

Julia programming language interoperability

Julia programming language

- Julia 1.0 released in 2018, now on 1.10.
- Designed for scientific software development.
- High level dynamic syntax but strongly typed when needed.
- Just-in-time (JIT) compiled, garbage collected, multiple dispatch for easy generic coding.
- Dynamic like Python, speeds comparable to C/C++.













Julia programming language interoperability











- How can I call my favorite Python and C/C++ libraries from Julia?
- How can I call my favorite Julia libraries from Python and C/C++?



Julia programming language interoperability

| from to |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | C-Interfaces | ISO_C_BINDING | nanobind / pybind11 clair | ccall CxxWrap.jl |
|  | Native | | | ctypes Python C-API | ccall Clang.jl |
|  | ISO_C_BINDING | | | f2py | ccall |
|  | Python C-API | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

Julia programming language interoperability

| <div>from</div> <div>to</div> |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | ccall CxxWrap.jl |
|  | | | | | ccall Clang.jl |
|  | | | | | ccall |
|  | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

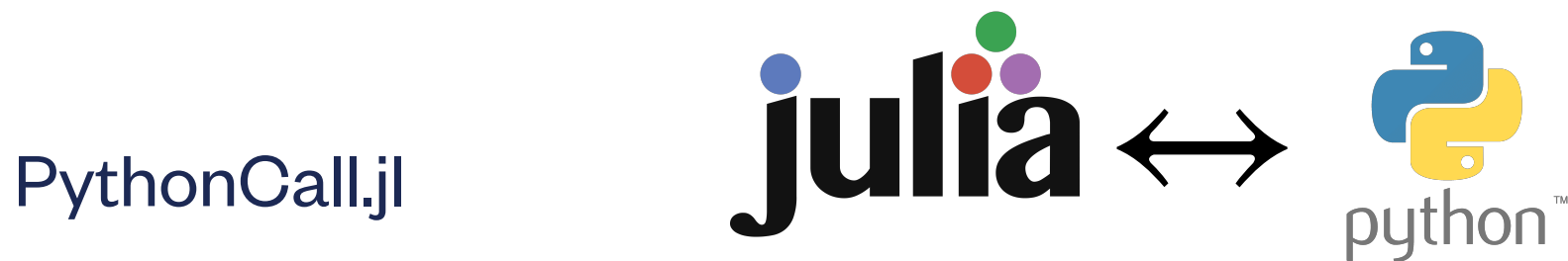
For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Julia programming language interoperability

| from to |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | ccall CxxWrap.jl |
|  | | | | | ccall Clang.jl |
|  | | | | | ccall |
|  | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Calling Python from Julia: PythonCall.jl



- Julia package for calling existing Python code.
- Automated wrapping and non-copying conversion of types.
- Install and manage Python packages through Julia (with CondaPkg.jl), or point to an existing Python installation.



```
import numpy as np

def scale_by_two(v):
    v *= 2

v = np.array([1., 2., 3.])
scale_by_two(v)

print(v) # [2. 4. 6.]
```

Scale array in-place in Python



```
function scale_by_two(v)
    v .*= 2
end

v = [1., 2., 3.]
scale_by_two(v)

println(v) # [2. 4. 6.]
```

Scale array in-place in Julia



```
import numpy as np
def scale_by_two(v):
    v *= 2

v = np.array([1., 2., 3.])
scale_by_two(v)

print(v) # [2. 4. 6.]
```

Scale array in-place in Python



```
function scale_by_two(v)
    v .*= 2
end

v = [1., 2., 3.]
scale_by_two(v)

println(v) # [2. 4. 6.]
```

Scale array in-place in Julia

Calling Python from Julia: PythonCall.jl



scale.py

```
def scale_by_two(v):  
    v *= 2
```

Define a function in Python



```
using PythonCall  
np = pyimport("numpy")  
scale = pyimport("scale")  
  
v = np.array([1., 2., 3.])  
scale.scale_by_two(v)  
  
println(v) # [2. 4. 6.]
```

Call from Julia

Calling Python from Julia: PythonCall.jl



scale.py

```
def scale_by_two(v):  
    v *= 2
```

Define a function in Python



```
using PythonCall  
np = pyimport("numpy")  
scale = pyimport("scale")  
  
v = np.array([1., 2., 3.])  
scale.scale_by_two(v)  
  
println(v) # [2. 4. 6.]
```

Call from Julia

Calling Python from Julia: PythonCall.jl



```
function scale_by_two(v)
    v .*= 2
end

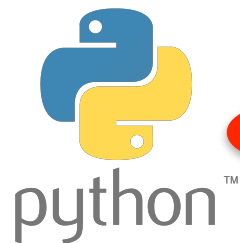
using PythonCall
np = pyimport("numpy")

v = np.array([1., 2., 3.])
# scale_by_two(v) # error, not supported
scale_by_two(PyArray(v))
println(v) # [2. 4. 6.]
```

Some operations require manual type conversions, non-copying conversions are available.

