# Rcpp syntactic sugar

**Dirk Eddelbuettel[a] and Romain François[b]**

[a]http://dirk.eddelbuettel.com; [b]https://romain.rbind.io/

This version was compiled on November 8, 2019

**This note describes *Rcpp sugar* which has been introduced in version 0.8.3 of Rcpp (Eddelbuettel *et al.*, 2019a; Eddelbuettel and François, 2011). *Rcpp sugar* brings a higher-level of abstraction to C++ code written using the Rcpp API. *Rcpp sugar* is based on expression templates (Abrahams and Gurtovoy, 2004; Vandevoorde and Josuttis, 2003) and provides some 'syntactic sugar' facilities directly in Rcpp. This is similar to how RcppArmadillo (Eddelbuettel *et al.*, 2019b) offers linear algebra C++ classes based on Armadillo (Sanderson, 2010).**

Rcpp | sugar | R | C++

## 1. Motivation

**Rcpp** facilitates development of internal compiled code in an R package by abstracting low-level details of the R API (R Core Team, 2018) into a consistent set of C++ classes.

Code written using **Rcpp** classes is easier to read, write and maintain, without loosing performance. Consider the following code example which provides a function `foo` as a C++ extension to R by using the **Rcpp** API:

```
RcppExport SEXP foo(SEXP x, SEXP y) {
    Rcpp::NumericVector xx(x), yy(y);
    int n = xx.size();
    Rcpp::NumericVector res(n);
    double x_ = 0.0, y_ = 0.0;
    for (int i=0; i<n; i++) {
        x_ = xx[i];
        y_ = yy[i];
        if (x_ < y_) {
            res[i] = x_ * x_;
        } else {
            res[i] = -(y_ * y_);
        }
    }
    return res;
}
```

The goal of the function `foo` code is simple. Given two `numeric` vectors, we create a third one. This is typical low-level C++ code that that could be written much more consicely in R thanks to vectorisation as shown in the next example.

```
foo <- function(x, y) {
    ifelse(x < y, x * x, -(y * y))
}
```

Put succinctly, the motivation of *Rcpp sugar* is to bring a subset of the high-level R syntax in C++. Hence, with *Rcpp sugar* , the C++ version of `foo` now becomes:

```
Rcpp::NumericVector foo(Rcpp::NumericVector x,
                        Rcpp::NumericVector y) {
    return ifelse(x < y, x * x, -(y * y));
}
```

Apart from being strongly-typed and the need for explicit `return` statement, the code is now identical between highly-vectorised R and C++.

*Rcpp sugar* is written using expression templates and lazy evaluation techniques (Abrahams and Gurtovoy, 2004; Vandevoorde and Josuttis, 2003). This not only allows a much nicer high-level syntax, but also makes it rather efficient (as we detail in section 4 below).

## 2. Operators

*Rcpp sugar* takes advantage of C++ operator overloading. The next few sections discuss several examples.

**2.1. Binary arithmetic operators.** *Rcpp sugar* defines the usual binary arithmetic operators : `+`, `-`, `*`, `/`.

```
// two numeric vectors of the same size
NumericVector x;
NumericVector y;

// expressions involving two vectors
NumericVector res = x + y;
NumericVector res = x - y;
NumericVector res = x * y;
NumericVector res = x / y;

// one vector, one single value
NumericVector res = x + 2.0;
NumericVector res = 2.0 - x;
NumericVector res = y * 2.0;
NumericVector res = 2.0 / y;

// two expressions
NumericVector res = x * y + y / 2.0;
NumericVector res = x * (y - 2.0);
NumericVector res = x / (y * y);
```

The left hand side (lhs) and the right hand side (rhs) of each binary arithmetic expression must be of the same type (for example they should be both `numeric` expressions).

The lhs and the rhs can either have the same size or one of them could be a primitive value of the appropriate type, for example adding a `NumericVector` and a `double`.

**2.2. Binary logical operators.** Binary logical operators create a `logical` sugar expression from either two sugar expressions of the same type or one sugar expression and a primitive value of the associated type.

```
// two integer vectors of the same size
NumericVector x;
NumericVector y;

// expressions involving two vectors
```

```
LogicalVector res = x < y;
LogicalVector res = x > y;
LogicalVector res = x <= y;
LogicalVector res = x >= y;
LogicalVector res = x == y;
LogicalVector res = x != y;
```