Foreign Fortran Documentation

Danny Hermes

Oct 03, 2018

Contents

We'll build a *shared library* in Fortran (90 and later) and show how to make **foreign calls** to the library from other programming languages:

- *Fortran*
- *C*
- *C++*
- *Python*
- *Go*

CHAPTER 1

Shared Library

We define a simple module that contains a particular subset of Fortran features. Many of them are exported with unmangled names (via `bind(c)`) so the ABI can be used in a predictable way independent of the compiler or platform.

1.1 Public Interface

int **KNOB**

The first public symbol is a mutable global:

Fortran Implementation:

```
integer(c_int) :: KNOB = 1337
```

KNOB does not have a bound name (i.e. it may be mangled) and it is expected to be accessed via a public getter and setter.

void **turn_knob** (int **new_value*)

Fortran Implementation:

```
! NOTE: This is missing ``bind(c, name='view_knob')`` because
!       the ``f2py`` parser has issues for some reason.
pure function view_knob() result(knob_value)
  integer(c_int) :: knob_value

  knob_value = KNOB

end function view_knob

subroutine turn_knob(new_value) bind(c, name='turn_knob')
  integer(c_int), intent(in) :: new_value

  KNOB = new_value

end subroutine turn_knob
```

C Signature:

```
void turn_knob(int *new_value);
```

As noted in the remark, the `view_knob()` getter also does not have a bound name because of the way a `function` (vs. a `subroutine`) is parsed by `f2py` (an issue has been filed).

In addition to a mutable global, there are two user defined types exposed and unmangled, hence each acts as a C `struct`.

`UserDefined`

Fortran Implementation:

```
type, bind(c) :: UserDefined
   real(c_double) :: buzz
   real(c_double) :: broken
   integer(c_int) :: how_many
end type UserDefined
```

C Signature:

```
typedef struct UserDefined {
  double buzz;
  double broken;
  int how_many;
} UserDefined;
```

double **`buzz`**

double **`broken`**

int **`how_many`**

`DataContainer`

Fortran Implementation:

```
type, bind(c) :: DataContainer
   real(c_double) :: data(4, 2)
end type DataContainer
```

C Signature:

```
typedef struct DataContainer { double data[8]; } DataContainer;
```

double[8] **`data`**

void **`foo`**(double *bar*, double *baz*, double **quux*)

The first subroutine exported by the public interface is an implementation of `f(x, y) = x + 3.75 y`.

Fortran Implementation:

```
subroutine foo(bar, baz, quux) bind(c, name='foo')
  real(c_double), intent(in), value :: bar, baz
  real(c_double), intent(out) :: quux

  quux = bar + 3.75_dp * baz

end subroutine foo
```

C Signature:

```
void foo(double bar, double baz, double *quux);
```

It accepts the inputs by value. Since pass-by-reference is the default behavior, an equivalent method is provided (though not as part of the unmangled ABI):

```
subroutine foo_by_ref(bar, baz, quux)
  real(dp), intent(in) :: bar, baz
  real(dp), intent(out) :: quux

  call foo(bar, baz, quux)

end subroutine foo_by_ref
```

void **foo_array**(int **size*, double **val*, double **two_val*)

Next, we define a method that accepts a variable size array and places twice the values of that array in the return value:

Fortran Implementation:

```
subroutine foo_array(size_, val, two_val) bind(c, name='foo_array')
  integer(c_int), intent(in) :: size_
  real(c_double), intent(in) :: val(size_, 2)
  real(c_double), intent(out) :: two_val(size_, 2)

  two_val = 2.0_dp * val

end subroutine foo_array
```

C Signature:

```
void foo_array(int *size, double *val, double *two_val);
```

void **make_udf**(double **buzz*, double **broken*, int **how_many*, *UserDefined* **quuz*)

The next subroutine creates an instance of the *UserDefined* data type, but **smuggles** the result out as raw bytes. The total size is `size(buzz) + size(broken) + size(how_many) = 2 c_double + c_int`. This is 20 bytes on most platforms, but as a struct it gets padded to 24 due to word size.

Fortran Implementation:

```
subroutine make_udf(buzz, broken, how_many, as_bytes) bind(c, name='make_udf')
  real(c_double), intent(in) :: buzz, broken
  integer(c_int), intent(in) :: how_many
  character(c_char), intent(out) :: as_bytes(24)
  ! Outside of signature
  type(UserDefined), target :: made_it

  made_it%buzz = buzz
  made_it%broken = broken
  made_it%how_many = how_many

  ! NOTE: We need ``sizeof(as_bytes) == sizeof(made_it)``
  as_bytes = transfer(made_it, as_bytes)

end subroutine make_udf
```

C Signature:

```
void make_udf(double *buzz, double *broken, int *how_many, UserDefined *quuz);
```

This concept of "data smuggling" is necessary for the use of user defined types with `f2py`, since it has no support for them.

void **udf_ptr**(intptr_t **ptr_as_int*)

A related way to smuggle data for use with `f2py` is to allocate memory for the struct and then pass a pointer to that memory as an opaque integer. Once this is done, the Fortran subroutine can convert the integer into a Fortran `pointer` and then write to the memory location owned by the foreign caller:

Fortran Implementation:

```
subroutine udf_ptr(ptr_as_int) bind(c, name='udf_ptr')
  integer(c_intptr_t), intent(in) :: ptr_as_int
  ! Outside of signature
  type(c_ptr) :: made_it_ptr
  type(UserDefined), pointer :: made_it

  made_it_ptr = transfer(ptr_as_int, made_it_ptr)
  call c_f_pointer(made_it_ptr, made_it)

  made_it%buzz = 3.125_dp
  made_it%broken = -10.5_dp
  made_it%how_many = 101

end subroutine udf_ptr
```

C Signature:

```
void udf_ptr(intptr_t *ptr_as_int);
```

This approach is problematic because it is so brittle. The memory must be handled by the caller rather than by Fortran directly. If the subroutine were responsible for the memory, the object would likely be allocated on the stack and the memory location re-used by subsequent calls to the subroutine.

void **make_container**(double **contained*, *DataContainer* **container*)

The next subroutine takes an array as input and sets the `data` attribute of a returned *`DataContainer`* instance as the input. This acts as a check that the operation happens as a data copy rather than a reference copy.

Fortran Implementation:

```
subroutine make_container(contained, container) &
     bind(c, name='make_container')
  real(c_double), intent(in) :: contained(4, 2)
  type(DataContainer), intent(out) :: container

  container%data = contained

end subroutine make_container
```

C Signature:

```
void make_container(double *contained, DataContainer *container);
```

void **just_print**(void)

The *`just_print()`* subroutine simply prints characters to the screen. However, printing requires `libgfortran`, which slightly complicates foreign usage.

Fortran Implementation:

```
subroutine just_print() bind(c, name='just_print')

  print *, "======== BEGIN FORTRAN ========"
  print *, "just_print() was called"
  print *, "========  END  FORTRAN ========"

end subroutine just_print
```

C Signature:

```
void just_print(void);
```

1.2 Object File

For some foreign usage of `example`, we'll directly use a compiled object file. To create `example.o`:

```
$ gfortran \
>   -J fortran/ \
>   -c fortran/example.f90 \
>   -o fortran/example.o
```

1.3 Shared Object

It's more common for foreign usage of native code to be done via a shared object file:

```
$ gfortran \
>   -shared -fPIC \
>   -J fortran/ \
>   fortran/example.f90 \
>   -o fortran/example.so
```

Here, we manually build a "position independent" shared library in the same directory as the source. However, in many cases, native code comes with an installer that puts the library in a standard place, e.g. a symlink to `libatlas` can be found in `/usr/lib/libatlas.so`. Shared object files are typically named `lib{pkg}.so` so that they can be included by the compiler with `-l{pkg}`. The compiler uses a default list of "search directories" when finding such shared libraries.

1.4 References

- Examples of user-defined types
- StackOverflow question about user-defined types
- The `sphinx-fortran` project was started to provide `autodoc` capabilities for Fortran libraries, but it is not actively maintained (as of this writing, August 2018)
- The AutoAPI redesign of `autodoc` will hopefully mature into a capable way of documenting Fortran code (and code from other languages) using Sphinx
- The FORD (FORtran Documentation) project is a modern way to generate documentation for Fortran code, though it is "Yet Another" documentation generator (example documentation)

- The `breathe` project / library seeks to be a bridge Python XML-based doxygen and Sphinx, though in practice the formatting of doxygen produced documentation is not in line with typical Sphinx documentation

CHAPTER 2

Fortran

This is the most vanilla example of all. We are simply calling the Fortran `example` module from a Fortran program.

The module is only "foreign" in the sense that we only interact with the object file `example.o` when creating the executable `fortran_example`:

```
$ gfortran \
>   -o fortran_example \
>   fortran/main.f90 \
>   fortran/example.o
```

However, this still requires the presence of a module file to build the executable

```
$ ls fortran/example.mod
fortran/example.mod
$ rm -f fortran/example.mod
$ gfortran \
>   -o fortran_example \
>   fortran/main.f90 \
>   fortran/example.o
fortran/main.f90:4:6:

   use example, only: &
      1
Fatal Error: Can't open module file 'example.mod' for reading
  at (1): No such file or directory
compilation terminated.
```

Finally, we run `fortran_example` to verify the behavior of several procedures in the public interface:

```
$ ./fortran_example
------------------------------------------------------------
quux = foo(1.000000, 16.000000) = 61.000000
------------------------------------------------------------
quuz = make_udf(1.250000, 5.000000, 1337)
```

(continues on next page)

(continued from previous page)

```
     = UserDefined(1.250000, 5.000000, 1337)
------------------------------------------------------------
foo_array(
    4,
    [[3.000000, 4.500000],
     [1.000000, 1.250000],
     [9.000000, 0.000000],
     [-1.000000, 4.000000]],
) =
    [[6.000000, 9.000000],
     [2.000000, 2.500000],
     [18.000000, 0.000000],
     [-2.000000, 8.000000]]
------------------------------------------------------------
ptr_as_int = c_loc(made_it)  ! type(c_ptr)
                             ! integer(c_intptr_t)
                             ! integer(kind=8)
ptr_as_int = 140733727194752  ! 0x7FFF1FD13E80
udf_ptr(ptr_as_int)  ! Set memory in ``made_it``
made_it = UserDefined(3.125000, -10.500000, 101)
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
------------------------------------------------------------
view_knob() = 1337
turn_knob(42)
view_knob() = 42
```

Using the shared library is as simple as declaring the public symbols used:

```
use example, only: &
     dp, foo, foo_array, make_udf, udf_ptr, just_print, &
     view_knob, turn_knob, UserDefined
```

Notice that the `view_knob()` subroutine is in the public **Fortran** interface even though it doesn't have a bound name in the ABI.

