Table of Contents

Implementing Derived C++ classes with R methods

We can invoke C++ methods from within R by automating the creation of the wrapper functions and routines for the individual methods. It is also possible to have R functions be called from C++ methods or C routines. And it is relatively easy to arrange to define a new C++ class that inherits from an existing C++ class and implements some or all of the methods with R functions. This allows us to create new C++ classes that can be used as regular C++ classes within existing C++ computations but which are implemented in an interpreted language. This gives us a way to prototype C++ classes within a high-level interpeted language and use them transparently within low-level compiled code.

Let's look at an example from wxWidgets that we have made use of when parsing and rendering HTML documents. The wxHtmlWindow widget provides a parser instance (wxHtmlParser) and before we parse a document, we can add handlers for HTML tags or our own "made-up" tags. We create wxHtmlTagHandler objects and register them with the parser using the parser's AddTagHandler method. Then when that parser encounters a tag, it finds the registered handler corresponding to that tag and calls its HandleTag() method. This is a nice example of programming to an interface and object oriented programming. The parser knows very little about the individual tag handlers, but can invoke the important methods GetSupportedTags() and HandleTag() for each handler to identify which tags each handler supports and to cause a handler to process a given tag.

When processing a tag, it is sometimes convenient for the tag handler to be able instruct the parser to continue the sub-nodes of this tag. The protected ParseInner() method allows us to do exactly this.

We want to be able to create instances of tag handlers conveniently in R using R functions to implement the methods. We can do this by creating a generic extension of the basic wxHtmlTagHandler class. We'll call this derived class RwxHtmlTagHandler.

```R
class RwxHtmlTagHandler : public wxHtmlTagHandler {

};
```

What we want for this class is a way to map R functions to methods. If there is an R function provided for a given method, then the implementation of that method for this class calls that R function. If there is no R function for that method, then we call the inherited method from the base class. We can define

a protected/private field in the class for each method and have the method refer to that to see if the R implementation of the method is available. Alternatively, we can maintain a (hash) table of R functions identified by the method "name" and look in that central location for a method's R implementation. Either approach works fine.

Let's be concrete with our example. Our derived class would provide a method for the SetParser method as

```
                                                                                     R
wxString SetParser(wxHtmlParser *parser) {

  if(R_SetParser_m == R_NilValue) {
     wxHtmlTagHandler::SetParser(parser);
  } else {
     SEXP e, ans;
     int errorOccurred = 0;

     PROTECT(e = allocVector(LANGSXP, 3));
     SETCAR(e, R_SetParser_m);
     SETCAR(CDR(e), R_make_wxWidget_Ref(this, "RwxHtmlTagHandler"));
     SETCAR(CDR(CDR(e)), R_make_wxWidget_Ref(parser, "wxHtmlParser"));

     ans = R_tryEval(e, R_GlobalEnv);

     UNPROTECT(1);
  }
}
```

We haven't dealt in this code with raising a C++ exception if the evaluation of the R call throws an error. Note also that we have added an argument 'this' as the first argument to the R function. If we controlled the environment of the R functions, we could assign the object 'this' into that environment. However, this is somewhat tricky to do generally when the R programmer is leveraging the functions' environments to implement shared variables.

The default constructor for RwxHtmlTagHandler sets the R_SetParser_m method field to NULL or R_NilValue. Other constructors can set the value to a legitimate R function.

We are unlikely to want to override this particular method. The ones we are interested in from R are the two work-horse methods: GetSupportedTags() and HandleTag(). Interestingly, both of these are virtual methdods that we can override (like SetParser), but also they are pure methods defined in the base class wxHtmlTagHandler. This means that we have to provide implementations for them in our derived class! And we cannot invoke the inherited method. So the C++ code for GetSupportedTags() in our derived class might be

```
                                                                                     R
wxString GetSupportedTags() {

  if(R_GetSupportedTags_m == R_NilValue) {
     throw RUnimplementedMethod("GetSupportedTags", "RwxHtmlTagHandler");
  }

  SEXP e, r_ans;
  int errorOccurred = 0;
```

```
    PROTECT(e = allocVector(LANGSXP, 2));

    SETCAR(e, R_GetSupportedTags_m);
    SETCAR(CDR(e), R_make_wxWidget_Ref(this, "RwxHtmlTagHandler"));

    r_ans = R_tryEval(e, R_GlobalEnv);

    PROTECT(r_ans);
    if(TYPEOF(r_ans) != STRSXP)
        throw RIncorrectMethodReturnType(r_ans, "string", "GetSupportedTags", "RwxHtm

    wxString ans;
    ans = R_to_wxString(r_ans);

    UNPROTECT(2);
    return(ans);
}
```
(We need to unprotect the stack in our exception handler.)