Call a Julia function on a Python object, all in Julia

Calling Python from Julia: PythonCall.jl



```
function scale_by_two(v)
    v .*= 2
end

using PythonCall
np = pyimport("numpy")

v = np.array([1., 2., 3.])
# scale_by_two(v) # error, not supported
scale_by_two(PyArray(v))
println(v) # [2. 4. 6.]
```

Some operations require manual type conversions, non-copying conversions are available.

Call a Julia function on a Python object, all in Julia

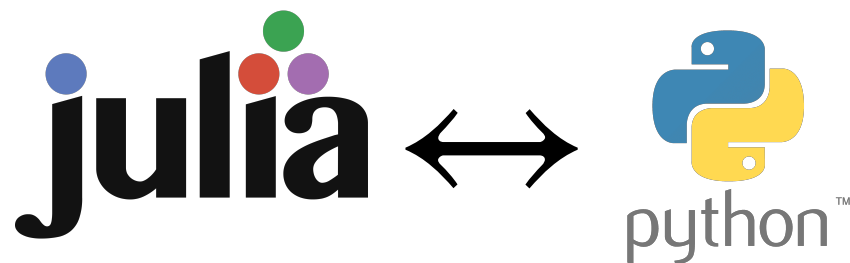
Julia programming language interoperability

| from to |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | ccall CxxWrap.jl |
|  | | | | | ccall Clang.jl |
|  | | | | | ccall |
|  | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Calling Julia from Python: JuliaCall

JuliaCall



- JuliaCall is a Python package for calling Julia from Python.
- JuliaCall is part of the PythonCall.jl project, and works in a very similar way.

Calling Julia from Python: JuliaCall



scale.jl

```
function scale_by_two(v)
    v .*= 2
end
```

Define a function in Julia



```
import numpy as np
from juliacall import Main as jl
jl.include("scale.jl")

v = np.array([1., 2., 3.])
jl.scale_by_two(v)

print(v) # [2., 4., 6.]
```

Call from Python

Calling Julia from Python: JuliaCall



scale.jl

```
function scale_by_two(v)
    v .*= 2
end
```

Define a function in Julia













```
import numpy as np
from juliacall import Main as jl
jl.include("scale.jl")

v = np.array([1., 2., 3.])
jl.scale_by_two(v)

print(v) # [2., 4., 6.]
```

Call from Python

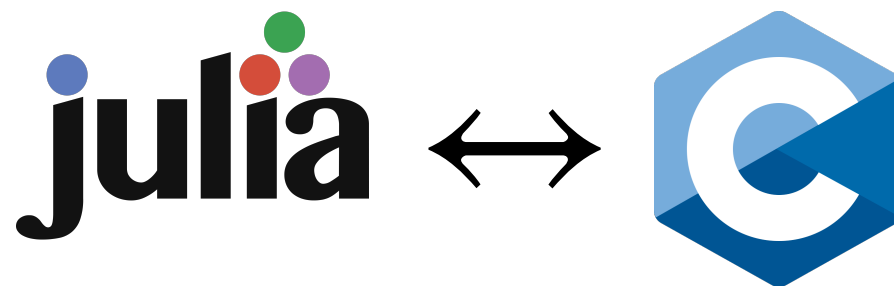
Julia programming language interoperability

| from to |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | ccall |
|  | | | | | CxxWrap.jl ccall |
|  | | | | | ccall |
|  | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Calling C from Julia: ccall

ccall



- The Julia standard library provides a function **ccall** (and a macro **@ccall**) for calling C functions from Julia.
- Performs some automated conversions back and forth between simple Julia types and C types.
- Requires manual wrapping/interfaces.

