

---

# shroud Documentation

*Release 0.13.0*

lInl

Sep 22, 2023



---

## Contents

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Installing</b>	<b>7</b>
<b>3</b>	<b>Tutorial</b>	<b>11</b>
<b>4</b>	<b>Input</b>	<b>23</b>
<b>5</b>	<b>Pointers and Arrays</b>	<b>37</b>
<b>6</b>	<b>Types</b>	<b>45</b>
<b>7</b>	<b>Namespaces</b>	<b>49</b>
<b>8</b>	<b>Structs and Classes</b>	<b>51</b>
<b>9</b>	<b>Default Arguments</b>	<b>63</b>
<b>10</b>	<b>Templates</b>	<b>65</b>
<b>11</b>	<b>Declarations</b>	<b>67</b>
<b>12</b>	<b>Output</b>	<b>73</b>
<b>13</b>	<b>C and C++</b>	<b>81</b>
<b>14</b>	<b>Fortran</b>	<b>85</b>
<b>15</b>	<b>Python</b>	<b>91</b>
<b>16</b>	<b>Cookbook</b>	<b>99</b>
<b>17</b>	<b>Typemaps</b>	<b>101</b>
<b>18</b>	<b>Statements</b>	<b>107</b>
<b>19</b>	<b>C Statements</b>	<b>109</b>
<b>20</b>	<b>Fortran Statements</b>	<b>115</b>

<b>21</b>	<b>Reference</b>	<b>121</b>
<b>22</b>	<b>Releases</b>	<b>139</b>
<b>23</b>	<b>Fortran Previous Work</b>	<b>143</b>
<b>24</b>	<b>Python Previous Work</b>	<b>147</b>
<b>25</b>	<b>Future Work</b>	<b>149</b>
<b>26</b>	<b>Sample Fortran Wrappers</b>	<b>151</b>
<b>27</b>	<b>Numpy Struct Descriptor</b>	<b>203</b>
<b>28</b>	<b>Glossary</b>	<b>205</b>
<b>29</b>	<b>Indices and tables</b>	<b>207</b>
	<b>Bibliography</b>	<b>209</b>

**Shroud** is a tool for creating a Fortran or Python interface to a C or C++ library. It can also create a C API for a C++ library.

The user creates a YAML file with the C/C++ declarations to be wrapped along with some annotations to provide semantic information and code generation options. **Shroud** produces a wrapper for the library. The generated code is highly-readable and intended to be similar to code that would be hand-written to create the bindings.

*verb*

1. wrap or dress (a body) in a shroud for burial.
2. cover or envelop so as to conceal from view.

Contents



# CHAPTER 1

---

## Introduction

---

**Shroud** is a tool for creating a Fortran or Python interface to a C or C++ library. It can also create a C API for a C++ library.

The user creates a YAML file with the C/C++ declarations to be wrapped along with some annotations to provide semantic information and code generation options. Shroud produces a wrapper for the library. The generated code is highly-readable and intended to be similar to code that would be hand-written to create the bindings.

Input is read from the YAML file which describes the types, variables, enumerations, functions, structures and classes to wrap. This file must be created by the user. Shroud does not parse C++ code to extract the API. That was considered a large task and not needed for the size of the API of the library that inspired Shroud's development. In addition, there is a lot of semantic information which must be provided by the user that may be difficult to infer from the source alone. However, the task of creating the input file is simplified since the C++ declarations can be cut-and-pasted into the YAML file.

In some sense, Shroud can be thought of as a fancy macro processor. It takes the function declarations from the YAML file, breaks them down into a series of contexts (library, class, function, argument) and defines a dictionary of format macros of the form key=value. There are then a series of macro templates which are expanded to create the wrapper functions. The overall structure of the generated code is defined by the classes and functions in the YAML file as well as the requirements of C++ and Fortran syntax.

Each declaration can have annotations which provide semantic information. This information is used to create more idiomatic wrappers. Shroud started as a tool for creating a Fortran wrapper for a C++ library. The declarations and annotations in the input file also provide enough information to create a Python wrapper.

## 1.1 Goals

- Simplify the creating of wrapper for a C++ library.
- Preserves the object-oriented style of C++ classes.
- Create an idiomatic wrapper API from the C++ API.
- Generate code which is easy to understand.

- No dependent runtime library.

## 1.2 Fortran

The Fortran wrapper is created by using the interoperability with C features added in Fortran 2003. This includes the `iso_c_binding` module and the `bind` and `value` keywords. Fortran cannot interoperate with C++ directly and uses C as the lingua franca. C++ can communicate with C via a common heritage and the `extern "C"` keyword. A C API for the C++ API is produced as a byproduct of the Fortran wrapping.

Using a C++ API to create an object and call a method:

```
Instance * inst = new Instance;
inst->method(1);
```

In Fortran this becomes:

```
type(instance) inst
inst = instance()
call inst%method(1)
```

---

**Note:** The ability to generate C++ wrappers for Fortran is not supported.

---

### 1.2.1 Issues

There is a long history of ad-hoc solutions to provide C and Fortran interoperability. Any solution must address several problems:

- Name mangling of externals. This includes namespaces and operator overloading in C++.
- Call-by-reference vs call-by-value differences
- Length of string arguments.
- Blank filled vs null terminated strings.

The 2003 Fortran standard added several features for interoperability with C:

- `iso_c_binding` - intrinsic module which defines fortran kinds for matching with C's types.
- `BIND` keyword to control name mangling of externals.
- `VALUE` attribute to allow pass-by-value.

In addition, Fortran 2003 provides object oriented programming facilities:

- Type extension
- Procedure Polymorphism with Type-Bound Procedures
- Enumerations compatible with C

*Further Interoperability of Fortran with C*, Technical Specification TS 29113, now part of Fortran 2018, introduced additional features:

- assumed-type
- assumed-rank



- `ALLOCATABLE`, `OPTIONAL`, and `POINTER` attributes may be specified for a dummy argument in a procedure interface that has the `BIND` attribute.
- `CONTIGUOUS` attribute

Shroud uses the features of Fortran 2003 as well as additional generated code to solve the interoperability problem to create an idiomatic interface.

### 1.2.2 Limitations

Not all features of C++ can be mapped to Fortran. Variadic function are not directly supported. Fortran supports `OPTIONAL` arguments but that does not map to variadic functions. `OPTIONAL` has a known number of possible argument while variadic does not.

Templates will be explicitly instantiated. The instances are listed in the YAML file and a wrapper will be created for each one. However, Fortran can not instantiate templates at compile time.

Lambda functions are not supported.

Some other features are not currently supported but will be in the future: complex type, exceptions.

### 1.2.3 Requirements

Fortran wrappers are generated as free-form source and require a Fortran 2003 compiler. C code requires C99.

## 1.3 Python

The Python wrappers use the [CPython API](#) to create a wrapper for the library.

### 1.3.1 Requirements

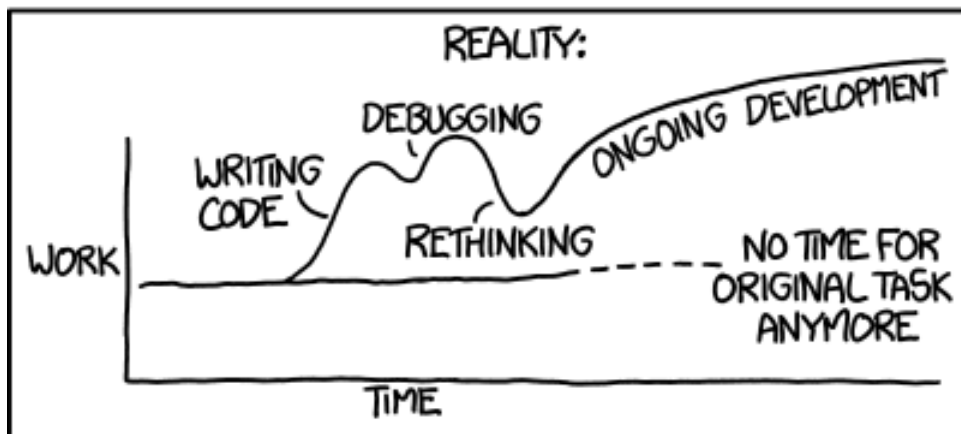
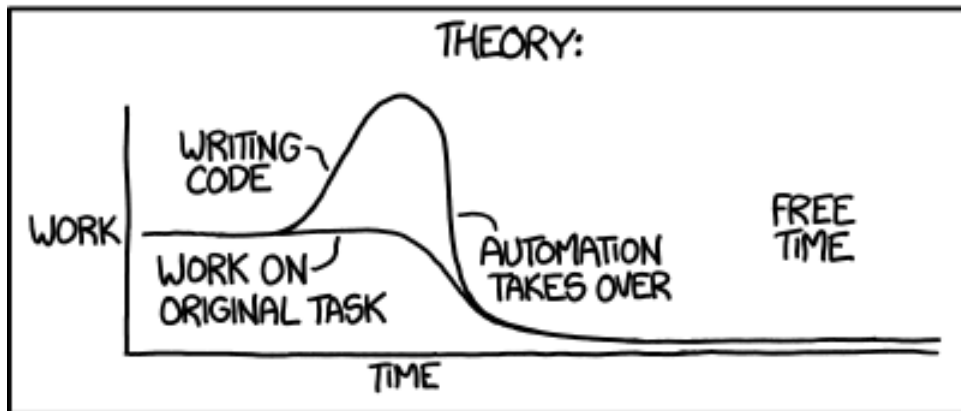
The generated code will require

- Python 2.7 or Python 3.4+
- NumPy can be used when using pointers with *rank*, *dimension* or *allocatable*, attributes.

## 1.4 XKCD

[XKCD](#)

"I SPEND A LOT OF TIME ON THIS TASK.  
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



## CHAPTER 2

---

### Installing

---

The easiest way to install Shroud is via `pip` which will fetch a file from [pypi](#)

```
pip install llnl-shroud
```

This will install Shroud into the same directory as `pip`. A virtual environment can be created if another destination directory is desired. For details see the [python docs](#)

The source is available from [github.com/LLNL/shroud](#) A `shiv` packaged executable is also available at [github releases](#). This is an executable file which contains Shroud and PyYAML and uses the Python3 in the user's path.

**Shroud** is written in Python and has been tested with version 2.7 and 3.4+. It requires the module:

- PyYAML <https://github.com/yaml/pyyaml>

After downloading the source:

```
python setup.py install
```

This will create the script *shroud* in the same directory as Python.

Since shroud installs into Python's bin directory, it may be desirable to setup a virtual environment to try it out:

```
$ cd my_project_folder
$ virtualenv my_project
$ source my_project/bin/activate
$ cd path/to/shroud/source
$ python setup.py install
```

This will create an executable at `my_project/bin/shroud`. This version requires the virtual environment to run and may be difficult to share with others.

It's possible to create a standalone executable with `shiv`:

```
$ cd path/to/shroud/source
$ shiv --python '/usr/bin/env python3' -c shroud -o dist/shroud.pyz .
```

A file *shroud.pyz* is created which bundles all of shroud and pyYAML into a single file. It uses the python on your path to run.

## 2.1 Building wrappers with CMake

Shroud can produce a CMake macro file with the option `-cmake`. This option can be incorporated into a CMakefile as:

```
if(EXISTS ${SHROUD_EXECUTABLE})
    execute_process(COMMAND ${SHROUD_EXECUTABLE}
                    --cmake ${CMAKE_CURRENT_BINARY_DIR}/SetupShroud.cmake
                    ERROR_VARIABLE SHROUD_cmake_error
                    OUTPUT_STRIP_TRAILING_WHITESPACE )
    if(${SHROUD_cmake_error})
        message(FATAL_ERROR "Error from Shroud: ${SHROUD_cmake_error}")
    endif()
    include(${CMAKE_CURRENT_BINARY_DIR}/SetupShroud.cmake)
endif()
```

The path to Shroud must be defined to CMake. It can be defined on the command line as:

```
cmake -DSHROUD_EXECUTABLE=/full/path/bin/shroud
```

The `add_shroud` macro can then be used in other `CMakeLists.txt` files as:

```
add_shroud(
    YAML_INPUT_FILE      ${YAML_INPUT_FILE}
    C_FORTRAN_OUTPUT_DIR c_fortran
)
```

CMake will treat all Fortran files as free format with the command:

```
set(CMAKE_Fortran_FORMAT FREE)
```

## 2.2 Building Python extensions

`setup.py` can be used to build the extension module from the files created by shroud. This example is drawn from the `run/tutorial` example. You must provide the paths to the input YAML file and the C++ library source files:

```
import os
from distutils.core import setup, Extension
import shroud
import numpy

outdir = 'build/source'
if not os.path.exists(outdir):
    os.makedirs(outdir)
config = shroud.create_wrapper('../.../tutorial.yaml',
                               path=['../.../'],
                               outdir=outdir)

tutorial = Extension(
    'tutorial',
```

(continues on next page)

(continued from previous page)

```
sources = config.pyfiles + ['./tutorial.cpp'],
include_dirs=[numpy.get_include(), '..']
)

setup(
    name='tutorial',
    version="0.0",
    description='shroud tutorial',
    author='xxx',
    author_email='yyy@zz',
    ext_modules=[tutorial],
)
```

The directory structure is layed out as:

```
tutorial.yaml
run
  tutorial
    tutorial.cpp # C++ library to wrap
    tutorial.hpp
  python
    setup.py # setup file shown above
  build
    source
      # create by shroud
      pyClassltype.cpp
      pySingletontype.cpp
      pyTutorialmodule.cpp
      pyTutorialmodule.hpp
      pyTutorialhelper.cpp
    lib
      tutorial.so # generated module
```



This tutorial will walk through the steps required to create a Fortran or Python wrapper for a simple C++ library.

### 3.1 Functions

The simplest item to wrap is a function in the file `tutorial.hpp`:

```
namespace tutorial {  
    void NoReturnNoArguments (void) ;  
}
```

This is wrapped using a YAML input file `tutorial.yaml`:

```
library: Tutorial  
cxx_header: tutorial.hpp  
  
declarations:  
- decl: namespace tutorial  
  declarations:  
  - decl: void NoReturnNoArguments ()
```

**library** is used to name output files and name the Fortran module. **cxx\_header** is the name of a C++ header file which contains the declarations for functions to be wrapped. **declarations** is a sequence of mappings which describe the functions to wrap.

Process the file with *Shroud*:

```
% shroud tutorial.yaml  
Wrote wrapTutorial.h  
Wrote wrapTutorial.cpp  
Wrote wrapftutorial.f  
  
Wrote pyClassItype.cpp
```

(continues on next page)

(continued from previous page)

```
Wrote pyTutorialmodule.hpp
Wrote pyTutorialmodule.cpp
Wrote pyTutorialutil.cpp
```

The C++ code to call the function:

```
#include "tutorial.hpp"

using namespace tutorial;
NoReturnNoArguments();
```

And the Fortran version:

```
use tutorial_mod
call no_return_no_arguments
```

The generated code is listed at [NoReturnNoArguments](#).

## 3.2 Arguments

### 3.2.1 Integer and Real

Integer and real types are handled using the `iso_c_binding` module which match them directly to the corresponding types in C++. To wrap `PassByValue`:

```
double PassByValue(double arg1, int arg2)
{
    return arg1 + arg2;
}
```

Add the declaration to the YAML file:

```
declarations:
- decl: double PassByValue(double arg1, int arg2)
```

Usage:

```
use tutorial_mod
real(C_DOUBLE) result
result = pass_by_value(1.d0, 4)
```

```
import tutorial
result = tutorial.PassByValue(1.0, 4)
```

## 3.3 Pointer Functions

Functions which return a pointer will create a Fortran wrapper with the `POINTER` attribute:

```
- decl: int * ReturnIntPtrDim(int *len+intent(out)+hidden) +dimension(len)
```



The C++ routine returns a pointer to an array and the length of the array in argument `len`. The Fortran API does not need to pass the `len` argument since the returned pointer will know its length. The *hidden* attribute will cause `len` to be omitted from the Fortran API, but still passed to the C wrapper.

It can be used as:

```
integer(C_INT), pointer :: intp(:)
integer len

intp => return_int_ptr_dim_pointer()
len = size(intp)
```

The generated code is listed at [returnIntPtrDimPointer](#).

A numeric pointer may also be processed differently by setting the *deref* attribute. Possible values are *pointer*, the default, for a Fortran pointer. *allocatable* to create a Fortran allocatable array and copy the data into it. *raw* returns a type (C\_PTR). In this case, the `len` argument should not be hidden so it can be used from Fortran. And *scalar* can be used when returning a pointer to a scalar. In this case, the `len` argument would be ignored.

## 3.4 Pointer arguments

When a C++ routine accepts a pointer argument it may mean several things

- output a scalar
- input or output an array
- pass-by-reference for a struct or class.

In this example, `len` and `values` are an input array and `result` is an output scalar:

```
void Sum(size_t len, const int *values, int *result)
{
    int sum = 0;
    for (size_t i=0; i < len; i++) {
        sum += values[i];
    }
    *result = sum;
    return;
}
```

When this function is wrapped it is necessary to give some annotations in the YAML file to describe how the variables should be mapped to Fortran:

```
- decl: void Sum(size_t len +implied(size(values)),
               const int *values +rank(1),
               int *result +intent(out))
```

In the `BIND(C)` interface only `len` uses the `value` attribute. Without the attribute Fortran defaults to pass-by-reference i.e. passes a pointer. The `rank` attribute defines the variable as a one dimensional, assumed-shape array. In the C interface this maps to an assumed-length array. C pointers, like assumed-length arrays, have no idea how many values they point to. This information is passed by the `len` argument.

The `len` argument defines the `implied` attribute. This argument is not part of the Fortran API since its presence is *implied* from the expression `size(values)`. This uses the Fortran intrinsic `size` to compute the total number of elements in the array. It then passes this value to the C wrapper:

Fortran usage:

```
use tutorial_mod
integer(C_INT) result
call sum([1,2,3,4,5], result)
```

Python usage. Since *result* is *intent(out)* it will be returned by the function.

```
import tutorial
result = tutorial.Sum([1, 2, 3, 4, 5])
```

See example [Sum](#) for generated code.

### 3.4.1 String

Character variables have significant differences between C and Fortran. The Fortran interoperability with C feature treats a character variable of default kind as an array of `character(kind=C_CHAR, len=1)`. The wrapper then deals with the C convention of NULL termination to Fortran's blank filled.

C++ routine:

```
const std::string ConcatenateStrings(
    const std::string& arg1,
    const std::string& arg2)
{
    return arg1 + arg2;
}
```

YAML input:

```
declarations:
- decl: const std::string ConcatenateStrings(
    const std::string& arg1,
    const std::string& arg2 )
```

The function is called as:

```
character(len=:), allocatable :: rv4c

rv4c = concatenate_strings("one", "two")
```

---

**Note:** This function is just for demonstration purposes. Any reasonable person would just use the concatenation operator in Fortran.

---

## 3.5 Default Arguments

Each function with default arguments will create a C and Fortran wrapper for each possible prototype. For Fortran, these functions are then wrapped in a generic statement which allows them to be called by the original name. A header files contains:

```
double UseDefaultArguments(double arg1 = 3.1415, bool arg2 = true)
```

and the function is defined as:

```
double UseDefaultArguments(double arg1, bool arg2)
{
    if (arg2) {
        return arg1 + 10.0;
    } else {
        return arg1;
    }
}
```

Creating a wrapper for each possible way of calling the C++ function allows C++ to provide the default values:

```
declarations:
- decl: double UseDefaultArguments(double arg1 = 3.1415, bool arg2 = true)
  default_arg_suffix:
  -
  - _arg1
  - _arg1_arg2
```

The *default\_arg\_suffix* provides a list of values of *function\_suffix* for each possible set of arguments for the function. In this case 0, 1, or 2 arguments.

Fortran usage:

```
use tutorial_mod
print *, use_default_arguments()
print *, use_default_arguments(1.d0)
print *, use_default_arguments(1.d0, .false.)
```

Python usage:

```
>>> import tutorial
>>> tutorial.UseDefaultArguments()
13.1415
>>> tutorial.UseDefaultArguments(1.0)
11.0
>>> tutorial.UseDefaultArguments(1.0, False)
1.0
```

The generated code is listed at [UseDefaultArguments](#).

There are additional options for dealing with default arguments described at [Default Arguments](#).

## 3.6 Overloaded Functions

C++ allows function names to be overloaded. Fortran supports this by using a generic interface. The C and Fortran wrappers will generate a wrapper for each C++ function but must explicitly mangle the name to distinguish the functions. The C++ compiler will mangle the names for you.

C++:

```
void OverloadedFunction(const std::string &name);
void OverloadedFunction(int idx);
```

By default the names are mangled by adding an index to the end. This can be controlled by setting *function\_suffix* in the YAML file:

```
declarations:
- decl: void OverloadedFunction(const std::string& name)
  function_suffix: _from_name
- decl: void OverloadedFunction(int indx)
  function_suffix: _from_index
```

```
call overloaded_function_from_name("name")
call overloaded_function_from_index(1)
call overloaded_function("name")
call overloaded_function(1)
```

```
tutorial.OverloadedFunction("name")
tutorial.OverloadedFunction(1)
```

## 3.7 Optional arguments and overloaded functions

Overloaded function that have optional arguments can also be wrapped:

```
- decl: int UseDefaultOverload(int num,
    int offset = 0, int stride = 1)
- decl: int UseDefaultOverload(double type, int num,
    int offset = 0, int stride = 1)
```

These routines can then be called as:

```
rv = use_default_overload(10)
rv = use_default_overload(1d0, 10)

rv = use_default_overload(10, 11, 12)
rv = use_default_overload(1d0, 10, 11, 12)
```

## 3.8 Templates

C++ template are handled by creating a wrapper for each instantiation of the function defined by the `cxx_template` field. The C and Fortran names are mangled by adding a type suffix to the function name.

C++:

```
template<typename ArgType>
void TemplateArgument(ArgType arg)
{
    return;
}
```

YAML:

```
- decl: |
    template<typename ArgType>
    void TemplateArgument(ArgType arg)
  cxx_template:
  - instantiation: <int>
  - instantiation: <double>
```

Fortran usage:

```
call template_argument(1)
call template_argument(10.d0)
```

Python usage:

```
tutorial.TemplateArgument(1)
tutorial.TemplateArgument(10.0)
```

Likewise, the return type can be templated but in this case no interface block will be generated since generic function cannot vary only by return type.

C++:

```
template<typename RetType>
RetType TemplateReturn()
{
    return 0;
}
```

YAML:

```
- decl: template<typename RetType> RetType TemplateReturn()
  cxx_template:
  - instantiation: <int>
  - instantiation: <double>
```

Fortran usage:

```
integer(C_INT) rv_integer
real(C_DOUBLE) rv_double
rv_integer = template_return_int()
rv_double = template_return_double()
```

Python usage:

```
rv_integer = TemplateReturn_int()
rv_double = TemplateReturn_double()
```

## 3.9 Generic Functions

C and C++ provide a type promotion feature when calling functions which Fortran does not support:

```
void FortranGeneric(double arg);

FortranGeneric(1.0f);
FortranGeneric(2.0);
```

When FortranGeneric is wrapped in Fortran it may only be used with the correct arguments:

```
call fortran_generic(1.)
      1
Error: Type mismatch in argument 'arg' at (1); passed REAL(4) to REAL(8)
```

It would be possible to create a version of the routine in C++ which accepts floats, but that would require changes to the library being wrapped. Instead it is possible to create a generic interface to the routine by defining which variables need their types changed. This is similar to templates in C++ but will only impact the Fortran wrapper. Instead of specify the Type which changes, you specify the argument which changes:

```
- decl: void FortranGeneric(double arg)
  fortran_generic:
- decl: (float arg)
  function_suffix: float
- decl: (double arg)
  function_suffix: double
```

It may now be used with single or double precision arguments:

```
call fortran_generic(1.0)
call fortran_generic(1.0d0)
```

A full example is at [GenericReal](#).

## 3.10 Types

### 3.10.1 Typedef

Sometimes a library will use a typedef to identify a specific use of a type:

```
typedef int TypeID;

int typefunc(TypeID arg);
```

Shroud must be told about user defined types in the YAML file:

```
declarations:
- decl: typedef int TypeID;
```

This will map the C++ type TypeID to the predefined type int. The C wrapper will use int:

```
int TUT_typefunc(int arg)
{
    tutorial::TypeID SHC_rv = tutorial::typefunc(arg);
    return SHC_rv;
}
```

### 3.10.2 Enumerations

Enumeration types can also be supported by describing the type to shroud. For example:

```
namespace tutorial
{
    enum EnumTypeID {
        ENUM0,
        ENUM1,
        ENUM2
    }
```

(continues on next page)

(continued from previous page)

```
};

EnumTypeID enumfunc(EnumTypeID arg);

} /* end namespace tutorial */
```

This enumeration is within a namespace so it is not available to C. For C and Fortran the type can be describe as an `int` similar to how the `typedef` is defined. But in addition we describe how to convert between C and C++:

```
declarations:
- decl: typedef int EnumTypeID
  fields:
    c_to_cxx : static_cast<tutorial::EnumTypeID>({c_var})
    cxx_to_c : static_cast<int>({cxx_var})
```

The typename must be fully qualified (use `tutorial::EnumTypeId` instead of `EnumTypeId`). The C argument is explicitly converted to a C++ type, then the return type is explicitly converted to a C type in the generated wrapper:

```
int TUT_enumfunc(int arg)
{
    tutorial::EnumTypeID SHCXX_arg = static_cast<tutorial::EnumTypeID>(arg);
    tutorial::EnumTypeID SHCXX_rv = tutorial::enumfunc(SHCXX_arg);
    int SHC_rv = static_cast<int>(SHCXX_rv);
    return SHC_rv;
}
```

Without the explicit conversion you're likely to get an error such as:

```
error: invalid conversion from 'int' to 'tutorial::EnumTypeID'
```

A enum can also be fully defined to Fortran:

```
declarations:
- decl: |
    enum Color {
        RED,
        BLUE,
        WHITE
    };
```

In this case the type is implicitly defined so there is no need to add it to the *types* list. The C header duplicates the enumeration, but within an `extern "C"` block:

```
// tutorial::Color
enum TUT_Color {
    TUT_tutorial_Color_RED,
    TUT_tutorial_Color_BLUE,
    TUT_tutorial_Color_WHITE
};
```

Fortran creates integer parameters for each value:

```
! enum tutorial::Color
integer(C_INT), parameter :: tutorial_color_red = 0
integer(C_INT), parameter :: tutorial_color_blue = 1
integer(C_INT), parameter :: tutorial_color_white = 2
```

**Note:** Fortran’s `ENUM`, `BIND(C)` provides a way of matching the size and values of enumerations. However, it doesn’t seem to buy you too much in this case. Defining enumeration values as `INTEGER`, `PARAMETER` seems more straightforward.

---

### 3.10.3 Structure

A structure in C++ can be mapped directly to a Fortran derived type using the `bind(C)` attribute provided by Fortran 2003. For example, the C++ code:

```
struct struct1 {  
    int ifield;  
    double dfield;  
};
```

can be defined to Shroud with the YAML input:

```
- decl: |  
    struct struct1 {  
        int ifield;  
        double dfield;  
    };
```

This will generate a C struct which is compatible with C++:

```
struct s_TUT_struct1 {  
    int ifield;  
    double dfield;  
};  
typedef struct s_TUT_struct1 TUT_struct1;
```

A C++ struct is compatible with C; however, its name may not be accessible to C since it may be defined within a namespace. By creating an identical struct in the C wrapper, we’re guaranteed visibility for the C API.

---

**Note:** All fields must be defined in the YAML file in order to ensure that `sizeof` operator will return the same value for the C and C++ structs.

---

This will generate a Fortran derived type which is compatible with C++:

```
type, bind(C) :: struct1  
    integer(C_INT) :: ifield  
    real(C_DOUBLE) :: dfield  
end type struct1
```

A function which returns a struct value can have its value copied into a Fortran variable where the fields can be accessed directly by Fortran. A C++ function which initialized a struct can be written as:

```
- decl: struct1 returnStructByValue(int i, double d);
```

The C wrapper casts the C++ struct to the C struct by using pointers to the struct then returns the value by dereferencing the C struct pointer.



```
TUT_struct1 TUT_return_struct_by_value(int i, double d)
{
    Cstruct1 SHCXX_rv = returnStructByValue(i, d);
    TUT_cstruct1 * SHC_rv = static_cast<TUT_cstruct1 *>(
        static_cast<void *>(&SHCXX_rv));
    return *SHC_rv;
}
```

This function can be called directly by Fortran using the generated interface:

```
function return_struct_by_value(i, d) &
    result(SHT_rv) &
    bind(C, name="TUT_return_struct_by_value")
    use iso_c_binding, only : C_DOUBLE, C_INT
    import :: struct1
    implicit none
    integer(C_INT), value, intent(IN) :: i
    real(C_DOUBLE), value, intent(IN) :: d
    type(struct1) :: SHT_rv
end function return_struct
```

To use the function:

```
type(struct1) var

var = return_struct(1, 2.5)
print *, var%ifield, var%dfield
```

## 3.11 Classes

Each class is wrapped in a Fortran derived type which shadows the C++ class by holding a `type(C_PTR)` pointer to an C++ instance. Class methods are wrapped using Fortran's type-bound procedures. This makes Fortran usage very similar to C++.

Now we'll add a simple class to the library:

```
class Class1
{
public:
    void Method1() {};
};
```

To wrap the class add the lines to the YAML file:

```
declarations:
- decl: class Class1
  declarations:
  - decl: Class1() +name(new)
    format:
      function_suffix: _default
  - decl: ~Class1() +name(delete)
  - decl: int Method1()
```

The constructor and destructor have no method name associated with them. They default to **ctor** and **dtor**. The names can be overridden by supplying the **+name** annotation. These declarations will create wrappers over the `new` and `delete` C++ keywords.

The C++ code to call the function:

```
#include <tutorial.hpp>
tutorial::Class1 *cptr = new tutorial::Class1();

cptr->Method1();
```

And the Fortran version:

```
use tutorial_mod
type(class1) cptr

cptr = class1_new()
call cptr%method1
```

Python usage:

```
import tutorial
obj = tutorial.Class1()
obj.method1()
```

For more details see *Structs and Classes*.

## CHAPTER 4

---

### Input

---

The input to Shroud is a YAML formatted file. YAML is a human friendly data serialization standard. [yaml] Structure is shown through indentation (one or more spaces). Sequence items are denoted by a dash, and key value pairs within a map are separated by a colon:

```
library: Tutorial

declarations:
- decl: typedef int TypeID

- decl: void Function1()

- decl: class Class1
  declarations:
  - decl: void Method1()
```

Each decl entry corresponds to a line of C or C++ code. The top level declarations field represents the source file while nested declarations fields corresponds to curly brace blocks. The above YAML file represent the source file:

```
typedef int TypeID;

void Function1();

class Class1
{
    void Method1();
}
```

A block can be used to group a collection of decl entires. Any option or format fields will apply to all declarations in the group:

```
declarations:
- block: True
  options:
```

(continues on next page)

(continued from previous page)

```

F_name_impl_template: {library}_{underscore_name}
format:
  F_impl_filename: localfile.f
declarations:
- decl: void func1()
- decl: void func2()

```

Shroud use curly braces for format strings. If a string starts with a curly brace YAML will interpret it as a map/dictionary instead of as part of the string. To avoid this behavior, strings which start with a curly brace should be quoted:

```
name : "{fmt}"
```

Strings may be split across several lines by indenting the continued line:

```

- decl: void Sum(int len, const int *values+rank(1),
               int *result+intent(out))

```

Some values consist of blocks of code. The pipe, |, is used to indicate that the string will span several lines and that newlines should be preserved:

```

C_invalid_name: |
  if (! isNameValid({cxx_var})) {{
    return NULL;
  }}

```

Note that to insert a literal {, a double brace, {{, is required since single braces are used for variable expansion. {cxx\_var} in this example. However, using the pipe, it is not necessary to quote lines that contain other YAML meta characters such as colon and curly braces.

For example, YAML will get confused by the :: characters and try to create a dictionary with the key integer, parameter ::.

```

splicer_code:
  f:
    module_top:
      - integer, parameter :: INDEXTYPE = 5

```

Literal newlines, /n, are respected. Line lengths are controlled by the options *C\_line\_length* and *F\_line\_length* and default to 72.:

```

C_invalid_name: |
  if (! isNameValid({cxx_var})) {{+
  return NULL;
  -}}

```

The only formatting option is to control output line lengths. This is required for Fortran which has a maximum line length of 132 in free form which is generated by shroud. If you care where curly braces go in the C source then it is best to set *C\_line\_length* to a large number then use an external formatting tool such as *indent* or *uncrustify*.

## 4.1 Customizing Behavior in the YAML file

### 4.1.1 Fields

A field only applies to the type, enumeration, function, structure or class to which it belongs. It is not inherited. For example, `cxx_header` is a field which is used to define the header file for class `Names`. Likewise, setting `library` within a class does not change the library name.

```
library: testnames

declarations:
- decl: class Names
  cxx_header: names.hpp
  declarations:
  - decl: void method1
```

### 4.1.2 Options

Options are used to customize the behavior of Shroud. They are defined in the YAML file as a dictionary. Options can be defined at the global, class, or function level. Each level creates a new scope which can access all upper level options. This allows the user to modify behavior for all functions or just a single one:

```
options:
  option_a = false
  option_b = false
  option_c = false

declarations:
- class: class1
  options:
  #   option_a = false      # inherited
    option_b = true
  #   option_c = false      # inherited
  declarations:
  - decl: void function1
    options:
  #   option_a = false      # inherited
  #   option_b = true       # inherited
    option_c = true
```

## 4.2 Format

A format dictionary contains strings which can be inserted into generated code. Generated filenames are also entries in the format dictionary. Format dictionaries are also scoped like options. For example, setting a format in a class also effects all of the functions within the class.

### 4.2.1 How code is formatted

Format strings contain “replacement fields” surrounded by curly braces `{ }`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{ }` and `}}`. [\[Python\\_Format\]](#)

There are some metacharacters that are used for formatting the line:

`\f`

Add an explicit formfeed

`\t`

A tab is used to suggest a place to break the line for a continuation before it exceeds option `C_line_length` or `F_line_length`. Any whitespace after a tab will be trimmed if the line is actually split at the tab. If a continuation was not needed (there was enough space on the current line) then the tab has no effect:

```
arg1,\t arg2
```

`+ -`

Increase or decrease indentation level. Used at the beginning or end of a line:

```
if (condition) { {+
do_one();
-} } else { {+
do_two();
-} }
```

The double curly braces are replaced by a single curly. This will be indented as:

```
if (condition) {
    do_one();
} else {
    do_two();
}
```

`#`

If the first character is a `#`, ignore indentation and write in column 0. Useful for preprocessing directives.

`^`

If the first character is `^`, ignore indentation and write in column 0. Useful for comments or labels.

`@`

If the first character is `@`, treat the following character literally. Used to ignore a metacharacter:

```
struct aa = { {++
0// set field to 0
@0,
-} };
```

Formatted as:

```
struct aa = {
// set field to 0
    0,
};
```

## 4.3 Attributes

Annotations or attributes apply to specific arguments or results. They describe semantic behavior for an argument. An attribute may be set to true by listing its name or it may have a value in parens:

```
- decl: Class1() +name(new)
- decl: void Sum(int len, const int *values+rank(1)+intent(in))
- decl: const std::string getName() +len(30)
```

Attributes may also be added external to *decl*:

```
- decl: void Sum(int len, const int *values)
  attrs:
    values:
      intent: in
      rank: 1
- decl: const std::string getName()
  fattrs:
    len: 30
```

Attributes must be added before default arguments since a default argument may include a plus symbol:

```
- decl: void Sum(int len, const int *values+rank(1)+intent(in) =nullptr)
```

### 4.3.1 api

Controls the API used by the C wrapper. The values are *capi*, *buf*, *capsule*, *capptr*, *cdesc* and *cfi*. Normally, this attribute is determined by Shroud internally. Scalar native types such as `int` and `double` will use *capi* since the argument can be passed directly to C using the *interoperability with C* feature of Fortran.

Otherwise a ‘bufferify’ wrapper will also be created. Pointers to native and `char` use additional metadata extracted by the Fortran wrapper via intrinsics `LEN` and `SIZE`. In addition, *intent(in)* strings will be copied and null-terminated. This uses *api(buf)*.

*cdesc* will pass down a pointer to a struct which contains metadata for the argument instead of passing additional fields. The advantage is the struct can also be used to return metadata from the C wrapper to the Fortran wrapper. The struct is named by the format fields *C\_array\_type* and *F\_array\_type*.

The option *F\_CFI*, will use the *Further interoperability with C* features and pass `CFI_cdesc_t` arguments to the C where where the metadata is extracted. This uses *api(cfi)*.

The *capsule* and *capptr* APIs are used by the capsule created by shadow types created for C++ classes. In both cases the result is passed from Fortran to C as an extra argument for function which return a class. With *capptr*, the C wrapper will return a pointer to the capsule argument while *capsule* will not return a value for the function. This is controlled by the *C\_shadow\_result* option.

There is currently one useful case where the user would want to set this attribute. To avoid creating a wrapper which copies and null terminates a `char *` argument the user can set *api(capi)*. The address of the formal parameter will be passed to the user’s library. This is useful when null termination does not make sense. For example, when the argument is a large buffer to be written to a file. The C library must have some other way of determining the length of the argument such as another argument with the explicit length.

### 4.3.2 assumedtype

When this attribute is applied to a `void *` argument, the Fortran assumed-type declaration, `type(*)`, will be used. Since Fortran defaults to pass-by-reference, the argument will be passed to C as a `void *` argument. The C function will need some other mechanism to determine the type of the argument before dereferencing the pointer. Note that *assumed-type* is part of Fortran 2018.

### 4.3.3 blanknull

Used with `const char *` arguments to convert a blank string to a `NULL` pointer instead of an empty C string (`'\0'`). Can be applied to all arguments with the option **F\_blanknull**.

### 4.3.4 capsule

Name of capsule argument. Defaults to `C_var_capsule_template`.

### 4.3.5 cdesc

Pass argument from Fortran to C wrapper as a pointer to a context type. This struct contains the address, type, rank and size of the argument. A ‘bufferify’ function will be created for the context type.

### 4.3.6 charlen

*charlen* is used to define the size of a `char *arg+intent(out)` argument in the Python wrapper. This deals with the case where *arg* is provided by the user and the function writes into the provided space. This technique has the inherent risk of overwriting memory if the supplied buffer is not long enough. For example, when used in C the user would write:

```
#define API_CHARLEN
char buffer[API_CHARLEN];
fill_buffer(buffer);
```

The Python wrapper must know the assumed length before calling the function. It will then be converted into a *str* object by `PyString_FromString`.

Fortran does not use this attribute since the *buffer* argument is supplied by the user. However, it is useful to provide the parameter by adding a splicer block in the YAML file:

```
splicer_code:
  f:
    module_top:
      - "integer, parameter :: MAXNAME = 20"
```

**Warning:** Using *charlen* and *dimension* together is not currently supported.

### 4.3.7 default

Default value for C++ function argument. This value is implied by C++ default argument syntax.

### 4.3.8 deref

Define how to dereference function results and pointers which are returned via an argument. It’s also used with objects which represent an array such as `std::string` or `std::vector`. This may be used in conjunction with *dimension* to create arrays. For example, `int **out +intent(out)+deref(pointer)+dimension(10)`.

allocatable



For Fortran, add `ALLOCATABLE` attribute to argument. An `ALLOCATE` statement is added and the contents of the C++ argument is copied. If *owner(caller)* is also defined, the C++ argument is released. The caller is responsible to `DEALLOCATE` the array.

For Python, create a NumPy array (same as *pointer* attribute)

arg

Added by Shroud when a function result needs to be passed as an additional argument from the Fortran wrapper to the C wrapper.

copy

Copy results into the Fortran argument. This helps reduce memory management problems since there is no dynamic memory. In addition, this helps with non-contiguous C++ memory such as arrays or vectors of `char *` or `std::string`. Fortran can not deal with ragged arrays directly and will copy into the contiguous argument.

pointer

For *intent(in)* arguments, a `POINTER` Fortran attribute will be added. This allows a dynamic memory address to be pass to the library.

```
void giveMemory(arg *data +intent(in)+deref(pointer))
```

For *intent(out)* arguments this indicates that memory from the library is being passed back to the user and will be assigned using `c_f_pointer`.

If *owner(caller)* is also defined, an additional argument is added which is used to release the memory.

For Python, create a list or NumPy array.

```
- decl: double *ReturnPtrFun() +dimension(10)
- decl: void ReturnPtrArg(double **arg +intent(out)+dimension(10))
- decl: double *ReturnScalar() +deref(pointer)
```

A *pointer* to scalar will also return a NumPy array in Python. Use *+deref(scalar)* to get a scalar.

raw

For Fortran, return a `type(C_PTR)`.

