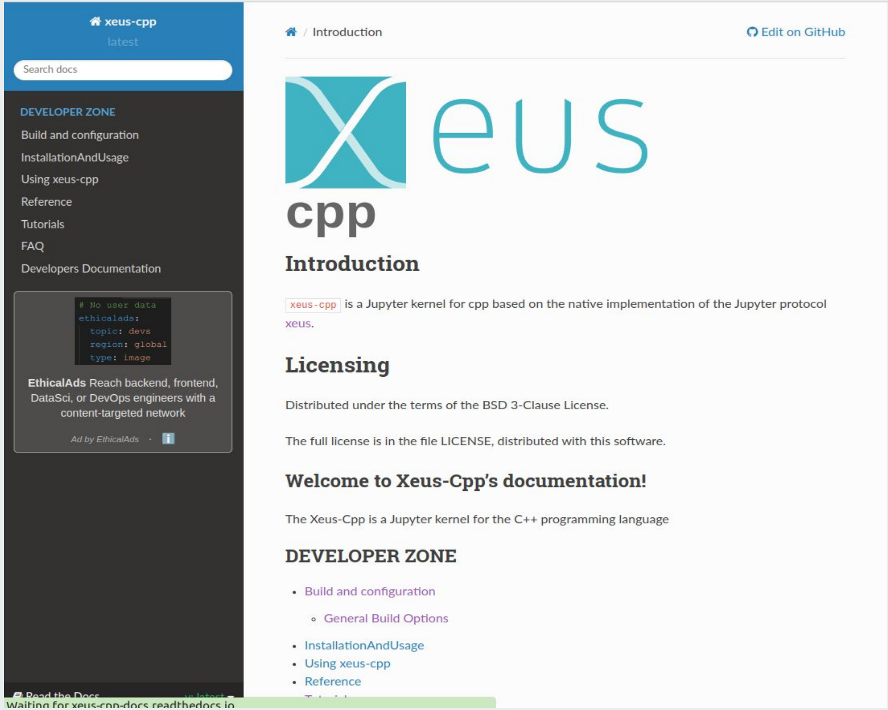
This documentation is generated directly from the source code with doxygen. Since Xeus-Clang-Repl is constantly under active development, what

Generated on Mon Aug 7 2023 15:50:10 for Xeus-Clang-REPL by Doxygen 1.9.1.

See the Main Xeus-Clang-REPL Web Page for more information.

https://xeus-clang-repl-docs.readthedocs.io/en/latest/

Add xeus-cpp Documentation



Introduction

`xeus-cpp` is a Jupyter kernel for cpp based on the native implementation of the Jupyter protocol xeus.

Licensing

Distributed under the terms of the BSD 3-Clause License.

The full license is in the file LICENSE, distributed with this software.

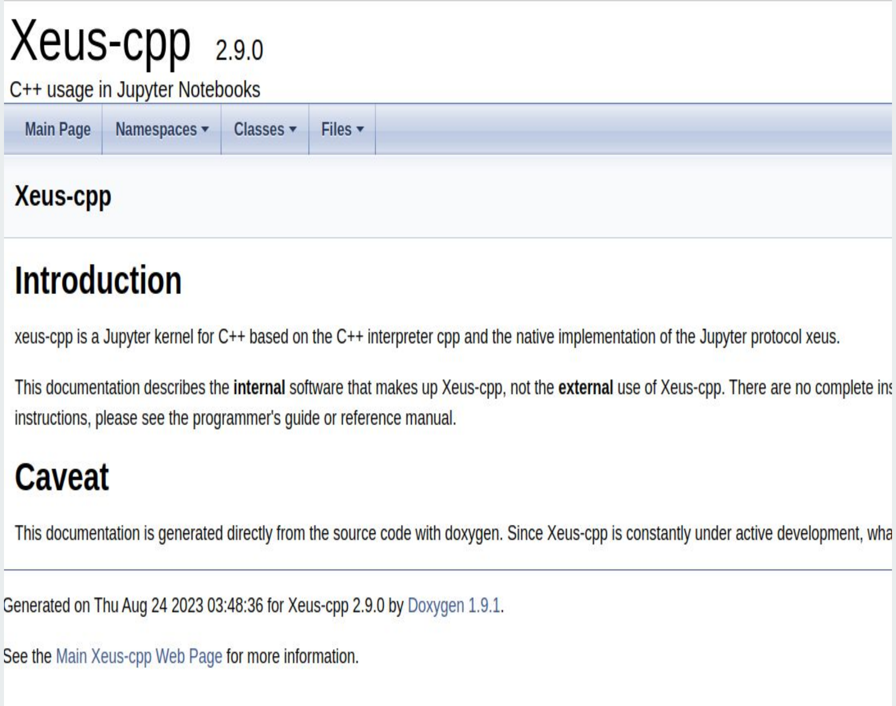
Welcome to Xeus-Cpp's documentation!

