

Module 7 (Rcpp part I)

Contents

1	Rcpp: Introduction	2
1.1	Introduction	2
1.2	The C++ programming language	2
1.3	Rcpp: Motivation	2
1.4	Rcpp: Usage	3
1.5	Summary	5
1.6	Bibliography	5
2	Rcpp: C++ primer	5
2.1	Introduction	5
2.2	C++ scalar types	5
2.3	Arithmetic operations	6
2.4	Defining functions	7
2.5	Conditional statements	7
2.6	Creating and assigning variables	9
2.7	Loops	10
2.8	Summary	11
2.9	Bibliography	12
3	R basic atomic types in Rcpp	12
3.1	Introduction	12
3.2	Accessing R vectors in Rcpp	12
3.3	Creating vectors	13
3.4	Copying/cloning vectors	14
3.5	NA handling	15
3.6	NULL	17
3.7	Summary	17
3.8	Bibliography	17
4	Rcpp: Functions from the R/C API	17
4.1	Introduction	17
4.2	cmath	17
4.3	cstring	17
4.4	R/C API	17
4.4.1	Numerical and mathematical functions	17
4.4.2	Utility functions	18
4.4.3	Distribution functions	19
4.4.4	Integration	21
4.5	Summary	22
4.6	Bibliography	22

1 Rcpp: Introduction

1.1 Introduction

Rcpp may be used to remove performance bottlenecks in R applications and/or functions.

To use it, we will need the following tools:

- Windows users: Rtools
- OS X users: Xcode
- Linux users: `yum install gcc gcc-c++ git` etc.
- Oracle VirtualBox VM: gagolewski.rexamine.com/resources/vms/

Moreover, please install the following R packages: Rcpp, inline, roxygen2, devtools.

1.2 The C++ programming language

C++ is a **compiled** (not: interpreted) all-purpose programming language. It is portable, object-oriented, generic, and provides low-level memory manipulation facilities. It is developed by Bjarne Stroustrup since 1979 as an extension of the C programming language. It was initially standardized in 1998 as ISO/IEC 14882:1998. C++ characterizes with high performance, efficiency, and flexibility.

Table 1 depicts an exemplary comparison of the code speed. It is quite safe to assume that C/C++ generates the fastest machine code, of course as far as some advanced machine-level tweaks, parallelism, etc. are not concerned. Note that different compiler-specific optimizations may affect code's performance.

Table 1: Exemplary benchmarks – implementations of different algorithms in some popular programming languages

	Fortran	Julia	Python	R	Matlab	Go
fib	0.26	0.91	30.37	411.36	1992.00	1.03
parse_int	5.03	1.60	13.95	59.40	1463.16	4.79
quicksort	1.11	1.14	31.98	524.29	101.84	1.25
mandel	0.86	0.85	14.19	106.97	64.58	2.36
pi_sum	0.80	1.00	16.33	15.42	1.29	1.41
rand_mat_stat	0.64	1.66	13.52	10.84	6.61	8.12

Benchmark times relative to C; see <http://julialang.org/>

1.3 Rcpp: Motivation

R is implemented in C (and R, and Fortran). In result, everything we can do in R may be implemented in C/C++. For example, the following functions call some compiled code directly:

```
sum
## function (... , na.rm = FALSE) .Primitive("sum")
c
## function (... , recursive = FALSE) .Primitive("c")
```

See also e.g. a call to `.External2` in `uniroot()`.

The **R/C API** provides the fastest way (in terms of performance) to communicate with R from the compiled code. However, it is definitely not the most convenient one. All the R objects are handled with the type `SEXP`, which is a pointer to the `SEXPREC` structure.

SEXPTYPE	R equivalent
REALSXP	numeric with storage mode double
INTSXP	integer
LGLSXP	logical
STRSXP	character
VECSXP	list (generic vector)
NILSXP	NULL
SYMSXP	name/symbol
CLOSXP	function or function closure
ENVSXP	environment

For example, this is a C/C++ code equivalent to an R call to `c(123.45, 67.89)`:

```
SEXP createVectorOfLength2() { // R/C API
    SEXP result;
    result = PROTECT(allocVector(REALSXP, 2));
    REAL(result)[0] = 123.45;
    REAL(result)[1] = 67.89;
    UNPROTECT(1);
    return result;
}
```

The R/C API may be difficult to learn and use for many R users. Thus, in this module, we will discuss the Rcpp package. It simplifies writing compiled code. Just compare the above to:

```
NumericVector createVectorOfLength2() { // Rcpp
    return NumericVector::create(123.45, 67.89);
}
```


Rcpp is a set of convenient C++ wrappers for the whole R/C API. We may use it to remove performance bottlenecks in our R code, implement code that is difficult to vectorize, or whenever we need some advanced algorithms, recursion, or data structures.