CHAPTER 3

C

This example interacts with the object file `fortran/example.o` when creating the executable `c_example`. Since the example will use *`just_print()`*, it relies on `libgfortran` so we link against it (potentially having used `gfortran -print-search-dirs` to determine where it is located):

```
$ gcc \
>   c/example.c \
>   fortran/example.o \
>   -o c_example \
>   -L/usr/lib/gcc/x86_64-linux-gnu/5 \
>   -L/usr/lib/x86_64-linux-gnu \
>   -lgfortran
```

Alternatively, `gfortran` can consume a `c/example.o` object file created from the C source. This removes the worry of including `libgfortran` though it's somewhat strange to compile an executable from C code with a Fortran compiler.

```
$ gcc \
>   -I c/ \
>   -c c/example.c \
>   -o c/example.o
$ gfortran \
>   c/example.o \
>   fortran/example.o \
>   -o c_example
```

Finally, we run `c_example` to verify the behavior of several procedures in the public interface:

```
$ ./c_example
------------------------------------------------------------
quux = foo(1.000000, 16.000000) = 61.000000
------------------------------------------------------------
quuz = make_udf(1.250000, 5.000000, 1337)
     = UserDefined(1.250000, 5.000000, 1337)
------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
foo_array(
    4,
    [[3.000000, 4.500000],
     [1.000000, 1.250000],
     [9.000000, 0.000000],
     [-1.000000, 4.000000]],
) =
    [[6.000000, 9.000000],
     [2.000000, 2.500000],
     [18.000000, 0.000000],
     [-2.000000, 8.000000]]
------------------------------------------------------------
ptr_as_int = &made_it  // intptr_t
                       // ssize_t
                       // long
ptr_as_int = 140727221075056  // 0x7ffd9c05bc70
udf_ptr(ptr_as_int)  // Set memory in ``made_it``
made_it = UserDefined(3.125000, -10.500000, 101)
------------------------------------------------------------
contained =
  [[0.000000, 4.000000],
   [1.000000, 9.000000],
   [1.000000, 2.000000],
   [3.000000, 1.000000]]
container = make_container(contained)
container.data =
  [[0.000000, 4.000000],
   [1.000000, 9.000000],
   [1.000000, 2.000000],
   [3.000000, 1.000000]]
&contained      = 140727221075216  // 0x7ffd9c05bd10
&container      = 140727221075280  // 0x7ffd9c05bd50
&container.data = 140727221075280  // 0x7ffd9c05bd50
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
------------------------------------------------------------
view_knob() = 1337
turn_knob(42)
view_knob() = 42
```

Note that in order to call `view_knob()`, the mangled name must be used

```c
int view_knob(void) {
  return __example_MOD_view_knob();
}
```

CHAPTER 4

C++

This example interacts with the object file `fortran/example.o` when creating the executable `cpp_example`. Since the example will use *`just_print()`*, it relies on `libgfortran` so we link against it (potentially having used `gfortran -print-search-dirs` to determine where it is located):

```
$ g++ \
>   -std=c++11 \
>   -I c/ \
>   cpp/example.cpp \
>   fortran/example.o \
>   -o cpp_example \
>   -L/usr/lib/gcc/x86_64-linux-gnu/5 \
>   -L/usr/lib/x86_64-linux-gnu \
>   -lgfortran
```

The calling script in C++ is only partially complete:

```
$ ./cpp_example
------------------------------------------------------------
quux = foo(1.000000, 16.000000) = 61.000000
------------------------------------------------------------
quuz = make_udf(1.250000, 5.000000, 1337)
     = UserDefined(1.250000, 5.000000, 1337)
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
```

CHAPTER 5

Python

There are several ways to make foreign calls from within a Python program, each with it's own pros and cons. Of the methods listed, only *f2py* is specific to Fortran. The rest can be used for any native code that can be built into the artifacts needed for that particular method.

5.1 ctypes

This example interacts with the shared object file `fortran/example.so`. This shared object file can be loaded directly

```
>>> import ctypes
>>> so_file = "fortran/example.so"
>>> lib_example = ctypes.cdll.LoadLibrary(so_file)
>>> lib_example
<CDLL 'fortran/example.so', handle 1b8d1b0 at 0x7f4f1cf0b128>
```

Once loaded, each function in the ABI can be accessed as an attribute of the `CDLL` object. For example:

```
>>> lib_example.make_udf
<_FuncPtr object at 0x7f4f1d15f688>
>>> lib_example.foo
<_FuncPtr object at 0x7f4f1d15f750>
```

See the `ctypes` documentation for more information on making foreign calls.

