

## Table of Contents

.....	1
Passing Inputs .....	2
Pointers and Arrays .....	3
Array Types .....	3
Structures .....	4
Variable number of arguments .....	5
Future Possibilities & Directions .....	7

The idea behind this package is to allow R programmers to dynamically invoke arbitrary compiled routines without having to explicitly write and compile C code to perform the invocation. This may not be of immense value as one has to compile the original routines to which you are interfacing. Furthermore, a compiled interface will be faster than one that examines its inputs at run-time. However, the dynamic interface is somewhat interesting.

The package is an interface to `libffi`. There are two main functions of interest to users. The first is `prepCIF()` which is used to create a "template" call to any native routine with a particular signature - both type of inputs and return value. Having created this interface object, we can use it to make one or more calls to any routine with that signature. We do this with `callCIF()`.

We specify the type of the return value and of each of the parameters in our call to `prepCIF()`. There are objects describing each of the basic types such as a double, various types of integers and pointers.

**Table 1.**

R variable	description
<code>doubleType</code>	a <i>double</i> in C
<code>floatType</code>	a <i>float</i> in C
<code>longdoubleType</code>	for platforms that support a <i>long double</i> type
<code>pointerType</code>	a generic <i>void *</i> pointer
<code>sint16Type</code>	a signed 16-bit/2 byte integer
<code>sint32Type</code>	a regular signed 32-bit/4-byte integer
<code>sint64Type</code>	a signed 64-bit integer
<code>sint8Type</code>	a signed single byte integer
<code>uint16Type</code>	an unsigned 16-bit integer
<code>uint32Type</code>	an unsigned 32-bit integer
<code>uint64Type</code>	an unsigned 64-bit integer
<code>uint8Type</code>	an unsigned 8-bit integer
<code>voidType</code>	the empty/void type of specific use when the routine of interest has no return value.

R variable	description
stringType	a type object introduced for this package for representing strings, i.e. <i>char *</i> . This might be expanded in the future for different types of strings, i.e. signed and unsigned characters, wide characters, etc.

In addition to these primitive types, one can create new types for describing C-level *structs*. See [struct-Type\(\)](#).

Let's consider a very simple example of calling a C routine that takes no arguments and returns no value.

```
#include <stdio.h>
void
voidCall()
{
    fprintf(stderr, "In voidCall\n");
}
```

This is included in the compiled code loaded in the *Rffi* package. We would create a CIF for calling a routine with this signature via

```
void = prepCIF(voidType)
```

We only have to specify the return type. We don't have to specify any additional information as there are no parameters. Note that we don't specify the name of the C routine to call. This CIF can be used to call any routine with such a signature. And we can use it for multiple calls to the same or different routines.

So now we can call the C routine *voidCall* via [callCIF\(\)](#). We have to specify the CIF and the name of the routine.

```
callCIF(void, "voidCall")
```

You'll see Again, since there are no inputs to this routine, we don't have to specify any.

Consider a C routine that takes no inputs and returns a real value:

```
double
rdouble()
{
    return(3.1415);
}
```

The same steps allow us to call this routine, but we need to specify the type of the return value differently.

```
cif = prepCIF(doubleType)
callCIF(cif, "rdouble")
```

This returns the value 3.1415 as an R numeric vector of length 1.

## Passing Inputs

Let's move on to where we pass values from R to the native routine. We'll use a routine (also in *Rffi.so/dll*) called *foo* that takes an integer and a double. It returns a double - the sum of the two.

```
#include <stdio.h>
double
foo(int x, double y)
```

---

```
{
  fprintf(stderr, "In foo %d %lf\n", x, y);
  return(x + y);
}
```

To create the CIF, we need to specify the return type as `doubleType` and then a list of the parameter types. These are `sint32Type` and `doubleType`. Since there are various different types of integers, we have to be specific about which one is to be used. We use a regular 4-byte/32-bit integer that has a bit for sign so that negative values are possible. So our call is

```
cif = CIF(doubleType, list(sint32Type, doubleType))
```

And now we can the routine with

```
callCIF(cif, "foo", -1L, pi)
```

What if we passed a numeric value instead of an integer for the first argument? For example,

```
callCIF(cif, "foo", -1.3, pi)
```

In the current version of the code, the numeric value is coerced to the target type - an integer. As a result, the value -1 is passed to the native routine.

What about if we pass a string such as "abc"? This is coerced to an integer and results in a **NA**.