For Python, return a `PyCapsule`.

scalar

Treat the pointee as a scalar. For Fortran, return a scalar and not a pointer to the scalar. For Python, this will not create a NumPy object.

### 4.3.9 dimension

A list of array extents for pointer or reference variables. All arrays use the language's default lower-bound (1 for Fortran and 0 for Python). Used to define the dimension of pointer arguments with *intent(out)* and function results. It can also be used with class member variables to create a getter which returns a Fortran pointer. A dimension without any value is an error – *+dimension*.

The expression is evaluated in the C wrapper. It can be passed back to the Fortran wrapper via a `cdesc` argument of type *F\_array\_type* when the attribute *deref* is set to *allocatable* or *pointer*. This allows the shape to be used in an `ALLOCATE` statement or a call to `C_F_POINTER`.

For *Futher interoperability with C*, set with option `F_CFI`, the shape is used directly in the C wrapper in a call to `CFI_allocate` or `CFI_establish`.

```
struct {
    int len;
    double *array +dimension(len);
};
```

An expression can also contain a *intent(out)* argument of the function being wrapped.

```
int * get_array(int **count +intent(out)+hidden) +dimension(count)
```

Argument `count` will be used to define the shape of the function result but will not be part of the wrapped API since it is *hidden*.

*rank* and *dimension* can not be specified together.

The dimension may also be assumed-rank, *dimension(..)*, to allow scalar or any rank. If option `F_CFI` is *true*, then assumed-rank will be added to the function interface and the C wrapper will extract the rank from the `CFI_cdesc_t` argument. Otherwise, a generic function will be created for each rank requested by options `F_assumed_rank_min` and `F_assumed_rank_max`.

### 4.3.10 external

This attribute is only valid with function pointers. It will ensure that a Fortran wrapper is created which uses the `external` statement for the argument. This will allow any function to be used as the dummy argument for the function pointer.

### 4.3.11 free\_pattern

A name in the **patterns** section which lists code to be used to release memory. Used with function results. It is used in the `C_memory_dtor_function` and will have the variable `void *ptr` available as the pointer to the memory to be released. See [Memory Management](#) for details.

### 4.3.12 hidden

The argument will not appear in the Fortran API.

For the native C API it will appear as a regular argument. For the bufferify C API, it will be a local variable which is passed to the C++ function.

It is useful for a function which returns the length of another pointer arguments. This value is save in the `F_array_type` argument or the `CFI_cdesc_t` struct.

For example, setting the shape of a pointer function:

```
int * ReturnIntPtr(int *len+intent(out)+hidden) +dimension(len)+deref(pointer)
```

Will create a Fortran wrapper which returns a ``POINTER`` which is ``len`` long but does not have an argument for the length.

```
integer(C_INT), pointer :: rv(:)
rv = return_int_ptr()
! size(rv) is argument len
```

### 4.3.13 implied

The value of an arguments to the C++ function may be implied by other arguments. If so the *implied* attribute can be used to assign the value to the argument and it will not be included in the wrapped API.

Used to compute value of argument to C++ based on argument to Fortran or Python wrapper. Useful with array sizes:

```
int Sum(const int * array, int len +implied(size(array))
```

Several functions will be converted to the corresponding code for Python wrappers: `size`, `len` and `len_trim`.

- `size(array[,dim])` Determine the extent of **array** along a specified dimension **dim**, or the total number of elements in **array** if **dim** is absent.
  - **array** name of argument
  - **dim** rank of array to check. If none, entire array.
- `len(string)` Returns the length of a character string.
- `len_trim(string)` Returns the length of a character string, ignoring any trailing blanks.

### 4.3.14 intent

The Fortran intent of the argument. Valid values are `in`, `out`, `inout`.

**in** The argument will only be read from.

**inout** The argument will be read from and written to.

**out** The argument will be written to.

Nonpointer arguments can only be *intent(in)*. If the argument is `const`, the default is `in`.

In Python, *intent(out)* arguments are not used as input arguments to the function but are returned as values.

Internally, Shroud also assigns the values of *function*, *ctor* and *dtor*.

### 4.3.15 len

When used with a function, it will be the length of the return value of the function using the declaration:

```
character(kind=C_CHAR, len={c_var_len}) :: {F_result}
```

### 4.3.16 name

Name of the method. Useful for constructor and destructor methods which have default names `ctor` and `dtor`. Also useful when class member variables use a convention such as `m_variable`. The *name* can be set to *variable* to avoid polluting the Fortran interface with the `m_` prefix. Fortran and Python both have an explicit scope of `self%variable` and `self.variable` instead of an implied `this`.

### 4.3.17 owner

Specifies who is responsible to release the memory associated with the argument/result.

The terms follow Python's reference counting . [\[Python\\_Refcount\]](#) The default is set by option *default\_owner* which is initialized to *borrow*.

caller

The memory belongs to the user who is responsible to delete it. A shadow class must have a destructor wrapped in order to delete the memory.

library

The memory belongs to the library and should not be deleted by the user. This is the default value.

### 4.3.18 pass

Used to define the argument which is the passed-object dummy argument for type-bound procedures when treating a struct as a class. In C, which does not support the `class` keyword, a `struct` can be used as a class by defining option `wrap_struct_as=class`. Other functions can be associated with the class by setting option `class_method` to the name of the struct.

See detail at [Object-oriented C](#)

### 4.3.19 rank

Add an assumed-shape dimension with the given rank. *rank* must be 0-7. A rank of 0 implies a scalar argument.

```
double *array +rank(2)
```

Creates the declaration:

```
real(C_DOUBLE) :: array(:, :)
```

Use with `+intent(in)` arguments when the wrapper should accept any extent instead of using Fortran's assumed-shape with `dimension(:)`.

This can be simpler than the *dimension* attribute for multidimension arrays. *rank* and *dimension* can not be specified together.

For the `bind(C)` interface, an assumed-size array will be created for any array with `rank > 0`.

```
real(C_DOUBLE) :: array(*)
```

### 4.3.20 readonly

May be added to struct or class member to avoid creating a setter function. If the member is *const*, this attribute is added by Shroud.

### 4.3.21 value

If true, pass-by-value; else, pass-by-reference. This attribute is implied when the argument is not a pointer or reference. This will also default to `intent(IN)` since there is no way to return a value.

---

**Note:** The Fortran wrapper may use an intrinsic function for some attributes. For example, *len*, *len\_trim*, and *size*. If there is an argument with the same name, the generated code may not compile.

Shroud preserves the names of the arguments since Fortran allows them to be used in function calls - `call worker(len=10)`

---

## 4.4 Statements

The code generated for each argument and return value can be controlled by statement dictionaries. Shroud has many entries built in which are used for most arguments. But it is possible to add custom code to the wrapper by providing additional fields. Most wrappers will not need to provide this information.

An example from strings.yaml:

```
- decl: const string * getConstStringPtrLen() +len=30
  doxygen:
    brief: return a 'const string *' as character(30)
    description: |
      It is the caller's responsibility to release the string
      created by the C++ library.
      This is accomplished with C_finalize_buf which is possible
      because +len(30) so the contents are copied before returning.
  fstatements:
    c_buf:
      final:
        - delete {cxx_var};
```

An example from vectors.yaml:

```
- decl: void vector_iota_out_with_num(std::vector<int> &arg+intent(out))
  fstatements:
    c_buf:
      return_type: long
      ret:
        - return Darg->size;
    f:
      result: num
      f_module:
        iso_c_binding: ["C_LONG"]
      declare:
        - "integer(C_LONG) :: {F_result}"
      call:
        - "{F_result} = {F_C_call}({F_arg_c_call})"
```

## 4.5 Patterns

To address the issue of semantic differences between Fortran and C++, *patterns* may be used to insert additional code. A *pattern* is a code template which is inserted at a specific point in the wrapper. They are defined in the input YAML file:

```
declarations:
- decl: const string& getString2+len=30()
  C_error_pattern: C_invalid_name
patterns:
  C_invalid_name: |
    if ({cxx_var}.empty()) {{
      return NULL;
    }}
```

The **C\_error\_pattern** will insert code after the call to the C++ function in the C wrapper and before any `post_call`

sections from the types. The buffered version of a function will append `_buf` to the `C_error_pattern` value. The *pattern* is formatted using the context of the return argument if present, otherwise the context of the function is used. This means that `c_var` and `c_var_len` refer to the argument which is added to contain the function result for the `_buf` pattern.

The function `getString2` is returning a `std::string` reference. Since C and Fortran cannot deal with this directly, the empty string is converted into a NULL pointer:: will blank fill the result:

```
const char * STR_get_string2()
{
    const std::string & SHCXX_rv = getString2();
    // C_error_pattern
    if (SHCXX_rv.empty()) {
        return NULL;
    }
    const char * SHC_rv = SHCXX_rv.c_str();
    return SHC_rv;
}
```

## 4.6 Splicers

No matter how many features are added to Shroud there will always exist cases that it does not handle. One of the weaknesses of generated code is that if the generated code is edited it becomes difficult to regenerate the code and preserve the edits. To deal with this situation each block of generated code is surrounded by ‘splicer’ comments:

```
const char * STR_get_char3()
{
    // splicer begin function.get_char3
    const char * SH_rv = getChar3();
    return SH_rv;
    // splicer end function.get_char3
}
```

These comments delineate a section of code which can be replaced by the user. The splicer’s name, `function.get_char3` in the example, is used to determine where to insert the code.

There are two ways to define splicers in the YAML file. First add a list of files which contain the splicer text:

```
splicer:
  f:
  - fsplicer.f
  c:
  - csplicer.c
```

In the listed file, add the begin and end splicer comments, then add the code which should be inserted into the wrapper inbetween the comments. Multiple splicer can be added to an input file. Any text that is not within a splicer block is ignored. Splicers must be sorted by language. If the input file ends with `.f` or `.f90` it is processed as splicers for the generated Fortran code. Code for the C wrappers must end with any of `.c`, `.h`, `.cpp`, `.hpp`, `.cxx`, `.hxx`, `.cc`, `.C`:

```
-- Lines outside blocks are ignore
// splicer begin function.get_char3
const char * SH_rv = getChar3();
SH_rv[0] = 'F';    // replace first character for Fortran
return SH_rv + 1;
// splicer end function.get_char3
```

This technique is useful when the splicers are very large or are generated by some other process.

The second method is to add the splicer code directly into the YAML file. A splicer can be added after the `decl` line. This splicer takes priority over other ways of defining splicers.

```
- decl: bool isNameValid(const std::string& name)
  splicer:
    c:
      - "return name != NULL;"
    f:
      - 'rv = name .ne. " "'
```

A splicer can be added in the `splicer_code` section. This can be used to add code to spliers which do not correspond directly to a declaration. Each level of splicer is a mapping and each line of text is an array entry:

```
splicer_code:
  c:
    function:
      get_char3:
        - const char * SH_rv = getChar3();
        - SH_rv[0] = 'F';    // replace first character for Fortran
        - return SH_rv + 1;
```

In addition to replacing code for a function wrapper, there are splicers that are generated which allow a user to insert additional code for helper functions or declarations:

```
! file_top
module {F_module_name}
  ! module_use
  implicit none
  ! module_top

  type class1
    ! class.{cxx_class}.component_part
  contains
    ! class.{cxx_class}.generic.{F_name_generic}
    ! class.{cxx_class}.type_bound_procedure_part
  end type class1

  ! additional_declarations

  contains

  ! function.{F_name_function}

  ! {cxx_class}.method.{F_name_function}

  ! additional_functions

end module {F_module_name}
```

C header:

```
// class.{class_name}.CXX_declarations

extern "C" {
// class.{class_name}.C_declarations
}
```

C implementation:

```
// class.{class_name}.CXX_definitions

extern "C" {
    // class.{class_name}.C_definitions

    // function.{underscore_name}{function_suffix}

    // class.{cxx_class}.method.{underscore_name}{function_suffix}
}
```

The splicer comments can be eliminated by setting the option **show\_splicer\_comments** to false. This may be useful to eliminate the clutter of the splicer comments.

## 4.7 file\_code

The `file_code` section allows the user to add some additional code to the wrapper which may conflict with code automatically added by Shroud for typemaps, statements or helpers. While splicer are simple text insertion, `file_code` inserts code semantically.

For C wrappers, including header files may duplicate headers added when creating the wrapper. By listing them in a `file_code` section instead of a splicer Shroud is able to manage all header files.

For Fortran wrappers, USE statements are managed collectively to avoid redundant USE statements.

```
file_code:
    wraptypemap.h:
        c_header: <stdint.h>
        cxx_header: <cstdint>
    wrapftypemap.f:
        f_module:
            iso_c_binding:
                - C_INT32_T
                - C_INT64_T
```



---

## Pointers and Arrays

---

Shroud will create code to map between C and Fortran pointers. The *interoperability with C* features of Fortran 2003 and the call-by-reference feature of Fortran provides most of the features necessary to pass arrays to C++ libraries. Shroud can also provide additional semantic information. Adding the *rank* attribute will declare the argument as an assumed-shape array with the given rank: `+rank(2)` creates `arg(:, :)`. The `+dimension(n)` attribute will instead give an explicit dimension: `+dimension(10, 20)` creates `arg(10, 20)`.

Using *dimension* on *intent(in)* arguments will use the dimension shape in the Fortran wrapper instead of assumed-shape. This adds some additional safety since many compiler will warn if the actual argument is too small. This is useful when the C++ function has an assumed shape. For example, it expects a pointer to 16 elements. The Fortran wrapper will pass a pointer to contiguous memory with no explicit shape information.

When a function returns a pointer, the default behavior of Shroud is to convert it into a Fortran variable with the `POINTER` attribute using `c_f_pointer`. This can be made explicit by adding `+deref(pointer)` to the function declaration in the YAML file. For example, `int *getData(void) +deref(pointer)` creates the Fortran function interface

```
function get_data() result(rv)
  integer(C_INT), pointer :: rv
end function get_data
```

The result of the the Fortran function directly accesses the memory returned from the C++ library.

An array can be returned by adding the *dimension* attribute to the function. The dimension expression will be used to provide the shape argument to `c_f_pointer`. The arguments to *dimension* are C++ expressions which are evaluated after the C++ function is called and can be the name of another argument to the function or a call another C++ function. As a simple example, this declaration returns a pointer to a constant sized array.

```
- decl: int *returnIntPtrToFixedArray(void) +dimension(10)
```

Example [returnIntPtrToFixedArray](#) shows the generated code.

If the dimension is unknown when the function returns, a `type(C_PTR)` can be returned with `+deref(raw)`. This will allow the user to call `c_f_pointer` once the shape is known. Instead of a Fortran pointer to a scalar, a scalar can be returned by adding `+deref(scalar)`.

A common idiom for C++ is to return pointers to memory via arguments. This would be declared as `int **arg +intent(out)`. By default, Shroud treats the argument similar to a function which returns a pointer: it adds the *deref(pointer)* attribute to treat it as a `POINTER` to a scalar. The *dimension* attribute can be used to create an array similar to a function result. If the *deref(allocatable)* attribute is added, then a Fortran array will be allocated to the size of *dimension* attribute and the argument will be copied into the Fortran memory.

A function which returns multiple layers of indirection will return a `type(C_PTR)`. This is also true for function arguments beyond `int **arg +intent(out)`. This pointer can represent non-contiguous memory and Shroud has no way to know the extent of each pointer in the array.

A special case is provided for arrays of `NULL` terminated strings, `char **`. While this also represents non-contiguous memory, it is a common idiom and can be processed since the length of each string can be found with `strlen`. See example [acceptCharArrayIn](#).

In Python wrappers, Shroud will allocate *intent(out)* arguments before calling the function. This requires the *dimension* attribute which defines the shape and must be known before the function is called. The argument will then be returned by the function along with the function result and other *intent(out)* arguments. For example, `int **arg +intent(out)+dimension(n)`. The value of the *dimension* attribute is used to define the shape of the array and must be known before the library function is called. The *dimension* attribute can include the Fortran intrinsic `size` to define the shape in terms of another array.

`char *` functions are treated differently. By default *deref* attribute will be set to *allocatable*. After the C++ function returns, a `CHARACTER` variable will be allocated and the contents copied. This will convert a `NULL` terminated string into the proper length of Fortran variable. For very long strings or strings with embedded `NULL`, `deref(raw)` will return a `type(C_PTR)`.

`void *` functions return a `type(C_PTR)` argument and cannot have *deref*, *dimension*, or *rank* attributes. A `type(C_PTR)` argument will be passed by value. For a `void **` argument, the `type(C_PTR)` will be passed by reference (the default). This will allow the C wrapper to assign a value to the argument. See example [passVoidStarStar](#).

If the C++ library function can also provide the length of the pointer, then it's possible to return a Fortran `POINTER` or `ALLOCATABLE` variable. This allows the caller to directly use the returned value of the C++ function. However, there is a price; the user will have to release the memory if *owner(caller)* is set. To accomplish this with `POINTER` arguments, an additional argument is added to the function which contains information about how to delete the array. If the argument is declared Fortran `ALLOCATABLE`, then the value of the C++ pointer are copied into a newly allocated Fortran array. The C++ memory is deleted by the wrapper and it is the caller's responsibility to deallocate the Fortran array. However, Fortran will release the array automatically under some conditions when the caller function returns. If *owner(library)* is set, the Fortran caller never needs to release the memory.

See [Memory Management](#) for details of the implementation.

A void pointer may also be used in a C function when any type may be passed in. The attribute *assumedtype* can be used to declare a Fortran argument as assumed-type: `type(*)`.

```
- decl: int passAssumedType(void *arg+assumedtype)
```

```
function pass_assumed_type(arg) &
  result(SHT_rv) &
  bind(C, name="passAssumedType")
  use iso_c_binding, only : C_INT, C_PTR
  implicit none
  type(*) :: arg
  integer(C_INT) :: SHT_rv
end function pass_assumed_type
```

## 5.1 Memory Management

Shroud will maintain ownership of memory via the **owner** attribute. It uses the value of the attribute to decide when to release memory.

Use **owner(library)** when the library owns the memory and the user should not release it. For example, this is used when a function returns `const std::string &` for a reference to a string which is maintained by the library. Fortran and Python will both get the reference, copy the contents into their own variable (Fortran `CHARACTER` or Python `str`), then return without releasing any memory. This is the default behavior.

Use **owner(caller)** when the library allocates new memory which is returned to the caller. The caller is then responsible to release the memory. Fortran and Python can both hold on to the memory and then provide ways to release it using a C++ callback when it is no longer needed.

For shadow classes with a destructor defined, the destructor will be used to release the memory.

The *c\_statements* may also define a way to destroy memory. For example, `std::vector` provides the lines:

```
destructor_name: std_vector_{cxx_T}
destructor:
- std::vector<{cxx_T}> *cxx_ptr = reinterpret_cast<std::vector<{cxx_T}>*>(ptr);
- delete cxx_ptr;
```

Patterns can be used to provide code to free memory for a wrapped function. The address of the memory to free will be in the variable `void *ptr`, which should be referenced in the pattern:

```
declarations:
- decl: char * getName() +free_pattern(free_getName)

patterns:
  free_getName: |
    decref(ptr);
```

Without any explicit *destructor\_name* or pattern, `free` will be used to release POD pointers; otherwise, `delete` will be used.

## 5.2 C and Fortran

Fortran keeps track of C++ objects with the struct **C\_capsule\_data\_type** and the `bind(C)` equivalent **F\_capsule\_data\_type**. Their names in the format dictionary default to `{C_prefix}SHROUD_capsule_data` and `{C_prefix}SHROUD_capsule_data`. In the Tutorial these types are defined in `typesTutorial.h` as:

```
// helper capsule_CLA_Class1
struct s_CLA_Class1 {
    void *addr;      /* address of C++ memory */
    int idtor;       /* index of destructor */
};
typedef struct s_CLA_Class1 CLA_Class1;
```

And `wrapftutorial.f`:

```
! helper capsule_data_helper
type, bind(C) :: CLA_SHROUD_capsule_data
  type(C_PTR) :: addr = C_NULL_PTR ! address of C++ memory
  integer(C_INT) :: idtor = 0 ! index of destructor
end type CLA_SHROUD_capsule_data
```

*addr* is the address of the C or C++ variable, such as a `char *` or `std::string *`. *idtor* is a Shroud generated index of the destructor code defined by *destructor\_name* or the *free\_pattern* attribute. These code segments are collected and written to function *C\_memory\_dtor\_function*. A value of 0 indicated the memory will not be released and is used with the **owner(library)** attribute.

Each class creates its own capsule struct for the C wrapper. This is to provide a measure of type safety in the C API. All Fortran classes use the same derived type since the user does not directly access the derived type.

A typical destructor function would look like:

```
// Release library allocated memory.
void TUT_SHROUD_memory_destructor(TUT_SHROUD_capsule_data *cap)
{
    void *ptr = cap->addr;
    switch (cap->idtor) {
        case 0:    // --none--
        {
            // Nothing to delete
            break;
        }
        case 1:    // new_string
        {
            std::string *cxx_ptr = reinterpret_cast<std::string *>(ptr);
            delete cxx_ptr;
            break;
        }
        default:
        {
            // Unexpected case in destructor
            break;
        }
    }
    cap->addr = nullptr;
    cap->idtor = 0;    // avoid deleting again
}
```

## 5.2.1 Character and Arrays

In order to create an allocatable copy of a C++ pointer, an additional structure is involved. For example, `getConstStringPtrAlloc` returns a pointer to a new string. From `strings.yaml`:

```
declarations:
- decl: const std::string * getConstStringPtrAlloc() +owner(library)
```

The C wrapper calls the function and saves the result along with metadata consisting of the address of the data within the `std::string` and its length. The Fortran wrappers allocates its return value to the proper length, then copies the data from the C++ variable and deletes it.

The metadata for variables are saved in the C struct **C\_array\_type** and the `bind(C)` equivalent **F\_array\_type**:

```
// helper array_context
struct s_STR_SHROUD_array {
    STR_SHROUD_capsule_data cxx;    /* address of C++ memory */
    union {
        const void * base;
        const char * ccharp;
    } addr;
}
```

(continues on next page)

(continued from previous page)

```

int type;          /* type of element */
size_t elem_len;   /* bytes-per-item or character len in c++ */
size_t size;       /* size of data in c++ */
int rank;          /* number of dimensions, 0=scalar */
long shape[7];
};
typedef struct s_STR_SHROUD_array STR_SHROUD_array;

```

The union for `addr` makes some assignments easier by removing the need for casts and also aids debugging. The union is replaced with a single type (`C_PTR`) for Fortran:

```

! helper array_context
type, bind(C) :: STR_SHROUD_array
! address of C++ memory
type(STR_SHROUD_capsule_data) :: cxx
! address of data in cxx
type(C_PTR) :: base_addr = C_NULL_PTR
! type of element
integer(C_INT) :: type
! bytes-per-item or character len of data in cxx
integer(C_SIZE_T) :: elem_len = 0_C_SIZE_T
! size of data in cxx
integer(C_SIZE_T) :: size = 0_C_SIZE_T
! number of dimensions
integer(C_INT) :: rank = -1
integer(C_LONG) :: shape(7) = 0
end type STR_SHROUD_array

```

The C wrapper does not return a `std::string` pointer. Instead it passes in a **C\_array\_type** pointer as an argument. It calls `getConstStringPtrAlloc`, saves the results and metadata into the argument. This allows it to be easily accessed from Fortran. Since the attribute is **owner(library)**, `cxx.idtor` is set to 0 to avoid deallocating the memory.

```

void STR_getConstStringPtrAlloc_bufferify(
    STR_SHROUD_array *SHT_rv_cdesc)
{
    // splicer begin function.getConstStringPtrAlloc_bufferify
    const std::string * SHCXX_rv = getConstStringPtrAlloc();
    ShroudStrToArray(SHT_rv_cdesc, SHCXX_rv, 0);
    // splicer end function.getConstStringPtrAlloc_bufferify
}

```

The Fortran wrapper uses the metadata to allocate the return argument to the correct length:

```

function get_const_string_ptr_alloc() &
    result(SHT_rv)
    character(len=:), allocatable :: SHT_rv
    ! splicer begin function.get_const_string_ptr_alloc
    type(STR_SHROUD_array) :: SHT_rv_cdesc
    call c_get_const_string_ptr_alloc_bufferify(SHT_rv_cdesc)
    allocate(character(len=SHT_rv_cdesc%elem_len):: SHT_rv)
    call STR_SHROUD_copy_string_and_free(SHT_rv_cdesc, SHT_rv, &
        SHT_rv_cdesc%elem_len)
    ! splicer end function.get_const_string_ptr_alloc
end function get_const_string_ptr_alloc

```

Finally, the helper function `SHROUD_copy_string_and_free` is called to set the value of the result and possible

free memory for **owner(caller)** or intermediate values:

```
// helper copy_string
// Copy the char* or std::string in context into c_var.
// Called by Fortran to deal with allocatable character.
void STR_ShroudCopyStringAndFree(STR_SHROUD_array *data, char *c_var, size_t c_var_
↪len) {
    const char *cxx_var = data->addr.ccharp;
    size_t n = c_var_len;
    if (data->elem_len < n) n = data->elem_len;
    std::strncpy(c_var, cxx_var, n);
    STR_SHROUD_memory_destructor(&data->cxx); // delete data->cxx.addr
}
```

---

**Note:** The three steps of call, allocate, copy could be replaced with a single call by using the *further interoperability with C* features of Fortran 2018 (a.k.a TS 29113). This feature allows Fortran ALLOCATABLE variables to be allocated by C. However, not all compilers currently support that feature. The current Shroud implementation works with Fortran 2003.

---

## 5.3 Python

NumPy arrays control garbage collection of C++ memory by creating a PyCapsule as the base object of NumPy objects. Once the final reference to the NumPy array is removed, the reference count on the PyCapsule is decremented. When 0, the *destructor* for the capsule is called and releases the C++ memory. This technique is discussed at [\[blog1\]](#) and [\[blog2\]](#)

## 5.4 Old

---

**Note:** C\_finalize is replaced by statement.final

---

Shroud generated C wrappers do not explicitly delete any memory. However a destructor may be automatically called for some C++ stl classes. For example, a function which returns a `std::string` will have its value copied into Fortran memory since the function's returned object will be destroyed when the C++ wrapper returns. If a function returns a `char *` value, it will also be copied into Fortran memory. But if the caller of the C++ function wants to transfer ownership of the pointer to its caller, the C++ wrapper will leak the memory.

The **C\_finalize** variable may be used to insert code before returning from the wrapper. Use **C\_finalize\_buf** for the buffer version of wrapped functions.

For example, a function which returns a new string will have to `delete` it before the C wrapper returns:

```
std::string * getConstStringPtrLen()
{
    std::string * rv = new std::string("getConstStringPtrLen");
    return rv;
}
```

Wrapped as:

```
- decl: const string * getConstStringPtrLen+len=30()
format:
  C_finalize_buf: delete {cxx_var};
```

The C buffer version of the wrapper is:

```
void STR_get_const_string_ptr_len_bufferify(char * SHF_rv, int NSHF_rv)
{
    const std::string * SHCXX_rv = getConstStringPtrLen();
    if (SHCXX_rv->empty()) {
        std::memset(SHF_rv, ' ', NSHF_rv);
    } else {
        ShroudStrCopy(SHF_rv, NSHF_rv, SHCXX_rv->c_str());
    }
    {
        // C_finalize
        delete SHCXX_rv;
    }
    return;
}
```

The unbuffer version of the function cannot destroy the string since only a pointer to the contents of the string is returned. It would leak memory when called:

```
const char * STR_get_const_string_ptr_len()
{
    const std::string * SHCXX_rv = getConstStringPtrLen();
    const char * SHC_rv = SHCXX_rv->c_str();
    return SHC_rv;
}
```





## 6.1 Numeric Types

The numeric types usually require no conversion. In this case the type map is mainly used to generate declaration code for wrappers:

```
type: int
fields:
  c_type: int
  cxx_type: int
  f_type: integer(C_INT)
  f_kind: C_INT
  f_module:
    iso_c_binding:
      - C_INT
  f_cast: int({f_var}, C_INT)
```

One case where a conversion is required is when the Fortran argument is one type and the C++ argument is another. This may happen when an overloaded function is generated so that a `C_INT` or `C_LONG` argument may be passed to a C++ function expecting a `long`. The **f\_cast** field is used to convert the argument to the type expected by the C++ function.

## 6.2 Bool

The first thing to notice is that **f\_c\_type** is defined. This is the type used in the Fortran interface for the C wrapper. The type is `logical(C_BOOL)` while **f\_type**, the type of the Fortran wrapper argument, is `logical`.

The **f\_statements** section describes code to add into the Fortran wrapper to perform the conversion. *c\_var* and *f\_var* default to the same value as the argument name. By setting **c\_local\_var**, a local variable is generated for the call to the C wrapper. It will be named `SH_{f_var}`.

There is no Fortran intrinsic function to convert between default `logical` and `logical(C_BOOL)`. The **pre\_call** and **post\_call** sections will insert an assignment statement to allow the compiler to do the conversion.

If a function returns a `bool` result then a wrapper is always needed to convert the result. The **result** section sets **need\_wrapper** to force the wrapper to be created. By default a function with no argument would not need a wrapper since there will be no **pre\_call** or **post\_call** code blocks. Only the C interface would be required since Fortran could call the C function directly.

See example [checkBool](#).

## 6.3 Char

Any C++ function which has `char` or `std::string` arguments or result will create an additional C function which include additional arguments for the length of the strings. Most Fortran compiler use this convention when passing `CHARACTER` arguments. Shroud makes this convention explicit for two reasons:

- It allows an interface to be used. Functions with an interface will not pass the hidden, non-standard length argument, depending on compiler.
- Returning character argument from C to Fortran is non-portable.

The C wrapper will create a NULL terminated copy a string with the *intent(in)* attribute. The assumption is that the trailing blanks are not part of the data but only padding. Return values and *intent(out)* arguments add a *len* annotation with the assumption that the wrapper will copy the result and blank fill the argument so it need to know the declared length.

A buffer for *intent(out)* arguments is also create which is one longer than the Fortran string length. This allows space for a C terminating NULL. This buffer is passed to the C library which will copy into it. Upon return, the buffer is copied and blank filled into the user's argument and the intermediate buffer released.

Library functions which return a scalar `char` have a wrapper generated which pass a `char *` argument to the C wrapper where the first element is assigned (`*arg` a.k.a `arg[0]`). Returning a `char` proved to be non-portable while passing the result by reference works on the tested compilers.

The `bufferify` function will be named the same as the original function with the option **C\_bufferify\_suffix** appended to the end. The Fortran wrapper will use the original function name, but call the C wrapper which accepts the length arguments.

Python wrappers may need an additional attribute for *intent(out)* strings to let Shroud know how much space to pass to the function. A function may pass a `char *` argument which the C library will copy into. While this is not a recommended practice since it's easy to overwrite memory, Shroud can deal with it by setting the `+charlen(n)` attribute where *n* is the number of character in the array passed to the function. This is required for Python since strings are immutable. The buffer will be converted into a Python str object then returned to the user. This is not an issue in Fortran since the output buffer is passed in by the caller and will have a known size.

By default, a Fortran blank input string will be converted to an empty string before being passed to the C library. i.e. " " in Fortran is converted to `'\0'` in C. This behavior can be changed to convert the empty string into a NULL pointer by setting the `+blanknull` attribute. This is often more natural for the C library to indicate the absence of a value. The option `F_blanknull` can be used to make this the default for all `const char *` arguments.

On some occasions the copy and null terminate behavior is not wanted. For example, to avoid copying a large buffer or the memory must be operated on directly. In this case using the attribute `+api(capi)` will use the native C API instead of the `bufferify` API for the argument. The library will need some way to determine the length of the string since it will not be passed to the C wrapper. As an alternative the `bufferify` function can be avoided altogether by setting the **F\_create\_bufferify\_function** option to *false*.

The character type maps use the **c\_statements** section to define code which will be inserted into the C wrapper. These actions vary depending on the intent of *in*, *out*, *inout* and *result*.

## 6.4 MPI\_Comm

MPI\_Comm is provided by Shroud and serves as an example of how to wrap a non-native type. MPI provides a Fortran interface and the ability to convert MPI\_comm between Fortran and C. The type map tells Shroud how to use these routines:

```
type: MPI_Comm
fields:
  cxx_type: MPI_Comm
  c_header: mpi.h
  c_type: MPI_Fint
  f_type: integer
  f_kind: C_INT
  f_c_type: integer(C_INT)
  f_c_module:
    iso_c_binding:
      - C_INT
  cxx_to_c: MPI_Comm_c2f({cxx_var})
  c_to_cxx: MPI_Comm_f2c({c_var})
```

This mapping makes the assumption that `integer` and `integer(C_INT)` are the same type.



# CHAPTER 7

---

## Namespaces

---

Namespaces in C++ are used to ensure the symbols in a library will not conflict with any symbols in another library. Fortran and Python both use a module to accomplish the same thing.

The global variable *namespace* is a blank delimited list of namespaces used as the initial namespace. This namespace will be used when accessing symbols in the library, but it will not be used when generating names for wrapper functions.

For example, the library wrapped is associated with the namespace `outer`. There are three functions all with the same name, `worker`. In C++ these functions are accessed by using a fully qualified name: `outer::worker`, `outer::inner1::worker` and `outer::inner2::worker`.

```
namespace outer {
  namespace inner1 {
    void worker();
  } // namespace inner1

  namespace inner2 {
    void worker();
  } // namespace inner2

  void worker();
} // namespace outer
```

The YAML file would look like:

```
library: wrapped
namespace: outer
format:
  C_prefix: WWW_

declarations:
- decl: namespace inner1
  declarations:
- decl: void worker();
```

(continues on next page)

(continued from previous page)

```
- decl: namespace inner2
  declarations:
- decl: void worker();
- decl: void worker();
```

For each namespace, Shroud will generate a C++ header file, a C++ implementation file, a Fortran file and a Python file. The nested namespaces are added to the format field *C\_name\_scope*.

For the C wrapper, all symbols are globally visible and must be unique. The format fields *C\_prefix* and *C\_name\_scope* are used to generate the names. This will essentially “flatten” the namespaces into legal C identifiers.

```
void WWW_worker();
void WWW_inner1_worker();
void WWW_inner2_worker();
```

In Fortran each namespace creates a module. Each module will have a function named *worker*. This makes the user responsible for distinguishing which implementation of *worker* is to be called.

```
subroutine work1
  ! Use a single module, unambiguous
  use wrapped_mod
  call worker
end subroutine work1

subroutine work2
  ! Rename symbol from namespace inner1
  use wrapped_mod
  use wrapped_inner1_mod, inner_worker => worker
  call worker
  call inner_worker
end subroutine work2
```

Each namespace creates a Python module.

```
import wrapped
wrapped.worker()
wrapped.inner1.worker()
```

Several fields in the format dictionary are updated for each namespace: *namespace\_scope*, *C\_name\_scope*, *F\_name\_scope*.

## 7.1 std namespace

Shroud has builtin support for `std::string` and `std::vector`.

---

## Structs and Classes

---

All problems in computer science can be solved by another level of indirection. — David Wheeler

While structs and classes are very similar in C++, Shroud wraps them much differently. Shroud treats structs as they are in C and creates a corresponding derived type for the struct. Shroud wraps classes by creating a shadow class which holds a pointer to the instance then uses Fortran type bound procedures to implement methods.

### 8.1 Class

Classes are wrapped by creating a *shadow class* for the C++ class. A pointer to an instance is saved along with a memory management flag.

Using the tutorial as an example, a simple class is defined in the C++ header as:

```
class Class1
{
public:
    void Method1() {};
};
```

And is wrapped in the YAML as:

```
declarations:
- decl: class Class1
  declarations:
  - decl: int Method1()
  - decl: Class1()
    format:
      function_suffix: _default
  - decl: Class1(int flag)
    format:
      function_suffix: _flag
  - decl: ~Class1() +name(delete)
```

While C++ will provide a default constructor and destructor, they must be listed explicitly to wrap them. They are not assumed since they may be private. The default name of the constructor is `ctor`. The name can be specified with the *name* attribute. If the constructor is overloaded, each constructor must be given the same *name* attribute.

The *function\_suffix* is added to distinguish overloaded constructors. The default is to have a sequential numeric suffix. The *function\_suffix* must not be explicitly set to blank since the name is used by the Fortran *generic* interface.

If no constructor is wrapped, then some other factory method should be available to create instances of the class. There is no way to create it directly from Fortran.

When the constructor is wrapped the destructor should also be wrapper or some other method should be wrapped to release the memory.

### 8.1.1 C

Each class in the YAML file will create a struct in the C wrapper. All of these structs are identical but are named after the class. This is intended to provide some level of type safety by making it harder to accidentally use the wrong class with a method. Shroud refers to this as a capsule.

```
// helper capsule_CLA_Class1
struct s_CLA_Class1 {
    void *addr;      /* address of C++ memory */
    int idtor;       /* index of destructor */
};
typedef struct s_CLA_Class1 CLA_Class1;
```

The C wrapper will extract the address of the instance then call the method.

```
int CLA_Class1_Method1(CLA_Class1 * self)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
    // splicer begin class.Class1.method.Method1
    int SHC_rv = SH_this->Method1();
    return SHC_rv;
    // splicer end class.Class1.method.Method1
}
```

All constructors are very similar. They call the C++ constructor then saves the pointer to the instance. The *idtor* field is the index of the destructor maintained by Shroud to destroy the instance.

```
CLA_Class1 * CLA_Class1_ctor_flag(int flag, CLA_Class1 * SHC_rv)
{
    // splicer begin class.Class1.method.ctor_flag
    classes::Class1 *SHCXX_rv = new classes::Class1(flag);
    SHC_rv->addr = static_cast<void *>(SHCXX_rv);
    SHC_rv->idtor = 1;
    return SHC_rv;
    // splicer end class.Class1.method.ctor_flag
}
```

Finally the wrapper for the destructor. The *addr* field is cleared to avoid a dangling pointer.

```
void CLA_Class1_delete(CLA_Class1 * self)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
```

(continues on next page)



(continued from previous page)

```

// splicer begin class.Class1.method.delete
delete SH_this;
self->addr = nullptr;
// splicer end class.Class1.method.delete
}

```

A function which returns a class, including constructors, is passed a pointer to a *F\_capsule\_data\_type*. The argument's members are filled in by the function. The function will return a `type(C_PTR)` which contains the address of the *F\_capsule\_data\_type* argument. The prototype for the C wrapper function allows it to be used in expressions similar to the way that `strcpy` returns its destination argument. The option *C\_shadow\_result* can be set to *False* to change the function to return *void* instead. The Fortran wrapper API will be unaffected.

C++ functions which return *const* pointers will not create a *const* C wrapper. This is because the C wrapper will return a pointer to the capsule and not the instance.

## 8.1.2 Fortran

The Fortran wrapper uses the object-oriented features added in Fortran 2003. There is one derived type for the library which is used as the capsule. This derived type uses `bind(C)` since it is passed to the C wrapper. Each class uses the same capsule derived type since it is considered an implementation detail and the user should not access it. Then each class creates an additional derived type as the *shadow* class which contains a capsule and has type-bound procedures associated with it.

```

! helper capsule_data_helper
type, bind(C) :: CLA_SHROUD_capsule_data
    type(C_PTR) :: addr = C_NULL_PTR ! address of C++ memory
    integer(C_INT) :: idtor = 0 ! index of destructor
end type CLA_SHROUD_capsule_data

```

The capsule is added to the Fortran shadow class. This derived type can contain type-bound procedures and may not use the `bind(C)` attribute.

```

type class1
    type(SHROUD_CLA_capsule_data) :: cxxmem
contains
    procedure :: delete => class1_delete
    procedure :: method1 => class1_method1
end type class1

```

The wrapper for the method passes the object as the first argument. The argument uses the format field *F\_this* to name the variable and defaults to `obj`. It can be renamed if it conflicts with another argument.

```

function class1_method1(obj) &
    result(SHT_rv)
    use iso_c_binding, only : C_INT
    class(class1) :: obj
    integer(C_INT) :: SHT_rv
    ! splicer begin class.Class1.method.method1
    SHT_rv = c_class1_method1(obj%cxxmem)
    ! splicer end class.Class1.method.method1
end function class1_method1

```

A generic interface with the same name as the class is created to call the constructors for the class. The constructor will initialize the Fortran shadow class.

```
interface class1
  module procedure class1_ctor_default
  module procedure class1_ctor_flag
end interface class1
```

The Fortran wrapped class can be used very similar to its C++ counterpart.

```
use classes_mod
type(class1) var      ! Create Fortran variable.
integer(C_INT) i
var = class1()         ! Allocate C++ class instance.
i = var%method1()
call var%delete
```

Some additional type-bound procedures are created to allow the user to get and set the address of the C++ memory directly. This can be used when the address of the instance is created in some other manner and it needs to be used in Fortran. There is no way to free this memory and it must be released outside of Fortran.