5.1.1 Usage

Each user defined type can be described by subclasses of `ctypes.Structure`, which are used to describe the fields in a C `struct`:

```
class UserDefined(ctypes.Structure):
    _fields_ = [
        ("buzz", ctypes.c_double),
        ("broken", ctypes.c_double),
        ("how_many", ctypes.c_int),
    ]

    def __repr__(self):
        template = (
            "UserDefined(buzz={self.buzz}, "
            "broken={self.broken}, "
            "how_many={self.how_many})"
        )
        return template.format(self=self)


class DataContainer(ctypes.Structure):
    _fields_ = [("data_", ctypes.c_double * 8)]

    @property
    def data(self):
        result = np.ctypeslib.as_array(self.data_)
        return result.reshape((4, 2), order="F")
```

Note in particular that NumPy provides the `numpy.ctypeslib` module for `ctypes`-based interoperability.

To go the other direction, i.e. from a NumPy `array` to a `double*` pointer:

```
def numpy_pointer(array):
    return array.ctypes.data_as(ctypes.POINTER(ctypes.c_double))
```

In order to call *`udf_ptr()`*, a `UserDefined` instance is created and an opaque `intptr_t` value is passed by reference:

```
made_it, ptr_as_int = prepare_udf()
lib_example.udf_ptr(ctypes.byref(ptr_as_int))
```

In order to convert the pointer to the data held in the `ctypes.Structure` instance to an `intptr_t`, the `UserDefined*` pointer is first converted to a `void*` pointer:

```
def prepare_udf():
    made_it = UserDefined()
    raw_pointer = ctypes.cast(ctypes.pointer(made_it), ctypes.c_void_p)
    intptr_t = get_intptr_t()
    ptr_as_int = intptr_t(raw_pointer.value)
    return made_it, ptr_as_int
```

To call `view_knob()`, the mangled name must be used:

```
def view_knob(lib_example):
    return lib_example.__example_MOD_view_knob()
```

5.1.2 Output

```
$ python python/check_ctypes.py
------------------------------------------------------------
quux = foo(c_double(1.0), c_double(16.0)) = c_double(61.0)
------------------------------------------------------------
quuz = make_udf(c_double(1.25), c_double(5.0), c_int(1337))
     = UserDefined(buzz=1.25, broken=5.0, how_many=1337)
needsfree(quuz) = True
address(quuz) = 139757150344968  # 0x7f1bbf4d1708
*address(quuz) =
    UserDefined(buzz=1.25, broken=5.0, how_many=1337)
------------------------------------------------------------
val =
[[ 3.    4.5 ]
 [ 1.    1.25]
 [ 9.    0.  ]
 [-1.    4.  ]]
two_val = foo_array(c_int(4), val)
two_val =
[[ 6.   9. ]
 [ 2.   2.5]
 [18.   0. ]
 [-2.   8. ]]
------------------------------------------------------------
ptr_as_int = address(made_it)  # intptr_t / ssize_t / long
ptr_as_int = c_long(139757150344992)  # 0x7f1bbf4d1720
udf_ptr(ptr_as_int)  # Set memory in ``made_it``
made_it = UserDefined(buzz=3.125, broken=-10.5, how_many=101)
needsfree(made_it) = True
*ptr_as_int =
    UserDefined(buzz=3.125, broken=-10.5, how_many=101)
------------------------------------------------------------
contained =
[[0. 4.]
 [1. 9.]
 [1. 2.]
 [3. 1.]]
container = make_container(contained)
container.data =
[[0. 4.]
 [1. 9.]
 [1. 2.]
 [3. 1.]]
address(contained)      = 43439344  # 0x296d4f0
address(container)      = 139757150084784  # 0x7f1bbf491eb0
address(container.data) = 139757150084784  # 0x7f1bbf491eb0
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
------------------------------------------------------------
view_knob() = 1337
turn_knob(c_int(42))
view_knob() = 42
```

5.2 cffi

This example interacts with the shared object file `fortran/example.so`. Similar to `ctypes`, the `cffi` library enables the creation of a foreign function interface (FFI) via `dlopen`:

```
>>> import cffi
>>> ffi = cffi.FFI()
>>> so_file = "fortran/example.so"
>>> lib_example = ffi.dlopen(so_file)
>>> lib_example
<cffi.api._make_ffi_library.<locals>.FFILibrary object at 0x7fdd0e364ba8>
```

After dynamically loading the path, we need to manually define each member of the ABI that we'll use (both the functions and the structs):

```
ffi.cdef("void foo(double bar, double baz, double *quux);")
ffi.cdef(
    "typedef struct UserDefined {\n"
    "  double buzz;\n"
    "  double broken;\n"
    "  int how_many;\n"
    "} UserDefined;"
)
ffi.cdef(
    "void make_udf(double *buzz, double *broken,\n"
    "              int *how_many, UserDefined *quux);"
)
ffi.cdef("void foo_array(int *size, double *val, double *two_val);")
ffi.cdef("void udf_ptr(intptr_t *ptr_as_int);")
ffi.cdef("void just_print();")
ffi.cdef("int __example_MOD_view_knob(void);")
ffi.cdef("void turn_knob(int *new_value);")
```

In order to convert a NumPy array to a type that can be used with `cffi`, we use the existing `ctypes` interface:

```
def numpy_pointer(array, ffi):
    if array.dtype != np.float64:
        raise TypeError("Unexpected data type", array.dtype)
    return ffi.cast("double *", array.ctypes.data)
```

5.2.1 Output

```
$ python python/check_cffi.py
------------------------------------------------------------
quux = foo(1.0, 16.0) = 61.0
------------------------------------------------------------
quuz = make_udf(1.25, 5.0, 1337)
     = UserDefined(1.25, 5.0, 1337)
------------------------------------------------------------
val =
[[ 3.    4.5 ]
 [ 1.    1.25]
 [ 9.    0.  ]
 [-1.    4.  ]]
two_val = foo_array(4, val)
```