Calling C from Julia: ccall



scale.c

```
void scale_by_two(double *v, int length)
{
    for(int i = 0; i < length; i++) v[i] *= 2;
}
```



```
v = [1., 2., 3.]
@ccall "./scale.so".scale_by_two(v::Ptr{Cdouble}, length(v)::Cint)::Cvoid
println(v) # [2., 4., 6.]
```

`gcc -fPIC -shared -o scale.so scale.c`

Calling C from Julia: ccall



scale.c











```
void scale_by_two(double *v, int length)
{
    for(int i = 0; i < length; i++) v[i] *= 2;
}
```



```
v = [1., 2., 3.]
@ccall"./scale.so".scale_by_two(v::Ptr{Cdouble}, length(v)::Cint)::Cvoid
println(v) # [2., 4., 6.]
```

gcc -fPIC -shared -o scale.so scale.c

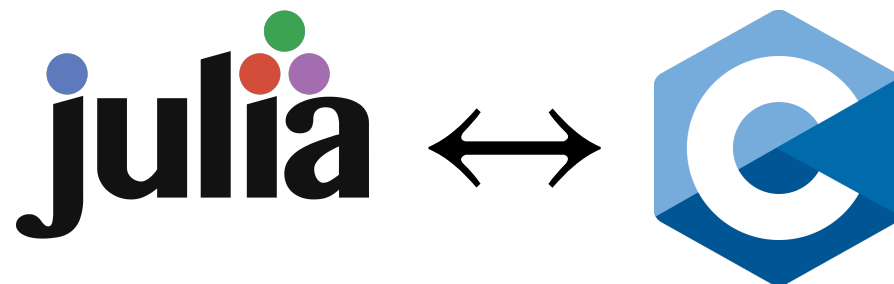
Julia programming language interoperability

| from to |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | ccall CxxWrap.jl |
|  | | | | | Clang.jl |
|  | | | | | ccall |
|  | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Calling C from Julia: Clang.jl

Clang.jl



- Automatically generates Julia function wrappers and types from C functions and structs.
- Used by many well established Julia libraries (SparseArrays.jl, CUDA.jl, MPI.jl, etc.) for generating wrappers for C libraries.

Calling C from Julia: Clang.jl



scale.h

```
void scale_by_two(double *v, int length);
```

scale.c

```
#include <stdio.h>
#include "scale.h"

void scale_by_two(double *v, int length)
{
    for(int i = 0; i < length; i++) v[i] *= 2;
}
```

Define functions in C and compile into a shared library



scale.jl

```
# Generated by Clang.jl
function scale_by_two(v, length)
    ccall(:scale_by_two, "scale.so", Cvoid, (Ptr{Cdouble}, Cint), v, length)
end
```

Clang.jl automatically generates Julia wrappers for the compiled C functions that are defined in the header file (scale.h).

```
v = [1., 2., 3.]
scale_by_two(v, length(v))

println(v) # [2., 4., 6.]
```

Call functions generated by Clang.jl in Julia

Calling C from Julia: Clang.jl



scale.h

```
void scale_by_two(double *v, int length);
```

scale.c

```
#include <stdio.h>
#include "scale.h"
```

```
void scale_by_two(double *v, int length)
{
    for(int i = 0; i < length; i++) v[i] *= 2;
}
```

Define functions in C and compile into a shared library



scale.jl

```
# Generated by Clang.jl
function scale_by_two(v, length)
    ccall((@cfunction{void, (Ptr{Cdouble}, Cint)}, v, length)
end
```

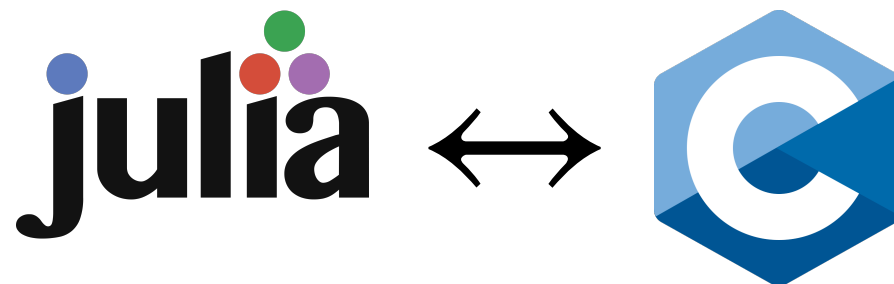
Clang.jl automatically generates Julia wrappers for the compiled C functions that are defined in the header file (scale.h).

```
v = [1., 2., 3.]
scale_by_two(v, length(v))
println(v) # [2., 4., 6.]
```