The Xeus-Cpp is a Jupyter kernel for the C++ programming language

DEVELOPER ZONE

- Build and configuration
  - General Build Options
- InstallationAndUsage
- Using xeus-cpp
- Reference

Waiting for xeus-cpp-docs.readthedocs.io...



Introduction

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

This documentation describes the **internal** software that makes up Xeus-cpp, not the **external** use of Xeus-cpp. There are no complete ins instructions, please see the programmer's guide or reference manual.

Caveat

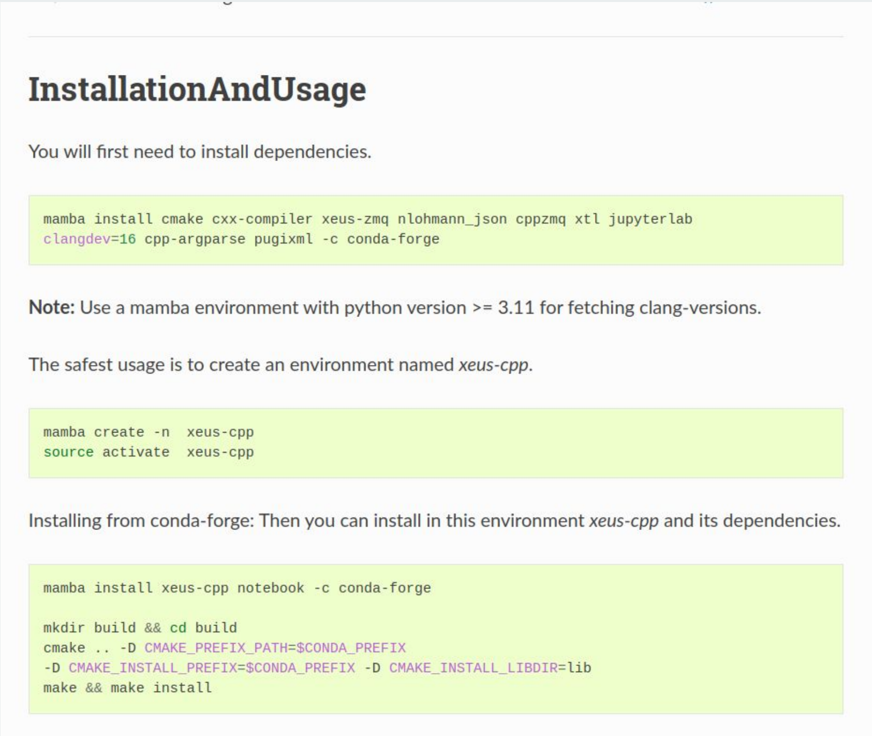
This documentation is generated directly from the source code with doxygen. Since Xeus-cpp is constantly under active development, wha

Generated on Thu Aug 24 2023 03:48:36 for Xeus-cpp 2.9.0 by Doxygen 1.9.1.

See the Main Xeus-cpp Web Page for more information.

https://xeus-cpp-docs.readthedocs.io/en/latest/

xeus-cpp docs glimpse



InstallationAndUsage

You will first need to install dependencies.

```
mamba install cmake cxx-compiler xeus-zmq nlohmann_json cppzmq xtl jupyterlab
clangdev=16 cpp-argparse pugixml -c conda-forge
```

Note: Use a mamba environment with python version >= 3.11 for fetching clang-versions.