(continues on next page)

(continued from previous page)

```
two_val =
[[ 6.   9. ]
 [ 2.   2.5]
 [18.   0. ]
 [-2.   8. ]]
------------------------------------------------------------
ptr_as_int = address(made_it)  # intptr_t / ssize_t / long
ptr_as_int = 14735136  # 0xe0d720
udf_ptr(ptr_as_int)  # Set memory in ``made_it``
made_it = UserDefined(3.125, -10.5, 101)
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
------------------------------------------------------------
view_knob() = 1337
turn_knob(42)
view_knob() = 42
```

5.3 f2py

Rather than calling into a shared library via `ctypes` or `cffi`, the `f2py` tool can be used to to wrap Fortran interfaces in Python functions. It does this by generating a custom CPython C extension and compiling it with a Fortran shared library.

`f2py` has many limitations, chief of which is absence of support for user defined types or derived types (equivalent of a C `struct`). The examples below demonstrate a few ways of getting around these limitations, including manual conversion of a custom typed variable to raw bytes.

5.3.1 Usage

First, the `f2py` tool must be used to create an extension module:

```
$ f2py \
>   --verbose \
>   -c \
>   --opt='-O3' \
>   -m example \
>   fortran/example.f90 \
>   skip: make_container view_knob
$ ls *.so
example.cpython-37m-x86_64-linux-gnu.so*
```

As we can see, this interacts directly with the Fortran source rather than with an object file or a shared library.

Inside the `example.so` module we've created, the only attribute is `example`, which represents the Fortran module in `example.f90`:

```
>>> import example
>>> example
<module 'example' from '../example.cpython-37m-x86_64-linux-gnu.so'>
>>> [name for name in dir(example) if not name.startswith("__")]
```

(continues on next page)

(continued from previous page)

```
['example']
>>> example.example
<fortran object>
```

It is within this wrapped Fortran module that our actual routines live:

```
>>> example_ns = example.example
>>> for name in dir(example_ns):
...     if not name.startswith("__"):
...         print(name)
...
foo
foo_array
foo_by_ref
just_print
make_udf
turn_knob
udf_ptr
```

The first task we'll accomplish is to call *foo()* and `foo_by_ref()`:

```
# foo()
bar = 1.0
baz = 16.0
msg_foo = "foo        ({}, {}) = {}".format(
    bar, baz, example_ns.foo(bar, baz)
)
print(msg_foo)
msg_foo_by_ref = "foo_by_ref({}, {}) = {}".format(
    bar, baz, example_ns.foo_by_ref(bar, baz)
)
print(msg_foo_by_ref)
```

As can be seen in the output below. Calling by reference results in the correct answer while calling by value (*foo()*) does not work correctly with `f2py`.

Next, we invoke the *make_udf()* routine to "smuggle" out a *UserDefined* value as raw bytes:

```
# make_udf()
buzz = 1.25
broken = 5.0
how_many = 1337
quuz_as_bytes = example_ns.make_udf(buzz, broken, how_many)
quuz = np_to_udf(quuz_as_bytes)
msg = MAKE_UDF_TEMPLATE.format(buzz, broken, how_many, quuz)
print(msg, end="")
```

In particular, this uses the `np_to_udf` helper to convert those bytes into a *UserDefined* object as defined in *ctypes*:

```
def np_to_udf(arr):
    address = arr.ctypes.data
    return UserDefined.from_address(address)
```

For *udf_ptr()*, the other routine which deals with a user defined type, we use the `prepare_udf` helper from *ctypes*. This allocates the memory for the *UserDefined* value in Python and then passes a `void*` pointer (as an integer) to the Fortran routine:

```
# udf_ptr()
made_it, ptr_as_int = prepare_udf()
ptr_as_int = ptr_as_int.value
example_ns.udf_ptr(ptr_as_int)
msg = UDF_PTR_TEMPLATE.format(ptr_as_int, ptr_as_int, made_it)
print(msg, end="")
```

Since `f2py` is included with NumPy, it has nicer support for NumPy arrays than either `ctypes` or `cffi`. This means we can call *`foo_array()`* directly with a NumPy array:

```
# foo_array()
val = np.asfortranarray([[3.0, 4.5], [1.0, 1.25], [9.0, 0.0], [-1.0, 4.0]])
two_val = example_ns.foo_array(val)
print(MSG_FOO_ARRAY.format(val, two_val))
```

Finally, we call *`just_print()`* to mix Python and Fortran usage of `STDOUT`:

```
# just_print()
print("just_print()")
example_ns.just_print()
```