Call functions generated by Clang.jl in Julia







Calling C from Julia: BinaryBuilder.jl

BinaryBuilder.jl



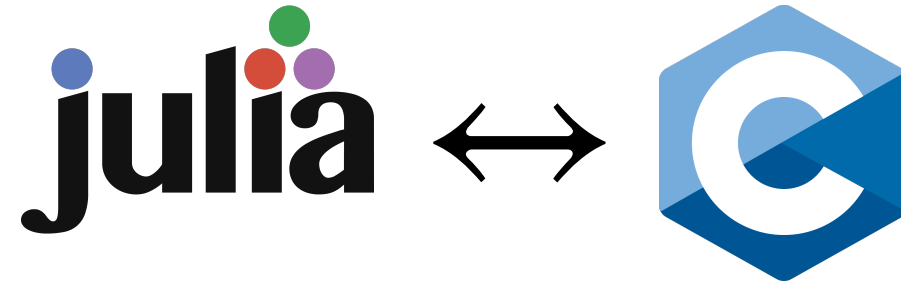
- See BinaryBuilder.jl for a standardized way of setting up cross-platform build scripts to ship binary dependencies as part of a Julia package.
- C libraries built with BinaryBuilder.jl can be accessed as a Julia package and called and wrapped with ccall and Clang.jl.

Julia programming language interoperability

| to \ from |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| | | | | | ccall CxxWrap.jl |
| | | | | | ccall Clang.jl |
| | | | | | ccall |
| | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

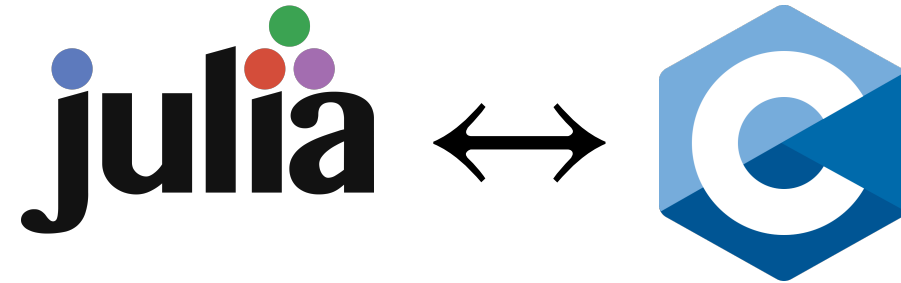
Calling Julia from C: PackageCompiler.jl



PackageCompiler.jl has two main purposes:

1. Compile Julia code for use within Julia (to decrease JIT compilation time).
2. Create a shared library of compiled Julia code which can be used by C, Fortran, C++, etc.

Calling Julia from C: PackageCompiler.jl



PackageCompiler.jl has two main purposes:

1. Compile Julia code for use within Julia (to decrease JIT compilation time).
2. Create a shared library of compiled Julia code which can be used by C, Fortran, C++, etc.

Calling Julia from C: PackageCompiler.jl



```
function scale_by_two(v)
    v .*= 2
end

Base.@ccallable function scale_by_two(v::Ptr{Cdouble}, length::Cint)::Cvoid
    scale_by_two(unsafe_wrap(Array, v, (length,)))
end
```

Define functions in Julia and wrap them in a C-compatible interface with **@ccallable**

```
#include "julia_init.h"
#include "scale.h"

int main(int argc, char *argv[])
{
    init_julia(argc, argv);

    double v[] = {1., 2., 3.};
    int length = sizeof(v) / sizeof(v[0]);
    scale_by_two(v, length);

    shutdown_julia(0);
    return 0;
}
```

```
using PackageCompiler
create_library("Scale", "ScaleCompiled"; lib_name="libscale")
```

```
gcc -o main main.o -L/ScaleCompiled/lib/libscale.so -lscale
```

Calling Julia from C: PackageCompiler.jl



```
function scale_by_two(v)
    v .*= 2
end

Base.@ccallable function scale_by_two(v::Ptr{Cdouble}, length::Cint)::Cvoid
    scale_by_two(Array{Cdouble, 1}(v, length))
end
```

Define functions in Julia and wrap them in a C-compatible interface with **@ccallable**

```
using PackageCompiler
create_library("Scale", "ScaleCompiled"; lib_name="libscale")
```

```
#include <stdio.h>
#include "julia_init.h"
#include "scale.h"

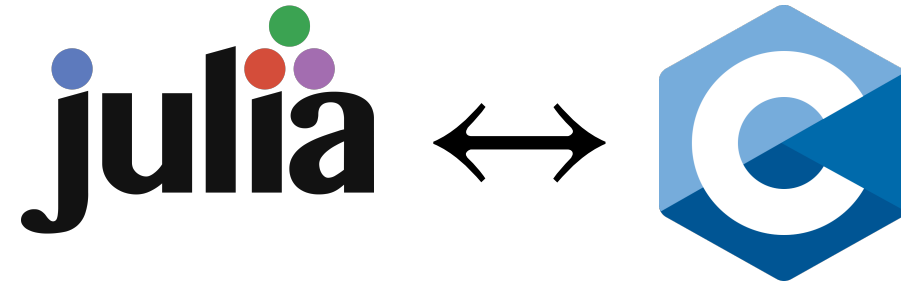
int main(int argc, char *argv[])
{
    init_julia(argc, argv);

    double v[] = {1., 2., 3.};
    scale_by_two(v);

    shutdown_julia(0);
    return 0;
}
```

```
gcc -o main main.o -L/ScaleCompiled/lib/libscale.so -lscale
```












Compiling Julia code



A lot of work is in progress to improve compilation of Julia code!