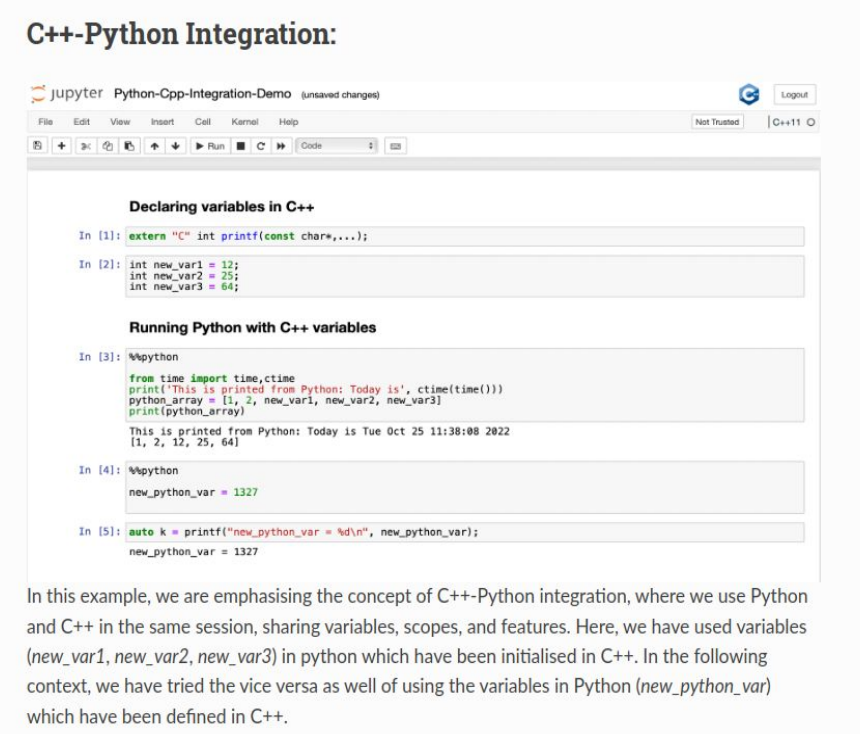
The safest usage is to create an environment named *xeus-cpp*.

```
mamba create -n  xeus-cpp
source activate  xeus-cpp
```

Installing from conda-forge: Then you can install in this environment *xeus-cpp* and its dependencies.

```
mamba install xeus-cpp notebook -c conda-forge

mkdir build && cd build
cmake .. -D CMAKE_PREFIX_PATH=$CONDA_PREFIX
-D CMAKE_INSTALL_PREFIX=$CONDA_PREFIX -D CMAKE_INSTALL_LIBDIR=lib
make && make install
```



C++-Python Integration:

Declaring variables in C++

```
In [1]: extern "C" int printf(const char*,...);

In [2]: int new_var1 = 12;
        int new_var2 = 25;
        int new_var3 = 64;
```

Running Python with C++ variables

```
In [3]: %%python

        from time import time,ctime
        print('This is printed from Python: Today is', ctime(time()))
        python_array = [1, 2, new_var1, new_var2, new_var3]
        print(python_array)

        This is printed from Python: Today is Tue Oct 25 11:38:08 2022
        [1, 2, 12, 25, 64]

In [4]: %%python

        new_python_var = 1327

In [5]: auto k = printf("new_python_var = %d\n", new_python_var);

        new_python_var = 1327
```

In this example, we are emphasising the concept of C++-Python integration, where we use Python and C++ in the same session, sharing variables, scopes, and features. Here, we have used variables (*new_var1*, *new_var2*, *new_var3*) in python which have been initialised in C++. In the following context, we have tried the vice versa as well of using the variables in Python (*new_python_var*) which have been defined in C++.

Miscellaneous

- Port **v1 to v2 configuration** readthedocs for compiler research projects which uses **builder os** reliant on python tools(python-3.*, mambaforge-22.*).
- Working on Saqib's patch to land on clang documentation **(support for graphviz extension for diagram convention)**.
- Contributing to **CppInterOp, xeus-clang-repl upstream** recently**.**
- **Fixing bugs** found in the current work and patching them up.

THANK YOU !