5.3.2 Output

```
$ python f2py/check_f2py.py
------------------------------------------------------------
example: <module 'example' from '.../f2py/example...so'>
dir(example.example): foo, foo_array, foo_by_ref, just_print, make_udf, udf_ptr
------------------------------------------------------------
foo       (1.0, 16.0) = 0.0
foo_by_ref(1.0, 16.0) = 61.0
------------------------------------------------------------
quuz = make_udf(1.25, 5.0, 1337)
     = UserDefined(buzz=1.25, broken=5.0, how_many=1337)
------------------------------------------------------------
val =
[[ 3.    4.5 ]
 [ 1.    1.25]
 [ 9.    0.  ]
 [-1.    4.  ]]
two_val = foo_array(val)
two_val =
[[  6.    9. ]
 [  2.    2.5]
 [ 18.    0. ]
 [ -2.    8. ]]
------------------------------------------------------------
ptr_as_int = address(made_it)  # intptr_t / ssize_t / long
ptr_as_int = 139859191412464  # 0x7f33816c36f0
udf_ptr(ptr_as_int)  # Set memory in ``made_it``
made_it = UserDefined(buzz=3.125, broken=-10.5, how_many=101)
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
```

5.3.3 What is Happening?

`f2py` actually generates a wrapped `{modname}module.c` file (so in our case `examplemodule.c`) and utilizes a `fortranobject` C library to create CPython C extensions:

```
>>> import os
>>> import numpy.f2py
>>>
>>> f2py_dir = os.path.dirname(numpy.f2py.__file__)
>>> f2py_dir
'.../lib/python3.7/site-packages/numpy/f2py'
>>> os.listdir(os.path.join(f2py_dir, 'src'))
['fortranobject.c', 'fortranobject.h']
```

It uses a C compiler with flags determined by `distutils` to link against NumPy and Python headers when compiling `{modname}module.c` and `fortranobject.c` and uses a Fortran compiler for `{modname}.f90`. Then it uses the Fortran compiler as linker:

```
$ gfortran \
>   -Wall \
>   -g \
>   -shared \
>   ${TEMPDIR}/.../{modname}module.o \
>   ${TEMPDIR}/.../fortranobject.o \
>   ${TEMPDIR}/{modname}.o \
>   -lgfortran \
>   -o \
>   ./{modname}.so
```

When trying to convert a Fortran subroutine to Python via `f2py`, a problem occurs if the subroutine uses a user defined type. For example, if we tried to use the *`make_container()`* routine:

```
$ f2py \
>   --verbose \
>   -c \
>   --opt='-O3' \
>   -m example \
>   fortran/example.f90 \
>   only: make_container
...
Building modules...
        Building module "example"...
                Constructing F90 module support for "example"...
Skipping type unknown_type
Skipping type unknown_type
                        Constructing wrapper function "example.make_container"...
getctype: No C-type found in "{'typespec': 'type', 'typename': 'datacontainer',
↪'attrspec': [], 'intent': ['out']}", assuming void.
getctype: No C-type found in "{'typespec': 'type', 'typename': 'datacontainer',
↪'attrspec': [], 'intent': ['out']}", assuming void.
getctype: No C-type found in "{'typespec': 'type', 'typename': 'datacontainer',
↪'attrspec': [], 'intent': ['out']}", assuming void.
Traceback (most recent call last):
...
  File ".../numpy/f2py/capi_maps.py", line 412,in getpydocsign
    sig = '%s : %s %s%s' % (a, opt, c2py_map[ctype], init)
KeyError: 'void'
```

This is because *make_container()* returns the *DataContainer* user defined type.

5.3.4 References

- (Lack of) support for user defined types in `f2py`
- The `f90wrap` interface generator adds support for user defined types to `f2py`. However, the author of this document has no experience with `f90wrap`.

5.4 Cython

Cython is a mature and heavily used extension of the Python programming language. It allows writing optionally typed Python code, implementing part of a Python module completely in C and many other performance benefits.

It is **very** mature. For example, a `.pyx` Cython file can be compiled both to a standard CPython C extension as well as providing basic support for the PyPy emulation layer `cpyext`.

5.4.1 Usage

The ABI for the Fortran module is provided in a Cython declaration `example_fortran.pxd` which we will reference throughout:

```
cimport example_fortran
```

Using this, values can be passed by value into *foo()* using typical C pass by value convention:

```
def foo(double bar, double baz):
    cdef double quux
    example_fortran.foo(bar, baz, &quux)
    return quux
```

For the *UserDefined* Fortran type, the `example_fortran.pxd` defines a matching `struct`:

```
ctypedef struct UserDefined:
    double buzz
    double broken
    int how_many
```

This can then be used for *make_udf()*:

```
def make_udf(double buzz, double broken, int how_many):
    cdef example_fortran.UserDefined made_it
    example_fortran.make_udf(&buzz, &broken, &how_many, &made_it)
    return made_it
```

and for *udf_ptr()*:

```
from libc.stdint cimport intptr_t
def udf_ptr():
    cdef example_fortran.UserDefined made_it
    cdef intptr_t ptr_as_int = <intptr_t> (&made_it)
    example_fortran.udf_ptr(&ptr_as_int)
    return made_it
```

In either case, the *UserDefined* value is created by each function (i.e. from Python, not from Fortran) and then a pointer to that memory is passed along to the relevant Fortran routine:

```
cdef example_fortran.UserDefined made_it
```

When calling *foo_array()* we allow NumPy arrays and Cython allows us to specify that the array is 2D and Fortran-contiguous. We also turn off bounds checking since the only array indices used are `0`:

```
@cython.boundscheck(False)
@cython.wraparound(False)
def foo_array(np.ndarray[double, ndim=2, mode='fortran'] val not None):
    cdef int size
    cdef np.ndarray[double, ndim=2, mode='fortran'] two_val

    size = np.shape(val)[0]
    two_val = np.empty_like(val)
    example_fortran.foo_array(
        &size,
        &val[0, 0],
        &two_val[0, 0],
    )
    return two_val
```