Rather than raising an error if there is no R function specified for this method, we may just insist that it is not possible to create or modify this object without such a function being available. In other words, one cannot call a constructor function and give an invalid value for this field, and one cannot set the value of the field during the lifetime of the object to be something inappropriate. We can do this by controlling the setting of this field.

R

```
void R_GetSupportedTags(SEXP val) {
    if(val == NULL || LENGTH(val) == 0)
      throw RInvalidMethodFunction("GetSupportedTags", "RwxHtmlTagHandler");

       /* Perhaps check on the number of formal arguments >=
          that of the C++ method (0 in this case). */
    R_GetSupportedTags_m = val;
  }
```

The essential details of the methods we have sketched are quite easy to automate. We start from a description of the base class and its methods. For each method, we define a field in the derived class named R_method_name_m, where the specific name of the method is substituted for "method_name". Then, we implement a C++ method that checks the value of this field to see if it is set and arranges to call the R function, if it is. The call marshals the arguments given in the C++ method call to R objects and inserts them sequentially into the function call expression. How this marshaling is done depends on the data types of the parameters and is handled by the usual code we use for marshaling for C routines. We invoke the function by evaluating the call expression and then we marshal the result back and return it from the C++ routine.

If the R function method field has not been set, then we call the inherited method, unless that is a pure method in the base class. If it is, we signal an error when we discover the missing/unspecified R function.

There are several others issues that remain in this process. The first is how to deal with overloaded or polymorphic methods defined in the base class and consequently in the derived class. Secondly, we need

to specify constructor methods and allow the creator of the class to provide R implementations of these constructors. And the third is concerned with protected methods and how we can make these accessible from R and still be protected. And lastly, we need to deal with protected and public fields and make these available to the R methods, i.e. provide accessors to them.

Overloaded methods

Let's consider a different example that we construct for our own discussion purposes here. Suppose we have two methods in our base class A named plot() and that one takes an array of integer values and a length, and the other takes a wxArrayInt object which contains knowledge of its length.

R
```
class A {
  public:
    void plot(int *vals, int n);
    void plot(wxArrayInt &vals);
};
```

Now, our basic algorithm that uses the name of the method for the field name to hold the R function breaks down. We have two methods that map to the same name. Of course, there is an obvious mechanism to deal with this and that is to use the names of the parameter types in each method's name and we get unique names. That is, we would use plot_int_int and plot_wxArrayInt as the method identifiers and define fields R_plot_int_int_m and R_plot_wxArrayInt_m. The names are not important for the C++ code that we generate. They are important for the R user trying to specify a method for one of these plot() methods and trying to identify the correct one. For the present, this name mangling scheme will work fine and the R programmer will have to navigate it. The class will have documentation (both static and dynamic) to illustrate the mapping of the names to the method definition.

Protected Methods

The wxHtmlTagHandler class has the ParseInner method. Only the object should be able to see and invoke this method, or more specifically, only the methods for this derived class should be able to invoke this method. We can generate the C++ and R wrapper routines and functions respectively to access the protected method in the usual way by first creating a public method in the derived class that calls the protected method. This is just a proxy that avoids calling the protected method from code outside of the class. This could be done via a friend class or by creating a new public method in the derived class, e.g.

R
```
public void RParseInner(const wxHtmlTag &tag) {
     ParseInner(tag);
 }
```

With this added to the derived class, we can use our regular tools to generate an R interface to this method.