## Pointers and Arrays

Next we turn our attention to passing non-scalar values, specifically pointers. We'll deal with structures later. Consider a routine which takes a collection of real values via a pointer of type *double*. It also takes the number of elements as an unsigned integer. The routine *retPointer* provides such a routine in *test.c* and available in the *Rffi*. We can create CIF

```
cif = CIF(pointerType, list(pointerType, uint32Type))
```

noting that it returns a new array of *doubles*. We can then invoke this with

```
x = c(1, 2, 3, 4)
y = callCIF(cif, "retPointer", x, length(x))
```

The resulting value in *y* is an external pointer which contains the address of the returned array newly allocated in *retPointer*. We can pass this to another native routine. Alternatively, we can call native code to extract values and generally manipulate it. We might do this with a regular hand-built routine that is invocable from R via the *.Call()* interface as we are dealing with SEXP objects throughout, i.e. inputs and outputs. One such routine is contained in *test.c* - *R\_copyDoubleArray*. We call it with the external pointer to the *double \** and the number of elements, e.g.

```
val = .Call("R_copyDoubleArray", y$value, length(x))
```

If we wanted to release the memory associated with the array, we could call *free*. Again, we could use a manually created routine via *.Call()*. But we can also do it via *libffi*.

```
free.cif = CIF(voidType, list(pointerType))
callCIF(free.cif, "free", y$value)
```

## Array Types

Accessing an array is much the same as accessing a pointer to a collection of elements. We need to know the type of the elements in the array and the number of elements.

---

# Structures

libffi also allows us to work with structures. Consider a simple structure defined as

```
typedef struct {
    short s;
    int i;
    double d;
    char *string;
} MyStruct;
```

Next, consider a routine that returns an instance of this structure.

```
char *MyString = "a string";
MyStruct
getStruct()
{
    MyStruct c = {-1, 11, 99.2};
    c.string = MyString;
    return(c);
}
```

How can we call this from R and get the structure back? We start by defining a new type to describe this structure. We use `structType()` for this. We give a list of the types of the elements within the structure.

```
myStruct.type = structType(list(s = sint16Type, i = sint32Type, d = doubleType, st
```

Note that the *short* corresponds to the `sint16` type.

We can optionally provide names. It is a good idea if we know them as we use the same names as in the C code to simplify interaction. At present we don't use the names, but we may in the future.

Now that we have this type information, we can use it as we did the regular built-in types, e.g. `doubleType`. So we create our CIF to call the routine `getStruct`:

```
cif = CIF(myStruct.type)
```

And now we can call this

```
callCIF(cif, "getStruct")
```

The result is an R list with 4 elements corresponding to the elements of the C structure.

Now let's explore passing an R object to a routine that expects a structure. We can use the following routine

```
void
doStruct(MyStruct s)
{
    fprintf(stderr, "doStruct: s = %d, i = %d, d = %lf, string = %s\n",
              (int) s.s, s.i, s.d, s.string);
}
```

We create the CIF for this routine

```
cif = CIF(voidType, list(myStruct.type))
```

Now we can call this as

---

```
callCIF(cif, "doStruct", list(-1L, 1L, 3.1415, "R string"))
```

Let's deal with pointers to structs.

```
library(Rffi)
myStruct.type = structType(list(s = sint16Type, i = sint32Type, d = doubleType, st
cif = CIF(pointerType)
ref = callCIF(cif, "getStructP")
```

Now we want to access the individual elements. Let's get the second element - i:

```
getStructField(ref, 2L, myStruct.type)
getStructField(ref, "i", myStruct.type) # the same as using 2L
```

And we can loop over the elements and extract each individually.

To convert the entire reference to an R object, we can use

```
getStructValue(ref, myStruct.type)
```

This is faster than accessing each element from R as above.

Rather than using `getStructField()` and having to specify the type description for the particular type, we might want to use the more convenient `$()` operator. For example,

```
ref$i
```

is easier than

```
getStructField(ref, "i", myStruct.type)
```

To do this, we can define a new class for our reference to the particular data type:

```
setClass("RCReference", list(ref = "externalptr"))
setClass("MyStructRef", contains = "RCReference")
```

Next, we can define a method for `$()`:

```
setMethod("$", "MyStructRef",
  function(x, name) {
    myStruct.type = structType(list(s = sint16Type, i = sint32Type, d =
    getStructField(x@ref, name, myStruct.type)
  })
```

Now, we can create an instance of this class and use the `$()` notation:

```
tmp = new("MyStructRef", ref = ref)
print(tmp$i)
print(tmp$d)
```

We can of course create similar methods for `$<` and `[` and `[<`.

If we use [RGCCTranslationUnit](#) or [RCIndex](#), we can obtain information about data structures and their fields programmatically and then automate the generation of these methods.