Calling *just_print()* simply requires wrapping a C call in a Python function (i.e. `def` not `cdef`):

```
def just_print():
    example_fortran.just_print()
```

When invoking `view_knob()`, we must do a little extra work. The `f2py` parser has a bug when a Fortran `function` (vs. a `subroutine`) has `bind(c, name=...)`. In order to allow `f2py` to wrap `example.f90`, we don't specify the non-mangled name in the ABI, hence must reference the mangled name from the object file:

```
int view_knob "__example_MOD_view_knob" ()
```

Luckily the mangled name can be aliased in the `.pxd` declaration and then calling `view_knob()` in Cython is straightforward:

```
def view_knob():
    return example_fortran.view_knob()
```

Similarly *turn_knob()* is also straightforward:

```
def turn_knob(int new_value):
    example_fortran.turn_knob(&new_value)
```

5.4.2 Output

```
$ python cython/check_cython.py
------------------------------------------------------------
quux = foo(1.0, 16.0) = 61.0
------------------------------------------------------------
quuz = make_udf(1.25, 5.0, 1337)
     = {'buzz': 1.25, 'broken': 5.0, 'how_many': 1337}
------------------------------------------------------------
val =
```

(continues on next page)

(continued from previous page)

```
[[ 3.    4.5 ]
 [ 1.    1.25]
 [ 9.    0.  ]
 [-1.    4.  ]]
two_val = foo_array(val)
two_val =
[[ 6.   9. ]
 [ 2.   2.5]
 [18.   0. ]
 [-2.   8. ]]
------------------------------------------------------------
made_it = udf_ptr()
        = {'buzz': 3.125, 'broken': -10.5, 'how_many': 101}
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
------------------------------------------------------------
example.get_include() =
.../foreign-fortran/cython/venv/lib/python.../site-packages/example/include
------------------------------------------------------------
view_knob() = 1337
turn_knob(42)
view_knob() = 42
```

5.4.3 `sdist` and installed files

On a standard CPython install on Linux, a source dist (`sdist`) contains the following:

```
.
├── example
│   ├── example.f90
│   ├── example_fortran.pxd
│   ├── fast.c
│   ├── include
│   │   └── example.h
│   └── __init__.py
├── example.egg-info
│   ├── dependency_links.txt
│   ├── PKG-INFO
│   ├── SOURCES.txt
│   └── top_level.txt
├── MANIFEST.in
├── PKG-INFO
├── setup.cfg
└── setup.py

3 directories, 13 files
```

Once this gets installed, the following files are present:

```
.
├── example_fortran.pxd
├── fast.cpython-37m-x86_64-linux-gnu.so
```

(continues on next page)

(continued from previous page)

```
├── include
│   └── example.h
├── __init__.py
├── lib
│   └── libexample.a
└── __pycache__
    └── __init__.cpython-37.pyc

3 directories, 6 files
```

5.4.4 `cimport`-ing this library

This library provides an `example/example_fortran.pxd` declaration file that can be used to `cimport` the library without having to worry about the Python layer:

```
cimport example.example_fortran
```

In this case, the library referenced in `example_fortran.pxd` is made available in the `example` package:

```
>>> import os
>>> import example
>>>
>>> include_dir = example.get_include()
>>> include_dir
'.../foreign-fortran/cython/venv/lib/python.../site-packages/example/include'
>>> os.listdir(include_dir)
['example.h']
>>>
>>> lib_dir = example.get_lib()
>>> lib_dir
'.../foreign-fortran/cython/venv/lib/python.../site-packages/example/lib'
>>> os.listdir(lib_dir)
['libexample.a']
```

See `cython/use_cimport/setup.py` for an example of how to wrap:

```
>>> wrapper.morp()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
>>> example.foo(1.5, 2.5)
10.875
>>> wrapper.triple_foo(1.5, 2.5)
32.625
```

5.4.5 Gotcha

If `libraries=['gfortran']` is not specified in `setup.py` when building the CPython C extension module (`example.so`), then the print statements in *`just_print()`* (as defined in in `example.f90`) cause

```
$ IGNORE_LIBRARIES=true python setup.py build_ext --inplace
running build_ext
...
```

(continues on next page)

(continued from previous page)

```
$ python -c 'import example'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File ".../cython/package/example/__init__.py", line 5, in <module>
    from example import fast
ImportError: .../cython/package/example/fast...so: undefined symbol: _gfortran_
↪transfer_character_write
```

5.4.6 References

- Decently helpful article and pre-article to that one about using Cython to wrap Fortran. But this article fails to point out its approach can leave out some symbols (e.g. the `check_cython` example when `libgfortran` isn't included)
- Description on the uber-useful `fortran90.org` on how to interface with C

CHAPTER 6

Go

Fortran can be invoked directly from Go code by using the `cgo` language extension. A simple example is provided in the Golang source tree, which is quite helpful. If the Fortran source is in the same directory as a Go library, `cgo` automatically builds the Fortran files and includes them.