For example, a C++ function creates an instance then passes the address of it to Fortran function `worker`. The shadow class is initialized with the address and can then be used in an object-oriented fashion:

```
subroutine worker(addr) bind(C)
use classes_mod
type(C_PTR), intent(IN) :: addr
type(class1) var
integer(C_INT) i

call var%set_instance(addr)
i = var%method1()
```

Two instances of the class can be compared using the associated method.

```
type(class1) var1, var2
var1 = get_class(1)    ! A library function to fetch an instance
var2 = get_class(2)
if (var1%associated(var2) then
  print *, "Identical instances"
endif
```

These functions names are controlled by format fields `F_name_associated`, `F_name_instance_get` and `F_name_instance_set`. If the names are blank, the functions will not be created.

The `.eq.` operator is also defined.

A full example is at [Constructor and Destructor](#).

### 8.1.3 Python

An `PyObject` is created for each C++ class. It contains the same values as the capsule.

```
typedef struct {
PyObject_HEAD
  classes::Class1 * obj;
  int idtor;
  // splicer begin class.Class1.C_object
  // splicer end class.Class1.C_object
} PY_Class1;
```

The `idtor` argument is used to release memory and described at [Memory Management](#). The splicer allows additional fields to be added by the developer which may be used in function wrappers.

Additional fields can be added to the splicer for custom behavior.

## 8.2 Chained functions

C++ allows function calls to be chained by returning the `this` argument. Several functions can be called in succession on the same object.

```
auto var = Class1()->returnThis()
```

The `return_this` field indicates that the function may be chained so the wrapper can generate appropriate code.

```
- decl: Class1 * returnThis();
  return_this: True
```

### 8.2.1 C

The C wrapper returns `void` instead of a pointer to the `this` argument.

```
void CLA_Class1_returnThis(CLA_Class1 * self)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
    // splicer begin class.Class1.method.returnThis
    SH_this->returnThis();
    // splicer end class.Class1.method.returnThis
}
```

### 8.2.2 Fortran

Fortran does not permit his behavior. The function is treated as a subroutine.

```
subroutine class1_return_this(obj)
    class(class1) :: obj
    ! splicer begin class.Class1.method.return_this
    call c_class1_return_this(obj%cxxmem)
    ! splicer end class.Class1.method.return_this
end subroutine class1_return_this
```

The chaining must be done as a sequence of calls.

```
use classes_mod
type(class1) var

var = class1()
call var%return_this()
```

## 8.3 Class static methods

To wrap the method:

```
class Singleton {
    static Singleton& getReference();
};
```

Use the YAML input:

```
- decl: class Singleton
  declarations:
    - decl: static Singleton& getReference()
```

### 8.3.1 Fortran

Class static methods are supported using the NOPASS keyword in Fortran.

```
type singleton
  type(CLA_SHROUD_capsule_data) :: cxxmem
  ! splicer begin class.Singleton.component_part
  ! splicer end class.Singleton.component_part
contains
  procedure, nopass :: get_reference => singleton_get_reference
  ! splicer begin class.Singleton.type_bound_procedure_part
  ! splicer end class.Singleton.type_bound_procedure_part
end type singleton
```

Called from Fortran as:

```
type(singleton) obj0
obj0 = obj0%get_reference()
```

Note that *obj0* is not assigned a value before the function *get\_reference* is called.

## 8.4 Class Inheritance

Class inheritance is supported. Note that the subclass declaration uses a colon and must be quoted. Otherwise YAML will treat it as another mapping entry.

```
- decl: class Shape
  declarations:
    - decl: Shape()
    - decl: int get_ivar() const

- decl: "class Circle : public Shape"
  declarations:
    - decl: Circle()
```

### 8.4.1 Fortran

Inheritance is implemented using the EXTENDS Fortran keyword. Only single inheritance is supported.

```

type shape
  type (CLA_SHROUD_capsule_data) :: cxxmem
contains
  procedure :: get_ivar => shape_get_ivar
end type shape

type, extends(shape) :: circle
end type circle

```

## 8.4.2 Python

Python uses the `PyTypeObject.tp_base` field.

## 8.5 Forward Declaration

A class may be forward declared by omitting declarations. All other fields, such as `format` and `options` must be provided on the initial `decl` of a Class. This will define the type and allow it to be used in following declarations. The class's declarations can be added later:

```

declarations:
- decl: class Class1
  options:
    foo: True

- decl: class Class2
  declarations:
  - decl: void accept1(Class1 & arg1)

- decl: class Class1
  declarations:
  - decl: void accept2(Class2 & arg2)

```

## 8.6 Member Variables

For each member variable of a C++ class a C and Fortran wrapper function will be created to get or set the value. The Python wrapper will create a descriptor. It is not necessary to list all members of the class, only the one which are to be exposed in the wrapper. `private` members cannot be wrapped.

```

class Class1
{
public:
  int m_flag;
  int m_test;
};

```

It is added to the YAML file as:

```

- decl: class Class1
  declarations:
  - decl: int m_flag +readonly;
  - decl: int m_test +name(test);

```

The *readonly* attribute will not create the setter function or descriptor. Python will report:

```
>>> obj = tutorial.Class1()
>>> obj.m_flag = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: attribute 'm_flag' of 'tutorial.Class1' objects is not writable
```

The *name* attribute will change the name of generated functions and descriptors. This is helpful when using a naming convention like *m\_test* and you do not want *m\_* to be used in the wrappers.

The names of these functions are controlled by the options *SH\_class\_getter\_template* and *SH\_class\_setter\_template*. They are added as additional methods on the class.

For wrapping details see [Getter and Setter](#).

The getter and setter for a member which is a pointer to a native type can use a Fortran pointer if the member is given the *dimension* attribute.

```
- decl: class PointerData
  declarations:
- decl: int nitems;
- decl: int *items +dimension(nitems);
```

Notice that the *dimension* uses another field in the class. This will create a getter which can be called from Fortran. Likewise, the setter will require an argument of the same rank as the *dimension* attribute.

```
type(PointerData) var
integer(C_INT) :: nitems
integer(C_INT), pointer :: items(:)
integer(C_INT) :: updated(10)

var = PointerData()
nitems = var%get_nitems()
items => var%get_items()

call var%set_items(updated)
var%nitems = size(updated)      ! keep nitems and items consistent
```

The user must be consistent in the use of the getter and setter. For example, *item* is *nitems* long, but if the setter assigns an array which is shorter, the next call to the getter will still create a Fortran pointer which is *nitems* long.

Another point to note is that the variable *updated* should have the *TARGET* attribute since we're saving its address in *var*.

## 8.7 Struct

Shroud supports both structs and classes. But it treats them much differently. Whereas in C++ a struct and class are essentially the same thing, Shroud treats structs as a C style struct. They do not have associated methods. This allows them to be mapped to a Fortran derived type with the *bind(C)* attribute and a Python NumPy array.

For wrapping purposes, a struct is a C++ class without a *vtable*. It will contain POD types. Unlike classes where all member variables do not need to be wrapped, a struct should be fully defined. This is necessary to allow an array of structs to be created in the wrapper language then passed to C++.

A struct is defined in the yaml file as:

```
- decl: struct Cstruct1
  declarations:
  - decl: int ifield;
  - decl: double dfield;
```

It can also be defined as one decl entry:

```
- decl: struct Cstruct1 {
      int ifield;
      double dfield;
    };
```

The `struct` statement can also be used to declare a variable of a previously defined structure. This is required for C but is optional for C++ where a `struct` statement defines a type. To distinguish a variable declaration from a struct declaration, the trailing semicolon is required.

```
- decl: struct Cstruct1 var;
```

## 8.7.1 Fortran

This is translated directly into a Fortran derived type with the `bind(C)` attribute.

```
type, bind(C) :: cstruct1
  integer(C_INT) :: ifield
  real(C_DOUBLE) :: dfield
end type cstruct1
```

All creation and access of members can be done using Fortran.

```
type(cstruct1) st(2)

st(1)%ifield = 1_C_INT
st(1)%dfield = 1.5_C_DOUBLE
st(2)%ifield = 2_C_INT
st(2)%dfield = 2.6_C_DOUBLE
```

## 8.7.2 Python

Python can treat a struct in several different ways by setting option `PY_struct_arg`. First, treat it the same as a class. An extension type is created with descriptors for the field methods. Second, as a numpy descriptor. This allows an array of structs to be used easily. Finally, as a tuple of Python objects.

When treated as a class, a constructor is created which will create an instance of the class. This is similar to the default constructor for structs in C++ but will also work with a C struct.

```
import cstruct
a = cstruct.Cstruct1(1, 2.5)
a = cstruct.Cstruct1()
```

When treated as a NumPy array no memory will be copied since the NumPy array contains a pointer to the C++ memory.

```
import cstruct
dt = cstruct.Cstruct1_dtype
a = np.array([(1, 1.5), (2, 2.6)], dtype=dt)
```

The descriptor is created in the wrapper *NumPy Struct Descriptor*.

### 8.7.3 Member Variables

Generally, getter and setter functions are not required since Fortran can directly access the member fields. But when the member is a pointer it is more convient to have a getter and setter which works with Fortran pointers.

```
struct PointerData
{
    int nitems;
    int *items;
};
```

The generated getter and setter are not type-bound functions and must be passed the struct variable:

```
type(PointerData) var
integer(C_INT) :: nitems
integer(C_INT) :: items(10)
integer(C_INT), pointer :: out(:)

var%nitems = 10
call pointerdata_set_items(var, items)

out => pointerdata_get_items(var)
```

The names of these functions are controlled by the options *SH\_struct\_getter\_template* and *SH\_struct\_setter\_template*. They are added to the same scope as the struct.

Option *F\_struct\_getter\_setter* can be set to *false* to avoid creating the getter and setter functions.

## 8.8 Object-oriented C

Object oriented programing is a model and not a language feature. This model has been used for years in C by creating a struct for the object, then functions for the methods. C++ will implicitly pass the `this` argument. C methods explicitly pass the struct as an argument. Fortran and Python both pass an explicit object then wrap it in syntacatic sugar to allow a `self.method()` syntax to be used. Shroud allows a struct and collection of functions to be treated as a class.

First, define a struct and set the *wrap\_struct\_as* options to *class*.

```
- decl: struct Cstruct_as_class {
    int x1;
    int y1;
};
options:
    wrap_struct_as: class
```

Create a constructor function. The *class\_ctor* options associates this with the struct.



```
- decl: Cstruct_as_class *Create_Cstruct_as_class(void)
options:
  class_ctor: Cstruct_as_class
```

Then add methods. The *class\_method* option associates this with the struct. The format field *F\_name\_function* is used to name the method. The default method name is the same as the function name. But since this name will be used in the context of the object, it can be much shorter. The *pass* attribute marks this as the ‘object’.

```
- decl: int Cstruct_as_class_sum(const Cstruct_as_class *point +pass)
options:
  wrap_python: False
  class_method: Cstruct_as_class
format:
  F_name_function: sum
```

Additional options are *wrap\_class\_as* and *class\_baseclass*.

## 8.8.1 Fortran

A *shadow* class is created for the struct. This is the same as wrapping a C++ class. Getters and setters are created for the member variable. And the *sum* method is added.

```
type cstruct_as_class
  type(STR_SHROUD_capsule_data) :: cxxmem
  ! splicer begin class.Cstruct_as_class.component_part
  ! splicer end class.Cstruct_as_class.component_part
contains
  procedure :: get_x1 => cstruct_as_class_get_x1
  procedure :: set_x1 => cstruct_as_class_set_x1
  procedure :: get_y1 => cstruct_as_class_get_y1
  procedure :: set_y1 => cstruct_as_class_set_y1
  procedure :: sum => cstruct_as_class_sum
  ! splicer begin class.Cstruct_as_class.type_bound_procedure_part
  ! splicer end class.Cstruct_as_class.type_bound_procedure_part
end type cstruct_as_class
```

Now the struct is treated as a class in the Fortran wrapper.

```
use struct_mod
type(cstruct_as_class) var      ! Create Fortran variable.
integer(C_INT) i
var = cstruct_as_class()      ! Create struct in C++.
i = var%sum()
```

Similar to Python, Fortran passes the object as an explicit argument. Unlike C++ which uses an implicit *this* variable. By default, the first argument of the function is assumed to be the object. However, this can be changed using the *pass* attribute. This will add the Fortran keyword *PASS* to the corresponding argument.

A full example is at *Struct as a Class*.



---

## Default Arguments

---

Default arguments allows a C++ function to be called without providing one or more trailing arguments. Shroud can handle default args in several different ways based on the value of option *F\_default\_args*.

### 9.1 Generic Default Arguments

Since a default argument can have any C++ value, the C++ compiler must be used to provide the values. Shroud does this by creating a function for each possible way to call the function. These functions are then combined into a generic interface with the C++ function name. This is the default behavior of Shroud but can be made explicit by setting option *F\_default\_args* to *generic*.

For example, the function

```
void apply(IndexType num_elems, IndexType offset = 0, IndexType stride = 1);
```

can be called with 1, 2 or 3 arguments. C wrapper functions are created with the prototypes:

```
void apply(IndexType num_elems);
void apply(IndexType num_elems, IndexType offset);
void apply(IndexType num_elems, IndexType offset, IndexType stride);
```

The C++ compiler will provided the missing arguments using the default values.

The generated functions will have the same name as the C++ function with a suffix added to create unique names. By default this is a integer sequence number. The suffix can be controlled by adding a **default\_arg\_suffix** entry to the YAML file. One suffix is provided for each generated overloaded function.

```
- decl: void apply(IndexType num_elems, IndexType offset = 0, IndexType stride = 1);
  default_arg_suffix:
  - _nelems
  - _nelems_offset
  - _nelems_offset_stride
```

## 9.2 Require Default Arguments

Shroud provides the option to require all arguments by setting *F\_default\_args* to *require*. This is intended to help when there are overloaded functions with default arguments. The Fortran type system is not as rich as C++ and some Fortran generic function may be ambiguous. This can happen since C++ `enum` is converted to an `integer`.

## 9.3 Optional Default Arguments

When the default values can be represented in Fortran the `OPTIONAL` attribute can be used with default arguments to allow the Fortran wrapper to supply the value for arguments which are not present in the call to the function. This is generated when the option *F\_default\_args* is set to *optional*. No overloaded functions are generated. The C wrapper will require all arguments to be provided.

This provides the ability to call the function from Fortran in a way not supported by C++. Each argument can be provided individually using keyword arguments. The function can be called as

```
call apply(100, stride=2)
```

and `offset` will be provided the default value of 0.

Since the value is provided by Fortran, this only works with integer and real values.

# CHAPTER 10

---

## Templates

---

Shroud will wrap templated classes and functions for explicit instantiations. The template is given as part of the `decl` and the instantiations are listed in the `cxx_template` section:

```
- decl: |
    template<typename ArgType>
    void TemplateArgument(ArgType arg)
  cxx_template:
  - instantiation: <int>
  - instantiation: <double>
```

options and format may be provide to control the generated code:

```
- decl: template<typename T> class vector
  cxx_header: <vector>
  cxx_template:
  - instantiation: <int>
    format:
      C_impl_filename: wrapvectorforint.cpp
    options:
      optblah: two
  - instantiation: <double>
```

For a class template, the `class_name` is modified to included the instantiation type. If only a single template parameter is provided, then the template argument is used. For the above example, `C_impl_filename` will default to `wrapvector_int.cpp` but has been explicitly changed to `wrapvectorforint.cpp`.

Functions can be created which return a templated class:

```
- decl: vector<int> getVector()
```

The result type must be instantiated via the `cxx_template` block before it can be used.



In order for Shroud to create an idiomatic wrapper, it needs to know how arguments are intended to be used. This information is supplied via attributes. This section describes how to describe the arguments to Shroud in order to implement the desired semantic.

## 11.1 No Arguments

A C function with no arguments and which does not return a value, can be “wrapped” by creating a Fortran interface which allows the function to be called directly. A C++ function will require an `extern C` function to create an C wrapper to deal with the C++ name mangling.

An example is detailed at [NoReturnNoArguments](#).

## 11.2 Numeric Arguments

Integer and floating point numbers are supported by the *interoperability with C* feature of Fortran 2003. This includes the integer types `short`, `int`, `long` and `long long`. Size specific types `int8_t`, `int16_t`, `int32_t`, and `int64_t` are also supported. Floating point types are `float` and `double`.

---

**Note:** Fortran has no support for unsigned types. `size_t` will be the correct number of bytes, but will be signed.

---

In the following examples, `int` can be replaced by any numeric type.

**int arg** Pass a numeric value to C. The attribute `intent(in)` is defaulted. The Fortran 2003 attribute `VALUE` is used to change from Fortran’s default call-by-reference to C’s call-by-value. This argument can be called directly by Fortran and no C wrapper is necessary. See example [PassByValue](#).

**const int \*arg** Scalar call-by-reference. `const` pointers are defaulted to `+intent(in)`.

**int \*arg +intent(out)** If the intent is to return a scalar value from a function, add the `intent(out)` attribute. See example [PassByReference](#).

**const int \*arg +rank(1)** The `rank(1)` attribute will create an assumed-shape Fortran dimension for the argument as `arg(:)`. The C library function needs to have some way to determine the length of the array. The length could be assumed by the library function. A better option is to add another argument which will explicitly pass the length of the array from Fortran - `int larg+implied(size(arg))`. An *implied* argument will not be part of the wrapped API but will still be passed to the C++ function. See example [Sum](#).

**int \*arg +intent(out)+deref(allocatable)+dimension(n)** Adds the Fortran attribute `ALLOCATABLE` to the argument, then use the `ALLOCATE` statment to allocate memory using *dimension* attribute as the shape. See example [truncate\\_to\\_int](#).

**intent \*\*arg +intent(out)** Return a pointer in an argument. This is converted into a Fortran `POINTER` to a scalar. See example [getPtrToScalar](#).

**intent \*\*arg +intent(out)+dimension(ncount)** Return a pointer in an argument. This is converted into a Fortran `POINTER` to an array by the *dimension* attribute. See example [getPtrToDynamicArray](#).

**intent \*\*arg +intent(out)+deref(raw)** Return a pointer in an argument. The Fortran argument will be a `type(C_PTR)`. This gives the caller the flexibility to dereference the pointer themselves using `c_f_pointer`. This is useful when the shape is not know when the function is called. See example [getRawPtrToFixedArray](#).

**int \*\*\*arg +intent(out)** Pointers nested to a deeper level are treated as a Fortran `type(C_PTR)` argument. This gives the user the most flexibility. The `type(C_PTR)` can be passed back to to library which should know how to cast it. There is no checks on the pointer before passing it to the library so it's very easy to pass bad values. The user can also explicitly dereferences the pointers using `c_f_pointer`. See example [getRawPtrToInt2d](#).

**int \*\*arg +intent(in)** Multiple levels of indirection are converted into a `type(C_PTR)` argument. See below for an exception for `char **`. See example [checkInt2d](#).

**int &min +intent(out)** A declaration to a scalar gets converted into pointers in the C wrapper. See example [getMinMax](#).

**int \*&arg** Return a pointer in an argument. From Fortran, this is the same as `int **arg`. See above examples.

## 11.3 Numeric Functions

**int \*func()** Return a Fortran `POINTER` to a scalar. See example [returnIntPtrToScalar](#).

**int \*func() +dimension(10)** Return a Fortran `POINTER` to an array with a fixed length. See example [returnIntPtrToFixedArray](#).

**int \*func() +deref(scalar)** Return a scalar. See example [returnIntScalar](#).

**int \*ReturnIntPtrDimPointer(int \*len+intent(out)+hidden) +dimension(len) +deref(pointer)** Return a Fortran `POINTER` to an array with a variable length. The length is returned in the argument *len*. It is marked *hidden* since it is not required for the Fortran or Python API. The returned array will know its length. See example [ReturnIntPtrDimPointer](#) The *deref* attribute can be changed to return a `type(C_PTR)`, `ALLOCATABLE` or a scalar. See example [ReturnIntPtrDimAlloc](#)

## 11.4 Bool

C and C++ functions with a `bool` argument generate a Fortran wrapper with a `logical` argument. One of the goals of Shroud is to produce an idiomatic interface. Converting the types in the wrapper avoids the awkwardness of requiring the Fortran user to passing in `.true._c_bool` instead of just `.true..` Using an integer for a `bool` argument is not portable since some compilers use 1 for `.true.` and others use -1.

**bool arg** Non-pointer arguments default to `intent(IN)`. See example [checkBool](#).



## 11.5 Char

Fortran, C, and C++ each have their own semantics for character variables.

- Fortran `character` variables know their length and are blank filled
- C `char *` variables are assumed to be NULL terminated.
- C++ `std::string` know their own length and can provide a NULL terminated pointer.

It is not sufficient to pass an address between Fortran and C++ like it is with other native types. In order to get idiomatic behavior in the Fortran wrappers it is often necessary to copy the values. This is to account for blank filled vs NULL terminated.

**const char \*arg** Create a NULL terminated string in Fortran using `trim(arg) // C_NULL_CHAR` and pass to C. Since the argument is `const`, it is treated as `intent(in)`. A `bufferify` function is not required to convert the argument. This is the same as `char *arg+intent(in)`. See example [acceptName](#).

**char \*arg** Pass a `char` pointer to a function which assign to the memory. `arg` must be NULL terminated by the function. Add the `intent(out)` attribute. The `bufferify` function will then blank-fill the string to the length of the Fortran `CHARACTER(*)` argument. It is the users responsibility to avoid overwriting the argument. See example [returnOneName](#).

Fortran must provide a `CHARACTER` argument which is at least as long as the amount that the C function will write into. This includes space for the terminating NULL which will be converted into a blank for Fortran.

**char \*arg, int larg** Similar to above, but pass in the length of `arg`. The argument `larg` does not need to be passed to Fortran explicitly since its value is implied. The `implied` attribute is defined to use the `len` Fortran intrinsic to pass the length of `arg` as the value of `larg`: `char *arg+intent(out), int larg+implied(len(arg))`. See example [ImpliedTextLen](#).

**char \*\*names +intent(in)** This is a standard C idiom for an array of NULL terminated strings. Shroud takes an array of `CHARACTER(len=*) arg(:)` and creates the C data structure by copying the data and adding the terminating NULL. See example [acceptCharArrayIn](#).

## 11.6 std::string

**std::string & arg** `arg` will default to `intent(inout)`. See example [acceptStringReference](#).

## 11.7 char functions

Functions which return a `char *` provide an additional challenge. Taken literally they should return a `type(C_PTR)`. And if you call the C library function via the interface, that's what you get. However, Shroud provides several options to provide a more idiomatic usage.

Each of these declaration call identical C++ functions but they are wrapped differently.

**char \*getCharPtr1** Return a pointer and convert into an `ALLOCATABLE CHARACTER` variable. Fortran 2003 is required. The Fortran application is responsible to release the memory. However, this may be done automatically by the Fortran runtime. See example [getCharPtr1](#).

**char \*getCharPtr2** Create a Fortran function which returns a predefined `CHARACTER` value. The size is determined by the `len` argument on the function. This is useful when the maximum size is already known. Works with Fortran 90. See example [getCharPtr2](#).

**char \*getCharPtr3** Create a Fortran subroutine with an additional `CHARACTER` argument for the C function result. Any size character string can be returned limited by the size of the Fortran argument. The argument is defined by the `F_string_result_as_arg` format string. Works with Fortran 90. See example [getCharPtr3](#).

## 11.8 string functions

Functions which return `std::string` values are similar but must provide the extra step of converting the result into a `char *`.

**const string &** See example [getConstStringRefPure](#).

## 11.9 std::vector

A `std::vector` argument for a C++ function can be created from a Fortran array. The address and size of the array is extracted and passed to the C wrapper to create the `std::vector`

**const std::vector<int> &arg** `arg` defaults to `intent(in)` since it is `const`. See example [vector\\_sum](#).

**std::vector<int> &arg** See example [vector\\_iota\\_out](#).

See example [vector\\_iota\\_out\\_alloc](#).

See example [vector\\_iota\\_inout\\_alloc](#).

On `intent(in)`, the `std::vector` constructor copies the values from the input pointer. With `intent(out)`, the values are copied after calling the function.

---

**Note:** With `intent(out)`, if `vector_iota` changes the size of `arg` to be longer than the original size of the Fortran argument, the additional values will not be copied.

---

## 11.10 Void Pointers

The Fortran 2003 standard added the `type(C_PTR)` derived type which is used to hold a C `void *`. Fortran is not able to directly dereference `type(C_PTR)` variables. The function `c_f_pointer` must be used.

**void \*arg** If the intent is to be able to pass any variable to the function, add the `+assumedtype` attribute. `type(*)` is only available with TS 29113. The Fortran wrapper will only accept scalar arguments. To pass an array, add the `dimension` attribute See examples [passAssumedType](#) and [passAssumedTypeDim](#).

**void \*arg** Passes the value of a `type(C_PTR)` argument. See example [passVoidStarStar](#).

**void \*\*arg** Used to return a `void *` from a function in an argument. Passes the address of a `type(C_PTR)` argument. See example [passVoidStarStar](#).

## 11.11 Function Pointers

C or C++ arguments which are pointers to functions are supported. The function pointer type is wrapped using a Fortran abstract interface. Only C compatible arguments in the function pointer are supported since no wrapper for the function pointer is created. It must be callable directly from Fortran.

**int (\*incr)(int)** Create a Fortran abstract interface for the function pointer. Only functions which match the interface can be used as a dummy argument. See example [callback1](#).

**void (\*incr)()** Adding the `external` attribute will allow any function to be passed. In C this is accomplished by using a cast. See example [callback1c](#).

The `abstract interface` is named from option **F\_abstract\_interface\_subprogram\_template** which defaults to `{underscore_name}_{argname}` where *argname* is the name of the function argument.

If the function pointer uses an abstract declarator (no argument name), the argument name is created from option **F\_abstract\_interface\_argument\_template** which defaults to `arg{index}` where *index* is the 0-based argument index. When a name is given to a function pointer argument, it is always used in the `abstract interface`.

To change the name of the subprogram or argument, change the option. There are no format fields **F\_abstract\_interface\_subprogram** or **F\_abstract\_interface\_argument** since they vary by argument (or argument to an argument):

```
options:
  F_abstract_interface_subprogram_template: custom_funptr
  F_abstract_interface_argument_template: XX{index}arg
```

It is also possible to pass a function which will accept any function interface as the dummy argument. This is done by adding the `external` attribute. A Fortran wrapper function is created with an `external` declaration for the argument. The C function is called via an interface with the `bind(C)` attribute. In the interface, an `abstract interface` for the function pointer argument is used. The user's library is responsible for calling the argument correctly since the interface is not preserved by the `external` declaration.

## 11.12 Struct

See example [passStruct1](#).

See example [passStructByValue](#).



## 12.1 What files are created

Shroud will create multiple output file which must be compiled with C++ or Fortran compilers.

One C++ file will be created for the library and one file for each C++ class. In addition a utility file will be created with routines which are implemented in C but called from Fortran. This includes some memory management routines.

Fortran creates a file for the library and one per additional namespace. Since Fortran does not support forward referencing of derived types, it is necessary to add all classes from a namespace into a single module.

Each Fortran file will only contain one module to make it easier to create makefile dependencies using pattern rules:

```
%o %.mod : %.f
```

File names for the header and implementation files can be set explicitly by setting variables in the format of the global or class scope:

```
format:
  C_header_filename: top.h
  C_impl_filename: top.cpp
  F_impl_filename: top.f

declarations:
- decl: class Names
  format:
    C_header_filename: foo.h
    C_impl_filename: foo.cpp
    F_impl_filename: foo.f
```

The default file names are controlled by global options. The option values can be changed to avoid setting the name for each class file explicitly. It's also possible to change just the suffix of files:

```
options:
  YAML_type_filename_template: {library_lower}_types.yaml
```

(continues on next page)

(continued from previous page)

```

C_header_filename_suffix: h
C_impl_filename_suffix: cpp
C_header_filename_library_template: wrap{library}.{C_header_filename_suffix}
C_impl_filename_library_template: wrap{library}.{C_impl_filename_suffix}

C_header_filename_namespace_template: wrap{file_scope}.{C_header_filename_suffix}
C_impl_filename_namespace_template: wrap{file_scope}.{C_impl_filename_suffix}

C_header_filename_class_template: wrap{cxx_class}.{C_header_filename_suffix}
C_impl_filename_class_template: wrap{cxx_class}.{C_impl_filename_suffix}

F_filename_suffix: f
F_impl_filename_library_template: wrapf{library_lower}.{F_filename_suffix}
F_impl_filename_namespace_template: wrapf{file_scope}.{F_filename_suffix}

```

A file with helper functions may also be created. For C the file is named by the format field *C\_impl\_utility*. It contains files which are implemented in C but are called from Fortran via BIND (C).

## 12.2 How names are created

Shroud attempts to provide user control of names while providing reasonable defaults. Each name is based on the library, class, function or argument name in the current scope. Most names have a template which may be used to control how the names are generated on a global scale. Many names may also be explicitly specified by a field.

For example, a library has an `initialize` function which is in a namespace. In C++ it is called as:

```

#include "library.hpp"

library::initialize()

```

By default this will be a function in a Fortran module and can be called as:

```

use library

call initialize

```

Since `initialize` is a rather common name for a function, it may be desirable to rename the Fortran wrapper to something more specific. The name of the Fortran implementation wrapper can be changed by setting *F\_name\_impl*:

```

library: library

declarations:
- decl: namespace library
  declarations:
  - decl: void initialize
    format:
      F_name_impl: library_initialize

```

To rename all functions, set the template in the toplevel *options*:

```

library: library

options:

```

(continues on next page)

(continued from previous page)

```

F_name_impl_template: "{library}_{underscore_name}{function_suffix}"

declarations:
- decl: namespace library
  declarations:
  - decl: void initialize

```

C++ allows overloaded functions and will mangle the names behind the scenes. With Fortran, the mangling must be explicit. To accomplish this Shroud uses the *function\_suffix* format string. By default, Shroud will use a sequence number. By explicitly setting *function\_suffix*, a more meaningful name can be provided:

```

- decl: void Function6(const std::string& name)
  format:
    function_suffix: _from_name
- decl: void Function6(int indx)
  format:
    function_suffix: _from_index

```

This will create the Fortran functions `function6_from_name` and `function6_from_index`. A generic interface named `function6` will also be created which will include the two generated functions.

Likewise, default arguments will produce several Fortran wrappers and a generic interface for a single C++ function. The format dictionary only allows for a single *function\_default* per function. Instead the field *default\_arg\_suffix* can be set. It contains a list of *function\_suffix* values which will be applied from the minimum to the maximum number of arguments:

```

- decl: int overload1(int num,
  int offset = 0, int stride = 1)
  default_arg_suffix:
  - _num
  - _num_offset
  - _num_offset_stride

```

Finally, multiple Fortran wrappers can be generated from a single templated function. Each instantiation will generate an additional Fortran Wrapper and can be distinguished by the *template\_suffix* entry of the format dictionary.

If there is a single template argument, then *template\_suffix* will be set to the *flat\_name* field of the instantiated argument. For example, `<int>` defaults to `_int`. This works well for POD types. The entire qualified name is used. For `<std::string>` this would be `std_string`. Classes which are deeply nested can produce very long values for *template\_suffix*. To deal with this, the *function\_template* field can be set on Class declarations:

```

- decl: namespace internal
  declarations:
  - decl: class ImplWorker1
    format:
      template_suffix: instantiation3

```

By default `internal_implworker1` would be used for the *template\_suffix*. But in this case `instantiation3` will be used.

For multiple template arguments, *template\_suffix* defaults to a sequence number to avoid long function names. In this case, specifying an explicit *template\_suffix* can produce a more user friendly name:

```

- decl: template<T,U> void FunctionTU(T arg1, U arg2)
  cxx_template:
  - instantiation: <int, long>

```

(continues on next page)

(continued from previous page)

```

format:
  template_suffix: instantiation1
- instantiation: <float, double>
format:
  template_suffix: instantiation2

```

The Fortran functions will be named `function_tu_instantiation1` and `function_tu_instantiation2`.

## 12.3 Additional Wrapper Functions

Functions can be created in the Fortran wrapper which have no corresponding function in the C++ library. This may be necessary to add functionality which may unnecessary in C++. For example, a library provides a function which returns a string reference to a name. If only the length is desired no extra function is required in C++ since the length is extracted used a `std::string` method:

```

ExClass1 obj("name")
int len = obj.getName().length();

```

Calling the Fortran `getName` wrapper will copy the string into a Fortran array but you need the length first to make sure there is enough room. You can create a Fortran wrapper to get the length without adding to the C++ library:

```

declarations:
- decl: class ExClass1
  declarations:
  - decl: int GetNameLength() const
    format:
      C_code: |
        {C_pre_call}
        return {CXX_this}->getName().length();

```

The generated C wrapper will use the `C_code` provided for the body:

```

int AA_exclass1_get_name_length(const AA_exclass1 * self)
{
    const ExClass1 *SH_this = static_cast<const ExClass1 *>(
        static_cast<const void *>(self));
    return SH_this->getName().length();
}

```

The `C_pre_call` format string is generated by Shroud to convert the `self` argument into `CXX_this` and must be included in `C_code` to get the definition.

## 12.4 Helper functions

Shroud provides some additional file static function which are inserted at the beginning of the wrapped code. Some helper functions are used to communicate between C and Fortran. They are global and written into the `fmt.C_impl_utility` file. The names of these files will have `C_prefix` prefixed to create unique names.

C helper functions

**ShroudStrCopy**(char \*dest, int ndest, const char \*src, int nsrc) Copy *src* into *dest*, blank fill to *ndest* characters Truncate if *dest* is too short to hold all of *src*. *dest* will not be NULL terminated.



**int ShroudLenTrim(const char \*src, int nsrc)** Returns the length of character string *src* with length *nsrc*, ignoring any trailing blanks.

Each Python helper is prefixed by format variable *PY\_helper\_prefix* which defaults to *SHROUD\_*. This is used to avoid conflict with other wrapped functions.

The option *PY\_write\_helper\_in\_util* will write all of the helper functions into the file defined by *PY\_utility\_filename*. This can be useful to avoid clutter when there are a lot of classes which may create lots of duplicate helpers. The helpers will no longer be file static and instead will also be prefixed with *C\_prefix* to avoid conflicting with helpers created by another Shroud wrapped library.

### 12.4.1 Header Files

The header files for the library are included by the generated C++ source files.

The library source file will include the global *cxx\_header* field. Each class source file will include the class *cxx\_header* field unless it is blank. In that case the global *cxx\_header* field will be used.

To include a file in the implementation list it in the global or class options:

```
cxx_header: global_header.hpp

declarations:
- decl: class Class1
  cxx_header: class_header.hpp
- decl: typedef int CustomType
  c_header: type_header.h
  cxx_header : type_header.hpp
```

The *c\_header* field will be added to the header file of contains functions which reference the type. This is used for files which are not part of the library but which contain code which helps map C++ constants to C constants

A global *fortran\_header* field will insert *#include* lines to be used with the Fortran preprocessor (typically a variant of the C preprocessor). This will work with the *cpp\_if* lines in declarations which will conditionally compile a wrapper.

### 12.4.2 Local Variable

*SH\_* prefix on local variables which are created for a corresponding argument. For example the argument *char \*name*, may need to create a local variable named *std::string SH\_name*.

Shroud also generates some code which requires local variables such as loop indexes. These are prefixed with *SHT\_*. This name is controlled by the format variable *c\_temp*.

Results are named from *fmt.C\_result* or *fmt.F\_result*.

Format variable which control names are

- *c\_temp*
- *C\_local*
- *C\_this*
- *CXX\_local*
- *CXX\_this*
- *C\_result*

- F\_result - SHT\_rv (return value)
- F\_this - obj
- LUA\_result
- PY\_result

## 12.5 C Preprocessor

It is possible to add C preprocessor conditional compilation directives to the generated source. For example, if a function should only be wrapped if `USE_MPI` is defined the `cpp_if` field can be used:

```
- decl: void testmpi(MPI_Comm comm)
  format:
    function_suffix: _mpi
  cpp_if: ifdef HAVE_MPI
- decl: void testmpi()
  format:
    function_suffix: _serial
  cpp_if: ifndef HAVE_MPI
```

The function wrappers will be created within `#ifdef/#endif` directives. This includes the C wrapper, the Fortran interface and the Fortran wrapper. The generated Fortran interface will be:

```
interface testmpi
#ifdef HAVE_MPI
  module procedure testmpi_mpi
#endif
#ifdef HAVE_MPI
  module procedure testmpi_serial
#endif
end interface testmpi
```

Class generic type-bound function will also insert conditional compilation directives:

```
- decl: class ExClass3
  cpp_if: ifdef USE_CLASS3
  declarations:
- decl: void exfunc()
  cpp_if: ifdef USE_CLASS3_A
- decl: void exfunc(int flag)
  cpp_if: ifndef USE_CLASS3_A
```

The generated type will be:

```
type exclass3
  type(SHROUD_capsule_data), private :: cxxmem
  contains
    procedure :: exfunc_0 => exclass3_exfunc_0
    procedure :: exfunc_1 => exclass3_exfunc_1
#ifdef USE_CLASS3_A
    generic :: exfunc => exfunc_0
#endif
#ifdef USE_CLASS3_A
    generic :: exfunc => exfunc_1
```

(continues on next page)

(continued from previous page)

```
#endif  
end type exclass3
```

A `cpp_if` field in a class will add a conditional directive around the entire class.

Finally, `cpp_if` can be used with types. This would be required in the first example since `mpi.h` should only be included when `USE_MPI` is defined:

```
typemaps:  
- type: MPI_Comm  
  fields:  
    cpp_if: ifdef USE_MPI
```

When using `cpp_if`, it is useful to set the option `F_filename_suffix` to `F`. This will cause most compilers to process the Fortran source with `cpp` before compilation. The `fortran_header` field can be added to the YAML file to insert `#include` directives at the top of the Fortran source files.

The `typemaps` field can only appear at the outermost layer and is used to augment existing typemaps.

## 12.6 Debugging

Shroud generates a JSON file with all of the input from the YAML and all of the format dictionaries and type maps. This file can be useful to see which format keys are available and how code is generated.



# CHAPTER 13

---

## C and C++

---

A C API is created for a C++ library. Wrapper functions are within an `extern "C"` block so they may be called by C or Fortran. But the file must be compiled with the C++ compiler since it is wrapping a C++ library.

When wrapping a C library, additional functions may be created which pass meta-data arguments. When called from Fortran, its wrappers will provide the meta-data. When called directly by a C application, the meta-data must be provided by the user.

### 13.1 Names

Shroud will flatten scoped C++ library names to create the C API. Since C does not support scopes such as classes and namespaces, a name such as `ns1::function` must be flattened into `ns1_function` to avoid conflict with a similarly named function `ns2::function`.

Names are also controlled by the **C\_api\_case** option. It can be set to *lower*, *upper*, *underscore* or *preserve*. This option is used to set the format field **C\_name\_api** which in turn is used in the option **C\_name\_template**. The default is *preserve*. This creates a stronger correlation between the C API and the C++ API.

To further help control the scope of C names, all externals add a prefix. It defaults to the first three letters of the **library** but may be changed by setting the format **C\_prefix**:

```
format:  
  C_prefix: NEW_
```

### 13.2 Wrapper

As each function declaration is parsed a format dictionary is created with fields to describe the function and its arguments. The fields are then expanded into the function wrapper.

C wrapper:

```
extern "C" {

{C_return_type} {C_name}({C_prototype})
{
    {C_code}
}

}
```

The wrapper is within an `extern "C"` block so that **C\_name** will not be mangled by the C++ compiler.

**C\_return\_code** can be set from the YAML file to override the return value:

```
- decl: void vector_string_fill(std::vector< std::string > &arg+intent(out))
  format:
    C_return_type: int
    C_return_code: return SH_arg.size();
```

The C wrapper (and the Fortran wrapper) will return `int` instead of `void` using **C\_return\_code** to compute the value. In this case, the wrapper will return the size of the vector. This is useful since C and Fortran convert the vector into an array.

## 13.3 Struct Type

While C++ considers a struct and a class to be similar, Shroud assumes a struct is intended to be a C compatible data structure. It has no methods which will cause a v-table to be created. This will cause an array of structs to be identical in C and C++.

The main use of wrapping a struct for C is to provide access to the name. If the struct is defined within a `namespace`, then a C application will be unable to access the struct. Shroud creates an identical struct as the one defined in the YAML file but at the global level.

## 13.4 Class Types

A C++ class is represented by the *C\_capsule\_data\_type*. This struct contains a pointer to the C++ instance allocated and an index passed to generated *C\_memory\_dtor\_function* used to destroy the memory:

```
struct s_{C_capsule_data_type} {
    void *addr;      /* address of C++ memory */
    int idtor;       /* index of destructor */
};
typedef struct s_{C_capsule_data_type} {C_capsule_data_type};
```

In addition, an identical struct is created for each class. Having a unique struct and typedef for each class add a measure of type safety to the C wrapper:

```
struct s_{C_type_name} {
    void *addr;      /* address of C++ memory */
    int idtor;       /* index of destructor */
};
typedef struct s_{C_type_name} {C_type_name};
```

`idtor` is the index of the destructor code. It is used with memory management and discussed in *Memory Management*.