As with protected fields, we want to ensure that only those functions are entitled to invoke protected methods. So we need to limit the visibility. We do not want to allow arbitrary code to be passed a reference to the object and be able to access its protected methods and fields. Within C/C++ code, the compiler enforces the correct access to the class methods. However, when we implement a method within an R function and want to access a protected method of field within the base class, the compiler cannot help use. Instead, when we transfer control from the R function to invoke a C++ method, we go through a proxy routine. It can check

whether the object is legitimately calling a protected method. This checks the mark that the derived class method assigned to the object when invoking the R method function acting as the implementation of that method in the derived class. Having verified the legitimacy of the call to a protected method, it invokes a public wrapper method in the derived class which then invokes the actual inherited protected method in the base class. To be specific, if we have a class A with a protected method foo, and we create a derived class RA, we would create a new method in RA named, e.g., Rfoo. This has the same signature as the method A::foo() and the implementation of RA::Rfoo() is merely

```R
RA::foo(...) {
   A::foo(...);
}
```

One approach to doing this is to mark the R object that is passed to

Inherited Methods

Often, we implement a virtual method and want to call the inherited method in the parent class as part of that method. For example, we want to do what the parent class does, but then adapt the result or add some action. Alternatively, we may want to perform so computations before calling the inherited method or call it conditionally depending on the inputs and the state of our object. Regardless of the reason, we want our "method function" to be able to do something like:

```R
foo =
function(this, arg1, arg2)
{
   arg1 = 2 * arg1
   ans = foo(this, arg1, arg2, super = TRUE)
   sqrt(ans)
}
```

Here we do some pre-processing of the arguments, call the inherited method and then post-process the result.

How can we achieve this?
<fix> We need to invoke the call to the inherited or base-class method from within the body of a method of our derived class. We cannot do this directly from arbitrary C/C++ code. This is untrue!!!!!! See the way we have done this in the past within this package! </fix>
So we need to have a method within our derived class that calls the inherited method. One approach for this is the following. For each virtual method such as our method

```R
foo(int arg1, int arg2)
```

we define a corresponding method with an additional parameter which is used to indicate a call to the inherited method. Such methods might be defined, for example, as

```R
foo(int arg1, int arg2, bool super)
{
  if(super)
     return(BaseClass::foo(arg1, arg2));
  else
```

```R
      return(foo(arg1, arg2));
}
```

What this means is that for each virtual method in our base class, we need to define a new method which adds a single argument. The particular type is not important. However, it should be such that it does not conflict with any overloaded methods. For example, if we had two methods named foo defined as

```R
foo(int arg1)
foo(int arg1, bool arg2)
```

the adding a bool parameter to each would lead to confusion. We would have

```R
foo(int)
foo(int, bool)
foo(int, bool )
foo(int bool, bool, bool)
```

and so two separate instances of

```R
foo(int, bool)
```

For this reason, we can introduce a new type which is named InheritedMethodIndicator variable. The we can introduce

```R
void foo(int arg1, InheritedMethodIndicator super)
void foo(int arg1, bool arg2, InheritedMethodIndicator super)
```

and there is no conflict. Only if somebody has a type with this name do we have difficulties. So we allow this to be specified in the code generation to avoid such situations. We can also detect whether such a type is present in the TU.

With these additional methods defined in the derived class, we can then generate C++ routines tied to the newly created derived class by which we can invoke them from R. We generate wrapper routines that are something similar to the following:

```R
R_RMyClass_foo(SEXP r_this, SEXP r_arg1, SEXP r_super)
{
  InheritedMethodIndictor super = (InheritedMethodIndicator) INTEGER(r_super)[0];
  int arg1 = INTEGER(r_arg1)[0];
  RMyClass *This;
  This = (RMyClass*) R_getNativeReference(r_this, "RMyClass", NULL);

  This->foo(arg1, super);
  return(R_NilValue);
}
```

And we implement the new method in the derived class as

```R
void RMyClass::foo(int arg1, InheritedMethodIndicator super)
{
   if(super)
     MyClass::foo(arg1);
```

```
    else
      foo(arg1);
}
```

We generate equivalent R code to invoke this method

```R
setMethod("foo", c("RMyClass", "integer", "InheritedMethodIndicator"),
    function(this, arg1, super) {
      .Call("R_RMyClass_foo", this, arg1, super)
    }
```

Fields

Consider our initial example of wxHtmlTagHandler. There is a protected field named m_Parser. A derived C++ class will be able to access that field directly rather than having to use an accessor function. An R function that acts as a method for a derived class of course does not have direct access to this field. We can generate an accessor function for this field and provide an R function that calls this to either set or get the value of the field. Providing this accessor method and an associated R function is relatively straightforward and mimics what we do for public fields in a C++ class or fields in a C struct.