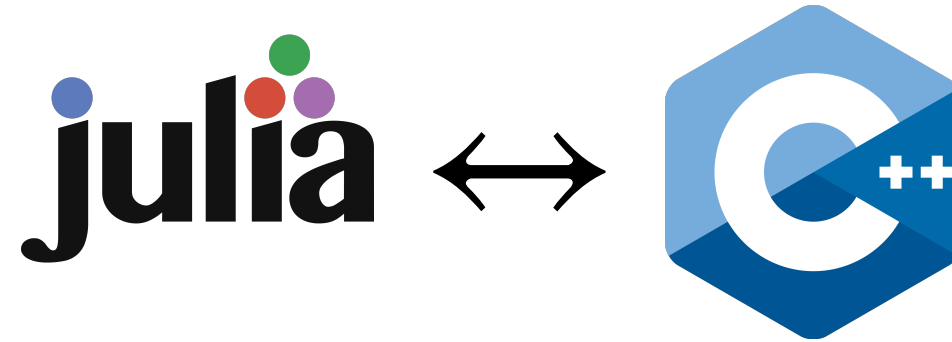
- StaticCompiler.jl: allows compiling small binaries, but can't compile all Julia code (code must be non-allocating).
- See JuliaCon talk “Jeff Bezanson - What's the deal with Julia binary sizes?” (<https://www.youtube.com/watch?v=kNslvU3WD4M>)

Julia programming language interoperability

| from \ to | |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | |  |
|  | | | | | | ccall |
|  | | | | | | Clang.jl |
|  | | | | | | ccall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Calling C++ from Julia: CxxWrap.jl



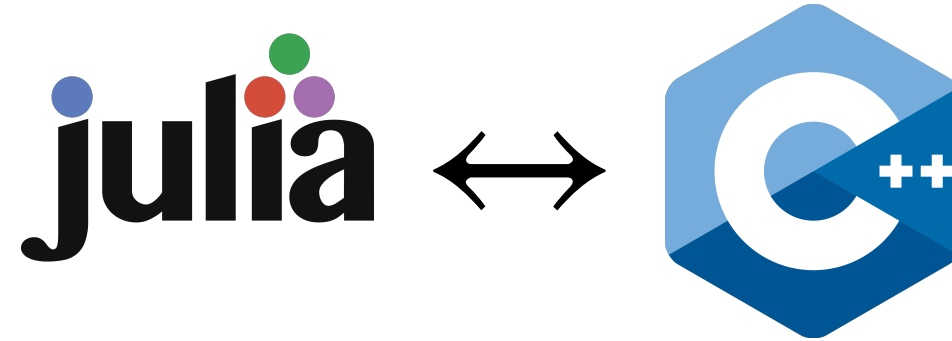
- CxxWrap.jl is the most mature option in Julia for calling C++ from Julia.
- **It is not automated!** You have to manually write wrappers in C++ which you then call from Julia.
- CxxWrap.jl is comparable to pybind11 or Boost.Python.
- Alternatively, use ccall/Clang.jl if you just need a C-interface.

Julia programming language interoperability

| to \ from |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| |  | | | | |
| |  | | | | |
| |  | | | | |
| |  | | | | |
|  |  | PackageCompiler.jl | | JuliaCall | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |











For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>

Calling Julia from C++: jluna



- jluna provides automated wrapping of Julia code into C++.
- It does not appear to be widely used at the moment.
- Alternatively, use PackageCompiler.jl if you just need a C-interface.

Julia programming language interoperability

| <div>from</div> <div>to</div> |  |  |  |  |  |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  | | | | | ccall CxxWrap.jl |
|  | | | | | ccall Clang.jl |
|  | | | | | ccall |
|  | | | | | PythonCall |
|  | jluna PackageCompiler.jl | PackageCompiler.jl | | JuliaCall | |

For interoperability with other languages (Matlab, R, Java, Mathematica, etc.), see <https://github.com/JuliaInterop>