Chapter 25: Rewriting R Code in C++

Ryan Heslin

May 30, 2022

```
library(Rcpp)
```

Declare a simple function.

```
cppFunction("int myprod(int x, int y, int z) {
  int myprod = x * y * z;
  return myprod;
}")
myprod
```

```
function (x, y, z)
.Call(<pointer: 0x7f8d5cd59180>, x, y, z)
<environment: 0x559c8f58f008>
```

```
#> function (x, y, z)
#> .Call(<pointer: 0x107536a00>, x, y, z)
myprod(1, 2, 3)
```

```
[1] 6
```

```
#> [1] 6
```

A naiive summing for loop. Much faster than implementing this in R.

```
cppFunction("double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}")
```

It is usually best to store C++ code in `.cpp` files with the following header:

```
#include <Rcpp.h>
using namespace Rcpp;
```

All exported functions need to be prefixed:

```
// [[Rcpp::export]]
```

The `SourceCPP` function sources these.

1.

Respectively `mean`, `cumsum`, `any`, `which.min`, and `pmin`

```
cppFunction("bool allC(LogicalVector x) {
int n = x.size();
for(int i = 0; i<n; ++i) {
    if( ! x[i]) {
        return false;
    }
}
return true;
}")
test <- as.logical(sample(0:1, 100000, replace = TRUE))

bench::mark(
  all(test),
  allC(test)
)
```

```
# A tibble: 2 x 6
  expression      min   median ‘itr/sec‘ mem_alloc
  <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
1 all(test)  140.98ns 150.99ns  5196126.        0B
2 allC(test)   1.09us   1.18us   714592.    2.49KB
# ... with 1 more variable: ‘gc/sec‘ <dbl>
```

Below, I show that the CPP functions behave the same as their R equivalents.

```
library(testthat)
sourceCpp(here::here("misc/functions.cpp"))
expect_equal(cum_min(mtcars$cyl), cummin(mtcars$cyl))
expect_equal(cum_min(mtcars$cyl), cummin(mtcars$cyl))
expect_equal(cum_prod(mtcars$cyl), cumprod(mtcars$cyl))
expect_equal(cummin(c(1, 2, NA, 7, 4)), cum_min(c(1, 2, NA, 7, 4)))
expect_equal(cummin(c(NA, 1, 2)), cum_min(c(NA, 1, 2)))
```

```
write_test <- function(test_fun, args) {
  bquote()
}
library(testthat)
test <- sample(100)
with_nas <- ifelse(test < 10, NA, test)
expect_equal(diff(test, lag = 1, differences = 4), diffC(test, 1, 4))
expect_equal(diff(test, 9, 1), diffC(test, 9, 1))
expect_equal(diff(test, 9, 3), diffC(test, 9, 3))
expect_equal(diff(test, 1, 1), diffC(test, 1, 1))
expect_equal(diff(with_nas), diffC(with_nas))
expect_equal(diff(with_nas, 7, 14), diffC(with_nas, 7, 14))
expect_equal(diff(rep(NA, 100), 7, 14), diffC(rep(NA, 100), 7, 14))
```

I'm too fast.

```
bench::mark(diff(test, 5, 5), diffC(test, 5, 5))
```

```
# A tibble: 2 x 6
  expression             min   median `itr/sec`
  <bch:expr>        <bch:tm> <bch:tm>     <dbl>
1 diff(test, 5, 5)    9.21us  10.42us    83145.
2 diffC(test, 5, 5)   2.37us   2.73us   300097.
# ... with 2 more variables: mem_alloc <bch:byt>,
#   `gc/sec` <dbl>
```

```
expect_equal(range(test), rangeC(test, FALSE))
expect_equal(range(1L), rangeC(1L, TRUE))
expect_equal(rangeC(c(NA, NA, 1), TRUE), c(1, 1))
expect_equal(rangeC(c(NA, NA), TRUE), c(Inf, -Inf))
```

```
with_nas <- c(1, 2, 3, NA)
expect_equal(var(-(1:10)), varC(-(1:10)))
expect_equal(varC(1), NA_real_)
expect_equal(var(test), varC(test))
expect_equal(var(with_nas, na.rm = TRUE), varC(with_nas, na_rm = TRUE))
expect_equal(varC(with_nas), NA_real_)
expect_equal(var(ifelse(test < 10, NA, test), na.rm = TRUE), varC(ifelse(test < 10, NA, test), na_rm = T

bench::mark(var(test), varC(test))
```

```
# A tibble: 2 x 6
  expression      min   median `itr/sec` mem_alloc
  <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
```

```
1 var(test)    6.19us   6.89us   137250.      848B
2 varC(test)   1.88us   2.12us   435206.    3.32KB
# ... with 1 more variable: ‘gc/sec‘ <dbl>
```

Other Classes and NAs

This is the trick for calling R functions from C:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
RObject callWithOne(Function f) {
  return f(1);
}
```

Attribute modification:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector attribs() {
  NumericVector out = NumericVector::create(1, 2, 3);

  out.names() = CharacterVector::create("a", "b", "c");
  out.attr("my-attr") = "my-value";
  out.attr("class") = "my-class";

  return out;
}
```

Scalars, especially `int`, have various annoying hacks for representing `NA`. Luckily, there are type-specific `NA` constants for vector types.

The Standard Template Library

Where the real power comes in, starting with efficient accumulators.

STL vectors are mutable. They can be subset with `[` and efficiently grown with `.push_back()`to add a new element. Sets behave as you’d expect. Maps store key-value pairs.

I couldn’t replicate `%in%`’s behavior with `NA`, but I question whether it’s sensible anyway.

```
# Not standard NA behavior, but more sensible
test <- c(3, 4, 5, 6, -5, NA)
expect_equal(1:3 %in% test, inC(1:3, test))
# expect_equal(NA %in% test, inC(NA_real_, test))
expect_equal(test %in% 1:5, inC(test, 1:5))

expect_equal(UniqueCC(test, numeric()), unique(test))
expect_equal(UniqueCC(c(1, 1, 2, 3), 1), unique(c(1, 1, 2, 3), incomparables = 1))

expect_equal(which.max(test), whichmaxC(test))

expect_equal(median(test, na.rm = TRUE), medianC(test, TRUE))
expect_equal(median(test), medianC(test))
expect_equal(median(1:10), medianC(1:10))
expect_equal(median(3), medianC(3))

x <- c(1, 4, 0, -1)
y <- 10:15
z <- 1:5
nil <- integer()
expect_equal(intersect(x, y), intersectC(x, y))
expect_equal(intersect(x, z), intersectC(x, z))
expect_equal(intersect(x, nil), intersectC(x, nil))

expect_equal(union(x, y), unionC(x, y))
expect_equal(union(x, z), unionC(x, z))
expect_equal(union(x, nil), unionC(x, nil))

expect_equal(setdiff(x, y), setdiffC(x, y))
expect_equal(setdiff(x, z), setdiffC(x, z))
expect_equal(setdiff(x, nil), setdiffC(x, nil))
expect_equal(setdiff(y, z), setdiffC(y, z))
expect_equal(setdiff(nil, x), setdiffC(nil, x))
```

I can't quite get this to handle NAs correctly. I'll need a fresh reproach.

```
maxC <- function(..., na_rm = FALSE) maxC_impl(list(...), na_rm)
expect_equal(max(1:5, 1:4), maxC(1:5, 1:4))
expect_equal(max(test, 1, -Inf, na.rm = TRUE), maxC(test, 1, -Inf, na.rm = TRUE))
# expect_equal(max(test, 1, -Inf), maxC(test, 1, -Inf, FALSE))
```