1.4 Rcpp: Usage

Let us compute the n -th Fibonacci number.

$$\begin{aligned}
 F(0) &= 1 \\
 F(1) &= 1 \\
 F(n) &= F(n-1) + F(n-2) \quad n \geq 2
 \end{aligned}$$

Output sequence: (1, 1, 2, 3, 5, 8, 13, 21, ...)

 **Exercise:** Write an R function to calculate $F(n)$.

Here is an R solution:

```
fib1 <- function(n) {
  if (n <= 1) return(1)
  last12 <- c(1, 1)
  for (i in 2:n)
    last12 <- c(last12[1]+last12[2], last12[1])
  last12[1]
}

sapply(0:7, fib1)
## [1] 1 1 2 3 5 8 13 21
```

And this is how we may implement it in Rcpp:

```
Rcpp::cppFunction("
  int fib2(int n) {
    if (n <= 1) return 1;
    int last1 = 1;
    int last2 = 1;
    for (int i=2; i<=n; ++i) {
      int last3 = last2;
      last2 = last1;
      last1 = last2+last3;
    }
    return last1;
  }
")
```

Note that `Rcpp::cppFunction` compiles, links, and loads a dynamic library.

```
print(fib2) # a library has been built
## function (n)
## .Primitive(".Call")(<pointer: 0x7fcbeb76ae60>, n)
sapply(0:7, fib2)
## [1] 1 1 2 3 5 8 13 21
```

What is important, Rcpp automatically takes care of checking the types of functions' arguments:

```
fib2(1L)
## [1] 1
fib2(1.5)
## [1] 1
fib2("1")
## Error: not compatible with requested type
fib2(c(1, 2, 3))
## Error: expecting a single value
```

Here are some benchmarks:

```
microbenchmark::microbenchmark(fib1(25), fib2(25))
## Unit: microseconds
##      expr      min       lq   median       uq      max neval
## fib1(25) 28.605 32.5665 34.001 38.8295 49.560   100
## fib2(25)  1.993  2.2805  2.631  3.1515 10.987   100
```

This looks very encouraging. Typically, we may get a speed gain of ca. 2–50x+. However, the code writing time increases significantly. Thus, Rcpp makes sense if **writing time** << **execution time** or in code re-used by others, e.g. whenever we would like to guarantee the best possible performance.

Apart from an inline usage or Rcpp, we may put our C++ code in separate source files. Here are the contents of the `test.cpp` file:

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

// [[Rcpp::export]]
int fib2(int n) {
  if (n <= 1) return 1;
  int last1 = 1;
  int last2 = 1;
  for (int i=2; i<=n; ++i) {
    int last3 = last2;
    last2 = last1;
    last1 = last2+last3;
  }
  return last1;
}
```

To compile the source file, we call:

```
Rcpp::sourceCpp("test.cpp")
```

or press (CTRL+SHIFT+S) in RStudio. The `fib2()` function may now be called in R.

By the way, we may change the name of an exported function:

```
// [[Rcpp::export(".test")]]  
int test(int n) {  
    return n;  
}
```

In this case the exported R function will be hidden (since the specified name is prefaced by a “.” character).

Moreover, some special C++ comments may be included in C++ source files:

```
/** R  
# R code here  
*/
```

The R code will be evaluated when the source file is compiled.

1.5 Summary

Rcpp may be used when:

- You care for speed.
- You need to implement code that is difficult to vectorize.
- You need advanced algorithms and data structures.

1.6 Bibliography

- Eddelbuettel D., *Seamless R and C++ Integration with Rcpp*, Springer, 2013
- en.cppreference.com – C++ reference
- Stroustrup B., *A Tour of C++*, Addison-Wesley, 2013
- Stroustrup B., *The C++ Programming Language*, Addison-Wesley, 2000
- Josuttis N.M., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 2012
- rcpp.org – Rcpp homepage
- Rcpp API documentation (Doxygen)
- Using Rcpp with RStudio
- High performance functions with Rcpp by H. Wickham

2 Rcpp: C++ primer

2.1 Introduction

R has a **C-like syntax**. Thus, we should find C++ basics easy to grasp. However, there are some notable differences:

- each operation is scalar
- variables must be declared before use; also, their type must be specified
- statements end up with a semicolon
- the `return` statement must be executed explicitly in C++ funs
- ...

2.2 C++ scalar types

Here are the most important types of atomic vectors in R:

<code>typeof()</code>	<code>mode()</code>	Element
<code>logical</code>	<code>logical</code>	Boolean (TRUE, FALSE)
<code>integer</code>	<code>numeric</code>	Integer (e.g. 1L, 2:2)
<code>double</code>	<code>numeric</code>	Floating-point number (e.g. 1, 3.14)
<code>character</code>	<code>character</code>	Character string (e.g. "aaa", '32a')

Recall that there are no “scalar” types in R. For example, a single number is represented by a numeric vector of length 1.

C++, on the other hand, has scalar types:

Type	Meaning
<code>bool</code>	Boolean (true, false)
<code>int</code>	Integer (e.g. 1)
<code>double</code>	Floating-point number (e.g. 1.0)

Moreover, `Rcpp::String` represents a character string (char*-like). By the way, we will see that R’s logical vectors use the `int` type for representing their elements.

2.3 Arithmetic operations

Here are the scalar arithmetic operators in C++:

R	C/C++
<code>+</code>	<code>+</code>
<code>-</code>	<code>-</code>
<code>*</code>	<code>*</code>
<code>/</code>	<code>/</code> (double)
<code>%/%</code>	<code>/</code> (int)
<code>%%</code>	<code>%</code>
<code>^</code> and <code>**</code>	<code>pow(x,y)</code>

Operations may be grouped with parentheses (...) only.

Note that in C++, “1” denotes an integer constant. To input a floating point (double) constant, we write rather “1.0”. Note the following:

```
Rcpp::evalCpp("1/2") # integer division
## [1] 0
Rcpp::evalCpp("1.0/2.0") # real division
## [1] 0.5
```

The “(type)value” notation is used for type casting (conversion). Type *promotion* is performed automatically.

```
Rcpp::evalCpp("1/2.0") # == 1.0/2.0 (promotion)
## [1] 0.5
Rcpp::evalCpp("1.0/2") # == 1.0/2.0
## [1] 0.5
Rcpp::evalCpp("1/(double)2") # == 1/2.0 == 1.0/2.0
## [1] 0.5
Rcpp::evalCpp("(double)1/2") # == 1.0/2 == 1.0/2.0
## [1] 0.5
Rcpp::evalCpp("(double)(1/2)") # == (double)0
## [1] 0
```

2.4 Defining functions

When defining C++ functions, we must indicate the arguments' types. Also, return value type must be declared.

A function's definition syntax is as follows:

```
return_type function_name(arg_list) {
    // function body (a comment)
    /* function body
       (a comment) */
    return out_value;
}
```

where `arg_list` is either:

- empty,
- `arg_type arg_name`, or
- `arg_type1 arg_name1, ..., arg_typen arg_namen`.


Note that the `return` statement must always be included explicitly. What is more, each statement must end up with a semicolon (;).

 **Exercise:** Write an Rcpp function to calculate $ax^2 + bx + c$ given $a, b, c, x \in \mathbb{R}$.

A solution:

```
Rcpp::cppFunction("
    double polynom2(double x, double a, double b, double c) {
        return a*x*x+b*x+c;
    }
")

polynom2(2, 1, 0, 0)
## [1] 4
polynom2(2, 1, FALSE, TRUE) # coercion when possible
## [1] 5
polynom2(c(1,2), 1, 0, 0)
## Error: expecting a single value
polynom2(2, 1, 0, "0")
## Error: not compatible with requested type
```

 **Exercise:** Add default arguments: $a = 1, b = 0, c = 0$.

A solution:

```
Rcpp::cppFunction("
    double polynom2(double x, double a=1.0,
                    double b=0.0, double c=0.0) {
        return a*x*x+b*x+c;
    }
")

polynom2(2)
## [1] 4
polynom2(2, 2)
## [1] 8
```

2.5 Conditional statements

Here is the syntax of the `if` and `if...else` statements:

```
if (condition)
    statement_if_true;
```

```
if (condition)
  statement_if_true;
else
  statement_if_false;
```

We may use curly braces to group a sequence of statements:

```
if (condition) {
  // statement block
  statement1;
  statement2;
}
```


Moreover, we can write some nested if...else statements:

```
if (condition1)
  statement1;
else if (condition2)
  statement2;
// ...
else
  statement3;
```

Here are C++ scalar comparison operators:

R	C/C++
==	==
!=	!=
<	<
<=	<=
>	>
>=	>=

To create a string, we use double quotes. By the way, 'a' has a different meaning: it denotes a single character (a constant of type char).

 **Exercise:** Write an Rcpp function that maps a given $x \in \mathbb{R}$ to a string "positive", "negative", or "zero".

A solution:

```
Rcpp::cppFunction('
  String sign(double x) {
    if (x > 0) return "positive";
    else if (x < 0) return "negative";
    else return "zero";
  }
')

sapply(c(-1,0,1), sign)
## [1] "negative" "zero"      "positive"
```

If we use Rcpp inline, we should not forget about quotes' escaping.


```
Rcpp::cppFunction("
  String sign(double x) {
    if (x > 0) return \"positive\";
    else if (x < 0) return \"negative\";
    else return \"zero\";
  }
")

sapply(c(-1,0,1), sign)
## [1] "negative" "zero"      "positive"
```


Here are the C++ scalar logical operators:

R	C/C++
!	!
& (&&)	&&
()	

Note that & and | in C++ have a different meaning. These denote bitwise operations. On the other hand, && and || are *lazy*, just like in R.

 **Exercise:** Write an Rcpp function to test if a given real x is in $[0, 1]$.

A solution:

```
Rcpp::cppFunction('
  bool in01(double x) {
    if (x >= 0 && x <= 1) return true;
    else return false;
  }
')
sapply(c(-1,0,1,2), in01)
## [1] FALSE TRUE TRUE FALSE
```

Equivalently:

```
Rcpp::cppFunction('
  bool in01(double x) {
    return (x >= 0 && x <= 1);
  }
')
```

The switch statement has the following syntax:

```
switch (x) { // x – most often of integral type
  case 1:
    expression1;
    break;
  case 2:
  case 3:
    expression2;
    break;
  default:
    expressionD;
    break;
}
```

The above is equivalent to:

```
if (x == 1) expression1;
if (x == 2 || x == 3) expression2;
else expressionD;
```

2.6 Creating and assigning variables

Each variable must be **declared** prior its first use:

```
var_type var_name;
var_type var_name1, var_name2;
```

For example, “var_type” may be one of: int, double, etc. “var_name” should match the `[aA_] [aA_1]*` regular expression.

To assign a value to a variable, we use the assignment operator, “=”.

```
int i;
i = 5;
// or int i=5;
```

Do not confuse the “=” (assignment) operator with “==”, which tests for equality. The if statement accepts both.

```
Rcpp::cppFunction('
  int changeSignIfOne(int x) {
    if (x = 1) // :-(
      x = -x;
    return x;
  }
')
changeSignIfOne(1)
## [1] -1
changeSignIfOne(10)
## [1] -1
```

Note that `if (cond)` is equivalent to `if (cond != 0)`. We have `(a = b) == b` (and `a` changes its value).

C++ has also compound assignment operators. For example, `a += b` is equivalent to `a = a+b`. Among other operators of this kind we have: `+=`, `-=`, `*=`, `/=`, etc. What is more, `a++` and `++a` is equivalent to `a += 1`, i.e. `a = a+1`. On the other hand, `a--` and `--a` mean `a -= 1`.

However:

```
Rcpp::cppFunction('
  int f_post() {
    int a = 1; int b = a++; return b;
  }')

Rcpp::cppFunction('
  int f_pre() {
    int a = 1; int b = ++a; return b;
  }')

c(f_post(), f_pre())
## [1] 1 2
```

This is because `b = a++` is equivalent to:

1. `b = a;`
2. `a = a+1;`

On the other hand, `b = ++a` is equivalent to:

1. `a = a+1;`
2. `b = a;`

2.7 Loops

The while loop is just like R’s while:

```
while (cond) {
  expression;
}
```

Use `break` to break out of a innermost loop. Use `continue` to jump to next iteration.

 **Exercise:** Write a function to test if given n is prime.

Let us go on with the other loop constructs. How to assure that the expression is evaluated at least once (to “do some work until we are done”)? We may do that with the `do..while` loop:

```
do {
```

```
    expression;
} while (cond);
```


The for loop is most often used. Its syntax is as follows:

```
for (init; cond; post) {
    expression;
}
```

The above is equivalent to:

```
init;
while (cond) {
    expression;
    post;
}
```


Note that continue jumps to the post statement.

 **Exercise:** Calculate $\sum_{i=1}^n i$ (for a given n).


A solution:

```
Rcpp::cppFunction('
    int sum2n(int n) {
        int result = 0;
        for (int i=1; i<=n; i++) {
            // i from 1 to n by 1
            // i = 1,2,...,n
            result += i;
        }
        return result;
    }
')
```

```
sum2n(3)
## [1] 6
```

 **Exercise:** Estimate the value of π by using n MC experiments.

Hint: Generate $(U, V) \sim U([-1, 1] \times [-1, 1])$ n times, calculate the proportion of points that appear in the unit circle.

 **Exercise:** And now calculate $\sum_{i=1}^n i$ using **recursion**.

A solution:

```
Rcpp::cppFunction('
    int sum2n(int n) {
        if (n <= 0) return 0;
        else return sum2n(n-1)+n;
    }
')
```

```
sum2n(3)
## [1] 6
```

2.8 Summary

C++ has scalar types. These include e.g. bool, int, double. Beware of integer division: $1/2 == 0$, because the constants “1” and “2” denote integers. Also, keep in mind that each statement ends up with a semicolon.

2.9 Bibliography

- Eddelbuettel D., *Seamless R and C++ Integration with Rcpp*, Springer, 2013
- en.cppreference.com – C++ reference
- Stroustrup B., *A Tour of C++*, Addison-Wesley, 2013
- Stroustrup B., *The C++ Programming Language*, Addison-Wesley, 2000
- Josuttis N.M., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 2012

3 R basic atomic types in Rcpp

3.1 Introduction

In R, *vectorcentricism* is obvious. For example, matrices, time series, factors, data frames base on vectors. In this section we will implement Rcpp functions that deal with R vectors.

3.2 Accessing R vectors in Rcpp

Below we list types of R atomic vectors and compare them with their Rcpp counterparts.

<code>typeof()</code>	<code>mode()</code>	Rcpp class	elem type
logical	logical	LogicalVector	int
integer	numeric	IntegerVector	int
double	numeric	NumericVector	double
character	character	CharacterVector	Rcpp::String

We should keep in mind that in C/C++, the first element of a vector is at index 0. By the way, Rcpp::Vector's interface is roughly compatible with STL's `std::vector`, see module 8.

Let us write a function to calculate the sum of elements in a NumericVector.

```
Rcpp::cppFunction('
double sum2(NumericVector x) {
    int n = x.size(); // a method
    double result = 0.0;
    for (int i=0; i<n; ++i)
        result += x[i]; // or x(i)
    return result;
}
')
```

```
sum2(1:5) # int -> double
## [1] 15
```

What about its performance?

```
x <- runif(1);      microbenchmark(sum(x), sum2(x), unit="us")
## Unit: microseconds
##      expr   min      lq median      uq     max neval
##  sum(x) 0.306 0.3285 0.4710 0.5755 19.078   100
##  sum2(x) 2.089 2.2155 2.4485 3.0550 40.858   100

x <- runif(1000);   microbenchmark(sum(x), sum2(x), unit="us")
## Unit: microseconds
##      expr   min      lq median      uq     max neval
##  sum(x) 1.500 1.5485 1.6705 1.7705 18.153   100
##  sum2(x) 3.253 3.4255 3.5875 3.7850 36.314   100
```