## Variable number of arguments

A C routine can take a variable number of arguments by having `...` as the final parameter. A good example of this is the family of `printf` functions. *printf* is the simplest example to work with. We pass a string that describes the string to print which also contains markup for how to interpret and format the values of the remaining arguments into the generated content. In C we would call this as in the following examples:

---

```
printf("A double %lf, and an integer %d\n", 3.1415, 10);
printf("A string '%s', and a double with %.5lf, and an integer %d\n",
      "my string", 3.14159265, 10L);
```

The problem with a variable number of arguments is that we cannot describe the call interface for the routine just once. Each call requires its own call interface definition that is based on the actual call. In the first call to *printf*, we could use

```
cif = CIF(sint32Type, list(stringType, doubleType, sint32Type))
callCIF(cif, "printf", "A double %lf and an integer %d", 3.1415, 10L)
```

The first argument is always the format string. In the actual call, there are 2 format markups - %lf and %d - and there are 2 additional arguments. We know these correspond to a double and an integer. Hence we can define our CIF. Then we can invoke it. (The integer result is the number of characters in the generated content.)

For our second example, the signature is different. We have 3 format markups in our formatting string and these correspond to a string, a double and an integer. So our CIF is

```
cif = CIF(sint32Type, list(stringType, stringType, doubleType, sint32Type))
```

and we can call it as

```
callCIF(cif, "printf", "A string '%s', and a double with %.5lf, and an integer %d\n",
      "my string", 3.14159265, 10L)
```

The family of 'exec' routines on UNIX (i.e. *execl*, *execle*, *execlp*) are examples of a routines accepting a variable number of arguments. All the arguments are strings, and the end of the arguments is identified by a NULL string. The routine *call\_varargs\_null* emulates this interface (without replacing the current process!). We might call it with the name of an executable (R) and then 4 additional arguments, the last being **NULL**. For this, we need a CIF that expects 5 strings:

```
cif = CIF(voidType, replicate(5, stringType))
```

Then we can invoke the routine as

```
callCIF(cif, "call_varargs_null", "R", "--no-restore", "--slave", "foo.R", NULL)
```

To call the routine with no additional parameters, but just the name of the executable, we have to create a new CIF and then use that in the call:

```
cif = CIF(voidType, replicate(2, stringType))
callCIF(cif, "call_varargs_null", "R", NULL)
```

So the simple problem is that a single CIF won't work for a variable argument function. Instead, we need to generate a CIF tailored to each call. In the case of *printf*, we can inspect the format string and find the markup elements to determine the number and expected type of the additional arguments. This is non-trivial, but it does allow us to determine each call and coerce the arguments to the correct types. For other routines that accept a variable number of arguments, we have to determine how to find the expected number and types of parameters. In some cases, this will amount to using the number and type of arguments passed in the call from R. This has the potential to get the types wrong. But it may be the best we can do.

When interfacing to routines with a variable number of arguments, it is a good idea to write an R function that dynamically creates a new CIF for each call. For our *call\_varargs\_null* routine, this function might be defined as

```
varargsNULL =
function(filename, ...)
```

---

```
{
  tmp = c(filename, list(...))
  args = vector("list", length(tmp) + 1)
  args[seq_along(tmp)] = tmp
  cif = CIF(voidType, replicate(length(args), stringType))
  callCIF(cif, "call_varargs_null", .args = args)
}
```

Then we can call this as

```
varargsNULL("R", "a", "b", "c")
```

The first 3 lines of the function just arrange the arguments into a list with a NULL value to terminate the list for the C code.

An implementation for an interface to `printf` might be as simple as determining the number of arguments to the R function and computing their type

```
printf = function(fmt, ...)
{
  args = c(as.character(fmt), list(...))
  cif = CIF(sint32Type, lapply(args, getRFFIType))
  invisible(callCIF(cif, "printf", .args = args))
}
```

Now we need to define a function `getRFFIType()` that maps the name of an R data type to a corresponding FFI type. A simple incomplete version is as follows:

```
getRFFIType =
function(x, type = class(x))
{
  switch(type,
    numeric = doubleType,
    integer = sint32Type,
    character = stringType)
}
```

We should look at the length of the R vectors to determine if we need to use an array or a scalar.

```
printf("%lf, %d, %s\n", pi, 10L, "a string")
```

This approach cannot handle formats that do not correspond to non-R data types, e.g. float, unsigned integers, longs. To support these, we should process the `fmt` string. But we will leave that as a separate tutorial.

## Future Possibilities & Directions

At present, we have not added support for specifying the type of the pointer, e.g. a pointer to a double. This is entirely feasible and would be valuable as it provides a great deal more information. This can then be used to comprehend the contents of the values with more specificity.

Information about routines can be obtained via `RGCCTranslationUnit`. This can be used within the same R session to construct CIF objects for different routines and data structures.