The C wrapper for a function which returns a class instance will return a *C\_capsule\_data\_type* by value. Functions which take a class instance will receive a pointer to a *C\_capsule\_data\_type*.





This section discusses Fortran specific wrapper details. This will also include some C wrapper details since some C wrappers are created specifically to be called by Fortran.

## 14.1 Names

There are several options to mangle the C++ library names into Fortran names. By default, names are mangled to convert camel case into snake case. For example, `StructAsClass` into `struct_as_class`. Since Fortran is case insensitive, `StructAsClass` and `structasclass` are equivalent. By using snake case, the identifier should be easier for a reader to parse regardless of the case.

The behavior is controlled by the option **F\_api\_case** which may have the values *lower*, *upper*, *underscore*, or *preserve*. This option is used to set the format field **F\_name\_api** which in turn is used in several options used to define names consistently: **F\_C\_name\_template**, **F\_name\_impl\_template**, **F\_name\_function\_template**, **F\_name\_generic\_template**, **F\_abstract\_interface\_subprogram\_template**, **F\_derived\_name\_template**, **F\_typedef\_name\_template**.

A Fortran module will be created for the library. This allows the compiler to do its own mangling so it is unnecessary to add an additional prefix to function names. In contrast, the C wrappers add a prefix to each wrapper since all names are global.

## 14.2 Wrapper

As each function declaration is parsed a format dictionary is created with fields to describe the function and its arguments. The fields are then expanded into the function wrapper.

The template for Fortran code showing names which may be controlled directly by the input YAML file:

```
module {F_module_name}

! use_stmts
```

(continues on next page)

(continued from previous page)

```

implicit none

abstract interface
  subprogram {F_abstract_interface_subprogram_template}
    type :: {F_abstract_interface_argument_template}
  end subprogram
end interface

interface
  {F_C_pure_clause} {F_C_subprogram} {F_C_name}
    {F_C_result_clause} bind(C, name="{C_name}")
    ! arg_f_use
    implicit none
    ! arg_c_decl
  end {F_C_subprogram} {F_C_name}
end interface

interface {F_name_generic}
  module procedure {F_name_impl}
end interface {F_name_generic}

contains

{F_subprogram} {F_name_impl}
  arg_f_use
  arg_f_decl
  ! splicer begin
  declare      ! local variables
  pre_call
  call {arg_c_call}
  post_call
  ! splicer end
end {F_subprogram} {F_name_impl}

end module {F_module_name}

```

## 14.3 Class

Use of format fields for creating class wrappers.

```

type, bind(C) :: {F_capsule_data_type}
  type(C_PTR) :: addr = C_NULL_PTR ! address of C++ memory
  integer(C_INT) :: idtor = 0      ! index of destructor
end type {F_capsule_data_type}

type {F_derived_name}
  type({F_capsule_data_type}) :: {F_derived_member}
contains
  procedure :: {F_name_function} => {F_name_impl}
  generic :: {F_name_generic} => {F_name_function}, ...

  ! F_name_getter, F_name_setter, F_name_instance_get as underscore_name
  procedure :: [F_name_function_template] => [F_name_impl_template]

```

(continues on next page)

(continued from previous page)

```
end type {F_derived_name}
```

### 14.3.1 Standard type-bound procedures

Several type bound procedures can be created to make it easier to use class from Fortran.

Usually the *F\_derived\_name* is constructed from wrapped C++ constructor. It may also be useful to take a pointer to a C++ struct and explicitly put it into a the derived type. The functions *F\_name\_instance\_get* and *F\_name\_instance\_set* can be used to access the pointer directly.

Two predicate function are generated to compare derived types:

```
interface operator (.eq.)
  module procedure class1_eq
  module procedure singleton_eq
end interface

interface operator (.ne.)
  module procedure class1_ne
  module procedure singleton_ne
end interface

contains

function {F_name_scope}eq(a,b) result (rv)
  use iso_c_binding, only: c_associated
  type({F_derived_name}), intent(IN) :: a,b
  logical :: rv
  if (c_associated(a%{F_derived_member}%addr, b%{F_derived_member}%addr)) then
    rv = .true.
  else
    rv = .false.
  endif
end function {F_name_scope}eq

function {F_name_scope}ne(a,b) result (rv)
  use iso_c_binding, only: c_associated
  type({F_derived_name}), intent(IN) :: a,b
  logical :: rv
  if (.not. c_associated(a%{F_derived_member}%addr, b%{F_derived_member}%addr)) ↪then
    rv = .true.
  else
    rv = .false.
  endif
end function {F_name_scope}ne
```

## 14.4 Generic Interfaces

Shroud has the ability to create generic interfaces for the routines that are being wrapped. The generic interfaces groups several functions under a common name. The compiler will then call the corresponding function based on the argument types used to call the generic function.

In several cases generic interfaces are automatically created. Function overloading and default arguments both create generic interfaces.

### 14.4.1 Assumed Rank

Assumed rank arguments allow a scalar or any rank array to be passed as an argument. This is added as the attribute *dimension(..)*. Think of the *..* as a *:*, used to separate lower and upper bounds, which fell over. This feature is part of Fortran's *Further interoperability with C*. First as TS 29113, approved in 2012, then as part of the Fortran 2018 standard.

---

**Note:** Shroud does not support *Further Interoperability with C* directly, yet.

---

Assumed-rank arguments are support by Shroud for older versions of Fortran by creating a generic interface. If there are multiple arguments with assumed-rank, Shroud will give each argument the same rank for each generic interface. This handles the common case and avoids the combinatoral explosion of mixing ranks in a single function interface.

The ranks used are controlled by the options *F\_assumed\_rank\_min* and *F\_assumed\_rank\_max* which default to 0, for scalar, and 7.

```
- decl: int SumValues(const int *values+dimension(..), int nvalues)
  options:
    F_assumed_rank_max: 2
```

The generated generic interface can be used to pass a scalar, 1d or 2d array to the C function. In each case `result` is 5.

```
result = sum_array(5, 1)
result = sum_array([1,1,1,1,1], 5)
```

### 14.4.2 Grouping Functions Together

The first case allows multiple C wrapper routines to be called by the same name. This is done by setting the *F\_name\_generic* format field.

```
- decl: void UpdateAsFloat(float arg)
  options:
    F_force_wrapper: True
  format:
    F_name_generic: update_real
- decl: void UpdateAsDouble(double arg)
  options:
    F_force_wrapper: True
  format:
    F_name_generic: update_real
```

This allows the correct functions to be called based on the argument type.

---

**Note:** In this example *F\_force\_wrapper* is set to *True* since by default Shroud will not create explicit wrappers for the functions since only native types are used as arguments. The generic interface is using `module procedure`` which requires the Fortran wrapper. This should be changed in a future version of Shroud.

---

```
call update_real(22.0_C_FLOAT)
call update_real(23.0_C_DOUBLE)
```

Or more typically as:

```
call update_real(22.0)
call update_real(23.0d0)
```

### 14.4.3 Argument Coercion

The C compiler will coerce arguments in a function call to the type of the argument in the prototype. This makes it very easy to pass an `float` to a function which is expecting a `double`. Fortran, which defaults to pass by reference, does not have this feature since it is passing the address of the argument. This corresponds to C's behavior since it cannot coerce a `float *` to a `double *`. When passing a literal `0.0` as a `float` argument it is necessary to use `0.0_C_DOUBLE`.

Shroud can create a generic interface for function which will coerce arguments similar to C's behavior. The *fortran\_generic* section variations of arguments which will be used to create a generic interface. For example, when wrapping a function which takes a `double`, the `float` variation can also be created.

```
- decl: void GenericReal(double arg)
  fortran_generic:
- decl: (float arg)
  function_suffix: _float
- decl: (double arg)
  function_suffix: _double
```

This will create a generic interface `generic_real` with two module procedures `generic_real_float` and `generic_real_double`.

```
interface generic_real
  module procedure generic_real_float
  module procedure generic_real_double
end interface generic_real
```

This can be used as

```
call generic_real(0.0)
call generic_real(0.0d0)

call generic_real_float(0.0)
call generic_real_double(0.0d0)
```

When adding *decl* entries to the *fortran\_generic* list the original declaration must also be included, `double arg` in this case. When there are multiple arguments only the arguments which vary need to be declared. The other arguments will be the same as the original *decl* line.

The *function\_suffix* line will be used to add a unique string to the generated Fortran wrappers. Without *function\_suffix* each function will have an integer suffix which is increment for each function.

### 14.4.4 Scalar and Array Arguments

Shroud can produce a generic interface which allows an argument to be passed as a scalar or an array. This can help generalize some function calls where a scalar can be used instead of an array of length one. This was often used in Fortran code before interfaces were introduced in Fortran 90. But now when using an interface the compiler will report

an error when passing a scalar where an array is expected. Likewise, a C function with a pointer argument such as `int *` has no way of knowing how long the array is without being told explicitly. Thus in C it is easy to pass a pointer to a scalar.

In the *fortran\_generic* section one of the declarations can be given the *rank* attribute which causes the interface to expect an array. Note that the declaration for the C function does not include the *rank* attribute.

```
- decl: int SumArray(int *values, int nvalues)
  fortran_generic:
- decl: (int *values)
  function_suffix: _scalar
- decl: (int *values+rank(1))
  function_suffix: _array
```

The generated generic interface can be used to pass a scalar or array to the C function.

```
integer scalar, result
integer array(5)

scalar = 5
result = sum_array(scalar, 1)

array = [1,2,3,4,5]
result = sum_array(array, 5)
```

---

**Note:** Work in progress

---

This section discusses Python specific wrapper details.

## 15.1 Wrapper

## 15.2 Types

## 15.3 type fields

### 15.3.1 PY\_build\_arg

Argument for Py\_BuildValue. Defaults to *{cxx\_var}*. This field can be used to turn the argument into an expression such as *(int) {cxx\_var}* or *{cxx\_var}{cxx\_member}c\_str()* *PY\_build\_format* is used as the format:

```
Py_BuildValue("{PY_build_format}", {PY_build_arg});
```

### 15.3.2 PY\_build\_format

‘format unit’ for Py\_BuildValue. If *None*, use value of *PY\_format*. Defaults to *None*

### 15.3.3 PY\_format

‘format unit’ for PyArg\_Parse and Py\_BuildValue. Defaults to *O*

### 15.3.4 PY\_PyTypeObject

Variable name of PyTypeObject instance. Defaults to *None*.

### 15.3.5 PY\_PyObject

Typedef name of PyObject instance. Defaults to *None*.

### 15.3.6 PY\_ctor

Expression to create object. ex. `PyInt_FromLong({rv})` Defaults to *None*.

### 15.3.7 PY\_get

Expression to get value from an object. ex. `PyInt_AsLong({py_var})` Defaults to *None*.

### 15.3.8 PY\_to\_object\_idtor

Create an Python object for the type. Includes the index of the destructor function. Used with structs/classes that are created by functions and must be wrapped. `object = converter(address, idtor)`. Defaults to *None*.

### 15.3.9 PY\_to\_object

`PyBuild - object = converter(address)`. Defaults to *None*.

### 15.3.10 PY\_from\_object

`PyArg_Parse - status = converter(object, address)`. Defaults to *None*.

### 15.3.11 py\_ctype

The type returned by `PY_get` function. Defaults to *None* which implies it is the same as the `typemap`. i.e. `PyInt_AsLong` returns a `long`.

Defined for complex types because `PyComplex_AsCComplex` returns type `Py_complex`. See also `py-type_to_pyctor` and `pytype_to_cxx`.

### 15.3.12 pytype\_to\_pyctor

Expression to use with `PY_ctor`. Defaults to *None* which indicates no additional processing of the argument is required. Only needs to be defined when `py_ctype` is defined.

With complex types, it is used to extract the real and imaginary parts from `Py_complex` (defined with `py_ctype`) with `creal({ctor_expr})`, `cimag({ctor_expr})`. `ctor_expr` is the expression used with `Py_ctor`.



### 15.3.13 pytype\_to\_cxx

Expression to convert *py\_ctype* into a C++ value. Only needs to be defined when *py\_ctype* is defined.

Used with `complex` to convert `Py_complex` (defined with *py\_ctype*) to C using `{work_var}.real + {work_var}.imag * I` or C++ with `std::complex(\tcvalue.real, cvalue.imag)`.

### 15.3.14 cxx\_to\_pytype

Statements to convert *cxx\_var* to *py\_ctype*/ Only needs to be defined when *py\_ctype* is defined.

```
cxx_to_pytype: |
    {ctype_var}.real = creal({cxx_var});
    {ctype_var}.imag = cimag({cxx_var});
```

### 15.3.15 PYN\_descr

Name of `PyArray_Descr` variable which describe type. Used with structs. Defaults to *None*.

### 15.3.16 PYN\_typenum

NumPy type number. ex. `NPY_INT` Defaults to *None*.

## 15.4 Statements

The template for a function is:

```
static char {PY_name_impl}__doc__[] = "{PY_doc_string}";

static PyObject *'
{PY_name_impl}(
    {PY_PyObject} *{PY_param_self},
    PyObject *{PY_param_args},
    PyObject *{PY_param_kwds})
{
    {declare}

    // {parse_format} {parse_args}
    if (!PyArg_ParseTupleAndKeywords(
        {PY_param_args}, {PY_param_kwds}, "{PyArg_format}",
        SH_kw_list, {PyArg_vargs})) {
        return NULL;
    }

    // result pre_call

    // Create C from Python objects
    // Create C++ from C
    {post_parse}
    {
        create scope before fail
        {pre_call}    pre_call declares variables for arguments
```

(continues on next page)

(continued from previous page)

```
    call {arg_call}
    {post_call}

    per argument
    // Create Python object from C++
    {ctor}      {post_call}

    {PyObject} * {py_var} Py_BuildValue("{Py_format}", {vargs});
    {cleanup}
}
return;

fail:
{fail}
Py_XDECREF(arr_x);
}
```

The template for a setter is:

```
static PyObject *{PY_getter}(
    {PY_PyObject} *{PY_param_self},
    void *SHROUD_UNUSED(closure)) {
    {setter}
}
```

The template for a getter is:

```
static int {PY_setter}("
    {PY_PyObject} *{PY_param_self},
    PyObject *{py_var},
    void *SHROUD_UNUSED(closure)) {
    {getter}
    return 0;
}
```

Fields listed in the order they generate code. C variables are created before the call to `Py_ParseArgs`. C++ variables are then created in *post\_parse* and *pre\_call*. For example, creating a `std::string` from a `char *`.

### 15.4.1 allocate\_local\_var

Functions which return a struct/class instance (such as `std::vector`) need to allocate a local variable which will be used to store the result. The Python object will maintain a pointer to the instance until it is deleted.

### 15.4.2 c\_header

### 15.4.3 cxx\_header

### 15.4.4 c\_helper

Blank delimited list of helper functions required for the wrapper. The name may contain format strings and will be expand before it is used. ex. `to_PyList_{cxx_type}`. The function associated with the helper will be named *hnamefunc0*, *hnamefunc1*, ... for each helper listed.

### 15.4.5 need\_numpy

If *True*, add NumPy headers and initialize in the module.

### 15.4.6 fmtdict

Update format dictionary to override generated values. Each field will be evaluated before assignment.

`ctor_expr` - Expression passed to `Typemap.PY_ctor PyInt_FromLong({ctor_expr})`. Useful to add dereferencing if necessary. `PyInt_FromLong` is from `typemap.PY_ctor`.

```
fmtdict=dict(
    ctor_expr="{c_var}",
),
```

### 15.4.7 arg\_declare

By default a local variable will be declared the same type as the argument to the function.

For some cases, this will not be correct. This field will be used to replace the default declaration.

references

In some cases the declaration is correct but need to be initialized. For example, setting a pointer.

Assign a blank list will not add any declarations. This is used when only an output `std::string` or `std::vector` is created after parsing arguments.

This variables is used with `PyArg_ParseTupleAndKeywords`.

The argument will be non-const to allow it to be assigned later.

```
name="py_char*_out_charlen",
arg_declare=[
    "{c_const}char {c_var}[{charlen}]; // intent(out)",
],
```

### 15.4.8 declare

Code needed to declare local variable. Often used to define variables of type `PyObject *`.

### 15.4.9 cxx\_local\_var

Set when a C++ variable is created by `post_parse`. *scalar*

Used to set format fields *cxx\_member*

### 15.4.10 parse\_format

Works together with *parse\_args* to describe how to parse `PyObject` in `PyArg_ParseTupleAndKeywords`. *parse\_format* is used in the *format* arguments and *parse\_args* is append to the call as a vararg.

```
int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kw,
    const char *format, char *keywords[], ...)
```

The simplest use is to pass the object directly through so that it can be operated on by *post\_parse* or *pre\_call* to convert the object into a C/C++ variable. For example, convert a `PyObject` into an `int *`.

```
parse_format="O",
parse_args=["&{pytmp_var}"],
declare=[
    "PyObject * {pytmp_var};",
],
```

The format field *pytmp\_var* is created by Shroud, but must be declared if it is used.

It can also be used to provide a *converter* function which converts the object:

```
parse_format="O&",
parse_args=["{hnamefunc0}", "&{py_var}"],
```

From the Python manual: Note that any Python object references which are provided to the caller (of *PyArg\_Parse*) are borrowed references; do not decrement their reference count!

### 15.4.11 parse\_args

A list of wrapper variables that are passed to *PyArg\_ParseTupleAndKeywords*. Used with *parse\_format*.

### 15.4.12 cxx\_local\_var

Set to *scalar* or *pointer* depending on the declaration in *post\_declare* *post\_parse* or *pre\_call*.

### 15.4.13 post\_declare

Declaration of C++ variables after calling *PyArg\_ParseTupleAndKeywords*. Usually involves object constructors such as `std::string` or `std::vector`. Or for extracting struct and class pointers out of a *PyObject*.

These declarations should not include `goto fail`. This allows them to be created without a “jump to label ‘fail’ crosses initialization of” error.

“It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an initializer.”

### 15.4.14 post\_parse

Statements to execute after the call to *PyArg\_ParseTupleAndKeywords*. Used to convert C values into C++ values:

```
{var} = PyObject_IsTrue({var_obj});
```

Will not be added for class constructor objects. since there is no need to build return values.

Allow *intent(in)* arguments to be processed. For example, process `PyObject` into `PyArrayObject`.

### 15.4.15 pre\_call

Location to allocate memory for output variables. All *intent(in)* variables have been processed by *post\_parse* so their lengths are known.

### 15.4.16 arg\_call

List of arguments to pass to function.

### 15.4.17 post\_call

Convert result and *intent(out)* into `PyObject`. Set *object\_created* to `True` if a `PyObject` is created.

### 15.4.18 cleanup

Code to remove any intermediate variables.

### 15.4.19 fail

Code to remove allocated memory and created objects.

### 15.4.20 goto\_fail

If `True`, one of the other blocks such as *post\_parse*, *pre\_call*, and *post\_call* contain a call to `fail`. If any statements block sets *goto\_fail*, then the *fail* block will be inserted into the code/

### 15.4.21 object\_created

Set to `True` when a `PyObject` is created by *post\_call*. This prevents `Py_BuildValue` from converting it into an `Object`. For example, when a pointer is converted into a `PyCapsule` or when `NumPy` is used to create an object.

## 15.5 Predefined Types

### 15.5.1 Int

An `int` argument is converted to Python with the typemap:

```
type: int
fields:
    PY_format: i
    PY_ctor: PyInt_FromLong({c_deref}{c_var})
    PY_get: PyInt_AsLong({py_var})
    PYN_tynenum: NPY_INT
```

## 15.6 Pointers

When a function returns a pointer to a POD type several Python interfaces are possible. When a function returns an `int *` the simplest result is to return a `PyCapsule`. This is just the raw pointer returned by C++. It's also the least useful to the caller since it cannot be used directly. The more useful option is to assume that the result is a pointer to a scalar. In this case a NumPy scalar can be returned or a Python object such as `int` or `float`.

If the C++ library function can also provide the length of the pointer, then its possible to return a NumPy array. If `owner(library)` is set, the memory will never be released. If `owner(caller)` is set, the the memory will be released when the object is deleted.

The argument `int *result+intent(OUT)+dimension(3)` will create a NumPy array, then pass the pointer to the data to the C function which will presumably fill the contents. The NumPy array will be returned as part of the function result. The dimension attribute must specify a length.

## 15.7 Class Types

An extension type is created for each C++ class:

```
typedef struct {  
    PyObject_HEAD  
    {namespace_scope}{cxx_class} * {PY_obj};  
} {PY_PyObject};
```

### 15.7.1 Extension types

Additional type information can be provided in the YAML file to generate place holders for extension type methods:

```
- name: ExClass2  
  cxx_header: ExClass2.hpp  
  python:  
    type: [dealloc, print, compare, getattr, setattr,  
          getattro, setattro,  
          repr, hash, call, str,  
          init, alloc, new, free, del]
```

## 16.1 Function is really a macro or function pointer

When wrapping a C library, a function which is really a macro may not create a C wrapper. It is necessary to use the option `C_force_wrapper: true` to create a wrapper which will expand the macro and create a function which the Fortran wrapper may call. This same issue occurs when the function is really a function pointer.

When wrapping C++, a C wrapper is always created to create an extern C symbol that Fortran can call. So this problem does not occur.

## 16.2 `F_name_impl` with `fortran_generic`

Using the `F_name_impl` format string to explicitly name a Fortran wrapper combined with the `fortran_generic` field may present some surprising behavior. The routine `BA_change` takes a `long` argument. However, this is inconvenient in Fortran since the default integer is typically an `int`. When passing a constant you need to explicitly state the kind as `0_C_LONG`. Shroud lets you create a generic routine which will also accept 0. But if you explicitly name the function using `F_name_impl`, both Fortran generated functions will have the same name. The solution is to set format field `F_name_generic` and the option for `F_name_impl_template`.

```
- decl: int BA_change(const char *name, long n)
  format:
    F_name_generic: change
  options:
    F_name_impl_template: "{F_name_generic}{function_suffix}"
  fortran_generic:
    - decl: (int n)
      function_suffix: int
    - decl: (long n)
      function_suffix: long
```

Will generate the Fortran code

```
interface change
  module procedure change_int
  module procedure change_long
end interface change
```



A typemap is created for each type to describe to Shroud how it should convert a type between languages for each wrapper. Native types are predefined and a Shroud typemap is created for each `struct` and `class` declaration.

The general form is:

```
declarations:
- type: type-name
  fields:
    field1:
    field2:
```

*type-name* is the name used by C++. There are some fields which are used by all wrappers and other fields which are used by language specific wrappers.

## 17.1 type fields

These fields are common to all wrapper languages.

### 17.1.1 base

The base type of *type-name*. This is used to generalize operations for several types. The base types that Shroud uses are **bool**, **integer**, **real**, **complex**, **string**, **vector**, **struct** or **shadow**.

**integer** includes all integer types such as `short`, `int` and `long`.

### 17.1.2 cpp\_if

A c preprocessor test which is used to conditionally use other fields of the type such as *c\_header* and *cxx\_header*:

```
- type: MPI_Comm
  fields:
    cpp_if: ifdef USE_MPI
```

### 17.1.3 flat\_name

A flattened version of **cxx\_type** which allows the name to be used as a legal identifier in C, Fortran and Python. By default any scope separators are converted to underscores i.e. `internal::Worker` becomes `internal_Worker`. Imbedded blanks are converted to underscores i.e. `unsigned int` becomes `unsigned_int`. And template arguments are converted to underscores with the trailing `>` being replaced i.e. `std::vector<int>` becomes `std_vector_int`.

Complex types set this explicitly since C and C++ have much different type names. The *flat\_name* is always `double_complex` while *c\_type* is `double` `complex` and *cxx\_type* is `complex<double>`.

One use of this name is as the **function\_suffix** for templated functions.

### 17.1.4 idtor

Index of `capsule_data` destructor in the function *C\_memory\_dtor\_function*. This value is computed by Shroud and should not be set. It can be used when formatting statements as `{idtor}`. Defaults to 0 indicating no destructor.

### 17.1.5 sgroup

Groups different base types together. For example, base *integer* and *real* are both *sgroup native*. For many others, they're the same: `base=struct`, `sgroup=struct`.

## 17.2 C and C++

### 17.2.1 c\_type

Name of type in C. Default to *None*.

### 17.2.2 c\_header

Name of C header file required for type. This file is included in the interface header. Only used with *language=c*. Defaults to *None*.

See also *cxx\_header*.

For example, `size_t` requires `stddef.h`:

```
type: size_t
fields:
  c_type: size_t
  cxx_type: size_t
  c_header: <stddef.h>
```

### 17.2.3 c\_to\_cxx

Expression to convert from C to C++. Defaults to *None* which implies *{c\_var}*. i.e. no conversion required.

### 17.2.4 c\_templates

c\_statements for cxx\_T

A dictionary indexed by type of specialized *c\_statements* When an argument has a *template* field, such as type `vector<string>`, some additional specialization of *c\_statements* may be required:

```
c_templates:
  string:
    intent_in_buf:
      - code to copy CHARACTER to vector<string>
```

### 17.2.5 c\_return\_code

None

### 17.2.6 c\_union

None # Union of C++ and C type (used with structs and complex)

### 17.2.7 cxx\_type

Name of type in C++. Defaults to *None*.

### 17.2.8 cxx\_to\_c

Expression to convert from C++ to C. Defaults to *None* which implies *{cxx\_var}*. i.e. no conversion required.

### 17.2.9 cxx\_header

Name of C++ header file required for implementation.

```
c_type: size_t
c_header: '<stddef.h>'
cxx_header: '<cstddef>'
```

See also *c\_header*.

### 17.2.10 impl\_header

**impl\_header** is used for implementation, i.e. the `wrap.cpp` file. For example, `std::string` uses `<string>`. `<string>` should not be in the interface since the wrapper is a C API.

### 17.2.11 wrap\_header

**wrap\_header** is used for generated wrappers for shadow classes. Contains struct definitions for capsules from Fortran.

A C `int` is represented as:

```
type: int
fields:
  c_type: int
  cxx_type: int
```

## 17.3 Fortran

### 17.3.1 i\_module

Fortran modules needed for type in the interface. A dictionary keyed on the module name with the value being a list of symbols. Similar to **f\_module**. Defaults to *None*.

In this example, the symbol `indextype` is created by a typedef which creates a symbol in Fortran. This symbol, `indextype`, must be imported into the interface.

```
typedef int indextype;
```

```
indextype:
  --import--:
  - indextype
```

### 17.3.2 i\_type

Type declaration for `bind(C)` interface. Defaults to *None* which will then use *f\_type*.

### 17.3.3 f\_cast

Expression to convert Fortran type to C type. This is used when creating a Fortran generic functions which accept several type but call a single C function which expects a specific type. For example, type `int` is defined as `int({f_var}, C_INT)`. This expression converts *f\_var* to a `integer(C_INT)`. Defaults to *{f\_var}* i.e. no conversion.

### 17.3.4 f\_derived\_type

Fortran derived type name. Defaults to *None* which will use the C++ class name for the Fortran derived type name.

### 17.3.5 f\_kind

Fortran kind of type. For example, `C_INT` or `C_LONG`. Defaults to *None*.

### 17.3.6 f\_module

Fortran modules needed for type in the implementation wrapper. A dictionary keyed on the module name with the value being a list of symbols. Defaults to *None*.

```
f_module:
  iso_c_binding:
    - C_INT
```

### 17.3.7 f\_type

Name of type in Fortran. ( `integer(C_INT)` ) Defaults to *None*.

### 17.3.8 f\_to\_c

None Expression to convert from Fortran to C.

example

An int argument is converted to Fortran with the typemap:

```
typemap:
- type: int
  fields:
    f_type: integer(C_INT)
    f_kind: C_INT
    f_module:
      iso_c_binding:
        - C_INT
    f_cast: int({f_var}, C_INT)
```

A struct defined in another YAML file.

```
typemap:
- type: Cstruct1
  fields:
    base: struct
    cxx_header:
      - struct.hpp
    wrap_header:
      - wrapstruct.h
    c_type: STR_cstruct1
    f_derived_type: cstruct1
    f_module_name: struct_mod
```

## 17.4 Statements

Each language also provides a section that is used to insert language specific statements into the wrapper. These are named **c\_statements**, **f\_statements**, and **py\_statements**.

They are broken down into several resolutions. The first is the intent of the argument. *result* is used as the intent for function results.

**in** Code to add for argument with `intent (IN)`. Can be used to convert types or copy-in semantics. For example, `char * to std::string`.

**out** Code to add after call when `intent (OUT)`. Used to implement copy-out semantics.

**inout** Code to add after call when `intent (INOUT)`. Used to implement copy-out semantics.

**result** Result of function. Including when it is passed as an argument, *F\_string\_result\_as\_arg*.

Each intent is then broken down into code to be added into specific sections of the wrapper. For example, **declaration**, **pre\_call** and **post\_call**.

Each statement is formatted using the format dictionary for the argument. This will define several variables.

**c\_var** The C name of the argument.

**cxx\_var** Name of the C++ variable.

**f\_var** Fortran variable name for argument.

For example:

```
f_statements:
  intent_in:
    - '{c_var} = {f_var}  ! coerce to C_BOOL'
  intent_out:
    - '{f_var} = {c_var}  ! coerce to logical'
```

Note that the code lines are quoted since they begin with a curly brace. Otherwise YAML would interpret them as a dictionary.

See the language specific sections for details.

Shroud can be thought of as a fancy macro processor. The statement data structure is used to define code that should be used to create the wrapper. Combinations of language, type and attributes are used to select a statement entry.

## 18.1 Passing function result as an argument

This section explains how statements are used to generate code for functions which return a struct.

Compiler ABI do not agree on how some function results should be returned. To ensure portability, some function results must be passed as an additional argument. This is typically more complicated types such as struct or complex.

```
dict(  
    name="f_function_struct_scalar",  
    alias=[  
        "c_function_struct_scalar",  
    ],  
    f_arg_call=["{f_var}"],  
  
    c_arg_decl=["{c_type} *{c_var}"],  
    i_arg_decl=["{f_type}, intent(OUT) :: {c_var}"],  
    i_arg_names=["{c_var}"],  
    i_import=["{f_kind}"],  
    c_return_type="void", # Convert to function.  
    cxx_local_var="result",  
    c_post_call=[  
        "memcpy((void *) {c_var}, (void *) &{cxx_var}, sizeof({cxx_var}))",  
    ],  
)
```





# CHAPTER 19

---

## C Statements

---

---

**Note:** Work in progress

---

```
extern "C" {  
  
{C_return_type} {C_name}({C_prototype})  
{  
    {c_pre_call}  
    {c_call_code}    {call}    arg_call  
    {post_call_pattern}  
    {c_post_call}  
    {c_final}  
    {c_return}  
}  
}
```

C\_prototype -> c\_arg\_decl

A corresponding bind(C) interface can be created for Fortran.

```
{F_C_subprogram} {F_C_name}({F_C_arguments}) &  
    {F_C_result_clause} &  
bind(C, name="{C_name}")  
f_c_module / f_c_module_line  
f_c_import  
f_c_arg_decl  
f_c_result_decl  
end {F_C_subprogram} {F_C_name}
```

Where F\_C\_clause = F\_C\_arguments = f\_c\_arg\_names F\_C\_result\_clause = f\_c\_result\_var

## 19.1 Lookup statements

The statements for an argument are looked up by converting the type and attributes into an underscore delimited string.

- language - `c`
- intent - `in`, `out`, `inout`, `function`, `ctor`, `dtor`, `getter`, `setter`
- group from `typemap`. `native`
- pointer - `scalar`, `*`, `**`
- api - from attribute `buf`, `capsule`, `capptr`, `cdesc` and `cfi`.
- deref - from attribute `allocatable`, `pointer`, `raw`, `result-as-arg`, `scalar`

### 19.1.1 template

Each template argument is appended to the initial statement name. `targ`, `group` and `pointer`

## 19.2 c\_statements

### 19.2.1 name

A name can contain variants separated by `/`.

```
- name: c_in/out/inout_native*_cfi
```

This is equivalent to having three groups:

```
- name: c_in_native*_cfi
- name: c_out_native*_cfi
- name: c_inout_native*_cfi
```

### 19.2.2 iface\_header

List of header files which will be included in the generated header for the C wrapper. These headers must be C only and will be included after `ifdef __cplusplus`. Used for headers needed for declarations in `c_arg_decl`. Can contain headers required for the generated prototypes.

For example, `ISO_Fortran_binding.h` is C only.

### 19.2.3 impl\_header

A list of header files which will be added to the C wrapper implementation. These headers may include C++ code.

### 19.2.4 c\_helper

A blank delimited list of helper functions which will be added to the wrapper file. The list will be formatted to allow for additional flexibility:

```
c_helper: capsule_data_helper vector_context vector_copy_{cxx_T}
```

These functions are defined in `whelper.py`. There is no current way to add additional functions.

### 19.2.5 `cxx_local_var`

If a local C++ variable is created for an argument by `pre_call`, `cxx_local_var` indicates if the local variable is a **pointer**, **scalar** or **result**. .. This sets `cxx_var` is set to `SH_{c_var}`. This will properly dereference the variable when passed to the C++ function. It will also set the format fields `cxx_member`. For example, a `std::string` argument is created for the C++ function from the `char *` argument passed into the C API wrapper.

```
name="c_inout_string",
cxx_local_var="scalar",
pre_call=["{c_const}std::string {cxx_var}({c_var});"],
```

Set to `**return**` when the `*c_var*` is passed in as an argument and a C++ variable must be created.  
 Ex ``c_function_shadow``.  
 In this case, `*cxx_to_c*` is defined so a local variable will already be created, unless `*language=c*` in which case `*cxx_to_c*` is unneeded.

### 19.2.6 `c_arg_decl`

A list of declarations to append to the prototype in the C wrapper. Defaults to *None* which will cause Shroud to generate an argument from the wrapped function's argument. An empty list will cause no declaration to be added. Functions do not add arguments by default. A trailing semicolon will be provided.

---

**Note:** `c_arg_decl`, `i_arg_decl`, and `i_arg_names` must all exist in a group and have the same number of names.

---

### 19.2.7 `i_arg_decl`

A list of dummy argument declarations in the Fortran `bind(C)` interface. The variable to be declared is `c_var`. `i_module` can be used to add `USE` statements needed by the declarations. An empty list will cause no declaration to be added.

---

**Note:** `c_arg_decl`, `i_arg_decl`, and `i_arg_names` must all exist in a group and have the same number of names.

---

### 19.2.8 `i_arg_names`

Names of arguments to pass to C function. Defaults to `{F_C_var}`. An empty list will cause no declaration to be added.

---

**Note:** `c_arg_decl`, `i_arg_decl`, and `i_arg_names` must all exist in a group and have the same number of names.

---

### 19.2.9 i\_result\_decl

A list of declarations in the Fortran interface for a function result value.

### 19.2.10 i\_import

List of names to import into the Fortran interface. The names will be expanded before being used.

In this example, Shroud creates *F\_array\_type* derived type in the module and it is used in the interface.

```
i_import=["{F_array_type}"],
```

### 19.2.11 i\_module

Fortran modules used in the Fortran interface:

```
i_module=dict(iso_c_binding=["C_PTR"]),
```

### 19.2.12 i\_module\_line

Fortran modules used in the Fortran interface as a single line which allows format strings to be used.

```
i_module_line="iso_c_binding:{f_kind}",
```

The format is:

```
module ":" symbol [ ",", symbol ]* [ ";" module ":" symbol [ ",", symbol ]* ]
```

### 19.2.13 c\_arg\_call

### 19.2.14 c\_pre\_call

Code used with *intent(in)* arguments to convert from C to C++.

### 19.2.15 c\_call

Code to call function. This is usually generated. An exception which require explicit call code are constructors and destructors for shadow types.

### 19.2.16 c\_post\_call

Code used with *intent(out)* arguments and function results. Can be used to convert results from C++ to C.

### 19.2.17 c\_final

Inserted after *post\_call* and before *ret*. Can be used to release intermediate memory in the C wrapper.

### 19.2.18 c\_return

List of code for return statement. Usually generated but can be replaced. For example, with constructors.

Useful to convert a subroutine into a function. For example, convert a `void` function which fills a `std::vector` to return the number of items.

### 19.2.19 c\_return\_type

Explicit return type when it is different than the functions return type. For example, with shadow types.

```
c_return_type: long
c_return:
- return Darg->size;
```

*return\_type* can also be used to convert a C wrapper into a void function. This is useful for functions which return pointers but the pointer value is assigned to a subroutine argument which holds the pointer (For example, `CFI_cdesc_t`). The type (`C_PTR`) which would be return by the C wrapper is unneeded by the Fortran wrapper.

### 19.2.20 destructor\_name

A name for the destructor code in *destructor*. Must be unique. May include format strings:

```
destructor_name: std_vector_{cxx_T}
```

### 19.2.21 destructor

A list of lines of code used to delete memory. Usually allocated by a *pre\_call* statement. The code is inserted into *C\_memory\_dtor\_function* which will provide the address of the memory to destroy in the variable `void *ptr`. For example:

```
destructor:
- std::vector<{cxx_T}> *cxx_ptr = reinterpret_cast<std::vector<{cxx_T}> *>(ptr);
- delete cxx_ptr;
```

### 19.2.22 owner

Set *owner* of the memory. Similar to attribute *owner*.

Used where the `new`` operator is part of the generated code. For example where a class is returned by value or a constructor. The C wrapper must explicitly allocate a class instance which will hold the value from the C++ library function. The Fortran shadow class must keep this copy until the shadow class is deleted.

Defaults to *library*.

### 19.2.23 c\_temps

A list of suffixes for temporary variable names.

```
c_temps=["len"]
```

Create variable names in the format dictionary using  
``{fmt.c\_temp}{rootname}\_{name}``.  
For example, argument `*foo*` creates `*SHT_foo_len*`.

### 19.2.24 `c_local`

Similar to *temps* but uses `{fmt.C_local}{rootname}_{name}`. *temps* is intended for arguments and is typically used in a *mixin* group. *local* is used by group to generate names for local variables. This allows creating names without conflicting with *temps* from a *mixin* group.

### 19.2.25 `lang_c` and `lang_cxx`

Language specific versions of each field can be added to these dictionaries. The version which corresponds to the YAML file *language* field will be used.

```
lang_c=dict(  
    impl_header=["<stddef.h>"],  
)  
lang_cxx=dict(  
    impl_header=["<cstddef>"],  
)
```

---

## Fortran Statements

---

---

**Note:** Work in progress.

---

Typemaps are used to add code to the generated wrappers to replace the default code.

The statements work together to pass variables and metadata between Fortran and C.

### 20.1 fc\_statements

A Fortran wrapper is created out of several segments.

```
{F_subprogram} {F_name_impl} ({F_arguments}) {F_result_clause}
  f_module
  f_import
  f_arg_decl
  ! splicer begin
  f_declare
  f_pre_call
  f_call {} ( {f_arg_call} )
  f_post_call
  ! splicer end
end {F_subprogram} {F_name_impl}
```

The `bind(C)` interface is defined by the `cstatements` since it must match the C wrapper that is being called. The C wrapper may have a different API than the Fortran wrapper since the Fortran may pass down additional arguments.

#### 20.1.1 f\_helper

Blank delimited list of Fortran helper function names to add to generated Fortran code. These functions are defined in `whelper.py`. There is no current way to add user defined helper functions.

### 20.1.2 `f_module`

USE statements to add to Fortran wrapper. A dictionary of list of ONLY names:

```
f_module:
  iso_c_binding:
    - C_SIZE_T
```

### 20.1.3 `f_need_wrapper`

Shroud tries to only create an interface for a C function to avoid the extra layer of a Fortran wrapper. However, often the Fortran wrapper needs to do some work that the C wrapper cannot. This field can be set to True to ensure the Fortran wrapper is created. This is used when an assignment is needed to do a type coercion; for example, with logical types.

A wrapper will always be created if the **F\_force\_wrapper** option is set.

### 20.1.4 `f_arg_name`

List of name of arguments for Fortran subprogram. Will be formatted before being used to expand `{f_var}`.

Any function result arguments will be added at the end. Only added if `f_arg_decl` is also defined.

### 20.1.5 `f_arg_decl`

List of argument or result declarations. Usually constructed from YAML *decl* but sometimes needs to be explicit to add Fortran attributes such as `TARGET` or `POINTER`. Also used when a function result is converted into an argument. Added before splicer since it is part of the API and must not change. Additional declarations can be added within the splicer via `f_declare`.

```
f_arg_name=["{f_var}"],
f_arg_decl=[
  "character, value, intent(IN) :: {f_var}",
],
```

### 20.1.6 `f_arg_call`

List of arguments to pass to C wrapper. By default the arguments of the Fortran wrapper are passed to the C wrapper. The list of arguments can be set to pass additional arguments or expressions. The format field `f_var` the name of the argument.