For example, in our example, we have one protected field, m_Parser. We provide a C++ accessor routine

```R
SEXP R_wxHtmlTagHandler_get_m_Parser(SEXP r_This) {
      wxHtmlTagHandler *This;
      This = (wxHtmlTagHandler *) R_wx_widget_Ref(r_This, "wxHtmlTagHandler");
      SEXP r_ans;
      r_ans = R_make_wxWidget_Ref(This->m_Parser, "wxHtmlParser");
      return(r_ans);
  }
```

And the associated R function is

```R
wxHtmlTagHandler_m_Parser =
function(this)
  .Call("R_wxHtmlTagHandler_get_m_Parser", this)
```

The aspect that is different about providing access to a protected field is that it is not intended that general code can access this field. Of course, we have provided an accessor method and so it is still hidden. However, it does display the implementation and breaks the idea of data encapsulation and hiding. What we would like to do ideally is to ensure that only the R functions that act as methods for the new class are able to invoke this accessor method.

There are several approaches we might use for this. One is to change the environment of the functions when they are specified as the R functions for the C++ methods. We can set the environment of each function to a special environment that contains the accessor functions for the different fields. Only functions used in this way would be able to see these functions and so only the method functions would be able to access the protected fields.

The downside to this approach is that the programmer providing the R functions may have explicitly created them to share an environment so that they can modify and access shared R values using closures. We

don't want to break that model and so we cannot simply reset the environment of each function. We could, however, identify the top-most environment of each function before the global environment and reparent those to insert our own environment between that top-most one and the global environment. If the functions initially have the global environment as their environment, these approaches are the same, i.e. this more elaborate case is the more general version of the same thing.

In our environment, we would assign the function wxHtmlTagHandler_m_Parser and a method for [[for the class RwxHtmlTagHandler so that the fields could be accessed as `this[["m_Parser"]]`. Then, we walk the hierarchy of environments of each function until we reach the global environment and then we insert this instance specific environment. Unfortunately, that will affect all other users of this environment, so again, this will only work effectively in simple cases where the functions have no special environment.

An alternative solution is Luke Tierney's suggestion for mimicing a weak (non-mutable) form of closures in S-Plus. This involves adding additional parameters with default values to each of the functions and having the default values be expressions that evaluate to the fields. For example, the R programmer might create an instance of our RwxHtmlTagHandler class with a call that specifies the two function methods as

```R
RwxHtmlTagHandler(
  GetSupportedTags = function(this) {
     "myTag"
  },
  HandleTag = function(this, tag) {
       win = m_Parser$GetWindow()
       btn = wxButton(win, wxID_ANY, "My Label")
       insertEmbeddedComponent(btn, m_Parser)
     }
 )
```

The constructor function would then modify these two functions to add a parameter m_Parser, as referenced in the second of these functions - HandleTag. That function would become

```R
function(this, tag, m_Parser = wxHtmlTagHandler_m_Parser(this)) {
       win = m_Parser$GetWindow()
       btn = wxButton(win, wxID_ANY, "My Label")
       insertEmbeddedComponent(btn, m_Parser)
     }
```

without any intervention by the R programmer. When m_Parser is referenced, the default value is computed and all behaves normally.

This will work well but as the number of fields increases, the number of parameters will also. This is not really a problem as the default values are only computed if the fields are referenced in the body of the R function. We do not pay a computational penalty for each call.

There is still the problem that other R functions can access these methods. If the code were in a package name space, we could hide the accessors, but this is too limiting at present. It is not obvious we should care as programmers who access these protected fields from non-method functions should know they are protected and realize that they are taking advantage of implementation-specific details that are not necessarily going to be available in future versions. However, in spite of this practical consideration, it is nice to figure out a mechanism which preserves the C++ semantics that make the field more protected than simply relying on a programmer's sense.