We define the Go package `example` in the `golang/src` directory. The Fortran source and the C headers are side-by-side with the Go package:

```
$ tree golang/src/example/
golang/src/example/
├── example.f90 -> ../../../fortran/example.f90
├── example.go
└── example.h -> ../../../c/example.h

0 directories, 3 files
```

6.1 Package

Within the package, we first declare the package to use `cgo` and include the relevant header file:

```
// #include "example.h"
import "C"
```

We start by defining user-friendly equivalents of `C.struct_UserDefined` and `C.struct_DataContainer`

```
type UserDefined struct {
        Buzz    float64
        Broken  float64
        HowMany int32
}

type DataContainer struct {
        Data *[8]float64
```

(continues on next page)

(continued from previous page)

```
}

func (udf *UserDefined) String() string {
        return fmt.Sprintf(
                "%T(%f, %f, %d)",
                udf, udf.Buzz, udf.Broken, udf.HowMany,
        )
}
```

Adding *just_print()* to the package interface is just a matter of making `C.just_print` public:

```
func JustPrint() {
        C.just_print()
}
```

When passing in `*float64` (i.e. pointer types), the underlying values can be passed along to `C.make_udf` without having to copy any data, e.g. via `(*C.double)(buzz)`. The foreign call will populate the fields of a `C.struct_UserDefined`, which will need to be converted to normal Go types when a `UserDefined` value is created for the return. To avoid copying when constructing a `UserDefined` object, we dereference a field value (e.g. `&quuz.buzz`), convert the reference to a non-`C` pointer type (e.g. `(*float64)(&quuz.buzz)`) and then dereference the value:

```
func MakeUDF(buzz, broken *float64, howMany *int32) *UserDefined {
        var quuz C.struct_UserDefined
        C.make_udf(
                (*C.double)(buzz),
                (*C.double)(broken),
                (*C.int)(howMany),
                &quuz,
        )
        return &UserDefined{
                *(*float64)(&quuz.buzz),
                *(*float64)(&quuz.broken),
                *(*int32)(&quuz.how_many),
        }
}
```

When dealing with array types, the first value in a slice is dereferenced and then converted into a C pointer (e.g. `(*C.double)(&val[0])`):

```
func FooArray(size *int32, val []float64) []float64 {
        twoVal := make([]float64, len(val), cap(val))
        C.foo_array(
                (*C.int)(size),
                (*C.double)(&val[0]),
                (*C.double)(&twoVal[0]),
        )
        return twoVal
}
```

When calling *udf_ptr()*, a `UserDefined` value must be created, dereferenced, cast to `unsafe.Pointer` and then cast again to `uintptr`:

```
madeIt := example.UserDefined{}
ptrAsInt := uintptr(unsafe.Pointer(&madeIt))
example.UDFPtr(&ptrAsInt)
```

Only then can the `uintptr` be converted to a `C.intptr_t`:

```
func UDFPtr(ptrAsInt *uintptr) {
        C.udf_ptr(
                (*C.intptr_t)(unsafe.Pointer(ptrAsInt)),
        )
}

func MakeContainer(contained []float64) *DataContainer {
        var container C.struct_DataContainer
        C.make_container(
                (*C.double)(&contained[0]),
                &container,
        )
        dataPtr := (*[8]float64)(unsafe.Pointer(&container.data))
        return &DataContainer{dataPtr}
}
```

In the case of `view_knob()`, the mangled name must be used:

```
func ViewKnob() int32 {
        knobValue := C.__example_MOD_view_knob()
        return (int32)(knobValue)
}

func TurnKnob(newValue *int32) {
        C.turn_knob(
                (*C.int)(newValue),
        )
}
```

6.2 Output

The Go example can be run via `go run`. As with *C* and *C++*, the `gfortran` search path may need to be explicitly provided (with `-L` flags). This can be done with the `CGO_LDFLAGS` environment variable.

```
$ go run golang/main.go
------------------------------------------------------------
quux = foo(1.000000, 16.000000) = 61.000000
------------------------------------------------------------
quuz = make_udf(1.250000, 5.000000, 1337)
     = *example.UserDefined(1.250000, 5.000000, 1337)
------------------------------------------------------------
foo_array(
    4,
    [[3.000000, 4.500000],
     [1.000000, 1.250000],
     [9.000000, 0.000000],
     [-1.000000, 4.000000]],
) =
    [[6.000000, 9.000000],
     [2.000000, 2.500000],
     [18.000000, 0.000000],
     [-2.000000, 8.000000]]
------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
ptrAsInt = &madeIt
ptrAsInt = 842350544096  // 0xc4200144e0
udf_ptr(&ptrAsInt)  // Set memory in ``madeIt``
&madeIt = *example.UserDefined(3.125000, -10.500000, 101)
------------------------------------------------------------
contained =
  [[0.000000, 4.000000],
   [1.000000, 9.000000],
   [1.000000, 2.000000],
   [3.000000, 1.000000]]
container = make_container(contained)
container.Data =
  [[0.000000, 4.000000],
   [1.000000, 9.000000],
   [1.000000, 2.000000],
   [3.000000, 1.000000]]
&contained      = 842350560256  // 0xc420018400
&container      = 842350518320  // 0xc42000e030
 container.Data = 842350560320  // 0xc420018440
------------------------------------------------------------
just_print()
 ======== BEGIN FORTRAN ========
 just_print() was called
 ========  END  FORTRAN ========
------------------------------------------------------------
view_knob() = 1337
turn_knob(42)
view_knob() = 42
```

Index