When used with a **f\_function** statement, the argument will be added to the end of the call list.

```
f_arg_call=[
  "C_LOC({f_var}) "
],
```

```
f_arg_call=[
  "{f_var}",
  "len({f_var}, kind=C_INT) ",
],
```



### 20.1.7 f\_declare

A list of declarations needed by *f\_pre\_call* or *f\_post\_call*. Usually a *c\_local\_var* is sufficient. No executable statements should be used since all declarations must be grouped together. Implies *f\_need\_wrapper*. Added within the splicer to make it easier to replace in the YAML file.

### 20.1.8 f\_import

List of names to import into the Fortran wrapper. The names will be expanded before being used.

In this example, Shroud creates *F\_array\_type* derived type in the module and it is used in the interface.

```
f_import=["{F_array_type}"],
```

### 20.1.9 f\_module

Fortran modules used in the Fortran wrapper:

```
f_module=dict(iso_c_binding=["C_PTR"]),
```

### 20.1.10 f\_module\_line

Fortran modules used in the Fortran wrapper as a single line which allows format strings to be used.

```
f_module_line="iso_c_binding:{f_kind}",
```

The format is:

```
module ":" symbol [ ",", symbol ]* [ ";" module ":" symbol [ ",", symbol ]* ]
```

### 20.1.11 f\_pre\_call

Statement to execute before call, often to coerce types when *f\_cast* cannot be used. Implies *f\_need\_wrapper*.

### 20.1.12 f\_call

Code used to call the function. Defaults to `{F_result} = {F_C_call}({f_arg_call})`

For example, to assign to an intermediate variable:

```
f_declare=[
    "type(C_PTR) :: {c_local_ptr}",
],
f_call=[
    "{c_local_ptr} = {F_C_call}({f_arg_call})",
],
f_local=["ptr"],
```

### 20.1.13 `f_post_call`

Statement to execute after call. Can be use to cleanup after *f\_pre\_call* or to coerce the return value. Implies *f\_need\_wrapper*.

### 20.1.14 `f_result`

Name of result variable. Added as the `RESULT` clause of the subprogram statement. Can be used to change a subroutine into a function.

In this example, the subroutine is converted into a function which will return the number of items copied into the result argument.

```
- decl: void vector_iota_out_with_num2(std::vector<int> &arg+intent(out))
  fstatements:
    f:
      f_result: num
      f_module:
        iso_c_binding: ["C_LONG"]
      f_declare:
        - "integer(C_LONG) :: num"
      f_post_call:
        - "num = Darg%size"
```

When set to **subroutine** it will treat the subprogram as a subroutine.

### 20.1.15 `f_temps`

A list of suffixes for temporary variable names.

```
f_temps=["len"]
```

Create variable names in the format dictionary using  
``{fmt.c\_temp}{rootname}\_{name}``.  
For example, argument `*foo*` creates `*SHT_foo_len*`.

### 20.1.16 `f_local`

Similar to *f\_temps* but uses `{fmt.C_local}{rootname}_{name}`. *temps* is intended for arguments and is typically used in a *mixin* group. *f\_local* is used by group to generate names for local variables. This allows creating names without conflicting with *f\_temps* from a *mixin* group.

## 20.2 notimplemented

If True the statement is not implemented. The generated function will have `#if 0` surrounding the wrapper.

This is a way to avoid generating code which will not compile when the notimplemented wrapper is not needed. For example, the C wrapper for a C++ function when only the C bufferify wrapper is needed for Fortran. The statements should eventually be completed to wrap the function properly.

## 20.3 How typemaps are found

### 20.3.1 alias

List of other names which will be used for its contents.

```
name="fc_out_string**_cdesc_allocatable",
alias=[
    "f_out_string**_cdesc_allocatable",
    "c_out_string**_cdesc_allocatable",
],
```



## 21.1 Command Line Options

**help** Show this help message and exit.

**version** Show program's version number and exit.

**outdir OUTDIR** Directory for output files. Defaults to current directory.

**outdir-c-fortran OUTDIR\_C\_FORTRAN** Directory for C/Fortran wrapper output files, overrides *–outdir*.

**outdir-python OUTDIR\_PYTHON** Directory for Python wrapper output files, overrides *–outdir*.

**outdir-lua OUTDIR\_LUA** Directory for Lua wrapper output files, overrides *–outdir*.

**outdir-yaml OUTDIR\_YAML** Directory for YAML output files, overrides *–outdir*.

**logdir LOGDIR** Directory for log files. Defaults to current directory.

**cfiles CFILES** Output file with list of C and C++ files created.

**ffiles FFILES** Output file with list of Fortran created.

**path PATH** Colon delimited paths to search for splicer files, may be supplied multiple times to append to path.

**sitedir** Return the installation directory of shroud and exit. This path can be used to find `cmake/SetupShroud.cmake`.

**write-helpers BASE** Write files which contain the available helper functions into the files `BASE.c` and `BASE.f`.

**write-statements BASE** Write a file which contain the statements tree. Used for debugging.

**write-version** Write Shroud version into generated files. `--nowrite-version` will not write the version and is used by the testsuite to avoid changing every reference file when the version changes.

**yaml-types FILE** Write a YAML file with the default types.

## 21.2 Global Fields

**copyright** A list of lines to add to the top of each generate file. Do not include any language specific comment characters since Shroud will add the appropriate comment delimiters for each language.

**classes** A list of classes. Each class may have fields as detailed in *Class Fields*.

**cxx\_header** Blank delimited list of header files which will be included in the implementation file. The order will be preserved when generating wrapper files.

**format** Dictionary of Format fields for the library. Described in *Format Fields*.

**language** The language of the library to wrap. Valid values are `c` and `c++`. The default is `c++`.

**library** The name of the library. Used to name output files and modules. The first three letters are used as the default for `C_prefix` option. Defaults to *library*. Each YAML file is intended to wrap a single library.

**options** Dictionary of option fields for the library. Described in *Options*

**patterns** Code blocks to insert into generated code. Described in *Patterns*.

**splicer** A dictionary mapping file suffix to a list of splicer files to read:

```
splicer:  
  c:  
  - filename1.c  
  - filename2.c
```

**types** A dictionary of user define types. Each type is a dictionary of members describing how to map a type between languages. Described in *Typemaps* and *Types Map*.

## 21.3 Class Fields

**cxx\_header** C++ header file name which will be included in the implementation file. If unset then the global *cxx\_header* will be used.

**format** Format fields for the class. Creates scope within library. Described in *Format Fields*.

**declarations** A list of declarations in the class. Each function is defined by *Function Fields*

**fields:** A dictionary of fields used to update the typemap.

**options** Options fields for the class. Creates scope within library. Described in *Options*

## 21.4 Function Fields

Each function can define fields to define the function and how it should be wrapped. These fields apply only to a single function i.e. they are not inherited.

**C\_prototype** XXX override prototype of generated C function

**cxx\_template** A list that define how each templated argument should be instantiated:

```
decl: void Function7(ArgType arg)  
cxx_template:  
- instantiation: <int>  
- instantiation: <double>
```

**decl** Function declaration. Parsed to extract function name, type and arguments descriptions.

**default\_arg\_suffix** A list of suffixes to apply to C and Fortran functions generated when wrapping a C++ function with default arguments. The first entry is for the function with the fewest arguments and the final entry should be for all of the arguments.

**format** Format fields for the function. Creates scope within container (library or class). Described in [Format Fields](#).

**fortran\_generic** A dictionary of lists that define generic functions which will be created. This allows different types to be passed to the function. This feature is provided by C which will promote arguments. Each generic function will have a suffix which defaults to an underscore plus a sequence number. This change be changed by adding *function\_suffix* for a declaration.

```
decl: void GenericReal(double arg)
fortran_generic:
- decl: (float arg)
  function_suffix: suffix1
- decl: (double arg)
```

A full example is at :ref:`GenericReal <example\_GenericReal>`.

**options** Options fields for the function. Creates scope within container (library or class). Described in [Options](#)

**return\_this** If true, the method returns a reference to *this*. This idiom can be used to chain calls in C++. This idiom does not translate to C and Fortran. Instead the *C\_return\_type* format is set to *void*.

## 21.5 Options

**C\_API\_case** Controls mangling of C++ library names to C names via the format field *C\_name\_api*. Possible values are *lower*, *upper*, *underscore*, or *preserve*. Defaults to *preserve* and will be combined with *C\_prefix*. For example, **C\_name\_template** includes {*C\_prefix*} {*C\_name\_scope*} {*C\_name\_api*}.

**C\_extern\_C** Set to *true* when the C++ routine is `extern "C"`. Defaults to *false*.

**C\_force\_wrapper** If *true*, always create an explicit C wrapper. When *language* is *c++* a C wrapper is always created. When wrapping C, the wrapper is automatically created if there is work for it to do. For example, *pre\_call* or *post\_call* is defined. The user should set this option when wrapping C and the function is really a macro or a function pointer variable. This forces a function to be created allowing Fortran to use the macro or function pointer.

**C\_line\_length** Control length of output line for generated C. This is not an exact line width, but is instead a hint of where to break lines. A value of 0 will give the shortest possible lines. Defaults to 72.

**C\_shadow\_result** If *true*, the api for the function result will be set to *cappr*, otherwise it will be set to *capsule*. In both cases, the result is passed from Fortran to the C api as an additional argument. With *C\_shadow\_result* true, a pointer to the capsule is returned as the function result. If *false*, the C wrapper is a *void* function. *cappr* acts more like C library functions such as *strcpy* which return a pointer to the result. *capsule* makes for a simpler Fortran wrapper implementation since the function result is not used since it is identical to the result argument.

**class\_baseclass** Used to define a baseclass for a struct for *wrap\_struct\_as=class*". The baseclass must already be defined earlier in the YAML file. It must be in the same namespace as the struct.

```
- decl: struct Cstruct_as_class
  options:
    wrap_struct_as: class
- decl: struct Cstruct_as_subclass
  options:
```

(continues on next page)

(continued from previous page)

```
wrap_struct_as: class
class_baseclass: Cstruct_as_class
```

This is equivalent to the C++ code

```
.. code-block:: c++
```

```
class Cstruct_as_class;
class Cstruct_as_subclass : public Cstruct_as_class;
```

The corresponding Fortran wrapper will have

```
type cstruct_as_class
  type (STR_SHROUD_capsule_data) :: cxxmem
end type cstruct_as_class
type, extends (cstruct_as_class) :: cstruct_as_class
end type cstruct_as_subclass
```

**class\_ctor** Indicates that this function is a constructor for a struct. The value is the name of the struct. Useful for *wrap\_struct\_as=class* when used with C.

```
- decl: struct Cstruct_as_class {
    int x1;
    int y1;
};
options:
  wrap_struct_as: class
- decl: Cstruct_as_class *Create_Cstruct_as_class(void)
options:
  class_ctor: Cstruct_as_class
```

**class\_method** Indicates that this function is a method for a struct.

**CXX\_standard** C++ standard. Defaults to *2011*. See *nullptr*.

**debug** Print additional comments in generated files that may be useful for debugging. Defaults to *false*.

**debug\_index** Print index number of function and relationships between C and Fortran wrappers in the wrappers and json file. The number changes whenever a new function is inserted and introduces lots of meaningless differences in the test answers. This option is used to avoid the clutter. If needed for debugging, then set to *true*. **debug** must also be *true*. Defaults to *false*.

**doxygen** If True, create doxygen comments.

**F\_API\_case** Controls mangling of C++ library names to Fortran names via the format field *F\_name\_api*. Possible values are *lower*, *upper*, *underscore*, or *preserve*. Defaults to *underscore* to convert CamelCase to *camel\_case*. Since Fortran is case insensitive, users are not required to respect the case of the C++ name. Using *underscore* makes the names easier to read regardless of the case.

**F\_assumed\_rank\_min** Minimum rank of argument with assumed-rank. Defaults to 0 (scalar).

**F\_assumed\_rank\_max** Maximum rank of argument with assumed-rank. Defaults to 7.

**F\_blanknull** Default value of attribute *+blanknull* for `const char *` arguments. This attribute will convert blank Fortran strings to a NULL pointer.

**F\_CFI** Use the C Fortran Interface provided by *Futher Interoperability with C* from Fortran 2018 (initially defined in TS29113 2012).



**F\_create\_bufferify\_function** Controls creation of a *bufferify* function. If *true*, an additional C function is created which receives *bufferified* arguments - i.e. the `len`, `len_trim`, and `size` may be added as additional arguments. Set to *false* when you want to avoid passing this information. This will avoid a copy of `CHARACTER` arguments required to append a trailing null. Defaults to *true*.

**F\_create\_generic** Controls creation of a generic interface. It defaults to *true* for most cases but will be set to *False* if a function is templated on the return type since Fortran does not distinguish generics based on return type (similar to overloaded functions based on return type in C++).

**F\_default\_args** Decide how to handle C++ default argument functions. See [Default Arguments](#).

**generic** Create a wrapper for each variation from all arguments to no arguments defaulted. In Fortran, create a generic interface.

**optional** Make each default argument as a Fortran `OPTIONAL` argument.

**require** Require all arguments to be provided to the wrapper.

**F\_line\_length** Control length of output line for generated Fortran. This is not an exact line width, but is instead a hint of where to break lines. A value of 0 will give the shortest possible lines. Defaults to 72.

**F\_force\_wrapper** If *true*, always create an explicit Fortran wrapper. If *false*, only create the wrapper when there is work for it to do; otherwise, call the C function directly. For example, a function which only deals with native numeric types does not need a wrapper since it can be called directly by defining the correct interface. The default is *false*.

**F\_standard** The fortran standard. Defaults to *2003*. This effects the `modl` argument of the `allocate` statement.

**F\_return\_fortran\_pointer** Use `c_f_pointer` in the Fortran wrapper to return a Fortran pointer instead of a `type(C_PTR)` in routines which return a pointer. It does not apply to `char *`, `void *`, and routines which return a pointer to a class instance. Defaults to *true*.

**F\_string\_len\_trim** For each function with a `std::string` argument, create another C function which accepts a buffer and length. The C wrapper will call the `std::string` constructor, instead of the Fortran wrapper creating a NULL terminated string using `trim`. This avoids copying the string in the Fortran wrapper. Defaults to *true*.

**F\_struct\_getter\_setter** If true, a getter and setter will be created for struct members which are a pointer to native type. This allows a Fortran pointer to be used with the field instead of having to deal with the `type(C_PTR)` directly. Default to *true*

**F\_trim\_char\_in** Controls code generation for `const char *` arguments. If *True*, Fortran perform a `TRIM` and concatenates `C_NULL_CHAR`. If *False*, it will be done in C. If the only need for the C wrapper is to null-terminate a string (wrapping a c library and no other argument requires a wrapper), then the C wrapper can be avoid by moving the null-termination action to Fortran. Default is *True*.

`literalinclude`

Write some text lines which can be used with Sphinx's `literalinclude` directive. This is used to insert the generated code into the documentation. Can be applied at the top level or any declaration. Setting *literalinclude* at the top level implies *literalinclude2*.

`literalinclude2`

Write some text lines which can be used with Sphinx's `literalinclude` directive. Only effects some entities which do not map to a declarations such as some helper functions or types. Only effective at the top level.

Each Fortran interface will be included in its own `interface` block. This is to provide the interface context when code is added to the documentation.

**LUA\_API\_case** Controls mangling of C++ library names to Lua names via the format field *LUA\_name\_api*. Possible values are *lower*, *upper*, *underscore*, or *preserve*. Defaults to *preserve*.

**PY\_create\_generic** Controls creation of a multi-dispatch function with overloaded/templated functions. It defaults to *true* for most cases but will be set to *False* if a function is templated on the return type since Fortran does not distinguish generics based on return type (similar to overloaded functions based on return type in C++).

**PY\_write\_helper\_in\_util** When *True* helper functions will be written into the utility file *PY\_utility\_filename*. Useful when there are lots of classes since helper functions may be duplicated in several files. The value of format *PY\_helper\_prefix* will have *C\_prefix* append to create names that are unique to the library. Defaults to *False*.

**return\_scalar\_pointer** Determines how to treat a function which returns a pointer to a scalar (it does not have the *dimension* or *rank* attribute). **scalar** return as a scalar or **pointer** to return as a pointer. This option does not effect the C or Fortran wrapper. For Python, **pointer** will return a NumPy scalar. Defaults to *pointer*.

**show\_splicer\_comments** If *true* show comments which delineate the splicer blocks; else, do not show the comments. Only the global level option is used.

**wrap\_class\_as** Defines how a *class* should be wrapped. If *class*, wrap using a shadow type. If *struct*, wrap the same as a *struct*. Default is *class*.

**wrap\_struct\_as** Defines how a *struct* should be wrapped. If *struct*, wrap a *struct* as a Fortran derived-type. If *class*, wrap a *struct* the same as a *class* using a shadow type. Default is *struct*.

**wrap\_c** If *true*, create C wrappers. Defaults to *true*.

**wrap\_fortran** If *true*, create Fortran wrappers. Defaults to *true*.

**wrap\_python** If *true*, create Python wrappers. Defaults to *false*.

**wrap\_lua** If *true*, create Lua wrappers. Defaults to *false*.

### 21.5.1 Option Templates

Templates are set in options then expanded to assign to the format dictionary to create names in the generated code.

**C\_enum\_template** Name of enumeration in C wrapper. {C\_prefix}{C\_name\_scope}{enum\_name}

**C\_enum\_member\_template** Name of enumeration member in C wrapper.  
{C\_prefix}{C\_name\_scope}{enum\_member\_name}

**C\_header\_filename\_class\_template** wrap{file\_scope}.{C\_header\_filename\_suffix}

**C\_header\_filename\_library\_template** wrap{library}.{C\_header\_filename\_suffix}

**C\_header\_filename\_namespace\_template** wrap{scope\_file}.{C\_header\_filename\_suffix}

**C\_impl\_filename\_class\_template** wrap{file\_scope}.{C\_impl\_filename\_suffix}

**C\_impl\_filename\_library\_template** wrap{library}.{C\_impl\_filename\_suffix}

**C\_impl\_filename\_namespace\_template** wrap{scope\_file}.{C\_impl\_filename\_suffix}

**C\_memory\_dtor\_function\_template** Name of function used to delete memory allocated by C or C++. defaults to {C\_prefix}SHROUD\_memory\_destructor.

**C\_name\_template** {C\_prefix}{C\_name\_scope}{C\_name\_api}{function\_suffix}{f\_c\_suffix}{template\_suffix}

**C\_name\_typedef\_template** {C\_prefix}{C\_name\_scope}{typedef\_name}

**F\_C\_name\_template** {F\_C\_prefix}{F\_name\_scope}{F\_name\_api}{function\_suffix}{f\_c\_suffix}{template\_suffix}

**F\_abstract\_interface\_argument\_template** The name of arguments for an abstract interface used with function pointers. Defaults to {F\_name\_api}\_{argname} where *argname* is the name of the function argument. see [Function Pointers](#).

**F\_abstract\_interface\_subprogram\_template** The name of the abstract interface subprogram which represents a function pointer. Defaults to arg{index} where *index* is the 0-based argument index. See [Function Pointers](#).

**F\_array\_type\_template** {C\_prefix}SHROUD\_array

**F\_capsule\_data\_type\_template** Name of the derived type which is the BIND (C) equivalent of the struct used to implement a shadow class (**C\_capsule\_data\_type**). All classes use the same derived type. Defaults to {C\_prefix}SHROUD\_capsule\_data.

**F\_capsule\_type\_template** {C\_prefix}SHROUD\_capsule

**F\_derived\_name\_template** Defaults to {F\_name\_api}.

**F\_enum\_member\_template** Name of enumeration member in Fortran wrapper. {F\_name\_scope}{enum\_member\_lower} Note that *F\_enum\_template* does not exist since only the members are in the Fortran code, not the enum name itself.

**F\_name\_generic\_template** {F\_name\_api}

**F\_impl\_filename\_library\_template** wrapf{library\_lower}.{F\_filename\_suffix}

**F\_name\_impl\_template** {F\_name\_scope}{F\_name\_api}{function\_suffix}{template\_suffix}

**F\_module\_name\_library\_template** {library\_lower}\_mod

**F\_module\_name\_namespace\_template** {file\_scope}\_mod

**F\_name\_function\_template** {F\_name\_api}{function\_suffix}{template\_suffix}

**F\_typedef\_name\_template** {F\_name\_scope}{F\_name\_api}

**LUA\_class\_reg\_template** Name of *luaL\_Reg* array of function names for a class. {LUA\_prefix}{cxx\_class}\_Reg

**LUA\_ctor\_name\_template** Name of constructor for a class. Added to the library's table. {cxx\_class}

**LUA\_header\_filename\_template** lua{library}module.{LUA\_header\_filename\_suffix}

**LUA\_metadata\_template** Name of metatable for a class. {cxx\_class}.metatable

**LUA\_module\_filename\_template** lua{library}module.{LUA\_impl\_filename\_suffix}

**LUA\_module\_reg\_template** Name of *luaL\_Reg* array of function names for a library. {LUA\_prefix}{library}\_Reg

**LUA\_name\_impl\_template** Name of implementation function. All overloaded function use the same Lua wrapper so *function\_suffix* is not needed. {LUA\_prefix}{C\_name\_scope}{underscore\_name}

**LUA\_name\_template** Name of function as know by Lua. All overloaded function use the same Lua wrapper so *function\_suffix* is not needed. {function\_name}

**LUA\_userdata\_type\_template** {LUA\_prefix}{cxx\_class}\_Type

**LUA\_userdata\_member\_template** Name of pointer to class instance in userdata. self

**PY\_array\_arg** How to wrap arrays - numpy or list. Applies to function arguments and to structs when **PY\_struct\_arg** is *class* (struct-as-class). Defaults to *numpy*. Added to *fmt* for functions. Useful for *c\_helpers* in statements.

c\_helper="get\_from\_object\_{c\_type}\_{PY\_array\_arg}",

**PY\_module\_filename\_template** py{library}module.{PY\_impl\_filename\_suffix}

**PY\_header\_filename\_template** py{library}module.{PY\_header\_filename\_suffix}

**PY\_utility\_filename\_template** py{library}util.{PY\_impl\_filename\_suffix}

**PY\_PyTypeObject\_template** {PY\_prefix}{cxx\_class}\_Type

**PY\_PyObject\_template** {PY\_prefix}{cxx\_class}

**PY\_member\_getter\_template** Name of descriptor getter method for a class variable. `{PY_prefix}{cxx_class}_{variable_name}_getter`

**PY\_member\_setter\_template** Name of descriptor setter method for a class variable. `{PY_prefix}{cxx_class}_{variable_name}_setter`

**PY\_member\_object\_template** Name of struct member of type *PyObject* \* which contains the data for member pointer fields. `{variable_name}_obj`.

**PY\_name\_impl\_template** `{PY_prefix}{function_name}{function_suffix}{template_suffix}`

**PY\_numpy\_array\_capsule\_name\_template** Name of *PyCapsule* object used as base object of NumPy arrays. Used to make sure a valid capsule is passed to *PY\_numpy\_array\_dtor\_function*. `{PY_prefix}array_dtor`

**PY\_numpy\_array\_dtor\_context\_template** Name of `const char * []` array used as the *context* field for *PY\_numpy\_array\_dtor\_function*. `{PY_prefix}array_destructor_context`

**PY\_numpy\_array\_dtor\_function\_template** Name of *destructor* in *PyCapsule* base object of NumPy arrays. `{PY_prefix}array_destructor_function`

**PY\_struct\_array\_descr\_create\_template** Name of C/C++ function to create a *PyArray\_Descr* pointer for a structure. `{PY_prefix}{cxx_class}_create_array_descr`

**PY\_struct\_arg** How to wrap structs - numpy, list or class. Defaults to *numpy*.

**PY\_struct\_array\_descr\_variable\_template** Name of C/C++ variable which is a pointer to a *PyArray\_Descr* variable for a structure. `{PY_prefix}{cxx_class}_array_descr`

**PY\_struct\_array\_descr\_name\_template** Name of Python variable which is a `numpy.dtype` for a struct. Can be used to create instances of a C/C++ struct from Python. `np.array((1,3.14), dtype=tutorial.struct1_dtype)` `{cxx_class}_dtype`

**PY\_type\_filename\_template** `py{file_scope}type.{PY_impl_filename_suffix}`

**PY\_type\_impl\_template** Names of functions for type methods such as `tp_init`. `{PY_prefix}{cxx_class}_{PY_type_method}{function_suffix}{template_suffix}`

**PY\_use\_numpy** Allow NumPy arrays to be used in the module. For example, when assigning to a struct-as-class member.

**SH\_class\_getter\_template** Name of generated getter function for class members. The wrapped name will be mangled further to distinguish scope. Defaults to `get_{wrapped_name}`.

**SH\_class\_setter\_template** Name of generated setter function for class members. The wrapped name will be mangled further to distinguish scope. Defaults to `set_{wrapped_name}`.

**SH\_struct\_getter\_template** Name of generated getter function for struct members. The wrapped name will be mangled further to distinguish scope. Defaults to `{struct_name}_get_{wrapped_name}`.

**SH\_struct\_setter\_template** Name of generated setter function for struct members. The wrapped name will be mangled further to distinguish scope. Defaults to `{struct_name}_set_{wrapped_name}`.

**YAML\_type\_filename\_template** Default value for global field `YAML_type_filename` `{library_lower}_types.yaml`

## 21.6 Format Fields

Each scope (library, class, function) has its own format dictionary. If a value is not found in the dictionary, then the parent scopes will be recursively searched.

## 21.6.1 Library

**C\_array\_type** Name of structure used to store metadata about an array such as its address and size. Defaults to `{C_prefix}SHROUD_array`.

**C\_bufferify\_suffix** Suffix appended to generated routine which pass strings as buffers with explicit lengths. Defaults to `_bufferify`.

**C\_capsule\_data\_type** Name of struct used to share memory information with Fortran. Defaults to `SHROUD_capsule_data`.

**C\_header\_filename** Name of generated header file for the library. Defaulted from expansion of option `C_header_filename_library_template`.

**C\_header\_filename\_suffix** Suffix added to C header files. Defaults to `h`. Other useful values might be `hh` or `hxx`.

**C\_header\_utility** A header file with shared Shroud internal typedefs for the library. Default is `types{library}. {C_header_filename_suffix}`.

**C\_impl\_filename** Name of generated C++ implementation file for the library. Defaulted from expansion of option `C_impl_filename_library_template`.

**C\_impl\_filename\_suffix:** Suffix added to C implementation files. Defaults to `cpp`. Other useful values might be `cc` or `cxx`.

**C\_impl\_utility** A implementation file with shared Shroud helper functions. Typically routines which are implemented in C but called from Fortran via `BIND(C)`. The must have global scope. Default is `util{library}. {C_header_filename_suffix}`.

**C\_local** Prefix for C compatible local variable. Defaults to `SHC_`.

**C\_memory\_dtor\_function** Name of function used to delete memory allocated by C or C++.

**C\_name\_api** Root name that is used to create various names in the C API. Defaulted by the **C\_API\_case** option with values *lower*, *upper*, *underscore*, or *preserve*. If set explicitly then **C\_API\_case** will have no effect.

May be blank for namespaces to avoid adding the name to `C_name_scope`.

**C\_name\_scope** Underscore delimited name of namespace, class, enumeration. Used to ‘flatten’ nested C++ names into global C identifiers. Ends with trailing underscore to allow the next scope to be appended. Does not include toplevel *namespace*. For example, **C\_name\_template** includes `{C_prefix}{C_name_scope}{C_name_api}`.

`C_name_scope` will replace *class\_name* with the instantiated *class\_name*. which will contain a template arguments.

This is a computed using `C_name_api` and should not be set explicitly.

**C\_result** The name of the C wrapper’s result variable. It must not be the same as any of the routines arguments. It defaults to *rv*.

**C\_string\_result\_as\_arg** The name of the output argument for string results. Function which return `char` or `std::string` values return the result in an additional argument in the C wrapper. See also `F_string_result_as_arg`.

**c\_temp** Prefix for wrapper temporary working variables. Defaults to `SHT_`.

**C\_this** Name of the C object argument. Defaults to `self`. It may be necessary to set this if it conflicts with an argument name.

**CXX\_local** Prefix for C++ compatible local variable. Defaults to `SHCXX_`.

**CXX\_this** Name of the C++ object pointer set from the `C_this` argument. Defaults to `SH_this`.

**F\_array\_type** Name of derived type used to store metadata about an array such as its address and size. Default value from option *F\_array\_type\_template* which defaults to *{C\_prefix}SHROUD\_array*.

**F\_C\_prefix** Prefix added to name of generated Fortran interface for C routines. Defaults to *c\_*.

**F\_capsule\_data\_type** Name of derived type used to share memory information with C or C++. Member of *F\_array\_type*. Default value from option *F\_capsule\_data\_type\_template* which defaults to *{C\_prefix}SHROUD\_capsule\_data*.

Each class has a similar derived type, but with a different name to enforce type safety.

**F\_capsule\_delete\_function** Name of type-bound function of *F\_capsule\_type* which will delete the memory in the capsule. Defaults to *SHROUD\_capsule\_delete*.

**F\_capsule\_final\_function** Name of function used was `FINAL` of *F\_capsule\_type*. The function is used to release memory allocated by C or C++. Defaults to *SHROUD\_capsule\_final*.

**F\_capsule\_type** Name of derived type used to release memory allocated by C or C++. Default value from option *F\_capsule\_type\_template* which defaults to *{C\_prefix}SHROUD\_capsule*. Contains a *F\_capsule\_data\_type*.

**F\_derived\_member** A *F\_capsule\_data\_type* use to reference C++ memory. Defaults to *cxxmem*.

**F\_derived\_member\_base** The *F\_derived\_member* for the base class of a class. Only single inheritance is support via the `EXTENDS` keyword in Fortran.

**F\_filename\_suffix** Suffix added to Fortran files. Defaults to *f*. Other useful values might be *F* or *f90*.

**F\_module\_name** Name of module for Fortran interface for the library. Defaulted from expansion of option *F\_module\_name\_library\_template* which is *{library\_lower}\_mod*. Then converted to lower case.

**F\_name\_api** Root name that is used to create various names in the Fortran API. Controlled by the **F\_API\_case** option with values *lower*, *upper*, *underscore* or *preserve*. Used with options **templates**, **F\_C\_name\_template**, **F\_name\_impl\_template**, **F\_name\_function\_template**, **F\_name\_generic\_template**, **F\_abstract\_interface\_subprogram\_template**, **F\_derived\_name\_template**, **F\_typedef\_name\_template**.

**F\_name\_scope** Underscore delimited name of namespace, class, enumeration. Used with creating names in Fortran. Ends with trailing underscore to allow the next scope to be appended. Does not include toplevel *namespace*.

This is a computed using *F\_name\_api* and should not be set explicitly.

**F\_impl\_filename** Name of generated Fortran implementation file for the library. Defaulted from expansion of option *F\_impl\_filename\_library\_template*.

**F\_result** The name of the Fortran wrapper's result variable. It must not be the same as any of the routines arguments. It defaults to *SHT\_rv* (Shroud temporary return value).

**F\_result\_ptr** The name of the variable used with api *captr* for the function result for arguments which create a shadow type. Defaults to *SHT\_ptrv*, pointer to return value. Used by option *C\_shadow\_result*.

**F\_string\_result\_as\_arg** The name of the output argument. Function which return a `char *` will instead be converted to a subroutine which require an additional argument for the result. See also *C\_string\_result\_as\_arg*.

**F\_this** Name of the Fortran argument which is the derived type which represents a C++ class. It must not be the same as any of the routines arguments. Defaults to *obj*.

**file\_scope** Used in filename creation to identify library, namespace, class.

**library** The value of global field *library*.

**library\_lower** Lowercase version of *library*.

**library\_upper** Uppercase version of *library*.

**LUA\_header\_filename\_suffix** Suffix added to Lua header files. Defaults to *h*. Other useful values might be *hh* or *hxx*.

**LUA\_impl\_filename\_suffix** Suffix added to Lua implementation files. Defaults to `cpp`. Other useful values might be `cc` or `cxx`.

**LUA\_module\_name** Name of Lua module for library. `{library_lower}`

**LUA\_name\_api** Root name that is used to create various names in the Lua API. Defaulted by the **LUA\_API\_case** option with values *lower*, *upper*, *underscore*, or *preserve*. If set explicitly then **LUA\_API\_case** will have no effect.

**LUA\_prefix** Prefix added to Lua wrapper functions.

**LUA\_result** The name of the Lua wrapper's result variable. It defaults to *rv* (return value).

**LUA\_state\_var** Name of argument in Lua wrapper functions for `lua_State` pointer.

**namespace\_scope** The current C++ namespace delimited with `::` and a trailing `::`. Used when referencing identifiers: `{namespace_scope}id`.

**nullptr** Set to `NULL` or `nullptr` based on option *CXX\_standard*. Always `NULL` when *language* is C.

**PY\_ARRAY\_UNIQUE\_SYMBOL** C preprocessor define used by NumPy to allow NumPy to be imported by several source files.

**PY\_header\_filename\_suffix** Suffix added to Python header files. Defaults to `h`. Other useful values might be `hh` or `hxx`.

**PY\_impl\_filename\_suffix** Suffix added to Python implementation files. Defaults to `cpp`. Other useful values might be `cc` or `cxx`.

**PY\_module\_init** Name of module and submodule initialization routine. library and namespaces delimited by `_`. Setting *PY\_module\_name* will update *PY\_module\_init*.

**PY\_module\_name** Name of generated Python module. Defaults to library name or namespace name.

**PY\_module\_scope** Name of module and submodule initialization routine. library and namespaces delimited by `..`. Setting *PY\_module\_name* will update *PY\_module\_scope*.

**PY\_name\_impl** Name of Python wrapper implementation function. Each class and namespace is implemented in its own function with file static functions. There is no need to include the class or namespace in this name. Defaults to `{PY_prefix}{function_name}{function_suffix}`.

**PY\_prefix** Prefix added to Python wrapper functions.

**PY\_result** The name of the Python wrapper's result variable. It defaults to *SHTPy\_rv* (return value). If the function returns multiple values (due to *intent(out)*) and the function result is already an object (for example, a NumPy array) then **PY\_result** will be **SHResult**.

**file\_scope** library plus any namespaces. The namespaces listed in the top level variable *namespace* is not included in the value. It is assumed that *library* will be used to generate unique names. Used in creating a filename.

**stdlib** Name of C++ standard library prefix. blank when *language=c*. `std::` when *language=c++*.

**YAML\_type\_filename** Output filename for type maps for classes.

## 21.6.2 Enumeration

**cxx\_value** Value of enum from YAML file.

`enum_lower`

`enum_name`

`enum_upper`

`enum_member_lower`



`enum_member_name`

`enum_member_upper`

**flat\_name** Scoped name of enumeration mapped to a legal C identifier. Scope operator `::` replaced with `_`. Used with *C\_enum\_template*.

**C\_enum\_member** C name for enum member. Computed from option *C\_enum\_member\_template*.

**C\_value** Evaluated value of enumeration. If the enum does not have an explicit value, it will not be present.

**C\_scope\_name** Set to *flat\_name* with a trailing underscore. Except for non-scoped enumerations in which case it is blank. Used with *C\_enum\_member\_template*. Does not include the enum name in member names for non-scoped enumerations.

**F\_scope\_name** Value of *C\_scope\_name* converted to lower case. Used with *F\_enum\_member\_template*.

**F\_enum\_member** Fortran name for enum member. Computed from option *F\_enum\_member\_template*.

**F\_value** Evaluated value of enumeration. If the enum does not have an explicit value, it is the previous value plus one.

### 21.6.3 Class

**C\_header\_filename** Name of generated header file for the class. Defaulted from expansion of option *C\_header\_filename\_class\_template*.

**C\_impl\_file** Name of generated C++ implementation file for the library. Defaulted from expansion of option *C\_impl\_filename\_class\_template*.

**F\_derived\_name** Name of Fortran derived type for this class. Computed from option *F\_derived\_name\_template*.

**F\_name\_assign** Name of method that controls assignment of shadow types. Used to help with reference counting.

**F\_name\_associated** Name of method to report if shadow type is associated. If the name is blank, no function is generated.

**F\_name\_final** Name of function used in `FINAL` for a class.

**F\_name\_instance\_get** Name of method to get `type(C_PTR)` instance pointer from wrapped class. Defaults to *get\_instance*. If the name is blank, no function is generated.

**F\_name\_instance\_set** Name of method to set `type(C_PTR)` instance pointer in wrapped class. Defaults to *set\_instance*. If the name is blank, no function is generated.

**cxx\_class** The name of the C++ class from the YAML input file. Used in generating names for C and Fortran and filenames. When the class is templated, it will be converted to a legal identifier by adding the *template\_suffix* or a sequence number.

When *cxx\_class* is set in the YAML file for a class, its value will be used in *class\_scope*, *C\_name\_scope*, *F\_name\_scope* and *F\_derived\_name*.

**cxx\_type** The namespace qualified name of the C++ class, including information from *template\_arguments*, ex. `std::vector<int>`. Same as *cxx\_class* if *template\_arguments* is not defined. Used in generating C++ code.

**class\_scope** Used to to access class static functions. Blank when not in a class. `{cxx_class}::`

**C\_prefix** Prefix for C wrapper functions. The prefix helps to ensure unique global names. Defaults to the first three letters of *library\_upper*.

**PY\_helper\_prefix** Prefix added to helper functions for the Python wrapper. This allows the helper functions to have names which will not conflict with any wrapped routines. When option *PY\_write\_helper\_in\_util* is *True*, *C\_prefix* will be prefixed to the value to ensure the helper functions will not conflict with any routines in other wrapped libraries.



**PY\_type\_obj** Name variable which points to C or C++ memory. Defaults to *obj*.

**PY\_type\_dtor** Pointer to information used to release memory.

**PY\_PyTypeObject** Name of *PyTypeObject* variable for a C++ class. Computed from option *PY\_PyTypeObject*.

**PY\_PyTypeObject\_base** The name of *PyTypeObject* variable for base class of C++ class. Only single inheritance is support via the *tp\_base* field of *PyTypeObject* struct.

## 21.6.4 Function

**C\_call\_list** Comma delimited list of function arguments.

**C\_name** Name of the C wrapper function. Defaults to evaluation of option *C\_name\_template*.

**C\_prototype** C prototype for the function. This will include any arguments required by annotations or options, such as length or **F\_string\_result\_as\_arg**.

**C\_return\_type** Return type of the C wrapper function. If the **return\_this** field is true, then set to `void`.  
Set to function's return type.

**CXX\_template** The template component of the function declaration. `<{type}>`

**CXX\_this\_call** How to call the function. `{CXX_this}->` for instance methods and blank for library functions.

**F\_arg\_c\_call** Comma delimited arguments to call C function from Fortran.

**F\_arguments** Set from option *F\_arguments* or generated from YAML decl.

**F\_C\_arguments** Argument names to the `bind(C)` interface for the subprogram. Arguments are tab delimited to aid in creating continuations.

**F\_C\_call** The name of the C function to call. Usually *F\_C\_name*, but it may be different if calling a generated routine. This can be done for functions with string arguments.

**F\_C\_name** Name of the Fortran `BIND(C)` interface for a C function. Defaults to the lower case version of *F\_C\_name\_template*.

**F\_C\_pure\_clause** TODO

**F\_C\_result\_clause** Result clause for the `bind(C)` interface.

**F\_C\_subprogram** subroutine or function for the `bind(C)` interface. The C wrapper function may be different Fortran wrapper function since some function results may be converted into arguments.

**F\_C\_var** Name of dummy argument in the `bind(C)` interface.

**F\_pure\_clause** For non-void function, pure if the *pure* annotation is added or the function is `const` and all arguments are `intent(in)`.

**F\_name\_function** The name of the *F\_name\_impl* subprogram when used as a type procedure. Defaults to evaluation of option *F\_name\_function\_template*.

**F\_name\_generic** Defaults to evaluation of option *F\_name\_generic\_template*.

**F\_name\_impl** Name of the Fortran implementation function. Defaults to evaluation of option *F\_name\_impl\_template*.

**F\_result\_clause** `“ result({F_result}) “` for functions. Blank for subroutines.

**f\_c\_suffix** Set by Shroud to allow the Fortran wrapper to call a C wrapper with additional mangling. Usually set to the value of *C\_bufferify\_suffix* or *C\_cfi\_suffix*.

**function\_name** Name of function in the YAML file.

**function\_suffix** String append to a generated function name. Useful to distinguish overloaded function and functions with default arguments. Defaults to a sequence number with a leading underscore (e.g. `_0`, `_1`, ...) but can be set by using the function field `function_suffix`. Multiple suffixes may be applied – overloaded with default arguments.