So another approach is to make the instance of the this argument into a special R class when passed to these function methods. The instance of the *this* argument would be similar to the object returned by the constructor for *RwxHtmlTagHandler()*, but would have extra properties and methods. The *[[()* method for this extended type would be able to access the protected fields by name, e.g.

```R
HandleTag =
 function(this, tag) {
        win = this[["m_Parser"]]$GetWindow()
```

but the same code would not work outside of these methods. That is,

```R
obj =
RwxHtmlTagHandler(
  GetSupportedTags = function(this) {
     "myTag"
  },
  HandleTag = function(this, tag) {
        win = m_Parser$GetWindow()
        btn = wxButton(win, wxID_ANY, "My Label")
        insertEmbeddedComponent(btn, m_Parser)
     }
 )
obj[["m_Parser"]]
```

But this is still not quite what we want. If we introduce new methods that are not inherited from the base class, then we want the object to be able to access the protected fields. What we want is to stop other code from accessing the field. In this respect, we want to "bless" these additional functions to have access to the protected fields. And we can do this passing them to the constructor function and having them be carried with the object and accessed via an extended $ operation. And these methods would coerce the *this* argument to the extended R type with the specialized *[[()* method for accessing the protected fields. If this object is passed to another function, then we don't want code in that other function to be able to access the protected members of this object. So we would want to coerce the value back to its simple form without the access to the protected members. But we don't have the opportunity to do that.

So somehow when we call the protected method, we want to verify that it is being called from a function method for this object. The wrapper for the protected method can look at the call stack and verify that the calling function is such a function method. That calling function must have some distinguishing feature. This could be a class or another attribute that identifies it as such. Anyone who goes to the effort to fake this attribute is clearly breaking data encapsulation and "deserves" access!

Alternatively, we can return to our approach of adding parameters to the function methods or generally modifying the code in the function body. If we add a single parameter, say named *.key*, to each of the methods and give it a default value, this key can be used to perform a hand-shake to gain access to the wrappers for the protected members. Again, we rewrite the functions passed to the constructor functions, converting

```R
HandleTag =
   function(this, tag) {
       this[["m_Parser"]]$GetWindow()
```

```
    }
```

to

```
HandleTag =
    function(this, tag) {
     .key = 12413213
     this[["m_Parser"]]$GetWindow()
    }
```

where .key is some random number that is inserted into the body of the function and stored in the instance of the object. Only those functions that can provide the correct key can access the protected member wrapper functions successfully.

The [[method for our *this* object fetches and validates this **.key** variable using the code

```
setMethod("[[", c("RwxHtmlTagHandler"),
function(x, name) {
 parentEnv = sys.frames()[[sys.parent()]]
 if(!exists(".key", parentEnv))
   stop("Invalid access to protected member of ", class(x))

 k =
 if(get(".key", parentEnv)  != .Call("R_RwxHtmlTagHandler_R_getKey", x))
    stop("Access to protected member with incorrect key")

 .Call(paste("R_RwxHtmlTagHandler", name, sep = "_"), x)
})
```

There is potentially non-trivial overhead in the computations here. This is unfortunate, and the mechanism can be foiled by those who are prepared to fetch the key and insert it into the body of their calling function. But this has to be done dynamically as the key for each instance is different. Again, for those who really want to do this, they can, as they can in C++ by knowing offsets in the representation of the C++ object!

Example 1.

Constructors

Now we have discussed the different aspects of the derived class and the R code to implement methods, we are ready to think about the user interface to these classes for the R programmer. We have one single derived C++ class, but we can use it to provide R implementations merely by providing different functions as methods.

The R constructor function will of course take a list of functions that are to be used to implement the C ++ methods. These are given by name corresponding to the method name, be it simple or mangled. The methods that are pure in the base class become required arguments in our constructor. One can pass a value of **NULL** or function(...) {} as a no-op which will be detected only if the C++ method is actually invoked. So the constructor function takes shape as

```r
RwxHtmlTagHandler =
function(GetSupportedTags,
         HandleTag,
         SetParser = NULL,
         GetClassInfo = NULL,
         ParseInner = NULL,
         dynamic = FALSE,
         .key = runif(1)
         )
```

If we want to be able to change the methods during the life of an instance of this derived C++ class, we pass `dynamic = TRUE`. Otherwise, we raise an error in the accessor functions that set the R function methods.

