The Default RGCCTranslationUnit mappings.

Duncan Temple Lang

Table of Contents

This article describes the current thinking underlying the default bindings created by the RGCCTranslationUnit for interfacing R to C/C++ code.

Constants

There are several types of constants in C/C++ code. There are regular variables which have been declared with the `const` keyword. There are values defined in enumerations, and for these there are named and anonymous enumerations. There are also "literals" coming from #define directives in the pre-processor.

Variables

Variables are things that vary!, i.e. they are not constants. So we need to be able to get and set the current value of each variable. In C, we think of non-local/global variables which are available across files, i.e. non-static variables not within a routine. For C++, we also include static class variables. Fields within a C ++ object behave like fields in a struct and are accessed directly. While they vary, they are not top-level variables

C Structures

We create an interface to each C-level `struct` data type that we need or is explicitly created. We allow one to create new instances of these data types at the C-level from within R, specify finalizers which are called when the instance is no longer referenced in R, and to remove such finalizers

Note

add these facilities

and to access individual fields within the structure. We also create an R-level class that mirrors the fields in the C-level structure and we allow one to copy from C to R and from R to C using coercion methods, e.g `as(x, target)`

Constructors

We create R functions and C routines for allocating instances of structs in C. These are named *new_<name>*() , where <name> is replaced by the actual name of the structure.

Note

What precisely do we mean by name here - the A in struct A or the name in typedef in typedef struct _A { ... } A;

The constructor

Accessors

Destructors

One can create C-level instances of C-level `struct` objects via the generated *new_<name>*() function.

Note

Do this for primitive data types and arrays thereof.

These are allocated on the heap and so must be released when we no longer need them in order to avoid memory leaks. In such calls to constructor functions, one can specify whether to add a finalizer routine that will free the memory when the instance is no longer referenced/reachable from R (i.e. it can be garbaged collected in R's view). To use the (programmatically generated) default finalizer routine, specify **TRUE** for the *.finalizer* argument of the constructor function in R. Alternatively, if one wants to specify a different finalizer, one can use this *.finalizer* parameter to specify a *NativeSymbol* or *NativeSymbolInfo* identifying a native routine (which should expect a single argument - the external pointer object), or an R function which is also called with the external pointer as its sole argument(?). One can also chose to register a finalizer any time after the creation of the C-level instance using the *addFinalizer*() function in the RAutoGenRunTime package. At present, there is no mechanism in R to undo this, i.e. remove a registered finalizer.

Note

Add such a routine to R.

There will be circumstances in which one cannot use R's garbage collection mechanism. For example, when an instance of a C-level structure is inserted as a field into a containing C-level structure, we cannot release that initial object and have the reference from the container still be valid. For cases like this, the R programmer must manage the memory herself (or not!). Use the function *free*() in the RAutoGenRunTime to do this.

Note

Do we generate copy routines? We have all the work done as we can copy from C to R, allocate a new object, and then from R to C again. But there is a quicker way that does this all in C.

Note

Perhaps add something for weak references.

Output Arguments

Some routines have parameters that are used to transfer results to the caller. This is how we can return multiple values from a C routine. We pass an object in by reference or as a pointer to a variable and the called routine can then populate its value and so convey an updated result to the caller. These are out or inout variables depending on whetehr the called routine ignores the current content of the object or actually uses its value(s) as well as inserting values into it.

When we create the interface to a function that has "out" parameters

Note

We need to extend this to inout parameters, but this is relatively easy.

we provide default arguments for these out parameters which are references to C-level structures of the appropriate type. These are passed to the C routine and then returned as part of the result with their contents changed.

A *`.copy`* parameter is to the R function and this controls which of the out arguments are returned as part of the result and how. This is a named logical vector with as many elements as there are out parameters. The names are the names of the out parameters in the R function. A value of **`TRUE`** corresponds to make a deep copy of the referenced C object to an instance of the corresponding R class. A value of **`FALSE`** indicates that the result should be left as a reference. This is useful if we want to make further use of this object at the C level in subsequent R function calls. Finally, a value of **`NA`** indicates that we are not interested in the result and that it should be ignored.

Note

This should be done at the C-level so as to avoid creating a reference to this object. But this is an efficiency issue, not a semantic one so can wait.

One can augment a resolved routine of class *`ResolvedNativeRoutine`* in R by adding a field named `paramStyle` to the list of values. If no value is passed for the argument *`paramStyle`* in a call to *`createMethodBinding`*() with this routine, that field is used to describe the parameter styles. Alternatively, one can explicitly specify these values in the call. Often, we create a set of "hints" and extract the specific parameter styles from that for the routine and add them to the call.

Note

We should allow this to be done from the *`typeMap`* parameter.

Casting

For pointers to instances of C++ classes, one can coerce between types. For example, suppose we have two classes A and B, and a third class C which extends B. Then we have another class W which uses multiple inheritance to derive from both A and C. Then, we can coerce from an instance of W, say **w** in R, to any of A, B or C

```
R
as(w, "APtr")
  as(w, "BPtr")
  as(w, "CPtr")
```

This uses a static_cast call in the C++ code. because W's ancestor classes - A, B and C - are known when we generate the bindings. We can use an explicit call to *cast*() , such as

```
R
cast(w, "APtr", how = 'dynamic')
```

This allows us to specify which of the cast techniques to use: static, dynamic, const and reinterpret.

Remember that casting the C++ pointers may not lead to invoking methods in that class. For example, suppose we have two classes X and Y, with Y derived from X. And suppose X has a virtual method foo(int) that Y also implements. Then, when we call

```
R
foo(y, 1L)
```

we get Y's method. And if we coerce **y** to a reference/pointer to X,

```
R
foo(as(y, "XPtr"), 1L)
```

we still get Y's method. This is because C++ finds the virtual method and invokes that, as we would want.

In order to invoke X's foo(int) method, we need to use the .inherited argument, e.g.

```
R
foo(y, 1L, .inherited = TRUE)
foo(y, 1L, .inherited = "X")
```

The **TRUE** value invokes the method in the first base class. Alternatively, we can explicitly name any of the ancestor classes.

RAutoGenRunTime package

The generated code uses R and C code that is quite general and not tied to the particular C/C++ code to which we are interfacing . We could copy this code to each generated "package". And indeed this is fine. However, it is better engineering to separate this code into its own package and have each programmatically generated interface use a shared version of this. This allows updates for the shared code to be readily installed and used without having to reinstall all the dependent packages. The RAutoGenRunTime package provides these facilities. For example, it provides the function that verifies the .copy argument for functions involving 'out' parameters. This is called directly from the generated code as RAutoGenRunTime:::validateCopy and is not exported (at present) as it is not intended to be called by regular users.

Note

There are two ways to compare the source attribute to a list of target files. Use isSourceFile to compare to a collection of actual file names (without their paths) and use checkSource when we have the name of the files of interest without their extension, e.g. foo which would match foo.c and foo.h. This is not as specific.

Note

The class name for a reference to a struct is structNameRef. The external pointer uses the same name as the class name.