**LUA\_name** Name of function as known by LUA. Defaults to evaluation of option `LUA_name_template`.

**template\_suffix** String which is append to the end of a generated function names to distinguish template instatiations. Default values generated by Shroud will include a leading underscore. i.e `_int` or `_0`.

**underscore\_name** `function_name` converted from CamelCase to snake\_case.

## 21.6.5 Argument

`c_array_shape`

`c_array_size`

**c\_array\_size2** The *dimension* attributes multiplied together.

**c\_char\_len** The value of the *len* attribute. It will be evaluated in the C wrapper. Defaults to 0 to indicate no length given.

**c\_blanknull** Used as argument to `ShroudStrAlloc` to determine if a blank string, trimmed length is 0, should be a NULL pointer instead of an empty C string – `'\0'`. Set via attribute `+blanknull` on a `const char *` argument. Should be 0 or 1.

**c\_const** `const` if argument has the *const* attribute.

**c\_deref** Used to dereference `c_var`. `*` if it is a pointer, else blank.

**c\_var** The C name of the argument.

**c\_var\_cdesc** Name of variable of type ...

`c_var_cdesc2`

`c_var_extents`

`c_var_lower`

**chelper\_\*** Helper name for a function. Each name in statements `c_helper` will create a format name which starts with `chelper_` and end with the helper name. It will contain the name of the C function for the helper. Used by statements `c_pre_call` and `c_post_call` statements.

**cxx\_addr** Syntax to take address of argument. `&` or blank.

**cxx\_nonconst\_ptr** A non-const pointer to `cxx_addr` using `const_cast` in C++ or a cast for C.

**cxx\_member** Syntax to access members of `cxx_var`. If `cxx_local_var` is *object*, then set to `.`; if *pointer*, then set to `->`.

**cxx\_T** The template parameters for templated arguments. `std::vector<{cxx_T}>`

**cxx\_type** The C++ type of the argument.

**cxx\_var** Name of the C++ variable.

**size\_var** Name of variable which holds the size of an array in the Python wrapper.

### fmtc

Format strings used with C wrappers. Set for each argument.

**fmtf**

Format strings used with Fortran wrappers. Set for each argument.

**c\_var** The name of the argument passed to the C wrapper. This is initially the same as *f\_var* but when the statement field *c\_local\_var* is true, another name will be generated of the form `SH_{f_var}`. A declaration will also be added using `typemap.f_c_type`.

**default\_value** The value of a C++ default value argument.

**f\_array\_allocate** Fortran shape expression used with `ALLOCATE` statement when *dimension* attribute is set. For example, attribute `+dimension(10)` will create `(10)`.

**f\_array\_shape** Shape of array for use with `c_f_pointer`. For example, attribute `+dimension(10)` will create `“,tSHT_rv_cdesc%shape(1:1)“`. The leading comma is used since scalar will not add a `SHAPE` argument to `c_f_pointer`.

**f\_assumed\_shape** Set when *rank* attribute is set to the corresponding shape. `rank=1` sets to `(:)`, `rank=2` sets to `(:, :)`, etc. May also be set to `(. .)` when attribute `+dimension(..)` is used and option *F\_CFI* is True.

**f\_c\_module\_line** `Typemap.f_c_module` in a format usable by `FStmts.f_module_line`. The dictionary is converted into the string.

**f\_capsule\_data\_type** The name of the derived type used to share memory information with C or C++. *F\_capsule\_data\_type* for the argument type.

**f\_cdesc\_shape** Used to assign the rank of a Fortran variable to a `cdesc` variable. It will be blank for a scalar. ex: `\nSHT_arg_cdesc%shape(1:1) = shape(arg)`

**f\_char\_len** Defaults to `:` for deferred length used with allocatable variables. Used in statements as `character({f_char_len})`.

**f\_char\_type** Character type used in `ALLOCATE` statements. Based on *len* attributes. Defaults to blank for `CHARACTER` types which have an explicit length in the type declaration - `CHARACTER(20)`.. Otherwise set to `character(len={c_var_cdesc}%elem_len) :: ``` which uses the length computed by the C wrapper and stored in `elem_len`. For example, find the maximum length of strings in a ```char **` argument. Used in statements as `allocate({f_char_type}(f_var))`.

`f_declare_shape_prefix`

`f_declare_shape_array`

`f_get_shape_array`

**f\_kind** Value from `typemap`. ex `C_INT`. Can be used in `CStmts.f_module_line`.

`f_pointer_shape`

`f_shape_var`

**f\_type** Value from `typemap`. ex `integer(C_INT)`.

**f\_var** Fortran variable name for argument.

**fhelper\_\*** Helper name for a function. Each name in statements *f\_helper* will create a format name which starts with *fhelper\_* and end with the helper name. It will contain the name of the Fortran function for the helper. Used by statements *f\_pre\_call* and *f\_post\_call* statements.

**i\_dimension** Dimension used in `bind(C)` interface. May be assumed-size, `(*)` or assumed-rank, `(. .)`.

**i\_module\_line** Used with Fortran interface.

**i\_type** Used with Fortran interface.

**size** Expression to compute size of array argument using `SIZE` intrinsic.

### **fmtl**

Format strings used with Lua wrappers.

### **fmltpy**

Format strings used with Python wrappers.

**array\_size** Dimensions multiplied together. `dimension(2,3)` creates  $(2) * (3)$ .

**rank** Attribute value for *rank*.

## 21.6.6 Result

**cxx\_rv\_decl** Declaration of variable to hold return value for function.

## 21.6.7 Variable

**PY\_struct\_context** Prefix used to to access struct/class variables. Includes trailing syntax to access member in a struct i.e. `.` or `->`. `self->obj->`.

## 21.7 Types Map

Types describe how to handle arguments from Fortran to C to C++. Then how to convert return values from C++ to C to Fortran.

Since Fortran 2003 (ISO/IEC 1539-1:2004(E)) there is a standardized way to generate procedure and derived-type declarations and global variables which are interoperable with C (ISO/IEC 9899:1999). The `bind(C)` attribute has been added to inform the compiler that a symbol shall be interoperable with C; also, some constraints are added. Note, however, that not all C features have a Fortran equivalent or vice versa. For instance, neither C's unsigned integers nor C's functions with variable number of arguments have an equivalent in Fortran.<sup>1</sup>

**forward** Forward declaration. Defaults to *None*.

**typedef** Initialize from existing type Defaults to *None*.

**f\_return\_code** Fortran code used to call function and assign the return value. Defaults to *None*.

**f\_to\_c** Expression to convert Fortran type to C type. If this field is set, it will be used before `f_cast`. Defaults to *None*.

## 21.8 Doxygen

Used to insert directives for doxygen for a function.

**brief** Brief description.

**description** Full description.

**return** Description of return value.

---

<sup>1</sup> <https://gcc.gnu.org/onlinedocs/gfortran/Interoperability-with-C.html>

## 21.9 Patterns

**C\_error\_pattern** Inserted after the call to the C++ function in the C wrapper. Format is evaluated in the context of the result argument. *c\_var*, *c\_var\_len* refer to the result argument.

**C\_error\_pattern\_buf** Inserted after the call to the C++ function in the buffer C wrapper for functions with string arguments. Format is evaluated in the context of the result argument.

**PY\_error\_pattern** Inserted into Python wrapper.



Notes to help migrate between releases.

## 22.1 Unreleased

- Rename some fields in Statements to allow C and Fortran entries to exist in the same group by consistently using a `c_`, `i_` or `f_` prefix. This allows a single group to contains all the fields used for more complex conversions making it easier to follow the flow.

This will change the name of fields in *fstatements* in an input YAML file. These are used to changed the default behavior of a wrapper.

```
- decl: void vector_iota_out_with_num(std::vector<int> &arg+intent(out))
fstatements:
  c:
    c_return_type: long
    c_return:
      - return SHT_arg_cdesc->size;
```

Table 1: c statements

Old Name	New Name
arg_call	c_arg_call
pre_call	c_pre_call
call	c_call
post_call	c_post_call
final	c_final
ret	c_return
temps	c_temps
local	c_local
f_arg_decl	i_arg_decl
f_result_decl	i_result_decl
f_result_var	i_result_var
f_module	i_module
f_import	i_import

Table 2: f statements

Old Name	New Name
need_wrapper	f_need_wrapper
arg_name	f_arg_name
arg_decl	f_arg_decl
arg_c_call	f_arg_call
declare	f_declare
pre_call	f_pre_call
call	f_call
post_call	f_post_call
result	f_result
temps	f_temps
local	f_local

- Added format field *f\_c\_suffix*. Used in format fields *\*\*C\_name\_template\** and *F\_C\_name\_template\** to allow Fortran wrapper to call a C function with additional mangling such as *\*\*C\_cfi\_suffix* and *C\_bufferify\_suffix*. Previously this was appended directly to format field *\*function\_suffix*. If *\*\*C\_name\_template\** or *F\_C\_name\_template\** are explicitly set in the *YAML file* then *\*f\_c\_suffix* should be included in the value.
- Renamed format fields *hnamefunc*. These fields were added from the statement fields **c\_helper** and **f\_helper**, each a blank delimited list of names. A format field was added for each name with a 0-based suffix corresponding to the position in the list. Now, the format fields have the prefix of *chelper\_* or *fhelper\_* followed by the helpers name. For example, *fhelper\_copy\_array*. This makes it easier to match the corresponding helper and will help when using statement mixin groups since the order of names will no longer matter.

## 22.2 v0.13.0

### 22.2.1 Changes

- Some generated wrapper names have been changed to be more consistent. Added format field *F\_name\_api*. It is controlled by option **F\_API\_case** which may be set to *lower*, *upper*, *underscore* or *preserve*. Uses of format field *underscore\_name* should be changed to *F\_name\_api*. It's often used in name options such as **F\_name\_impl\_template** and **F\_name\_generic\_template**.



Likewise, C API names are controlled by option **C\_name\_api**. The default is *preserve*. The previous behavior can be restored by setting option **C\_API\_case** to *underscore*.

**F\_API\_case** defaults to *underscore* since Fortran is case insensitive. **F\_C\_case** defaults to *preserve* to make the C API closer to the C++ API.

- Changed the name of C and Python function splicer to use *function\_name* instead of *underscore\_name* to correspond to C++ library names.
- The *C\_memory\_dtor\_function* is now written to the utility file, *C\_impl\_utility*. This function contains code to delete memory from shadow classes. Previously it was written to file *C\_impl\_filename*. In addition, some helper functions are also written into this file. This may require changes to Makefiles to ensure this file is compiled.
- A single capsule derived type is created in the Fortran wrapper instead of one per class. This is considered an implementation detail and a user of the wrapper will not access them directly. However, it may show up in splicer code. It is used to pass values from the Fortran wrapper to the C wrapper. The old type names may of been referenced in explicit splicer code. In that case the name will need to be changed. The format field *F\_capsule\_data\_type\_class* is replaced by *F\_capsule\_data\_type*. The C wrapper continues to create a capsule struct for each class as a form of type safety in the C API.
- Class instance arguments which are passed by value will now pass the shadow type by reference. This allows the *addr* and *idtor* fields to be changed if necessary by the C wrapper.
- Replaced the *additional\_interfaces* splicer with *additional\_declarations*. This new splicer is outside of an interface block and can be used to add a generic interface that could not be added to *additional\_interfaces*. Existing *additional\_interfaces* splicers can be converted to *additional\_declarations* by wrapping the splicer with `INTERFACE/END INTERFACE`.

## 22.2.2 New Features

- Added support for C++ class inheritance. See [Class Inheritance](#)
- Added the ability to treat a struct as a class. See [Object-oriented C](#)
- Added the ability to declare members of a struct on individual `decl` lines in the YAML file similar to how class members are defined. Before the struct was defined in a single `decl` : .
- Allow structs to be templated.
- Added the ability to declare variables using the `enum` keyword. C++ creates a type for each enumeration.
- Generate generic interface which allows a scalar or array to be passed for an argument.
- Process assumed-rank dimension attribute, *dimension(..)*. Create a generic interface using scalar and each rank.
- Added some support for Futher Interoperability with C. Used when option *F\_CFI* is True (C/Fortran Interoperability).
- Support *deref(pointer)* for `char *` and `std::string` functions. Requires at least gfortran 6.1.0
- Added option *F\_trim\_char\_in*. Controls where CHARACTER arguments are NULL terminated. If *True* then terminated in Fortran else in C.
- Added attribute *+blanknull* to convert a blank Fortran string into a NULL pointer instead of a 1-d buffer with `'/0'`. Used with `const char *` arguments. This can be defaulted to True with the *F\_blanknull* option.
- Added *file\_code* dictionary to input YAML file. It contains directives to add header file and USE statements into generated files. These are collated with headers and USE statements added by typemaps, statements and helpers to avoid duplication.
- Allow typemaps with *base* as *integer* and *real* to be added to the input YAML file. This allows kind parameters to be defined via splicers then used by a typemap. i.e. `integer (INDEXTYPE)`

- Added option `C_shadow_result`. If true, the C wrapper will return a pointer to the capsule holding the function result. The capsule is also passed as an argument. If false the function is `void`.
- The getter for a class member function will return a Fortran pointer if the `dimension` attribute is added to the declaration. Likewise, the setter will expect an array of the same rank as `dimension`. Getter and setters will also be generated for struct fields which are pointers to native types. Option `F_struct_getter_setter` can be used to control their creation.
- Added ability to add `splicer` to `typedef` declarations. For example, to use the C preprocessor to set the type of the typedef. See `typedefs.yaml` for an example.
- Added support for out arguments which return a reference to a `std::vector` or pointer to an array of `std::string`.
- Create C and Fortran wrappers for `typedef` statements. Before `typedef` was treated as an alias. `typedef int TypeID` would substitute `integer(C_INT)` for every use of `TypeID` in the Fortran wrapper. Now a parameter is created: `integer, parameter :: type_id = C_INT`. Used as: `integer(type_id) :: arg`.

### 22.2.3 Fixed

- Order of header files in `cxx_header` is preserved in the generated code.
- Create a generic interface even if only one `decl` is in the `fortran_generic` list.
- `generic_function` now creates a C wrapper for each Fortran wrapper. This causes each Fortran interface to bind to a different C function which fixes a compile error with `xlF`.
- Add generic interfaces for class methods. Generic functions were only being added to the type-bound procedures. `class_generic(obj)` now works instead of only `obj%generic()`.
- Add continuations on Fortran `IMPORT` statements.
- Support an array of pointers - `void **addr+rank(1)`.
- Fix Fortran wrapper for `intent(INOUT)` for `void **`.
- Promote wrap options (ex `wrap_fortran`) up to container when True (library, class, namespace). This allows `wrap_fortran` to be False at the global level and set True on a function and get a wrapper. Before a False at the global level would never attempt to do any wrapping.
- Better support for `std::vector` with pointer template arguments. For examples, `<const double *>`.
- Parse `class`, `struct` and `enum` as part of declaration. This allows `typedef struct tag name` to be parsed properly.
- Create type table earlier in parse. This allows recursive structs such as `struct point { struct point *next; }` to be parsed.
- Fixed issues in converting function names from CamelCase
  - Remove redundant underscore `Create_Cstruct_as_class` was `c_create__cstruct_as_class` now `c_create_cstruct_as_class`
  - Add missing underscore `AFunction` was `afunction` now `a_function`.

---

### Fortran Previous Work

---

Communicating between languages has a long history.

#### 23.1 Babel

<https://computation.llnl.gov/projects/babel-high-performance-language-interoperability> Babel parses a SIDL (Scientific Interface Definition Language) file to generate source. It is a hub-and-spokes approach where each language it supports is mapped to a Babel runtime object. The last release was 2012-01-06. [http://en.wikipedia.org/wiki/Babel\\_Middleware](http://en.wikipedia.org/wiki/Babel_Middleware)

#### 23.2 Cfortran.h

Used with Fortran 77 and C.

- <https://www-zeus.desy.de/~burow/cfortran/>
- <https://cfortran.sourceforge.net/>

#### 23.3 Chasm

<http://chasm-interop.sourceforge.net/> - This page is dated July 13, 2005

Chasm is a tool to improve C++ and Fortran 90 interoperability. Chasm parses Fortran 90 source code and automatically generates C++ bridging code that can be used in C++ programs to make calls to Fortran routines. It also automatically generates C structs that provide a bridge to Fortran derived types. Chasm supplies a C++ array descriptor class which provides an interface between C and F90 arrays. This allows arrays to be created in one language and then passed to and used by the other language. <http://www.cs.uoregon.edu/research/pdt/users.php>

- **CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability**  
C.E. Rasmussen, K.A. Lindlan, B. Mohr, J. Striegnitz

- Bridging the language gap in scientific computing: the Chasm approach C. E. Rasmussen, M. J. Sottile, S. S. Shende, A. D. Malony (2005)

## 23.4 wrap

<https://github.com/scalability-llnl/wrap>

a PMPI wrapper generator

## 23.5 Trilinos

<http://trilinos.org/>

Trilinos wraps C++ with C, then the Fortran over the C. Described in the book Scientific Software Design. <http://www.amazon.com/Scientific-Software-Design-The-Object-Oriented/dp/0521888131>

- On the object-oriented design of reference-counted shadow objects Karla Morris, Damian W.I. Rouson, Jim Xia (2011)
- This Isn't Your Parents' Fortran: Managing C++ Objects with Modern Fortran Damian Rouson, Karla Morris, Jim Xia (2012)

Directory packages/ForTrilinos/src/skeleton has a basic template which must be edited to create a wrapper for a class.

Exascale Programming: Adapting What We Have Can (and Must) Work

In 2009 and 2010, the C++ based Trilinos project developed Fortran interface capabilities, called ForTrilinos. As an object-oriented (OO) collection of libraries, we assumed that the OO features of Fortran 2003 would provide us with natural mappings of Trilinos classes into Fortran equivalents. Over the two-year span of the ForTrilinos effort, we discovered that compiler support for 2003 features was very immature. ForTrilinos developers quickly came to know the handful of compiler developers who worked on these features and, despite close collaboration with them to complete and stabilize the implementation of Fortran 2003 features (in 2010), ForTrilinos stalled and is no longer developed.

<http://www.hpcwire.com/2016/01/14/24151/>

<https://github.com/Trilinos/ForTrilinos> <https://www.researchgate.net/project/ForTrilinos>

This is the new effort to provide Fortran interfaces to Trilinos through automatic code generation using SWIG. The previous effort (ca. 2008-2012) can be obtained by downloading Trilinos releases prior to 12.12.

[https://trilinos.github.io/ForTrilinos/files/ForTrilinos\\_Design\\_Document.pdf](https://trilinos.github.io/ForTrilinos/files/ForTrilinos_Design_Document.pdf)

## 23.6 SWIG

The custom version of swig available at <https://github.com/swig-fortran/swig>

- <https://github.com/swig-fortran/flibcpp>
- [http://www.icl.utk.edu/~luszczyk/conf/2019/siam\\_cse/siam-cse-johnsonsr.pdf](http://www.icl.utk.edu/~luszczyk/conf/2019/siam_cse/siam-cse-johnsonsr.pdf)
- <https://info.ornl.gov/sites/publications/Files/Pub127965.pdf>
- Documenting automated Fortran-C++ bindings with SWIG
- IDEAS-ECP Webinar: Automated Fortran-C++ Bindings for Large-Scale Scientific Applications

## 23.7 MPICH

MPICH uses a custom perl scripts which has routine names and types in the source.

[http://git.mpich.org/mpich.git/blob/HEAD:/src/binding/fortran/use\\_mpi/buildiface](http://git.mpich.org/mpich.git/blob/HEAD:/src/binding/fortran/use_mpi/buildiface)

## 23.8 GTK

gtk-fortran uses a python script which parse the C header files to generate the Fortran.

<https://github.com/jerryd/gtk-fortran/blob/master/src/cfwrapper.py> <https://github.com/vmagnin/gtk-fortran/wiki> <https://github.com/vmagnin/gtk-fortran/wiki/How-gtk-fortran-is-generated>

## 23.9 CDI

CDI is a C and Fortran Interface to access Climate and NWP model Data. <https://code.zmaw.de/projects/cdi>

“One part of CDI[1] is a such generator. It still has some rough edges and we haven’t yet decided what to do about functions returning char \* (it seems like that will need some wrapping unless we simply return TYPE(c\_ptr) and let the caller deal with that) but if you’d like to have a starting point in Ruby try interfaces/f2003/bindGen.rb from the tarball you can download” <https://groups.google.com/d/msg/comp.lang.fortran/oadwd3HHtGA/J8DD8kGeVw8J>

## 23.10 Forpy

This is a Fortran interface over the Python API written using the metaprogramming tool Fypp.

- Forpy: A library for Fortran-Python interoperability
- Fypp — Python powered Fortran metaprogramming

## 23.11 CNF

<http://www.starlink.ac.uk/docs/sun209.htx/sun209.html>

The CNF package comprises two sets of software which ease the task of writing portable programs in a mixture of FORTRAN and C. F77 is a set of C macros for handling the FORTRAN/C subroutine linkage in a portable way, and CNF is a set of functions to handle the difference between FORTRAN and C character strings, logical values and pointers to dynamically allocated memory.

## 23.12 h2m-AutoFortran

<https://github.com/Kaiveria/h2m-Autofortran-Tool>

The h2m-AutoFortran tool is designed to allow easy calls to C routines from Fortran programs. Given a header file in standard C, h2m will produce a Fortran module providing function interfaces which maintain interoperability with C. Features for which there are no Fortran equivalents will not be translated and warnings will be written to standard error. The h2m-AutoFortran tool is built into Clang, the LLVM C compiler. During translation, the Clang abstract syntax tree (AST) is used to assemble information about the header file.

## 23.13 Links

- [Technical Specification ISO/IEC TS 29113:2012](#)
- [Generating C Interfaces](#)
- [Shadow-object interface between Fortran95 and C++](#) Mark G. Gray, Randy M. Roberts, and Tom M. Evans (1999)
- [Generate C interface from C++ source code using Clang libtooling](#)
- [Memory leaks in derived types revisited](#) G. W. Stewart (2003)
- [A General Approach to Creating Fortran Interface for C++ Application Libraries](#)
- [C, Fortran, and single-character strings](#)
- [Emulating Multiple Inheritance in Fortran 2003/2008](#)

There are several available tools to creating a Python interface to a C or C++ library.

## 24.1 Ctypes

- <http://docs.python.org/lib/module-ctypes.html>

### 24.1.1 Pros

- No need for compiler.

### 24.1.2 Cons

- Difficult wrapping C++ due to mangling and object ABI.

## 24.2 SWIG

- <http://www.swig.org/>

## 24.3 PyBindgen

- <https://github.com/gjcarneiro/pybindgen>
- <http://pybindgen.readthedocs.io/en/latest/>

## 24.4 Cython

- <http://cython.org>
- <https://cython.readthedocs.io/en/latest/>

<http://blog.kevmod.com/2020/05/python-performance-its-not-just-the-interpreter/>

I ran Cython (a Python->C converter) on the previous benchmark, and it runs in exactly the same amount of time: 2.11s. I wrote a simplified C extension in 36 lines compared to Cython's 3600, and it too runs in 2.11s.

## 24.5 SIP

Sip was developed to create PyQt.

- <https://www.riverbankcomputing.com/software/sip/intro>

## 24.6 Shiboken

Shiboken was developed to create PySide.

- [https://wiki.qt.io/Qt\\_for\\_Python](https://wiki.qt.io/Qt_for_Python)
- <http://doc.qt.io/qtforpython/shiboken2/contents.html>

## 24.7 Boost Python

- [https://www.boost.org/doc/libs/1\\_66\\_0/libs/python/doc/html/index.html](https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html)

## 24.8 Pybind11

- <https://github.com/pybind/pybind11>
- <https://pybind11.readthedocs.io/en/stable/>

## 24.9 Links

- [Interfacing with C - Scipy lecture notes](#)
- [SciPy Cookbook](#)



- complex
- pointers to pointers and in particular `char **` are not supported. An argument like `Class **ptr+intent(out)` does not work. Instead use a function which return a pointer to `Class *`
- reference counting and garbage collection
- Support for *Further Interoperability of Fortran with C*. This includes the `ISO_Fortran_binding.h` header file.

The copying of strings solves the blank-filled vs null-terminated differences between Fortran and C and works well for many strings. However, if a large buffer is passed, it may be desirable to avoid the copy.

There is some initial work to support Python and Lua wrappers.

### 25.1 Possible Future Work

Use a tool to parse C++ and extract info.

- <https://github.com/CastXML/CastXML>
- <https://pypi.python.org/pypi/pygccxml>
- Wrapping C and C++ Libraries with CastXML | SciPy 2015 | Brad King, Bill Hoffman, Matthew McCormick <https://www.youtube.com/watch?v=O2lBgtaDdyk&index=20&list=PLYx7XA2nY5Gcpabmu61kKcToLz0FapmHu>



---

## Sample Fortran Wrappers

---

This chapter gives details of the generated code. It's intended for users who want to understand the details of how the wrappers are created.

All of these examples are derived from tests in the `regression` directory.

### 26.1 No Arguments

C library function in `clibrary.c`:

```
void NoReturnNoArguments(void)
{
    strncpy(last_function_called, "Function1", MAXLAST);
    return;
}
```

`clibrary.yaml`:

```
- decl: void NoReturnNoArguments()
```

Fortran calls C via the following interface:

```
interface
  subroutine no_return_no_arguments() &
    bind(C, name="NoReturnNoArguments")
    implicit none
  end subroutine no_return_no_arguments
end interface
```

If wrapping a C++ library, a function with a C API will be created that Fortran can call.

```
void TUT_NoReturnNoArguments(void)
{
```

(continues on next page)

(continued from previous page)

```
// splicer begin function.NoReturnNoArguments
tutorial::NoReturnNoArguments();
// splicer end function.NoReturnNoArguments
}
```

Fortran usage:

```
use tutorial_mod
call no_return_no_arguments
```

The C++ usage is similar:

```
#include "tutorial.hpp"

tutorial::NoReturnNoArguments();
```

## 26.2 Numeric Types

### 26.2.1 PassByValue

C library function in `clibrary.c`:

```
double PassByValue(double arg1, int arg2)
{
    strncpy(last_function_called, "PassByValue", MAXLAST);
    return arg1 + arg2;
}
```

`clibrary.yaml`:

```
- decl: double PassByValue(double arg1, int arg2)
```

Both types are supported directly by the `iso_c_binding` module so there is no need for a Fortran function. The C function can be called directly by the Fortran interface using the `bind(C)` keyword.

Fortran calls C via the following interface:

```
interface
    function pass_by_value(arg1, arg2) &
        result(SHT_rv) &
        bind(C, name="PassByValue")
    use iso_c_binding, only : C_DOUBLE, C_INT
    implicit none
    real(C_DOUBLE), value, intent(IN) :: arg1
    integer(C_INT), value, intent(IN) :: arg2
    real(C_DOUBLE) :: SHT_rv
end function pass_by_value
end interface
```

Fortran usage:

```
real(C_DOUBLE) :: rv_double
rv_double = pass_by_value(1.d0, 4)
call assert_true(rv_double == 5.d0)
```

## 26.2.2 PassByReference

C library function in `clibrary.c`:

```
void PassByReference(double *arg1, int *arg2)
{
    strncpy(last_function_called, "PassByReference", MAXLAST);
    *arg2 = *arg1;
}
```

`clibrary.yaml`:

```
- decl: void PassByReference(double *arg1+intent(in), int *arg2+intent(out))
```

Fortran calls C via the following interface:

```
interface
  subroutine pass_by_reference(arg1, arg2) &
    bind(C, name="PassByReference")
    use iso_c_binding, only : C_DOUBLE, C_INT
    implicit none
    real(C_DOUBLE), intent(IN) :: arg1
    integer(C_INT), intent(OUT) :: arg2
  end subroutine pass_by_reference
end interface
```

Example usage:

```
integer(C_INT) var
call pass_by_reference(3.14d0, var)
call assert_equals(3, var)
```

## 26.2.3 Sum

C++ library function from `pointers.cpp`:

```
void Sum(int len, const int *values, int *result)
{
    int sum = 0;
    for (int i=0; i < len; i++) {
        sum += values[i];
    }
    *result = sum;
    return;
}
```

`pointers.yaml`:

```
- decl: void Sum(int len +implied(size(values)),
                int *values +rank(1)+intent(in),
                int *result +intent(out))
```

The POI prefix to the function names is derived from the format field *C\_prefix* which defaults to the first three letters of the *library* field, in this case *pointers*. This is a C++ file which provides a C API via `extern "C"`.

`wrappointers.cpp`:

```
int POI_sumFixedArray(void)
{
    // splicer begin function.sumFixedArray
    int SHC_rv = sumFixedArray();
    return SHC_rv;
    // splicer end function.sumFixedArray
}
```

Fortran calls C via the following interface:

```
interface
    subroutine c_sum(len, values, result) &
        bind(C, name="POI_Sum")
        use iso_c_binding, only : C_INT
        implicit none
        integer(C_INT), value, intent(IN) :: len
        integer(C_INT), intent(IN) :: values(*)
        integer(C_INT), intent(OUT) :: result
    end subroutine c_sum
end interface
```

The Fortran wrapper:

```
interface
    function sum_fixed_array() &
        result(SHT_rv) &
        bind(C, name="POI_sumFixedArray")
        use iso_c_binding, only : C_INT
        implicit none
        integer(C_INT) :: SHT_rv
    end function sum_fixed_array
end interface
```

Example usage:

```
integer(C_INT) rv_int
call sum([1,2,3,4,5], rv_int)
call assert_true(rv_int .eq. 15, "sum")
```

## 26.2.4 truncate\_to\_int

Sometimes it is more convenient to have the wrapper allocate an `intent(out)` array before passing it to the C++ function. This can be accomplished by adding the *deref(allocatable)* attribute.

C++ library function from `pointers.c`:

```
void truncate_to_int(double *in, int *out, int size)
{
    int i;
    for(i = 0; i < size; i++) {
        out[i] = in[i];
    }
}
```

`pointers.yaml`:

```
- decl: void truncate_to_int(double * in      +intent(in)  +rank(1),
                           int *   out      +intent(out)
                           +deref(allocatable)+dimension(size(in)),
                           int      sizein  +implied(size(in))
```

Fortran calls C via the following interface:

```
interface
  subroutine c_truncate_to_int(in, out, sizein) &
    bind(C, name="truncate_to_int")
    use iso_c_binding, only : C_DOUBLE, C_INT
    implicit none
    real(C_DOUBLE), intent(IN) :: in(*)
    integer(C_INT), intent(OUT) :: out(*)
    integer(C_INT), value, intent(IN) :: sizein
  end subroutine c_truncate_to_int
end interface
```

The Fortran wrapper:

```
subroutine truncate_to_int(in, out)
  use iso_c_binding, only : C_DOUBLE, C_INT
  real(C_DOUBLE), intent(IN) :: in(:)
  integer(C_INT), intent(OUT) :: out(:)
  integer(C_INT) :: SH_sizein
  ! splicer begin function.truncate_to_int
  SH_sizein = size(in,kind=C_INT)
  call c_truncate_to_int(in, out, SH_sizein)
  ! splicer end function.truncate_to_int
end subroutine truncate_to_int
```

Example usage:

```
integer(c_int), allocatable :: out_int(:)
call truncate_to_int([1.2d0, 2.3d0, 3.4d0, 4.5d0], out_int)
```

## 26.3 Numeric Pointers

### 26.3.1 getRawPtrToFixedArray

C++ library function from pointers.c:

```
void getRawPtrToFixedArray(int **count)
{
  *count = (int *) &global_fixed_array;
}
```

pointers.yaml:

```
- decl: void getRawPtrToFixedArray(int **count+intent(out)+deref(row))
```

Fortran calls C via the following interface:

```
interface
  subroutine get_raw_ptr_to_fixed_array(count) &
    bind(C, name="getRowPtrToFixedArray")
    use iso_c_binding, only : C_PTR
    implicit none
    type(C_PTR), intent(OUT) :: count
  end subroutine get_raw_ptr_to_fixed_array
end interface
```

Example usage:

```
type(C_PTR) :: cptr_array
call get_raw_ptr_to_fixed_array(cptr_array)
```

### 26.3.2 getPtrToScalar

C++ library function from pointers.c:

```
void getPtrToScalar(int **nitems)
{
    *nitems = &global_int;
}
```

pointers.yaml:

```
- decl: void getPtrToScalar(int **nitems+intent(out))
```

This is a C file which provides the bufferify function.

wrappointers.c:

```
void POI_getPtrToScalar_bufferify(POI_SHROUD_array *SHT_nitems_cdesc)
{
    // splicer begin function.getPtrToScalar_bufferify
    int *nitems;
    getPtrToScalar(&nitems);
    SHT_nitems_cdesc->cxx.addr = nitems;
    SHT_nitems_cdesc->cxx.idtor = 0;
    SHT_nitems_cdesc->addr.base = nitems;
    SHT_nitems_cdesc->type = SH_TYPE_INT;
    SHT_nitems_cdesc->elem_len = sizeof(int);
    SHT_nitems_cdesc->rank = 0;
    SHT_nitems_cdesc->size = 1;
    // splicer end function.getPtrToScalar_bufferify
}
```

Fortran calls C via the following interface:

```
interface
  subroutine c_get_ptr_to_scalar(nitems) &
    bind(C, name="getPtrToScalar")
    use iso_c_binding, only : C_PTR
    implicit none
    type(C_PTR), intent(OUT) :: nitems
  end subroutine c_get_ptr_to_scalar
end interface
```



The Fortran wrapper:

```
subroutine get_ptr_to_scalar(nitems)
  use iso_c_binding, only : C_INT, c_f_pointer
  integer(C_INT), intent(OUT), pointer :: nitems
  ! splicer begin function.get_ptr_to_scalar
  type(POI_SHROUD_array) :: SHT_nitems_cdesc
  call c_get_ptr_to_scalar_bufferify(SHT_nitems_cdesc)
  call c_f_pointer(SHT_nitems_cdesc%base_addr, nitems)
  ! splicer end function.get_ptr_to_scalar
end subroutine get_ptr_to_scalar
```

Assigning to iscalar will modify the C++ variable. Example usage:

```
integer(C_INT), pointer :: iscalar
call get_ptr_to_scalar(iscalar)
iscalar = 0
```

### 26.3.3 getPtrToDynamicArray

C++ library function from pointers.c:

```
void getPtrToDynamicArray(int **count, int *len)
{
  *count = (int *) &global_fixed_array;
  *len = sizeof(global_fixed_array)/sizeof(int);
}
```

pointers.yaml:

```
- decl: void getPtrToDynamicArray(int **count+intent(out)+dimension(ncount),
                                int *ncount+intent(out)+hidden)
```

This is a C file which provides the bufferify function.

wrappointers.c:

```
void POI_getPtrToDynamicArray_bufferify(
  POI_SHROUD_array *SHT_count_cdesc)
{
  // splicer begin function.getPtrToDynamicArray_bufferify
  int *count;
  int ncount;
  getPtrToDynamicArray(&count, &ncount);
  SHT_count_cdesc->cxx.addr = count;
  SHT_count_cdesc->cxx.idtor = 0;
  SHT_count_cdesc->addr.base = count;
  SHT_count_cdesc->type = SH_TYPE_INT;
  SHT_count_cdesc->elem_len = sizeof(int);
  SHT_count_cdesc->rank = 1;
  SHT_count_cdesc->shape[0] = ncount;
  SHT_count_cdesc->size = SHT_count_cdesc->shape[0];
  // splicer end function.getPtrToDynamicArray_bufferify
}
```

Fortran calls C via the following interface:

```

interface
  subroutine c_get_ptr_to_dynamic_array(count, ncount) &
    bind(C, name="getPtrToDynamicArray")
    use iso_c_binding, only : C_INT, C_PTR
    implicit none
    type(C_PTR), intent(OUT) :: count
    integer(C_INT), intent(OUT) :: ncount
  end subroutine c_get_ptr_to_dynamic_array
end interface

```

The Fortran wrapper:

```

subroutine get_ptr_to_dynamic_array(count)
  use iso_c_binding, only : C_INT, c_f_pointer
  integer(C_INT), intent(OUT), pointer :: count(:)
  ! splicer begin function.get_ptr_to_dynamic_array
  type(POI_SHROUD_array) :: SHT_count_cdesc
  call c_get_ptr_to_dynamic_array_bufferify(SHT_count_cdesc)
  call c_f_pointer(SHT_count_cdesc%base_addr, count, &
    SHT_count_cdesc%shape(1:1))
  ! splicer end function.get_ptr_to_dynamic_array
end subroutine get_ptr_to_dynamic_array

```

Assigning to `iarray` will modify the C++ variable. Example usage:

```

integer(C_INT), pointer :: iarray(:)
call get_ptr_to_dynamic_array(iarray)
iarray = 0

```

### 26.3.4 getRawPtrToInt2d

`global_int2d` is a two dimensional array of non-contiguous rows. C stores the address of each row. Shroud can only deal with this as a type (`C_PTR`) and expects the user to dereference the address.

C++ library function from `pointers.c`:

```

static int global_int2d_1[] = {1,2,3};
static int global_int2d_2[] = {4,5};
static int *global_int2d[] = {global_int2d_1, global_int2d_2};

void getRawPtrToInt2d(int ***arg)
{
  *arg = (int **) global_int2d;
}

```

`pointers.yaml`:

```

- decl: void getRawPtrToInt2d(int ***arg +intent(out))

```

Fortran calls C via the following interface:

```

interface
  subroutine get_raw_ptr_to_int2d(arg) &
    bind(C, name="getRawPtrToInt2d")
    use iso_c_binding, only : C_PTR
    implicit none

```

(continues on next page)

(continued from previous page)

```

    type(C_PTR), intent(OUT) :: arg
  end subroutine get_raw_ptr_to_int2d
end interface

```

Example usage:

```

type(C_PTR) :: addr
type(C_PTR), pointer :: array2d(:)
integer(C_INT), pointer :: row1(:), row2(:)
integer total

call get_raw_ptr_to_int2d(addr)

! Dereference the pointers into two 1d arrays.
call c_f_pointer(addr, array2d, [2])
call c_f_pointer(array2d(1), row1, [3])
call c_f_pointer(array2d(2), row2, [2])

total = row1(1) + row1(2) + row1(3) + row2(1) + row2(2)
call assert_equals(15, total)

```

### 26.3.5 checkInt2d

Example of using the type (C\_PTR) returned *getRawPtrToInt2d*.

pointers.yaml:

```
- decl: int checkInt2d(int **arg +intent(in))
```

Fortran calls C via the following interface. Note the use of VALUE attribute.

```

interface
  function check_int2d(arg) &
    result(SHT_rv) &
    bind(C, name="checkInt2d")
    use iso_c_binding, only : C_INT, C_PTR
    implicit none
    type(C_PTR), intent(IN), value :: arg
    integer(C_INT) :: SHT_rv
  end function check_int2d
end interface

```

Example usage:

```

type(C_PTR) :: addr
integer total

call get_raw_ptr_to_int2d(addr)
total = check_int2d(addr)
call assert_equals(15, total)

```

### 26.3.6 getMinMax

No Fortran function is created. Only an interface to a C wrapper which dereference the pointers so they can be treated as references.

