Introduction

When we are using R6 objects and want to introduce some C++ code in our project, we may also want to interact with these objects using Rcpp. With this objective in mind, the key to interacting with R6 objects is that they are essentially environments, and so they can be treated as such.

In this example, we will define a ‘Person’ class with some private attributes and public methods. It will allow us to show how to create R6 objects in C++ and how to interact with them.

Creating a dummy R6 class

First, we define the class ‘Person’, with a name, an id, an item and some methods:

```
library(R6)

#' R6 class that defines a person
Person <- R6::R6Class("Person",
  public = list(
    #' @description
    #' Constructor of the 'Person' class
    #' @param name a string that defines this person's name
    #' @param id an integer that defines this person's id
    #' @return A new 'Person' object
    initialize = function(name, id){
      private$name <- name
      private$id <- id
    },

    get_name = function(){return(private$name)},

    get_id = function(){return(private$id)},

    #' @description
    #' Gives an item to the person
    #' @param item a string that defines the item
    give_item = function(item){private$item <- item},

    #' @description
    #' A public function that calls a private one
    good_foo = function(){
      return(paste0("Wrapped inside: {", private$bad_foo(), "}"))
    }
  ),
  private = list(
    #' @field name the name of the person
    name = NULL,
    #' @field id the id of the person
    id = NULL,
    #' @field item some item that the person has
    item = NULL,

    #' @description
    #' A private function that should not be called from outside the
object
```

```
    bad_foo = function(){return("This is a private function")}
  )
)
```

With this simple class, we will be able to create R6 objects, initialize them and call some of their methods.

Creating R6 objects in C++

To create R6 objects, we must first get the ‘new’ function that initializes them. In this case, given that we have loaded the Person class into the global environment, we will get it from there:

```
#include
using namespace Rcpp;

// [[Rcpp::export]]
List initialize_list(StringVector &names, IntegerVector &ids){
  List res(names.size());
  Environment g_env = Environment::global_env();
  Environment p_env = g_env["Person"];
  Function new_person = p_env["new"];

  for(unsigned int i = 0; i < names.size(); i++){
    Environment new_p;
    String name = names[i];
    int id = ids[i];
    new_p = new_person(name, id);
    res[i] = new_p;
  }

  return res;
}
```

```
names <- c("Jake", "Anne", "Alex")
ids <- 1:3

res <- initialize_list(names, ids)

print(res)
```

The previous example initializes a list with Persons. It is important to notice that the only function relevant to us from the global environment is the ‘new’ function created by the R6 class. All the functions defined inside Person will be contained in each instance of the class.

If the R6 object is defined inside some package, included our own package if we are developing one, we must get it from there. To do this, the previous function has to be modified as follows:

```
#include
using namespace Rcpp;

// [[Rcpp::export]]
List initialize_list(unsigned int size){
  List res(size);
  Environment package_env("package:some_package");
  Environment class_env = package_env["some_class"];
  Function new_instance = class_env["new"];
  for(unsigned int i = 0; i < size; i++){
    Environment new_i;
    new_i = new_instance();
    res[i] = new_i;
  }

  return res;
}
```

Calling the methods of an instance

Once we have instantiated an R6 object as an environment, we can call its methods after getting them from the instance:

```
#include
using namespace Rcpp;

// [[Rcpp::export]]
Environment create_person_item(){
  Environment g_env = Environment::global_env();
  Environment p_env = g_env["Person"];
  Function new_person = p_env["new"];
  Environment new_p;
  String name = "Jake";
  int id = 1;
  String item = "shovel";

  new_p = new_person(name, id);
  Function give_i = new_p["give_item"];
  give_i(item);

  return new_p;
}

res <- create_person_item()

print(res)
```

Note that if we are creating multiple instances, we have to get the desired method from each one of them.

Private attributes and methods cannot be accessed in this manner, trying to do so results in an error:

```
#include
using namespace Rcpp;

// [[Rcpp::export]]
String access_method(std::string foo_name){
  Environment g_env = Environment::global_env();
  Environment p_env = g_env["Person"];
  Function new_person = p_env["new"];
  Environment new_p;
  String name = "Jake";
  int id = 1;

  new_p = new_person(name, id);
  Function foo = new_p[foo_name];
  String res = foo();
  return res;
}

tryCatch({res <- access_method("bad_foo")},
         error=function(cond){print(paste0("Exception raised: ",
cond))})

res <- access_method("good_foo")

print(res)
```

The error is telling us that there is no function called “bad_foo” in the environment of the object “new_p”. In that environment, only public attributes and methods are present on the surface. Theoretically, we could still try to find the private attributes and methods inside the object, but it goes against the paradigm and it would be much easier to just make the method we want to call public.

Passing down R6 objects

Passing R6 objects as arguments to C++ functions is also straight forward by treating them as environments:

```
#include
using namespace Rcpp;

// [[Rcpp::export]]
void give_shovel(Environment &person){
  String item = "shovel";

  Function give_i = person["give_item"];
  give_i(item);
}

p <- Person$new("Jake", 1)

give_shovel(p)

print(p)
```

```
Public:
  clone: function (deep = FALSE)
  get_id: function ()
  get_name: function ()
  give_item: function (item)
  good_foo: function ()
  initialize: function (name, id)
Private:
  bad_foo: function ()
  id: 1
  item: shovel
  name: Jake
```

Note that there is no need to return the object, as the environment is passed by reference and changes inside it will be reflected outside the function.