It is often convenient to create the methods within another function and pass them to this function as a named list of functions. This allows us to use a closure to allow the function methods to share variables. So we might define a generator function that returns the collection of methods. In many respects, this is the R-side constructor and we can have different ones. It is slightly tedious to call this generator function, assign the result and then pass each element to the constructor. Rather, it is easier to pass the list of methods and have the constructor assign the elements to the individual variables. This provides two ways for the caller to provide the function methods.

For example, we may have a constructor function that we use to create handler methods for different tags

R
```r
tagHandlerConstructor =
function(supportedTags)
{
  list(GetSupportedTags =
              function()
                 supportedTags,
       HandleTag =
              function(this, tag) {
                  parser = this[["m_Parser"]]
          win = parser$GetWindow()
                  btn = wxButton(win, wxID_ANY, "My Label")
                  insertEmbeddedComponent(btn, )
                }
      )
}
```

and we would call this with the name of the tag that this was to process:

R
```r
tagHandlerConstructor("myTag")
```

And we would pass this to the *RwxHtmlTagHandler*() constructor function as

R
```r
RwxHtmlTagHandler(.methods = tagHandlerConstructor("myTag"))
```

And the body of that constructor function would process the elements of *.methods*.

R
```r
curFrame = sys.frame(sys.nframe())
for(i in names(.methods)) {
```

```
    assign(i, .methods[[i]], curFrame)
}
```

This is the general constructor that does not deal with calling the constructors of the base class. This is important as we want to be able to initialize them appropriately and we only get to do this (in general) when the C++ object is created. So our constructor(s) in R need to be able to pass the information to the newly constructed derived class and then call the relevant constructor of the base class. This is slightly more complex than we would like as we have to do this in the C++ code rather than in R as the C++ constructors are not callback directly, but only as part of the constructor for our derived class.

One might think that we have to set the R function method fields before calling the base constructor. This is sensible since the base constructor may call a virtual method that we are implementing with our R function. In fact, this won't happen - at least with g++.

> **Note**
> look up specification for C++.

Rather, the base constructor will end up calling its own methods and if they are pure (i.e. unimplemented) one can end up with an error of the form

R
```
pure virtual method called
terminate called without an active exception
Abort
```
from the g++ compiled code, at least. (If the constructor calls a pure virtual method, the compiler will signal an error. If it indirectly calls such a method, it is caught at run-time.)

Given this information, we don't need to set the values before the call to the base constructor. We could do this however, if we wanted. For example, suppose the base class wxHtmlTagHandler had a constructor of the form (which it doesn't)

R
```
wxHtmlTagHandler(const char *tagName);
```
Then the following C++ code will perform the creation of our derived class instance, setting the values of fields and then calling the base constructor. (The exact order is probably compiler-specific.)

R
```
RwxHtmlTagHandler(const char *tagName,
              SEXP GetSupportedTags, SEXP HandleTag, SEXP SetParser,
               SEXP GetClassInfo, SEXP ParseInner,
                bool dynamic, SEXP key)
        :  R_GetSupportedTags_m(GetSupportedTags),
           R_HandleTag_m(HandleTag),
           ...,
           wxHtmlTagHandler(tagName)
{

}
```

Alternatively, we can set the values in the body of the constructor since they will not be called during the constructor from the base class.

Since the virtual methods in the derived class will not be called from within the base constructor, we can, in fact, set the R functions as methods after calling the base constructor. And, we can do this within the single C++ constructor code for our derived class or from within the R-language constructor function in a separate call. That is, we can define the following C++ constructor

```R
RwxHtmlTagHandler(const char *tagName,
                  SEXP GetSupportedTags, SEXP HandleTag, SEXP SetParser,
                   SEXP GetClassInfo, SEXP ParseInner,
                    bool dynamic, SEXP key)
          :       wxHtmlTagHandler(tagName)
{
  _setMethods(GetSupportedTags, HandleTag, SetParser, GetClassInfo, ParseInner);
}
```

and then call it from R as

```R
RwxHtmlTagHandler =
function(tag,
         GetSupportedTags,
         HandleTag,
         SetParser = NULL,
         GetClassInfo = NULL,
         ParseInner = NULL,
         dynamic = FALSE,
         .key = runif(1)
        )
{
 .Call("R_wxHtmlTagHandler_new", as.character(tag),
        GetSupportedTags, HandleTag, SetParser, GetClassInfo, ParseInner, dynamic
}
```

which, in turn, calls our constructor.