C++ library function in `tutorial.cpp`:

```
void getMinMax(int &min, int &max)
{
    min = -1;
    max = 100;
}
```

`tutorial.yaml`:

```
- decl: void getMinMax(int &min +intent(out), int &max +intent(out))
```

The C wrapper:

```
void TUT_getMinMax(int * min, int * max)
{
    // splicer begin function.getMinMax
    tutorial::getMinMax(*min, *max);
    // splicer end function.getMinMax
}
```

Fortran calls C via the following interface:

```
interface
    subroutine get_min_max(min, max) &
        bind(C, name="TUT_getMinMax")
        use iso_c_binding, only : C_INT
        implicit none
        integer(C_INT), intent(OUT) :: min
        integer(C_INT), intent(OUT) :: max
    end subroutine get_min_max
end interface
```

Fortran usage:

```
call get_min_max(minout, maxout)
call assert_equals(-1, minout, "get_min_max minout")
call assert_equals(100, maxout, "get_min_max maxout")
```

### 26.3.7 returnIntPtrToScalar

`pointers.yaml`:

```
- decl: int *returnIntPtrToScalar(void)
```

Fortran calls C via the following interface:

```
interface
    function c_return_int_ptr_to_scalar() &
        result(SHT_rv) &
        bind(C, name="returnIntPtrToScalar")
```

(continues on next page)

(continued from previous page)

```

    use iso_c_binding, only : C_PTR
    implicit none
    type(C_PTR) SHT_rv
end function c_return_int_ptr_to_scalar
end interface

```

The Fortran wrapper:

```

function return_int_ptr_to_scalar() &
    result(SHT_rv)
    use iso_c_binding, only : C_INT, C_PTR, c_f_pointer
    integer(C_INT), pointer :: SHT_rv
    ! splicer begin function.return_int_ptr_to_scalar
    type(C_PTR) :: SHC_rv_ptr
    SHC_rv_ptr = c_return_int_ptr_to_scalar()
    call c_f_pointer(SHC_rv_ptr, SHT_rv)
    ! splicer end function.return_int_ptr_to_scalar
end function return_int_ptr_to_scalar

```

Example usage:

```

integer(C_INT), pointer :: irvscalar
irvscalar => return_int_ptr_to_scalar()

```

## 26.3.8 returnIntPtrToFixedArray

pointers.yaml:

```

- decl: int *returnIntPtrToFixedArray(void) +dimension(10)

```

This is a C file which provides the bufferify function.

wrappointers.c:

```

void POI_returnIntPtrToFixedArray_bufferify(
    POI_SHROUD_array *SHT_rv_cdesc)
{
    // splicer begin function.returnIntPtrToFixedArray_bufferify
    int * SHC_rv = returnIntPtrToFixedArray();
    SHT_rv_cdesc->cxx.addr = SHC_rv;
    SHT_rv_cdesc->cxx.idtor = 0;
    SHT_rv_cdesc->addr.base = SHC_rv;
    SHT_rv_cdesc->type = SH_TYPE_INT;
    SHT_rv_cdesc->elem_len = sizeof(int);
    SHT_rv_cdesc->rank = 1;
    SHT_rv_cdesc->shape[0] = 10;
    SHT_rv_cdesc->size = SHT_rv_cdesc->shape[0];
    // splicer end function.returnIntPtrToFixedArray_bufferify
}

```

Fortran calls C via the following interface:

```

interface
    subroutine c_return_int_ptr_to_fixed_array_bufferify(SHT_rv) &
        bind(C, name="POI_returnIntPtrToFixedArray_bufferify")

```

(continues on next page)

(continued from previous page)

```

import :: POI_SHROUD_array
implicit none
type(POI_SHROUD_array), intent(OUT) :: SHT_rv
end subroutine c_return_int_ptr_to_fixed_array_bufferify
end interface

```

The Fortran wrapper:

```

function return_int_ptr_to_fixed_array() &
  result(SHT_rv)
  use iso_c_binding, only : C_INT, c_f_pointer
  integer(C_INT), pointer :: SHT_rv(:)
  ! splicer begin function.return_int_ptr_to_fixed_array
  type(POI_SHROUD_array) :: SHT_rv_cdesc
  call c_return_int_ptr_to_fixed_array_bufferify(SHT_rv_cdesc)
  call c_f_pointer(SHT_rv_cdesc%base_addr, SHT_rv, &
    SHT_rv_cdesc%shape(1:1))
  ! splicer end function.return_int_ptr_to_fixed_array
end function return_int_ptr_to_fixed_array

```

Example usage:

```

integer(C_INT), pointer :: irvarray(:)
irvarray => return_int_ptr_to_fixed_array()

```

### 26.3.9 returnIntScalar

pointers.yaml:

```
- decl: int *returnIntScalar(void) +deref(scalar)
```

This is a C file which provides the bufferify function.

wrappointers.c:

```

int POI_returnIntScalar(void)
{
  // splicer begin function.returnIntScalar
  int * SHC_rv = returnIntScalar();
  return *SHC_rv;
  // splicer end function.returnIntScalar
}

```

Fortran calls C via the following interface:

```

interface
  function return_int_scalar() &
    result(SHT_rv) &
    bind(C, name="POI_returnIntScalar")
    use iso_c_binding, only : C_INT
    implicit none
    integer(C_INT) :: SHT_rv
  end function return_int_scalar
end interface

```

Example usage:

```
integer :: ivalue
ivalue = return_int_scalar()
```

### 26.3.10 returnIntPtrDimPointer

Return a Fortran pointer to an array. The length of the array is returned from C++ in the *len* argument. This argument sets the *hidden* attribute since it is not needed in the Fortran wrapper. It will be used in the `c_f_pointer` call to set the length of the array.

The input is in file `ownership.yaml`.

```
- decl: int * ReturnIntPtrDimPointer(int *len+intent(out)+hidden)
  fattrs:
    deref: pointer
    dimension: len
```

The C wrapper calls the C++ function from an `extern C` wrapper. It does not hide the *len* argument. This function does not use the *deref* attribute.

```
int * OWN_ReturnIntPtrDimPointer(int * len)
{
    // splicer begin function.ReturnIntPtrDimPointer
    int * SHC_rv = ReturnIntPtrDimPointer(len);
    return SHC_rv;
    // splicer end function.ReturnIntPtrDimPointer
}
```

The `bufferify` function passes an argument to contain the meta data of the array. It is written to `wrapownership.cpp`.

```
void OWN_ReturnIntPtrDimPointer_bufferify(
    OWN_SHROUD_array *SHT_rv_cdesc)
{
    // splicer begin function.ReturnIntPtrDimPointer_bufferify
    int len;
    int * SHC_rv = ReturnIntPtrDimPointer(&len);
    SHT_rv_cdesc->cxx.addr = SHC_rv;
    SHT_rv_cdesc->cxx.idtor = 0;
    SHT_rv_cdesc->addr.base = SHC_rv;
    SHT_rv_cdesc->type = SH_TYPE_INT;
    SHT_rv_cdesc->elem_len = sizeof(int);
    SHT_rv_cdesc->rank = 1;
    SHT_rv_cdesc->shape[0] = len;
    SHT_rv_cdesc->size = SHT_rv_cdesc->shape[0];
    // splicer end function.ReturnIntPtrDimPointer_bufferify
}
```

Fortran calls the `bufferify` function in `wrapownership.f`.

```
function return_int_ptr_dim_pointer() &
    result(SHT_rv)
    use iso_c_binding, only : C_INT, c_f_pointer
    integer(C_INT), pointer :: SHT_rv(:)
    ! splicer begin function.return_int_ptr_dim_pointer
    type(OWN_SHROUD_array) :: SHT_rv_cdesc
```

(continues on next page)

(continued from previous page)

```

call c_return_int_ptr_dim_pointer_bufferify(SHT_rv_cdesc)
call c_f_pointer(SHT_rv_cdesc%base_addr, SHT_rv, &
  SHT_rv_cdesc%shape(1:1))
! splicer end function.return_int_ptr_dim_pointer
end function return_int_ptr_dim_pointer

```

Fortran usage:

```

integer(C_INT), pointer :: ivalue(:)
integer len

ivalue => return_int_ptr_dim_pointer()
len = size(ivalue)

```

### 26.3.11 returnIntPtrDimAlloc

Convert a pointer returned from C++ into a Fortran allocatable array. To do this, memory is allocated in Fortran then the C++ values are copied into it. The advantage is that the user does not have to worry about releasing the C++ memory. The length of the array is returned from C++ in the *len* argument. This argument sets the *hidden* attribute since it is not needed in the Fortran wrapper.

The input is in file ownership.yaml.

```

- decl: int * ReturnIntPtrDimAlloc(int *len+intent(out)+hidden)
  fattrs:
    deref: allocatable
    dimension: len

```

The C wrapper calls the C++ function from an extern C wrapper. It does not hide the *len* argument. This function does not use the *deref* attribute.

```

int * OWN_ReturnIntPtrDimAlloc(int * len)
{
    // splicer begin function.ReturnIntPtrDimAlloc
    int * SHC_rv = ReturnIntPtrDimAlloc(len);
    return SHC_rv;
    // splicer end function.ReturnIntPtrDimAlloc
}

```

The bufferify function passes an argument to contain the meta data of the array. It is written to wrapownership.cpp.

```

void OWN_ReturnIntPtrDimAlloc_bufferify(OWN_SHROUD_array *SHT_rv_cdesc)
{
    // splicer begin function.ReturnIntPtrDimAlloc_bufferify
    int len;
    int * SHC_rv = ReturnIntPtrDimAlloc(&len);
    SHT_rv_cdesc->cxx.addr = SHC_rv;
    SHT_rv_cdesc->cxx.idtor = 0;
    SHT_rv_cdesc->addr.base = SHC_rv;
    SHT_rv_cdesc->type = SH_TYPE_INT;
    SHT_rv_cdesc->elem_len = sizeof(int);
    SHT_rv_cdesc->rank = 1;
    SHT_rv_cdesc->shape[0] = len;
    SHT_rv_cdesc->size = SHT_rv_cdesc->shape[0];

```

(continues on next page)



(continued from previous page)

```
// splicer end function.ReturnIntPtrDimAlloc_bufferify
}
```

Fortran calls the bufferify function in wrapfownership.f.

```
function return_int_ptr_dim_alloc() &
    result(SHT_rv)
    use iso_c_binding, only : C_INT, C_LOC, C_SIZE_T
    integer(C_INT), allocatable, target :: SHT_rv(:)
    ! splicer begin function.return_int_ptr_dim_alloc
    type(OWN_SHROUD_array) :: SHT_rv_cdesc
    call c_return_int_ptr_dim_alloc_bufferify(SHT_rv_cdesc)
    allocate(SHT_rv(SHT_rv_cdesc%shape(1)))
    call OWN_SHROUD_copy_array(SHT_rv_cdesc, C_LOC(SHT_rv), &
        size(SHT_rv, kind=C_SIZE_T))
    ! splicer end function.return_int_ptr_dim_alloc
end function return_int_ptr_dim_alloc
```

Fortran usage:

```
integer(C_INT), allocatable :: ivalue(:)
integer len

ivalue = return_int_ptr_dim_alloc()
len = size(ivalue)
```

## 26.4 Bool

### 26.4.1 checkBool

Assignments are done in the Fortran wrapper to convert between logical and logical (C\_BOOL).

C library function in clibrary:

```
void checkBool(const bool arg1, bool *arg2, bool *arg3)
{
    strncpy(last_function_called, "checkBool", MAXLAST);
    *arg2 = ! arg1;
    *arg3 = ! *arg3;
    return;
}
```

clibrary.yaml:

```
- decl: void checkBool(const bool arg1,
                      bool *arg2+intent(out),
                      bool *arg3+intent(inout))
```

Fortran calls C via the following interface:

```
interface
    subroutine c_check_bool(arg1, arg2, arg3) &
        bind(C, name="checkBool")
        use iso_c_binding, only : C_BOOL
```

(continues on next page)

(continued from previous page)

```

    implicit none
    logical(C_BOOL), value, intent(IN) :: arg1
    logical(C_BOOL), intent(OUT) :: arg2
    logical(C_BOOL), intent(INOUT) :: arg3
end subroutine c_check_bool
end interface

```

The Fortran wrapper:

```

subroutine check_bool(arg1, arg2, arg3)
  use iso_c_binding, only : C_BOOL
  logical, value, intent(IN) :: arg1
  logical, intent(OUT) :: arg2
  logical, intent(INOUT) :: arg3
  ! splicer begin function.check_bool
  logical(C_BOOL) :: SHT_arg1_cxx
  logical(C_BOOL) :: SHT_arg2_cxx
  logical(C_BOOL) :: SHT_arg3_cxx
  SHT_arg1_cxx = arg1 ! coerce to C_BOOL
  SHT_arg3_cxx = arg3 ! coerce to C_BOOL
  call c_check_bool(SHT_arg1_cxx, SHT_arg2_cxx, SHT_arg3_cxx)
  arg2 = SHT_arg2_cxx ! coerce to logical
  arg3 = SHT_arg3_cxx ! coerce to logical
  ! splicer end function.check_bool
end subroutine check_bool

```

Fortran usage:

```

logical rv_logical, wrk_logical
rv_logical = .true.
wrk_logical = .true.
call check_bool(.true., rv_logical, wrk_logical)
call assert_false(rv_logical)
call assert_false(wrk_logical)

```

## 26.5 Character

### 26.5.1 acceptName

Pass a NULL terminated string to a C function. The string will be unchanged.

C library function in `clibrary.c`:

```

void acceptName(const char *name)
{
    strncpy(last_function_called, "acceptName", MAXLAST);
}

```

`clibrary.yaml`:

```

- decl: void acceptName(const char *name)

```

Fortran calls C via the following interface:

```

interface
  subroutine c_accept_name(name) &
    bind(C, name="acceptName")
    use iso_c_binding, only : C_CHAR
    implicit none
    character(kind=C_CHAR), intent(IN) :: name(*)
  end subroutine c_accept_name
end interface

```

The Fortran wrapper:

```

subroutine accept_name(name)
  use iso_c_binding, only : C_NULL_CHAR
  character(len=*), intent(IN) :: name
  ! splicer begin function.accept_name
  call c_accept_name(trim(name)//C_NULL_CHAR)
  ! splicer end function.accept_name
end subroutine accept_name

```

No C wrapper is required since the Fortran wrapper creates a NULL terminated string by calling the Fortran intrinsic function `trim` and concatenating `C_NULL_CHAR` (from `iso_c_binding`). This can be done since the argument `name` is `const` which sets the attribute *intent(in)*.

Fortran usage:

```
call accept_name("spot")
```

## 26.5.2 returnOneName

Pass the pointer to a buffer which the C library will fill. The length of the string is implicitly known by the library to not exceed the library variable `MAXNAME`.

C library function in `clibrary.c`:

```

void returnOneName(char *name1)
{
  strcpy(name1, "bill");
}

```

`clibrary.yaml`:

```
- decl: void returnOneName(char *name1+intent(out)+charlen(MAXNAME))
```

The C wrapper:

```

void CLI_returnOneName_bufferify(char *name1, int SHT_name1_len)
{
  // splicer begin function.returnOneName_bufferify
  returnOneName(name1);
  ShroudStrBlankFill(name1, SHT_name1_len);
  // splicer end function.returnOneName_bufferify
}

```

Fortran calls C via the following interface:

```
interface
  subroutine c_return_one_name_bufferify(name1, SHT_name1_len) &
    bind(C, name="CLI_returnOneName_bufferify")
    use iso_c_binding, only : C_CHAR, C_INT
    implicit none
    character(kind=C_CHAR), intent(OUT) :: name1(*)
    integer(C_INT), value, intent(IN) :: SHT_name1_len
  end subroutine c_return_one_name_bufferify
end interface
```

The Fortran wrapper:

```
subroutine return_one_name(name1)
  use iso_c_binding, only : C_INT
  character(len=*), intent(OUT) :: name1
  ! splicer begin function.return_one_name
  integer(C_INT) SHT_name1_len
  SHT_name1_len = len(name1, kind=C_INT)
  call c_return_one_name_bufferify(name1, SHT_name1_len)
  ! splicer end function.return_one_name
end subroutine return_one_name
```

Fortran usage:

```
name1 = " "
call return_one_name(name1)
call assert_equals("bill", name1)
```

### 26.5.3 passCharPtr

The function `passCharPtr(dest, src)` is equivalent to the Fortran statement `dest = src`:

C++ library function in `strings.cpp`:

```
void passCharPtr(char *dest, const char *src)
{
    std::strcpy(dest, src);
}
```

`strings.yaml`:

```
- decl: void passCharPtr(char * dest+intent(out)+charlen(40),
                        const char *src)
```

The intent of `dest` must be explicit. It defaults to *intent(inout)* since it is a pointer. `src` is implied to be *intent(in)* since it is `const`. This single line will create five different wrappers.

The native C version. The only feature this provides to Fortran is the ability to call a C++ function by wrapping it in an `extern "C"` function. The user is responsible for providing the NULL termination. The result in `str` will also be NULL terminated instead of blank filled.:

```
void STR_passCharPtr(char * dest, const char * src)
{
    // splicer begin function.passCharPtr
    passCharPtr(dest, src);
```

(continues on next page)

(continued from previous page)

```

    // splicer end function.passCharPtr
}

```

The C wrapper:

```

void STR_passCharPtr_bufferify(char *dest, int SHT_dest_len,
    const char * src)
{
    // splicer begin function.passCharPtr_bufferify
    passCharPtr(dest, src);
    ShroudStrBlankFill(dest, SHT_dest_len);
    // splicer end function.passCharPtr_bufferify
}

```

Fortran calls C via the following interface:

```

interface
    subroutine c_pass_char_ptr(dest, src) &
        bind(C, name="STR_passCharPtr")
        use iso_c_binding, only : C_CHAR
        implicit none
        character(kind=C_CHAR), intent(OUT) :: dest(*)
        character(kind=C_CHAR), intent(IN) :: src(*)
    end subroutine c_pass_char_ptr
end interface

```

```

interface
    subroutine c_pass_char_ptr_bufferify(dest, SHT_dest_len, src) &
        bind(C, name="STR_passCharPtr_bufferify")
        use iso_c_binding, only : C_CHAR, C_INT
        implicit none
        character(kind=C_CHAR), intent(OUT) :: dest(*)
        integer(C_INT), value, intent(IN) :: SHT_dest_len
        character(kind=C_CHAR), intent(IN) :: src(*)
    end subroutine c_pass_char_ptr_bufferify
end interface

```

The Fortran wrapper:

```

subroutine pass_char_ptr(dest, src)
    use iso_c_binding, only : C_INT, C_NULL_CHAR
    character(len=*), intent(OUT) :: dest
    character(len=*), intent(IN) :: src
    ! splicer begin function.pass_char_ptr
    integer(C_INT) SHT_dest_len
    SHT_dest_len = len(dest, kind=C_INT)
    call c_pass_char_ptr_bufferify(dest, SHT_dest_len, &
        trim(src)//C_NULL_CHAR)
    ! splicer end function.pass_char_ptr
end subroutine pass_char_ptr

```

The function can be called without the user aware that it is written in C++:

```

character(30) str
call pass_char_ptr(dest=str, src="mouse")

```

## 26.5.4 ImpliedTextLen

Pass the pointer to a buffer which the C library will fill. The length of the buffer is passed in `ltext`. Since Fortran knows the length of CHARACTER variable, the Fortran wrapper does not need to be explicitly told the length of the variable. Instead its value can be defined with the *implied* attribute.

This can be used to emulate the behavior of most Fortran compilers which will pass an additional, hidden argument which contains the length of a CHARACTER argument.

C library function in `clibrary.c`:

```
void ImpliedTextLen(char *text, int ltext)
{
    strncpy(text, "ImpliedTextLen", ltext);
    strncpy(last_function_called, "ImpliedTextLen", MAXLAST);
}
```

`clibrary.yaml`:

```
- decl: void ImpliedTextLen(char *text+intent(out)+charlen(MAXNAME),
                           int ltext+implied(len(text)))
```

The C wrapper:

```
void CLI_ImpliedTextLen_bufferify(char *text, int SHT_text_len,
                                  int ltext)
{
    // splicer begin function.ImpliedTextLen_bufferify
    ImpliedTextLen(text, ltext);
    ShroudStrBlankFill(text, SHT_text_len);
    // splicer end function.ImpliedTextLen_bufferify
}
```

Fortran calls C via the following interface:

```
interface
    subroutine c_implied_text_len_bufferify(text, SHT_text_len, &
        ltext) &
        bind(C, name="CLI_ImpliedTextLen_bufferify")
        use iso_c_binding, only : C_CHAR, C_INT
        implicit none
        character(kind=C_CHAR), intent(OUT) :: text(*)
        integer(C_INT), value, intent(IN) :: SHT_text_len
        integer(C_INT), value, intent(IN) :: ltext
    end subroutine c_implied_text_len_bufferify
end interface
```

The Fortran wrapper:

```
subroutine implied_text_len(text)
    use iso_c_binding, only : C_INT
    character(len=*) , intent(OUT) :: text
    integer(C_INT) :: SH_ltext
    ! splicer begin function.implied_text_len
    integer(C_INT) SHT_text_len
    SHT_text_len = len(text, kind=C_INT)
    SH_ltext = len(text, kind=C_INT)
    call c_implied_text_len_bufferify(text, SHT_text_len, SH_ltext)
```

(continues on next page)

(continued from previous page)

```

! splicer end function.implied_text_len
end subroutine implied_text_len

```

Fortran usage:

```

character (MAXNAME) name1
call implied_text_len(name1)
call assert_equals("ImpliedTextLen", name1)

```

## 26.5.5 acceptCharArrayIn

Arguments of type `char **` are assumed to be a list of NULL terminated strings. In Fortran this pattern would be an array of `CHARACTER` where all strings are the same length. The Fortran variable is converted into the the C version by copying the data then releasing it at the end of the wrapper.

pointers.yaml:

```

- decl: void acceptCharArrayIn(char **names +intent(in))

```

This is a C file which provides the bufferify function.

wrappointers.c:

```

int POI_acceptCharArrayIn_bufferify(const char *names,
    size_t SHT_names_size, int SHT_names_len)
{
    // splicer begin function.acceptCharArrayIn_bufferify
    char **SHCXX_names = ShroudStrArrayAlloc(names, SHT_names_size,
        SHT_names_len);
    int SHC_rv = acceptCharArrayIn(SHCXX_names);
    ShroudStrArrayFree(SHCXX_names, SHT_names_size);
    return SHC_rv;
    // splicer end function.acceptCharArrayIn_bufferify
}

```

Most of the work is done by the helper function. This converts the Fortran array into NULL terminated strings by copying all of the values:

```

// helper ShroudStrArrayAlloc
// Copy src into new memory and null terminate.
static char **ShroudStrArrayAlloc(const char *src, int nsrc, int len)
{
    char **rv = malloc(sizeof(char *) * nsrc);
    const char *src0 = src;
    for(int i=0; i < nsrc; ++i) {
        int ntrim = ShroudLenTrim(src0, len);
        char *tgt = malloc(ntrim+1);
        memcpy(tgt, src0, ntrim);
        tgt[ntrim] = '\0';
        rv[i] = tgt;
        src0 += len;
    }
    return rv;
}

```

Fortran calls C via the following interface:

```
interface
  function c_accept_char_array_in(names) &
    result(SHT_rv) &
    bind(C, name="acceptCharArrayIn")
    use iso_c_binding, only : C_INT, C_PTR
    implicit none
    type(C_PTR), intent(IN) :: names(*)
    integer(C_INT) :: SHT_rv
  end function c_accept_char_array_in
end interface
```

The Fortran wrapper:

```
function accept_char_array_in(names) &
  result(SHT_rv)
  use iso_c_binding, only : C_INT, C_SIZE_T
  character(len=*), intent(IN) :: names(:)
  integer(C_INT) :: SHT_rv
  ! splicer begin function.accept_char_array_in
  SHT_rv = c_accept_char_array_in_bufferify(names, &
    size(names, kind=C_SIZE_T), len(names, kind=C_INT))
  ! splicer end function.accept_char_array_in
end function accept_char_array_in
```

Example usage:

```
character(10) :: in(3) = [ &
  "dog", &
  "cat", &
  "monkey" &
]
call accept_char_array_in(in)
```

## 26.6 std::string

### 26.6.1 acceptStringReference

C++ library function in `strings.c`:

```
void acceptStringReference(std::string & arg1)
{
  arg1.append("dog");
}
```

`strings.yaml`:

```
- decl: void acceptStringReference(std::string & arg1)
```

A reference defaults to *intent(inout)* and will add both the *len* and *len\_trim* annotations.

Both generated functions will convert `arg` into a `std::string`, call the function, then copy the results back into the argument.

Which will call the C wrapper:



```

void STR_acceptStringReference(char * arg1)
{
    // splicer begin function.acceptStringReference
    std::string SHCXX_arg1(arg1);
    acceptStringReference(SHCXX_arg1);
    strcpy(arg1, SHCXX_arg1.c_str());
    // splicer end function.acceptStringReference
}

```

The C wrapper:

```

void STR_acceptStringReference_bufferify(char *arg1, int SHT_arg1_len)
{
    // splicer begin function.acceptStringReference_bufferify
    std::string SHCXX_arg1(arg1, ShroudLenTrim(arg1, SHT_arg1_len));
    acceptStringReference(SHCXX_arg1);
    ShroudStrCopy(arg1, SHT_arg1_len, SHCXX_arg1.data(),
        SHCXX_arg1.size());
    // splicer end function.acceptStringReference_bufferify
}

```

An interface for the native C function is also created:

```

interface
    subroutine c_accept_string_reference(arg1) &
        bind(C, name="STR_acceptStringReference")
        use iso_c_binding, only : C_CHAR
        implicit none
        character(kind=C_CHAR), intent(INOUT) :: arg1(*)
    end subroutine c_accept_string_reference
end interface

```

Fortran calls C via the following interface:

```

interface
    subroutine c_accept_string_reference_bufferify(arg1, &
        SHT_arg1_len) &
        bind(C, name="STR_acceptStringReference_bufferify")
        use iso_c_binding, only : C_CHAR, C_INT
        implicit none
        character(kind=C_CHAR), intent(INOUT) :: arg1(*)
        integer(C_INT), value, intent(IN) :: SHT_arg1_len
    end subroutine c_accept_string_reference_bufferify
end interface

```

The Fortran wrapper:

```

subroutine accept_string_reference(arg1)
    use iso_c_binding, only : C_INT
    character(len=*), intent(INOUT) :: arg1
    ! splicer begin function.accept_string_reference
    integer(C_INT) SHT_arg1_len
    SHT_arg1_len = len(arg1, kind=C_INT)
    call c_accept_string_reference_bufferify(arg1, SHT_arg1_len)
    ! splicer end function.accept_string_reference
end subroutine accept_string_reference

```

The important thing to notice is that the pure C version could do very bad things since it does not know how much

space it has to copy into. The `bufferify` version knows the allocated length of the argument. However, since the input argument is a fixed length it may be too short for the new string value:

Fortran usage:

```
character(30) str
str = "cat"
call accept_string_reference(str)
call assert_true( str == "catdog")
```

## 26.7 char functions

### 26.7.1 getCharPtr1

Return a pointer and convert into an `ALLOCATABLE CHARACTER` variable. The Fortran application is responsible to release the memory. However, this may be done automatically by the Fortran runtime.

C++ library function in `strings.cpp`:

```
const char * getCharPtr1()
{
    return static_char;
}
```

`strings.yaml`:

```
- decl: const char * getCharPtr1()
```

The C wrapper copies all of the metadata into a `SHROUD_array` struct which is used by the Fortran wrapper:

```
void STR_getCharPtr1_bufferify(STR_SHROUD_array *SHT_rv_cdesc)
{
    // splicer begin function.getCharPtr1_bufferify
    const char * SHC_rv = getCharPtr1();
    SHT_rv_cdesc->cxx.addr = const_cast<char *>(SHC_rv);
    SHT_rv_cdesc->cxx.idtor = 0;
    SHT_rv_cdesc->addr.ccharp = SHC_rv;
    SHT_rv_cdesc->type = SH_TYPE_OTHER;
    SHT_rv_cdesc->elem_len = SHC_rv == nullptr ? 0 : std::strlen(SHC_rv);
    SHT_rv_cdesc->size = 1;
    SHT_rv_cdesc->rank = 0;
    // splicer end function.getCharPtr1_bufferify
}
```

Fortran calls C via the following interface:

```
interface
    subroutine c_get_char_ptr1_bufferify(SHT_rv) &
        bind(C, name="STR_getCharPtr1_bufferify")
        import :: STR_SHROUD_array
        implicit none
        type(STR_SHROUD_array), intent(OUT) :: SHT_rv
    end subroutine c_get_char_ptr1_bufferify
end interface
```

The Fortran wrapper uses the metadata in `DSHF_rv` to allocate a `CHARACTER` variable of the correct length. The helper function `SHROUD_copy_string_and_free` will copy the results of the C++ function into the return variable:

```
function get_char_ptr1() &
    result(SHT_rv)
    character(len=:), allocatable :: SHT_rv
    ! splicer begin function.get_char_ptr1
    type(STR_SHROUD_array) :: SHT_rv_cdesc
    call c_get_char_ptr1_bufferify(SHT_rv_cdesc)
    allocate(character(len=SHT_rv_cdesc%elem_len) :: SHT_rv)
    call STR_SHROUD_copy_string_and_free(SHT_rv_cdesc, SHT_rv, &
        SHT_rv_cdesc%elem_len)
    ! splicer end function.get_char_ptr1
end function get_char_ptr1
```

Fortran usage:

```
character(len=:), allocatable :: str
str = get_char_ptr1()
```

## 26.7.2 getCharPtr2

If you know the maximum size of string that you expect the function to return, then the `len` attribute is used to declare the length. The explicit `ALLOCATE` is avoided but any result which is longer than the length will be silently truncated.

C++ library function in `strings.cpp`:

```
const char * getCharPtr2()
{
    return static_char;
}
```

`strings.yaml`:

```
- decl: const char * getCharPtr2() +len(30)
```

The C wrapper:

```
void STR_getCharPtr2_bufferify(char *SHC_rv, int SHT_rv_len)
{
    // splicer begin function.getCharPtr2_bufferify
    const char * SHCXX_rv = getCharPtr2();
    ShroudStrCopy(SHC_rv, SHT_rv_len, SHCXX_rv, -1);
    // splicer end function.getCharPtr2_bufferify
}
```

Fortran calls C via the following interface:

```
interface
    subroutine c_get_char_ptr2_bufferify(SHT_rv, SHT_rv_len) &
        bind(C, name="STR_getCharPtr2_bufferify")
        use iso_c_binding, only : C_CHAR, C_INT
        implicit none
        character(kind=C_CHAR), intent(OUT) :: SHT_rv(*)
        integer(C_INT), value, intent(IN) :: SHT_rv_len
    end subroutine
end interface
```

(continues on next page)

(continued from previous page)

```

    end subroutine c_get_char_ptr2_bufferify
end interface

```

The Fortran wrapper:

```

function get_char_ptr2() &
    result(SHT_rv)
    use iso_c_binding, only : C_INT
    character(len=30) :: SHT_rv
    ! splicer begin function.get_char_ptr2
    integer(C_INT) SHT_rv_len
    SHT_rv_len = len(SHT_rv, kind=C_INT)
    call c_get_char_ptr2_bufferify(SHT_rv, SHT_rv_len)
    ! splicer end function.get_char_ptr2
end function get_char_ptr2

```

Fortran usage:

```

character(30) str
str = get_char_ptr2()

```

### 26.7.3 getCharPtr3

Create a Fortran subroutine with an additional CHARACTER argument for the C function result. Any size character string can be returned limited by the size of the Fortran argument. The argument is defined by the *F\_string\_result\_as\_arg* format string.

C++ library function in `strings.cpp`:

```

const char * getCharPtr3()
{
    return static_char;
}

```

`strings.yaml`:

```

- decl: const char * getCharPtr3()
  format:
    F_string_result_as_arg: output

```

The C wrapper:

```

void STR_getCharPtr3_bufferify(char *output, int noutput)
{
    // splicer begin function.getCharPtr3_bufferify
    const char * SHC_rv = getCharPtr3();
    ShroudStrCopy(output, noutput, SHC_rv, -1);
    // splicer end function.getCharPtr3_bufferify
}

```

Fortran calls C via the following interface:

```

interface
    subroutine c_get_char_ptr3_bufferify(output, noutput) &
        bind(C, name="STR_getCharPtr3_bufferify")

```

(continues on next page)

(continued from previous page)

```

    use iso_c_binding, only : C_CHAR, C_INT
    implicit none
    character(kind=C_CHAR), intent(OUT) :: output(*)
    integer(C_INT), value, intent(IN) :: noutput
  end subroutine c_get_char_ptr3_bufferify
end interface

```

The Fortran wrapper:

```

subroutine get_char_ptr3(output)
  character(*), intent(OUT) :: output
  ! splicer begin function.get_char_ptr3
  integer(C_INT) SHT_rv_len
  SHT_rv_len = len(output, kind=C_INT)
  call c_get_char_ptr3_bufferify(output, SHT_rv_len)
  ! splicer end function.get_char_ptr3
end subroutine get_char_ptr3

```

Fortran usage:

```

character(30) str
call get_char_ptrs(str)

```

## 26.8 string functions

### 26.8.1 getConstStringRefPure

C++ library function in strings.cpp:

```

const std::string& getConstStringRefPure()
{
    return static_str;
}

```

strings.yaml:

```

- decl: const string& getConstStringRefPure()

```

The C wrapper:

```

void STR_getConstStringRefPure_bufferify(STR_SHROUD_array *SHT_rv_cdesc)
{
    // splicer begin function.getConstStringRefPure_bufferify
    const std::string & SHCXX_rv = getConstStringRefPure();
    ShroudStrToArray(SHT_rv_cdesc, &SHCXX_rv, 0);
    // splicer end function.getConstStringRefPure_bufferify
}

```

The native C wrapper:

```

const char * STR_getConstStringRefPure(void)
{
    // splicer begin function.getConstStringRefPure

```

(continues on next page)

(continued from previous page)

```

const std::string & SHCXX_rv = getConstStringRefPure();
const char * SHC_rv = SHCXX_rv.c_str();
return SHC_rv;
// splicer end function.getConstStringRefPure
}

```

Fortran calls C via the following interface:

```

interface
  subroutine c_get_const_string_ref_pure_bufferify(SHT_rv) &
    bind(C, name="STR_getConstStringRefPure_bufferify")
    import :: STR_SHROUD_array
    implicit none
    type(STR_SHROUD_array), intent(OUT) :: SHT_rv
  end subroutine c_get_const_string_ref_pure_bufferify
end interface

```

The Fortran wrapper:

```

function get_const_string_ref_pure() &
  result(SHT_rv)
  character(len=:), allocatable :: SHT_rv
  ! splicer begin function.get_const_string_ref_pure
  type(STR_SHROUD_array) :: SHT_rv_cdesc
  call c_get_const_string_ref_pure_bufferify(SHT_rv_cdesc)
  allocate(character(len=SHT_rv_cdesc%elem_len) :: SHT_rv)
  call STR_SHROUD_copy_string_and_free(SHT_rv_cdesc, SHT_rv, &
    SHT_rv_cdesc%elem_len)
  ! splicer end function.get_const_string_ref_pure
end function get_const_string_ref_pure

```

Fortran usage:

```

str = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXX'
str = get_const_string_ref_pure()
call assert_true( str == static_str, "getConstStringRefPure")

```

## 26.9 std::vector

### 26.9.1 vector\_sum

C++ library function in `vectors.cpp`:

```

int vector_sum(const std::vector<int> &arg)
{
  int sum = 0;
  for(std::vector<int>::const_iterator it = arg.begin(); it != arg.end(); ++it) {
    sum += *it;
  }
  return sum;
}

```

`vectors.yaml`:

```
- decl: int vector_sum(const std::vector<int> &arg)
```

intent(in) is implied for the *vector\_sum* argument since it is *const*. The Fortran wrapper passes the array and the size to C.

The C wrapper:

```
int VEC_vector_sum_bufferify(int *arg, size_t SHT_arg_size)
{
    // splicer begin function.vector_sum_bufferify
    const std::vector<int> SHCXX_arg(arg, arg + SHT_arg_size);
    int SHC_rv = vector_sum(SHCXX_arg);
    return SHC_rv;
    // splicer end function.vector_sum_bufferify
}
```

Fortran calls C via the following interface:

```
interface
    function c_vector_sum_bufferify(arg, SHT_arg_size) &
        result(SHT_rv) &
        bind(C, name="VEC_vector_sum_bufferify")
        use iso_c_binding, only : C_INT, C_SIZE_T
        implicit none
        integer(C_INT), intent(IN) :: arg(*)
        integer(C_SIZE_T), intent(IN), value :: SHT_arg_size
        integer(C_INT) :: SHT_rv
    end function c_vector_sum_bufferify
end interface
```

The Fortran wrapper:

```
function vector_sum(arg) &
    result(SHT_rv)
    use iso_c_binding, only : C_INT, C_SIZE_T
    integer(C_INT), intent(IN) :: arg(:)
    integer(C_INT) :: SHT_rv
    ! splicer begin function.vector_sum
    SHT_rv = c_vector_sum_bufferify(arg, size(arg, kind=C_SIZE_T))
    ! splicer end function.vector_sum
end function vector_sum
```

Fortran usage:

```
integer(C_INT) intv(5)
intv = [1,2,3,4,5]
irv = vector_sum(intv)
call assert_true(irv .eq. 15)
```

## 26.9.2 vector\_iota\_out

C++ library function in *vectors.cpp* accepts an empty vector then fills in some values. In this example, a Fortran array is passed in and will be filled.

```
void vector_iota_out(std::vector<int> &arg)
{
```

(continues on next page)

(continued from previous page)

```

    for(unsigned int i=0; i < 5; i++) {
        arg.push_back(i + 1);
    }
    return;
}

```

vectors.yaml:

```

- decl: void vector_iota_out(std::vector<int> &arg,intent(out))

```

The C wrapper allocates a new `std::vector` instance which will be returned to the Fortran wrapper. Variable `Darg` will be filled with the meta data for the `std::vector` in a form that allows Fortran to access it. The value of `Darg->cxx.idtor` is computed by Shroud and used to release the memory (index of destructor).

```

void VEC_vector_iota_out_bufferify(VEC_SHROUD_array *SHT_arg_cdesc)
{
    // splicer begin function.vector_iota_out_bufferify
    std::vector<int> *SHCXX_arg = new std::vector<int>;
    vector_iota_out(*SHCXX_arg);
    SHT_arg_cdesc->cxx.addr = SHCXX_arg;
    SHT_arg_cdesc->cxx.idtor = 1;
    SHT_arg_cdesc->addr.base = SHCXX_arg->empty() ? nullptr : &SHCXX_arg->front();
    SHT_arg_cdesc->type = SH_TYPE_INT;
    SHT_arg_cdesc->elem_len = sizeof(int);
    SHT_arg_cdesc->size = SHCXX_arg->size();
    SHT_arg_cdesc->rank = 1;
    SHT_arg_cdesc->shape[0] = SHT_arg_cdesc->size;
    // splicer end function.vector_iota_out_bufferify
}

```

Fortran calls C via the following interface:

```

interface
    subroutine c_vector_iota_out_bufferify(SHT_arg_cdesc) &
        bind(C, name="VEC_vector_iota_out_bufferify")
        import :: VEC_SHROUD_array
        implicit none
        type(VEC_SHROUD_array), intent(OUT) :: SHT_arg_cdesc
    end subroutine c_vector_iota_out_bufferify
end interface

```

The Fortran wrapper passes a `SHROUD_array` instance which will be filled by the C wrapper.

```

subroutine vector_iota_out(arg)
    use iso_c_binding, only : C_INT, C_LOC, C_SIZE_T
    integer(C_INT), intent(OUT), target :: arg(:)
    ! splicer begin function.vector_iota_out
    type(VEC_SHROUD_array) :: SHT_arg_cdesc
    call c_vector_iota_out_bufferify(SHT_arg_cdesc)
    call VEC_SHROUD_copy_array(SHT_arg_cdesc, C_LOC(arg), &
        size(arg,kind=C_SIZE_T))
    ! splicer end function.vector_iota_out
end subroutine vector_iota_out

```

Function `SHROUD_copy_array_int` copies the values into the user's argument. If the argument is too short, not all values returned by the library function will be copied.



```

// helper copy_array
// Copy std::vector into array c_var(c_var_size).
// Then release std::vector.
// Called from Fortran.
void VEC_ShroudCopyArray(VEC_SHROUD_array *data, void *c_var,
    size_t c_var_size)
{
    const void *cxx_var = data->addr.base;
    int n = c_var_size < data->size ? c_var_size : data->size;
    n *= data->elem_len;
    std::memcpy(c_var, cxx_var, n);
    VEC_SHROUD_memory_destructor(&data->cxx); // delete data->cxx.addr
}

```

Finally, the `std::vector` is released based on the value of `idtor`:

```

// Release library allocated memory.
void VEC_SHROUD_memory_destructor(VEC_SHROUD_capsule_data *cap)
{
    void *ptr = cap->addr;
    switch (cap->idtor) {
    case 0: // --none--
    {
        // Nothing to delete
        break;
    }
    case 1: // std_vector_int
    {
        std::vector<int> *cxx_ptr =
            reinterpret_cast<std::vector<int> *>(ptr);
        delete cxx_ptr;
        break;
    }
    case 2: // std_vector_double
    {
        std::vector<double> *cxx_ptr =
            reinterpret_cast<std::vector<double> *>(ptr);
        delete cxx_ptr;
        break;
    }
    default:
    {
        // Unexpected case in destructor
        break;
    }
    }
    cap->addr = nullptr;
    cap->idtor = 0; // avoid deleting again
}

```

Fortran usage:

```

integer(C_INT) intv(5)
intv(:) = 0
call vector_iota_out(intv)
call assert_true(all(intv(:) .eq. [1,2,3,4,5]))

```

### 26.9.3 vector\_iota\_out\_alloc

C++ library function in `vectors.cpp` accepts an empty vector then fills in some values. In this example, the Fortran argument is `ALLOCATABLE` and will be sized based on the output of the library function.

```
void vector_iota_out_alloc(std::vector<int> &arg)
{
    for(unsigned int i=0; i < 5; i++) {
        arg.push_back(i + 1);
    }
    return;
}
```

The attribute `+deref(allocatable)` will cause the argument to be an `ALLOCATABLE` array.