Or alternatively, we can define the C++ constructor as

```R
RwxHtmlTagHandler(const char *tagName)
          :       wxHtmlTagHandler(tagName)
{
}
```

and define our

```R
RwxHtmlTagHandler =
function(tag,
         GetSupportedTags,
         HandleTag,
         SetParser = NULL,
         GetClassInfo = NULL,
         ParseInner = NULL,
         .dynamic = FALSE,
```

```
        .key = runif(1)
        )
{
 .Call("R_wxHtmlTagHandler_new", tag)
  # Set the methods, etc. after we have constructed the object.
 RwxHtmlTagHandler_setMethods(GetSupportedTags, HandleTag, ..., .dynamic, .key)
}
```

Both should work as there is no opportunity to call one of the methods that we have overridden.

We should note that we have to provide the method for handling the dynamic and key values. These are common to all derived classes that we create. Since we are automating the construction of the code, replicating this code is not an onerous task. However, it makes sense to move this to a special class that handles this for us and then have our derived class use multiple inheritance and inherit from both the primary base class and this helper class, which we will call RDerivedClass. So our constructor code becomes,

R
```
RwxHtmlTagHandler(const char *tagName,
                  SEXP GetSupportedTags, SEXP HandleTag, SEXP SetParser,
                   SEXP GetClassInfo, SEXP ParseInner,
                    bool dynamic, SEXP key)
         :   R_GetSupportedTags_m(GetSupportedTags),
             R_HandleTag_m(HandleTag),
             ...,
             wxHtmlTagHandler(tagName), RDerivedClass(key)
```

Where we handle the dynamic flag is debatable as we gain very little by deferring it to RDerivedClass. We still have to generate the code to set the function object for each method.

Constructors

One of the things that we have not dealt with in this section is how we handle multiple inheritance and calling constructors for different classes. For example, suppose the class C is derived from both A and B and that A has constructors defined as

R
```
A(int x, double y)
A(bool val)
```

and B has constructors

R
```
B(int z)
B(char *str)
```

Then, there are 4 possible combinations of constructors - the product of the number of constructors in each class. So, our derived class C might provide

R
```
C(int x, double y, int z) : A(x, y), B(z) {}
C(int x, double y, char *str) : A(x, y), B(str) {}
C(bool val, int z) : A(val), B(z) {}
C(bool val, char *str) : A(val), B(str) {}
```

So far, so good.

Now, how do we identify these in R. Well, we can use name mangling as usual. But consider the case where we have constructors for our base classes

```
A(int x, int y)
A(int x, int y, int z)
B()
B(int a)
```

Then, we would end up with 4 constructors in our derived class that would look like

```
C(int x, int y)  // A(int x, int y) & B()
C(int x, int y, int a) // A(int x, int y) & B(int)
C(int x, int y, int z) // A(int x, int y, int z) & B()
C(int x, int y, int z, int a) // A(int x, int y, int z) & B()
```

The simple rule of combining the signatures of the base class constructors leads to a duplication. The middle two constructors have same signature but are quite different. The first passes all of its arguments to A and uses the default constructor for B. This is a regular C++ difficulty, and not something introduced by our interface. The issue manifests itself in C++ and also in defining methods for the R-level constructor. Solving the C++ issue will lead to a natural disambiguation in R also.

Dynamic versus Fixed classes

We should note that the scheme we have outlined in this document means that the R functions implementing C++ methods are only specified at the time the instance of the derived C++ class is created. They are not fixed and can be changed for each instance of the derived class. In this way, we can have numerous instances of the derived class that behave quite differently. The intent is that this one derived C++ class acts as a template for several R classes and that these R classes are fixed in their behavior and specification of R function methods. In this way, when one has an instance of a particular R class, one knows what the associated methods are and there is no ambiguity. That said, it is convenient for debugging and development purposes to be able to alter the methods and slide different ones into an existing instance of the derived C++ class.