`vectors.yaml`:

```
- decl: void vector_iota_out_alloc(std::vector<int> &
  ↪arg+intent(out)+deref(allocatable))
```

The C wrapper:

```
void VEC_vector_iota_out_alloc_bufferify(
    VEC_SHROUD_array *SHT_arg_cdesc)
{
    // splicer begin function.vector_iota_out_alloc_bufferify
    std::vector<int> *SHCXX_arg = new std::vector<int>;
    vector_iota_out_alloc(*SHCXX_arg);
    SHT_arg_cdesc->cxx.addr = SHCXX_arg;
    SHT_arg_cdesc->cxx.idtor = 1;
    SHT_arg_cdesc->addr.base = SHCXX_arg->empty() ? nullptr : &SHCXX_arg->front();
    SHT_arg_cdesc->type = SH_TYPE_INT;
    SHT_arg_cdesc->elem_len = sizeof(int);
    SHT_arg_cdesc->size = SHCXX_arg->size();
    SHT_arg_cdesc->rank = 1;
    SHT_arg_cdesc->shape[0] = SHT_arg_cdesc->size;
    // splicer end function.vector_iota_out_alloc_bufferify
}
```

Fortran calls C via the following interface:

```
interface
    subroutine c_vector_iota_out_alloc_bufferify(SHT_arg_cdesc) &
        bind(C, name="VEC_vector_iota_out_alloc_bufferify")
        import :: VEC_SHROUD_array
        implicit none
        type(VEC_SHROUD_array), intent(OUT) :: SHT_arg_cdesc
    end subroutine c_vector_iota_out_alloc_bufferify
end interface
```

The Fortran wrapper passes a `SHROUD_array` instance which will be filled by the C wrapper. After the function returns, the `allocate` statement allocates an array of the proper length.

```
subroutine vector_iota_out_alloc(arg)
    use iso_c_binding, only : C_INT, C_LOC, C_SIZE_T
    integer(C_INT), intent(OUT), allocatable, target :: arg(:)
    ! splicer begin function.vector_iota_out_alloc
    type(VEC_SHROUD_array) :: SHT_arg_cdesc
```

(continues on next page)

(continued from previous page)

```

call c_vector_iota_out_alloc_bufferify(SHT_arg_cdesc)
allocate(arg(SHT_arg_cdesc%size))
call VEC_SHROUD_copy_array(SHT_arg_cdesc, C_LOC(arg), &
    size(arg,kind=C_SIZE_T))
! splicer end function.vector_iota_out_alloc
end subroutine vector_iota_out_alloc

```

inta is intent (out), so it will be deallocated upon entry to vector\_iota\_out\_alloc.

Fortran usage:

```

integer(C_INT), allocatable :: inta(:)
call vector_iota_out_alloc(inta)
call assert_true(allocated(inta))
call assert_equals(5, size(inta))
call assert_true( all(inta == [1,2,3,4,5]), &
    "vector_iota_out_alloc value")

```

## 26.9.4 vector\_iota\_inout\_alloc

C++ library function in vectors.cpp:

```

void vector_iota_inout_alloc(std::vector<int> &arg)
{
    for(unsigned int i=0; i < 5; i++) {
        arg.push_back(i + 11);
    }
    return;
}

```

vectors.yaml:

```

- decl: void vector_iota_out_alloc(std::vector<int> &
  ↪arg+intent(inout)+deref(allocatable))

```

The C wrapper creates a new std::vector and initializes it to the Fortran argument.

```

void VEC_vector_iota_inout_alloc_bufferify(int *arg,
    size_t SHT_arg_size, VEC_SHROUD_array *SHT_arg_cdesc)
{
    // splicer begin function.vector_iota_inout_alloc_bufferify
    std::vector<int> *SHCXX_arg = new std::vector<int>(
        arg, arg + SHT_arg_size);
    vector_iota_inout_alloc(*SHCXX_arg);
    SHT_arg_cdesc->cxx.addr = SHCXX_arg;
    SHT_arg_cdesc->cxx.idtor = 1;
    SHT_arg_cdesc->addr.base = SHCXX_arg->empty() ? nullptr : &SHCXX_arg->front();
    SHT_arg_cdesc->type = SH_TYPE_INT;
    SHT_arg_cdesc->elem_len = sizeof(int);
    SHT_arg_cdesc->size = SHCXX_arg->size();
    SHT_arg_cdesc->rank = 1;
    SHT_arg_cdesc->shape[0] = SHT_arg_cdesc->size;
    // splicer end function.vector_iota_inout_alloc_bufferify
}

```

Fortran calls C via the following interface:

```
interface
  subroutine c_vector_iota_inout_alloc_bufferify(arg, &
    SHT_arg_size, SHT_arg_cdesc) &
    bind(C, name="VEC_vector_iota_inout_alloc_bufferify")
  use iso_c_binding, only : C_INT, C_SIZE_T
  import :: VEC_SHROUD_array
  implicit none
  integer(C_INT), intent(IN) :: arg(*)
  integer(C_SIZE_T), intent(IN), value :: SHT_arg_size
  type(VEC_SHROUD_array), intent(OUT) :: SHT_arg_cdesc
end subroutine c_vector_iota_inout_alloc_bufferify
end interface
```

The Fortran wrapper will deallocate the argument after returning since it is *intent(inout)*. The *in* values are now stored in the `std::vector`. A new array is allocated to the current size of the `std::vector`. Fortran has no `realloc` statement. Finally, the new values are copied into the Fortran array and the `std::vector` is released.

```
subroutine vector_iota_inout_alloc(arg)
  use iso_c_binding, only : C_INT, C_LOC, C_SIZE_T
  integer(C_INT), intent(INOUT), allocatable, target :: arg(:)
  ! splicer begin function.vector_iota_inout_alloc
  type(VEC_SHROUD_array) :: SHT_arg_cdesc
  call c_vector_iota_inout_alloc_bufferify(arg, &
    size(arg, kind=C_SIZE_T), SHT_arg_cdesc)
  if (allocated(arg)) deallocate(arg)
  allocate(arg(SHT_arg_cdesc%size))
  call VEC_SHROUD_copy_array(SHT_arg_cdesc, C_LOC(arg), &
    size(arg, kind=C_SIZE_T))
  ! splicer end function.vector_iota_inout_alloc
end subroutine vector_iota_inout_alloc
```

`inta` is `intent(inout)`, so it will NOT be deallocated upon entry to `vector_iota_inout_alloc`. Fortran usage:

```
call vector_iota_inout_alloc(inta)
call assert_true(allocated(inta))
call assert_equals(10, size(inta))
call assert_true( all(inta == [1,2,3,4,5,11,12,13,14,15]), &
  "vector_iota_inout_alloc value")
deallocate(inta)
```

## 26.10 Void Pointers

### 26.10.1 passAssumedType

C library function in `clibrary.c`:

```
int passAssumedType(void *arg)
{
  strncpy(last_function_called, "passAssumedType", MAXLAST);
  return *(int *) arg;
}
```

`clibrary.yaml`:

---

```
- decl: int passAssumedType(void *arg+assumedtype)
```

Fortran calls C via the following interface:

```
interface
  function pass_assumed_type(arg) &
    result(SHT_rv) &
    bind(C, name="passAssumedType")
    use iso_c_binding, only : C_INT
    implicit none
    type(*) :: arg
    integer(C_INT) :: SHT_rv
  end function pass_assumed_type
end interface
```

Fortran usage:

```
use iso_c_binding, only : C_INT
integer(C_INT) rv_int
rv_int = pass_assumed_type(23_C_INT)
```

As a reminder, 23\_C\_INT creates an integer(C\_INT) constant.

---

**Note:** Assumed-type was introduced in Fortran 2018.

---

## 26.10.2 passAssumedTypeDim

C library function in `clibrary.c`:

```
void passAssumedTypeDim(void *arg)
{
    strncpy(last_function_called, "passAssumedTypeDim", MAXLAST);
}
```

`clibrary.yaml`:

```
- decl: int passAssumedTypeDim(void *arg+assumedtype+rank(1))
```

Fortran calls C via the following interface:

```
interface
  subroutine pass_assumed_type_dim(arg) &
    bind(C, name="passAssumedTypeDim")
    implicit none
    type(*) :: arg(*)
  end subroutine pass_assumed_type_dim
end interface
```

Example usage:

```
use iso_c_binding, only : C_INT, C_DOUBLE
integer(C_INT) int_array(10)
real(C_DOUBLE) double_array(2,5)
call pass_assumed_type_dim(int_array)
call pass_assumed_type_dim(double_array)
```

---

**Note:** Assumed-type was introduced in Fortran 2018.

---

### 26.10.3 passVoidStarStar

C library function in `clibrary.c`:

```
void passVoidStarStar(void *in, void **out)
{
    strncpy(last_function_called, "passVoidStarStar", MAXLAST);
    *out = in;
}
```

`clibrary.yaml`:

```
- decl: void passVoidStarStar(void *in+intent(in), void **out+intent(out))
```

Fortran calls C via the following interface:

```
interface
  subroutine pass_void_star_star(in, out) &
    bind(C, name="passVoidStarStar")
    use iso_c_binding, only : C_PTR
    implicit none
    type(C_PTR), value, intent(IN) :: in
    type(C_PTR), intent(OUT) :: out
  end subroutine pass_void_star_star
end interface
```

Example usage:

```
use iso_c_binding, only : C_INT, C_NULL_PTR, c_associated
integer(C_INT) int_var
cptr1 = c_loc(int_var)
cptr2 = C_NULL_PTR
call pass_void_star_star(cptr1, cptr2)
call assert_true(c_associated(cptr1, cptr2))
```

## 26.11 Function Pointers

### 26.11.1 callback1

C++ library function in `tutorial.cpp`:

```
int callback1(int in, int (*incr)(int))
{
    return incr(in);
}
```

`tutorial.yaml`:

```
- decl: int callback1(int in, int (*incr)(int));
```

The C wrapper:

```
int TUT_callback1(int in, int (*incr)(int))
{
    // splicer begin function.callback1
    int SHC_rv = tutorial::callback1(in, incr);
    return SHC_rv;
    // splicer end function.callback1
}
```

Creates the abstract interface:

```
abstract interface
  function callback1_incr(arg0) bind(C)
    use iso_c_binding, only : C_INT
    implicit none
    integer(C_INT), value :: arg0
    integer(C_INT) :: callback1_incr
  end function callback1_incr
end interface
```

Fortran calls C via the following interface:

```
interface
  function callback1(in, incr) &
    result(SHT_rv) &
    bind(C, name="TUT_callback1")
    use iso_c_binding, only : C_INT
    import :: callback1_incr
    implicit none
    integer(C_INT), value, intent(IN) :: in
    procedure(callback1_incr) :: incr
    integer(C_INT) :: SHT_rv
  end function callback1
end interface
```

Fortran usage:

```
module worker
  use iso_c_binding
contains
  subroutine userincr(i) bind(C)
    integer(C_INT), value :: i
    ! do work of callback
  end subroutine user

  subroutine work
    call callback1(1, userincr)
  end subroutine work
end module worker
```

## 26.11.2 callback1c

C library function in `clibrary.c`:

```
void callback1(int type, void (*incr)(void))
{
    // Use type to decide how to call incr
}
```

clibrary.yaml:

```
- decl: int callback1(int type, void (*incr)()+external)
```

Creates the abstract interface:

```
abstract interface
  subroutine callback1_incr() bind(C)
    implicit none
  end subroutine callback1_incr
end interface
```

Fortran calls C via the following interface:

```
interface
  subroutine c_callback1(type, incr) &
    bind(C, name="callback1")
    use iso_c_binding, only : C_INT
    import :: callback1_incr
    implicit none
    integer(C_INT), value, intent(IN) :: type
    procedure(callback1_incr) :: incr
  end subroutine c_callback1
end interface
```

The Fortran wrapper. By using external no abstract interface is used:

```
subroutine callback1(type, incr)
  use iso_c_binding, only : C_INT
  integer(C_INT), value, intent(IN) :: type
  external :: incr
  ! splicer begin function.callback1
  call c_callback1(type, incr)
  ! splicer end function.callback1
end subroutine callback1
```

Fortran usage:

```
module worker
  use iso_c_binding
contains
  subroutine userincr_int(i) bind(C)
    integer(C_INT), value :: i
    ! do work of callback
  end subroutine user_int

  subroutine userincr_double(i) bind(C)
    real(C_DOUBLE), value :: i
    ! do work of callback
  end subroutine user_int

  subroutine work
```

(continues on next page)



(continued from previous page)

```

    call callback1c(1, userincr_int)
    call callback1c(1, userincr_double)
end subroutine work
end module worker

```

## 26.12 Struct

Struct creating is described in *Fortran Structs*.

### 26.12.1 passStruct1

C library function in `struct.c`:

```

int passStruct1(const Cstruct1 *s1)
{
    strncpy(last_function_called, "passStruct1", MAXLAST);
    return s1->ifield;
}

```

`struct.yaml`:

```
- decl: int passStruct1(Cstruct1 *s1)
```

Fortran calls C via the following interface:

```

interface
    function pass_struct1(arg) &
        result(SHT_rv) &
        bind(C, name="passStruct1")
    use iso_c_binding, only : C_INT
    import :: cstruct1
    implicit none
    type(cstruct1), intent(IN) :: arg
    integer(C_INT) :: SHT_rv
end function pass_struct1
end interface

```

Fortran usage:

```

type(cstruct1) str1
str1%ifield = 12
str1%dfield = 12.6
call assert_equals(12, pass_struct1(str1), "passStruct1")

```

### 26.12.2 passStructByValue

C library function in `struct.c`:

```

int passStructByValue(Cstruct1 arg)
{
    int rv = arg.ifield * 2;

```

(continues on next page)

(continued from previous page)

```
// Caller will not see changes.
arg.ifield += 1;
return rv;
}
```

struct.yaml:

```
- decl: double passStructByValue(struct1 arg)
```

Fortran calls C via the following interface:

```
interface
  function pass_struct_by_value(arg) &
    result(SHT_rv) &
    bind(C, name="passStructByValue")
  use iso_c_binding, only : C_INT
  import :: cstruct1
  implicit none
  type(cstruct1), value, intent(IN) :: arg
  integer(C_INT) :: SHT_rv
end function pass_struct_by_value
end interface
```

Fortran usage:

```
type(cstruct1) str1
str1%ifield = 2_C_INT
str1%dfield = 2.0_C_DOUBLE
rvi = pass_struct_by_value(str1)
call assert_equals(4, rvi, "pass_struct_by_value")
! Make sure str1 was passed by value.
call assert_equals(2_C_INT, str1%ifield, "pass_struct_by_value ifield")
call assert_equals(2.0_C_DOUBLE, str1%dfield, "pass_struct_by_value dfield")
```

## 26.13 Class Type

### 26.13.1 constructor and destructor

The C++ header file from `classes.hpp`.

```
class Class1
{
public:
  int m_flag;
  int m_test;
  Class1() : m_flag(0), m_test(0) {};
  Class1(int flag) : m_flag(flag), m_test(0) {};
};
```

classes.yaml:

```
declarations:
- decl: class Class1
```

(continues on next page)

(continued from previous page)

```

declarations:
- decl: Class1()
  format:
    function_suffix: _default
- decl: Class1(int flag)
  format:
    function_suffix: _flag
- decl: ~Class1() +name(delete)

```

A C wrapper function is created for each constructor and the destructor.

The C wrappers:

```

CLA_Class1 * CLA_Class1_ctor_default(CLA_Class1 * SHC_rv)
{
    // splicer begin class.Class1.method.ctor_default
    classes::Class1 *SHCXX_rv = new classes::Class1();
    SHC_rv->addr = static_cast<void *>(SHCXX_rv);
    SHC_rv->idtor = 1;
    return SHC_rv;
    // splicer end class.Class1.method.ctor_default
}

```

```

CLA_Class1 * CLA_Class1_ctor_flag(int flag, CLA_Class1 * SHC_rv)
{
    // splicer begin class.Class1.method.ctor_flag
    classes::Class1 *SHCXX_rv = new classes::Class1(flag);
    SHC_rv->addr = static_cast<void *>(SHCXX_rv);
    SHC_rv->idtor = 1;
    return SHC_rv;
    // splicer end class.Class1.method.ctor_flag
}

```

```

void CLA_Class1_delete(CLA_Class1 * self)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
    // splicer begin class.Class1.method.delete
    delete SH_this;
    self->addr = nullptr;
    // splicer end class.Class1.method.delete
}

```

The corresponding Fortran interfaces:

```

interface
  function c_class1_ctor_default(SHT_rv) &
    result (SHT_prv) &
    bind(C, name="CLA_Class1_ctor_default")
    use iso_c_binding, only : C_PTR
    import :: CLA_SHROUD_capsule_data
    implicit none
    type(CLA_SHROUD_capsule_data), intent(OUT) :: SHT_rv
    type(C_PTR) :: SHT_prv
  end function c_class1_ctor_default
end interface

```

```
interface
  function c_class1_ctor_flag(flag, SHT_rv) &
    result(SHT_prv) &
    bind(C, name="CLA_Class1_ctor_flag")
  use iso_c_binding, only : C_INT, C_PTR
  import :: CLA_SHROUD_capsule_data
  implicit none
  integer(C_INT), value, intent(IN) :: flag
  type(CLA_SHROUD_capsule_data), intent(OUT) :: SHT_rv
  type(C_PTR) :: SHT_prv
end function c_class1_ctor_flag
end interface
```

```
interface
  subroutine c_class1_delete(self) &
    bind(C, name="CLA_Class1_delete")
  import :: CLA_SHROUD_capsule_data
  implicit none
  type(CLA_SHROUD_capsule_data), intent(INOUT) :: self
end subroutine c_class1_delete
end interface
```

And the Fortran wrappers:

```
function class1_ctor_default() &
  result(SHT_rv)
  use iso_c_binding, only : C_PTR
  type(class1) :: SHT_rv
  type(C_PTR) :: SHT_prv
  ! splicer begin class.Class1.method.ctor_default
  SHT_prv = c_class1_ctor_default(SHT_rv%cxxmem)
  ! splicer end class.Class1.method.ctor_default
end function class1_ctor_default
```

```
function class1_ctor_flag(flag) &
  result(SHT_rv)
  use iso_c_binding, only : C_INT, C_PTR
  integer(C_INT), value, intent(IN) :: flag
  type(class1) :: SHT_rv
  type(C_PTR) :: SHT_prv
  ! splicer begin class.Class1.method.ctor_flag
  SHT_prv = c_class1_ctor_flag(flag, SHT_rv%cxxmem)
  ! splicer end class.Class1.method.ctor_flag
end function class1_ctor_flag
```

```
subroutine class1_delete(obj)
  class(class1) :: obj
  ! splicer begin class.Class1.method.delete
  call c_class1_delete(obj%cxxmem)
  ! splicer end class.Class1.method.delete
end subroutine class1_delete
```

The Fortran shadow class adds the type-bound method for the destructor:

```
type, bind(C) :: SHROUD_class1_capsule
  type(C_PTR) :: addr = C_NULL_PTR ! address of C++ memory
```

(continues on next page)

(continued from previous page)

```

    integer(C_INT) :: idtor = 0      ! index of destructor
end type SHROUD_class1_capsule

type class1
    type(SHROUD_class1_capsule) :: cxxmem
contains
    procedure :: delete => class1_delete
end type class1

```

The constructors are not type-bound procedures. But they are combined into a generic interface.

```

interface class1
    module procedure class1_ctor_default
    module procedure class1_ctor_flag
end interface class1

```

A class instance is created and destroy from Fortran as:

```

use classes_mod
type(class1) obj

obj = class1()
call obj%delete

```

Corresponding C++ code:

```

include <classes.hpp>

classes::Class1 * obj = new classes::Class1;

delete obj;

```

## 26.13.2 Getter and Setter

The C++ header file from `classes.hpp`.

```

class Class1
{
public:
    int m_flag;
    int m_test;
};

```

`classes.yaml`:

```

declarations:
- decl: class Class1
  declarations:
  - decl: int m_flag +readonly;
  - decl: int m_test +name(test);

```

A C wrapper function is created for each getter and setter. If the *readonly* attribute is added, then only a getter is created. In this case `m_` is a convention used to designate member variables. The Fortran attribute is renamed as **test** to avoid cluttering the Fortran API with this convention.

The C wrappers:

```
int CLA_Class1_get_m_flag(CLA_Class1 * self)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
    // splicer begin class.Class1.method.get_m_flag
    // skip call c_getter
    return SH_this->m_flag;
    // splicer end class.Class1.method.get_m_flag
}
```

```
int CLA_Class1_get_test(CLA_Class1 * self)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
    // splicer begin class.Class1.method.get_test
    // skip call c_getter
    return SH_this->m_test;
    // splicer end class.Class1.method.get_test
}
```

```
void CLA_Class1_set_test(CLA_Class1 * self, int val)
{
    classes::Class1 *SH_this = static_cast<classes::Class1 *>
        (self->addr);
    // splicer begin class.Class1.method.set_test
    // skip call c_setter
    SH_this->m_test = val;
    // splicer end class.Class1.method.set_test
}
```

The corresponding Fortran interfaces:

```
interface
    function c_class1_get_m_flag(self) &
        result (SHT_rv) &
        bind(C, name="CLA_Class1_get_m_flag")
        use iso_c_binding, only : C_INT
        import :: CLA_SHROUD_capsule_data
        implicit none
        type(CLA_SHROUD_capsule_data), intent(IN) :: self
        integer(C_INT) :: SHT_rv
    end function c_class1_get_m_flag
end interface
```

```
interface
    function c_class1_get_test(self) &
        result (SHT_rv) &
        bind(C, name="CLA_Class1_get_test")
        use iso_c_binding, only : C_INT
        import :: CLA_SHROUD_capsule_data
        implicit none
        type(CLA_SHROUD_capsule_data), intent(IN) :: self
        integer(C_INT) :: SHT_rv
    end function c_class1_get_test
end interface
```

```

interface
  subroutine c_class1_set_test(self, val) &
    bind(C, name="CLA_Class1_set_test")
    use iso_c_binding, only : C_INT
    import :: CLA_SHROUD_capsule_data
    implicit none
    type(CLA_SHROUD_capsule_data), intent(IN) :: self
    integer(C_INT), value, intent(IN) :: val
  end subroutine c_class1_set_test
end interface

```

And the Fortran wrappers:

```

function class1_get_m_flag(obj) &
  result(SHT_rv)
  use iso_c_binding, only : C_INT
  class(class1) :: obj
  integer(C_INT) :: SHT_rv
  ! splicer begin class.Class1.method.get_m_flag
  SHT_rv = c_class1_get_m_flag(obj%cxxmem)
  ! splicer end class.Class1.method.get_m_flag
end function class1_get_m_flag

```

```

function class1_get_test(obj) &
  result(SHT_rv)
  use iso_c_binding, only : C_INT
  class(class1) :: obj
  integer(C_INT) :: SHT_rv
  ! splicer begin class.Class1.method.get_test
  SHT_rv = c_class1_get_test(obj%cxxmem)
  ! splicer end class.Class1.method.get_test
end function class1_get_test

```

```

subroutine class1_set_test(obj, val)
  use iso_c_binding, only : C_INT
  class(class1) :: obj
  integer(C_INT), value, intent(IN) :: val
  ! splicer begin class.Class1.method.set_test
  call c_class1_set_test(obj%cxxmem, val)
  ! splicer end class.Class1.method.set_test
end subroutine class1_set_test

```

The Fortran shadow class adds the type-bound methods:

```

type class1
  type(SHROUD_class1_capsule) :: cxxmem
contains
  procedure :: get_m_flag => class1_get_m_flag
  procedure :: get_test  => class1_get_test
  procedure :: set_test  => class1_set_test
end type class1

```

The class variables can be used as:

```

use classes_mod
type(class1) obj

```

(continues on next page)

(continued from previous page)

```
integer iflag

obj = class1()
call obj%set_test(4)
iflag = obj%get_test()
```

Corresponding C++ code:

```
include <classes.hpp>
classes::Class1 obj = new * classes::Class1;
obj->m_test = 4;
int iflag = obj->m_test;
```

### 26.13.3 Struct as a Class

While C does not support object-oriented programming directly, it can be emulated by using structs. The ‘base class’ struct is `Cstruct_as_class`. It is subclassed by `Cstruct_as_subclass` which explicitly duplicates the members of `Cstruct_as_class`. The C header file from `struct.h`.

```
struct Cstruct_as_class {
    int x1;
    int y1;
};
typedef struct Cstruct_as_class Cstruct_as_class;

/* The first members match Cstruct_as_class */
struct Cstruct_as_subclass {
    int x1;
    int y1;
    int z1;
};
typedef struct Cstruct_as_subclass Cstruct_as_subclass;
```

The C ‘constructor’ returns a pointer to an instance of the object.

```
Cstruct_as_class *Create_Cstruct_as_class(void);
Cstruct_as_class *Create_Cstruct_as_class_args(int x, int y);
```

The ‘methods’ pass an instance of the class as an explicit *this* object.

```
int Cstruct_as_class_sum(const Cstruct_as_class *point);
```

The methods are wrapped in `classes.yaml`:

```
declarations:
- decl: struct Cstruct_as_class {
      int x1;
      int y1;
    };
  options:
    wrap_struct_as: class

- decl: Cstruct_as_class *Create_Cstruct_as_class(void)
  options:
    class_ctor: Cstruct_as_class
```

(continues on next page)



(continued from previous page)

```

- decl: Cstruct_as_class *Create_Cstruct_as_class_args(int x, int y)
options:
  class_ctor: Cstruct_as_class

- decl: int Cstruct_as_class_sum(const Cstruct_as_class *point +pass)
options:
  class_method: Cstruct_as_class
format:
  F_name_function: sum

- decl: struct Cstruct_as_subclass {
    int x1;
    int y1;
    int z1;
  };
options:
  wrap_struct_as: class
  class_baseclass: Cstruct_as_class
- decl: Cstruct_as_subclass *Create_Cstruct_as_subclass_args(int x, int y, int z)
options:
  wrap_python: False
  class_ctor: Cstruct_as_subclass

```

This uses several options to creates the class features for the struct: *wrap\_struct\_as*, *class\_ctor*, *class\_method*.

```

type cstruct_as_class
  type(STR_SHROUD_capsule_data) :: cxxmem
  ! splicer begin class.Cstruct_as_class.component_part
  ! splicer end class.Cstruct_as_class.component_part
contains
  procedure :: get_x1 => cstruct_as_class_get_x1
  procedure :: set_x1 => cstruct_as_class_set_x1
  procedure :: get_y1 => cstruct_as_class_get_y1
  procedure :: set_y1 => cstruct_as_class_set_y1
  procedure :: sum => cstruct_as_class_sum
  ! splicer begin class.Cstruct_as_class.type_bound_procedure_part
  ! splicer end class.Cstruct_as_class.type_bound_procedure_part
end type cstruct_as_class

```

The subclass is created using the Fortran EXTENDS keyword. No additional members are added. The `cxxmem` field from `cstruct_as_class` will now point to an instance of the C struct `Cstruct_as_subclass`.

```

type, extends(cstruct_as_class) :: cstruct_as_subclass
  ! splicer begin class.Cstruct_as_subclass.component_part
  ! splicer end class.Cstruct_as_subclass.component_part
contains
  procedure :: get_x1 => cstruct_as_subclass_get_x1
  procedure :: set_x1 => cstruct_as_subclass_set_x1
  procedure :: get_y1 => cstruct_as_subclass_get_y1
  procedure :: set_y1 => cstruct_as_subclass_set_y1
  procedure :: get_z1 => cstruct_as_subclass_get_z1
  procedure :: set_z1 => cstruct_as_subclass_set_z1
  ! splicer begin class.Cstruct_as_subclass.type_bound_procedure_part
  ! splicer end class.Cstruct_as_subclass.type_bound_procedure_part
end type cstruct_as_subclass

```

The C wrapper to construct the struct-as-class. It calls the C function and fills in the fields for the shadow struct.

```
STR_Cstruct_as_class * STR_Create_Cstruct_as_class(  
    STR_Cstruct_as_class * SHC_rv)  
{  
    // splicer begin function.Create_Cstruct_as_class  
    Cstruct_as_class * SHCXX_rv = Create_Cstruct_as_class();  
    SHC_rv->addr = SHCXX_rv;  
    SHC_rv->idtor = 0;  
    return SHC_rv;  
    // splicer end function.Create_Cstruct_as_class  
}
```

A Fortran generic interface is created for the class:

```
interface cstruct_as_class  
    module procedure create_cstruct_as_class  
    module procedure create_cstruct_as_class_args  
end interface cstruct_as_class
```

And the Fortran constructor call the C wrapper function.

```
function create_cstruct_as_class() &  
    result(SHT_rv)  
    use iso_c_binding, only : C_PTR  
    type(cstruct_as_class) :: SHT_rv  
    type(C_PTR) :: SHT_prv  
    ! splicer begin function.create_cstruct_as_class  
    SHT_prv = c_create_cstruct_as_class(SHT_rv%cxxmem)  
    ! splicer end function.create_cstruct_as_class  
end function create_cstruct_as_class
```

The class can be used as:

```
type(cstruct_as_class) point1, point2  
type(cstruct_as_subclass) subpoint1  
  
call set_case_name("test_struct_class")  
  
! F_name_associated is blank so the associated function is not created.  
! Instead look at pointer directly.  
! call assert_false(point1%associated())  
call assert_false(c_associated(point1%cxxmem%addr))  
  
point1 = Cstruct_as_class()  
call assert_equals(0, point1%get_x1())  
call assert_equals(0, point1%get_y1())  
  
point2 = Cstruct_as_class(1, 2)  
call assert_equals(1, point2%get_x1())  
call assert_equals(2, point2%get_y1())  
  
call assert_equals(3, cstruct_as_class_sum(point2))  
call assert_equals(3, point2%sum())  
  
subpoint1 = Cstruct_as_subclass(1, 2, 3)  
call assert_equals(1, subpoint1%get_x1())  
call assert_equals(2, subpoint1%get_y1())  
call assert_equals(3, subpoint1%get_z1())  
call assert_equals(3, subpoint1%sum())
```

## 26.14 Default Value Arguments

The default values are provided in the function declaration.

C++ library function in `tutorial.cpp`:

```
double UseDefaultArguments(double arg1 = 3.1415, bool arg2 = true);
```

`tutorial.yaml`:

```
- decl: double UseDefaultArguments(double arg1 = 3.1415, bool arg2 = true)
  default_arg_suffix:
  -
  - _arg1
  - _arg1_arg2
```

A C++ wrapper is created which calls the C++ function with no arguments with default values and then adds a wrapper with an explicit argument for each default value argument. In this case, three wrappers are created. Since the C++ compiler provides the default value, it is necessary to create each wrapper.

`wrapTutorial.cpp`:

```
double TUT_UseDefaultArguments(void)
{
    // splicer begin function.UseDefaultArguments
    double SHC_rv = tutorial::UseDefaultArguments();
    return SHC_rv;
    // splicer end function.UseDefaultArguments
}
```

```
double TUT_UseDefaultArguments_arg1(double arg1)
{
    // splicer begin function.UseDefaultArguments_arg1
    double SHC_rv = tutorial::UseDefaultArguments(arg1);
    return SHC_rv;
    // splicer end function.UseDefaultArguments_arg1
}
```

```
double TUT_UseDefaultArguments_arg1_arg2(double arg1, bool arg2)
{
    // splicer begin function.UseDefaultArguments_arg1_arg2
    double SHC_rv = tutorial::UseDefaultArguments(arg1, arg2);
    return SHC_rv;
    // splicer end function.UseDefaultArguments_arg1_arg2
}
```

This creates three corresponding Fortran interfaces:

```
interface
  function c_use_default_arguments() &
    result (SHT_rv) &
    bind(C, name="TUT_UseDefaultArguments")
    use iso_c_binding, only : C_DOUBLE
    implicit none
    real(C_DOUBLE) :: SHT_rv
  end function c_use_default_arguments
end interface
```

```
interface
  function c_use_default_arguments_arg1(arg1) &
    result(SHT_rv) &
    bind(C, name="TUT_UseDefaultArguments_arg1")
    use iso_c_binding, only : C_DOUBLE
    implicit none
    real(C_DOUBLE), value, intent(IN) :: arg1
    real(C_DOUBLE) :: SHT_rv
  end function c_use_default_arguments_arg1
end interface
```

```
interface
  function c_use_default_arguments_arg1_arg2(arg1, arg2) &
    result(SHT_rv) &
    bind(C, name="TUT_UseDefaultArguments_arg1_arg2")
    use iso_c_binding, only : C_BOOL, C_DOUBLE
    implicit none
    real(C_DOUBLE), value, intent(IN) :: arg1
    logical(C_BOOL), value, intent(IN) :: arg2
    real(C_DOUBLE) :: SHT_rv
  end function c_use_default_arguments_arg1_arg2
end interface
```

In many case the interfaces would be enough to call the routines. However, in order to have a generic interface, there must be explicit Fortran wrappers:

```
function use_default_arguments() &
  result(SHT_rv)
  use iso_c_binding, only : C_DOUBLE
  real(C_DOUBLE) :: SHT_rv
  ! splicer begin function.use_default_arguments
  SHT_rv = c_use_default_arguments()
  ! splicer end function.use_default_arguments
end function use_default_arguments
```

```
function use_default_arguments_arg1(arg1) &
  result(SHT_rv)
  use iso_c_binding, only : C_DOUBLE
  real(C_DOUBLE), value, intent(IN) :: arg1
  real(C_DOUBLE) :: SHT_rv
  ! splicer begin function.use_default_arguments_arg1
  SHT_rv = c_use_default_arguments_arg1(arg1)
  ! splicer end function.use_default_arguments_arg1
end function use_default_arguments_arg1
```

```
function use_default_arguments_arg1_arg2(arg1, arg2) &
  result(SHT_rv)
  use iso_c_binding, only : C_BOOL, C_DOUBLE
  real(C_DOUBLE), value, intent(IN) :: arg1
  logical, value, intent(IN) :: arg2
  real(C_DOUBLE) :: SHT_rv
  ! splicer begin function.use_default_arguments_arg1_arg2
  logical(C_BOOL) :: SHT_arg2_cxx
  SHT_arg2_cxx = arg2 ! coerce to C_BOOL
  SHT_rv = c_use_default_arguments_arg1_arg2(arg1, SHT_arg2_cxx)
  ! splicer end function.use_default_arguments_arg1_arg2
```

(continues on next page)

(continued from previous page)

```
end function use_default_arguments_arg1_arg2
```

The Fortran generic interface adds the ability to call any of the functions by the C++ function name:

```
interface use_default_arguments
  module procedure use_default_arguments
  module procedure use_default_arguments_arg1
  module procedure use_default_arguments_arg1_arg2
end interface use_default_arguments
```

Usage:

```
real(C_DOUBLE) rv
rv = use_default_arguments()
rv = use_default_arguments(1.d0)
rv = use_default_arguments(1.d0, .false.)
```

## 26.15 Generic Real

C library function in `clibrary.c`:

```
void GenericReal(double arg)
{
    global_double = arg;
    return;
}
```

`generic.yaml`:

```
- decl: void GenericReal(double arg)
  fortran_generic:
  - decl: (float arg)
    function_suffix: float
  - decl: (double arg)
    function_suffix: double
```

Fortran calls C via the following interface:

```
interface
  subroutine c_generic_real(arg) &
    bind(C, name="GenericReal")
    use iso_c_binding, only : C_DOUBLE
    implicit none
    real(C_DOUBLE), value, intent(IN) :: arg
  end subroutine c_generic_real
end interface
```

There is a single interface since there is a single C function. A generic interface is created for each declaration in the `fortran_generic` block.

```
interface generic_real
  module procedure generic_real_float
  module procedure generic_real_double
end interface generic_real
```

A Fortran wrapper is created for each declaration in the *fortran\_generic* block. The argument is explicitly converted to a `C_DOUBLE` before calling the C function in `generic_real_float`. There is no conversion necessary in `generic_real_double`.

```
subroutine generic_real_float(arg)
  use iso_c_binding, only : C_DOUBLE, C_FLOAT
  real(C_FLOAT), value, intent(IN) :: arg
  ! splicer begin function.generic_real_float
  call c_generic_real(real(arg, C_DOUBLE))
  ! splicer end function.generic_real_float
end subroutine generic_real_float
```

```
subroutine generic_real_double(arg)
  use iso_c_binding, only : C_DOUBLE
  real(C_DOUBLE), value, intent(IN) :: arg
  ! splicer begin function.generic_real_double
  call c_generic_real(arg)
  ! splicer end function.generic_real_double
end subroutine generic_real_double
```

The function can be called via the generic interface with either type. If the specific function is called, the correct type must be passed.

```
call generic_real(0.0)
call generic_real(0.0d0)

call generic_real_float(0.0)
call generic_real_double(0.0d0)
```

In C, the compiler will promote the argument.

```
GenericReal(0.0f);
GenericReal(0.0);
```

---

Numpy Struct Descriptor

---

struct.yaml:

```
- decl: struct Cstruct1 {  
    int ifield;  
    double dfield;  
};
```

```
// Create PyArray_Descr for Cstruct1  
static PyArray_Descr *PY_Cstruct1_create_array_descr(void)  
{  
    int ierr;  
    PyObject *obj = NULL;  
    PyObject * lnames = NULL;  
    PyObject * ldescr = NULL;  
    PyObject * dict = NULL;  
    PyArray_Descr *dtype = NULL;  
  
    lnames = PyList_New(2);  
    if (lnames == NULL) goto fail;  
    ldescr = PyList_New(2);  
    if (ldescr == NULL) goto fail;  
  
    // ifield  
    obj = PyString_FromString("ifield");  
    if (obj == NULL) goto fail;  
    PyList_SET_ITEM(lnames, 0, obj);  
    obj = (PyObject *) PyArray_DescrFromType(NPY_INT);  
    if (obj == NULL) goto fail;  
    PyList_SET_ITEM(ldescr, 0, obj);  
  
    // dfield  
    obj = PyString_FromString("dfield");  
    if (obj == NULL) goto fail;  
    PyList_SET_ITEM(lnames, 1, obj);
```

(continues on next page)

(continued from previous page)

```
obj = (PyObject *) PyArray_DescrFromType(NPY_DOUBLE);
if (obj == NULL) goto fail;
PyList_SET_ITEM(ldescr, 1, obj);
obj = NULL;

dict = PyDict_New();
if (dict == NULL) goto fail;
ierr = PyDict_SetItemString(dict, "names", lnames);
if (ierr == -1) goto fail;
lnames = NULL;
ierr = PyDict_SetItemString(dict, "formats", ldescr);
if (ierr == -1) goto fail;
ldescr = NULL;
ierr = PyArray_DescrAlignConverter(dict, &dtype);
if (ierr == 0) goto fail;
return dtype;
fail:
Py_XDECREF(obj);
if (lnames != NULL) {
    for (int i=0; i < 2; i++) {
        Py_XDECREF(PyList_GET_ITEM(lnames, i));
    }
    Py_DECREF(lnames);
}
if (ldescr != NULL) {
    for (int i=0; i < 2; i++) {
        Py_XDECREF(PyList_GET_ITEM(ldescr, i));
    }
    Py_DECREF(ldescr);
}
Py_XDECREF(dict);
Py_XDECREF(dtype);
return NULL;
}
```



#### bufferify

The process of creating a buffer. The C wrapper may create an additional function for each wrapped function. This function has additional arguments needed for the Fortran interface to pass the metadata of the argument such as the size.

#### C wrapper

The wrapper with a C API. When wrapping C++, the C wrapper will be C++ code but functions will use `extern "C"` block to allow them to be called from C and Fortran using the `bind(C)` keyword.

#### capsule

Holds a pointer to a C++ class instance and some memory management flags. The name was inspired by the Python `PyCapsule` type.

#### Fortran wrapper

Fortran functions which call the C wrapper functions.

#### library

A collection of C++ declarations wrapped at the same time. This creates a Fortran or Python module.

#### native type

Integer or real of any size.

#### shadow class

A Fortran derived type which contains a capsule and type-bound functions to provide a Fortran object-oriented interface similar to a C++ class. Similar to a *PyObject* struct in Python.

#### splicer

A block of code which implements the wrapper. It is generated by Shroud but can be replaced with a user provided implementation. A splicer also provides a place to insert code into a generated file a well defined places. This allows user provided code to be preserved when regenerating wrappers.



## CHAPTER 29

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



---

## Bibliography

---

[Python\_Format] <https://docs.python.org/2/library/string.html#format-string-syntax>

[Python\_Refcount] <https://docs.python.org/3/c-api/intro.html#reference-count-details>

[yaml] [yaml.org](http://yaml.org)

[blog1] <http://blog.enthought.com/python/numpy-arrays-with-pre-allocated-memory>

[blog2] <http://blog.enthought.com/python/numpy/simplified-creation-of-numpy-arrays-from-pre-allocated-memory>