```
x <- runif(1000000); microbenchmark(sum(x), sum2(x), unit="us")
## Unit: microseconds
##      expr      min       lq     median       uq      max neval
##   sum(x) 1604.231 1625.215 1634.063 1659.858 2002.161   100
##  sum2(x) 1492.512 1500.061 1518.290 1537.147 1892.642   100
```


Generally, our approach is slower than the built-in R/C API based `sum()` function (note that we do not handle missing values at all above). We should not reinvent the wheel “trivial” base R functions are likely to be faster. As a rule of thumb, `Rcpp_time == base_R_time + Const_time`. For example, in the above example we have `Const_time == 2μs`.


Rcpp is a *wrapper* to the R/C API, which means that it has to:

- perform some additional operations on the input and output objects,
- agree R and C++ exception handling,
- protect object from garbage collection,
- assure that it is safe to use the random number generator,
- etc.

But we can beat R in speed if we will use a different **algorithm**. For example, R’s `sort()` is quite easy to “defeat”.

 **Exercise:** Implement an Rcpp function equivalent to `min(which(x>0))`, where `x` is a numeric vector, i.e. find the index of first positive element. When (and why) your code will be faster?


 **Exercise:** Let `x` – integer vector with elements $\in \{1, 2, 3, 4\}$. Write a function that gives the position of the first occurrence of `(1, 1, 2, 3, 1, 2, 4)` in a given integer vector or 0 iff not found. Hint: Develop a finite state machine.

 **Exercise:** Let `x`, `y` – integer vectors. Locate the first occurrence of the `y` pattern in `x`. Hint: Start with a naive $O(nm)$ algorithm, then try to come up with some more efficient.

3.3 Creating vectors

Here are few ways to create a numeric vector in C++ code:

```
NumericVector x1(10); // 10 x 0.0
NumericVector x2(10, 1.0); // 10 x 1.0
NumericVector x3 =
    NumericVector::create(1.0, 2.0, 3.0);
```


 **Exercise:** Write a function returning `(start, start+1, ..., end)` for given `start`, `end`.


A solution:


```
Rcpp::cppFunction('
    IntegerVector seq2(int start, int end) {
        int n = end-start+1;
        IntegerVector out(n);
        for (int i=0; i<n; ++i)
            out[i] = (double)(start+i);
        return out;
    }
')
```


```
seq2(1, 5)
## [1] 1 2 3 4 5
```

By the way, the above code contains a bug. Can you find it?

 **Exercise:** Given n , return a vector of primes $\leq n$. Hint: Implement the Eratosthenes sieve.

 **Exercise:** Given an integer vector, “trim” 0s from the front and from the end, e.g. $(0, 0, 0, 1, 2, 3, 0, 1, 2, 0, 0, 0) \rightarrow (1, 2, 3, 0, 1, 2)$. Return a new vector, do not modify the given one.

 **Exercise:** Given an integer vector, substitute all sequences of 0s with single 0s, e.g. $(0, 0, 1, 2, 0, 3, 4, 0, 0, 0, 5, 0, 0, 0) \rightarrow (0, 1, 2, 0, 3, 4, 0, 5, 0)$. Return a new vector, do not modify the given one.

 **Exercise:** Implement a C++ function to calculate $x+y$, where x, y – numeric vectors. Mimic the following behavior:

```
c(1, 2, 3) + c(-1, -2, -3)
## [1] 0 0 0
c(1, 2, 3) + 1
## [1] 2 3 4

c(-1, -2) + c(1, 2, 3) # Rf_warning("warning message");
## Warning: longer object length is not a multiple of shorter object length
## [1] 0 0 2
c() + c(1, 2, 3)
## numeric(0)
```

3.4 Copying/cloning vectors

Are R vectors passed by value to Rcpp functions?

```
Rcpp::cppFunction('
NumericVector chsgn(NumericVector x) {
  for (int i=0; i<x.size(); ++i)
    x[i] = -x[i];
  return x;
}
')
```

```
chsgn(c(-2, -1, 0, 1, 2))
## [1] 2 1 0 -1 -2
```

However...

```
x <- c(-2, -1, 0, 1, 2)
y <- x
chsgn(x)
## [1] 2 1 0 -1 -2
print(x)
## [1] 2 1 0 -1 -2
print(y)
## [1] 2 1 0 -1 -2
```

For performance reasons, only a **shallow copy** of the underlying R object is made. A `NumericVector` is a very thin wrapper around an R object: In fact, SEXP is just a pointer, so the underlying data are not copied. For your own safety, always add a **const** keyword before each argument specifier:

```
Rcpp::cppFunction('
NumericVector chsgn(const NumericVector x) {
  for (int i=0; i<x.size(); ++i)
    x[i] = -x[i];
  return x;
}
')
## Error: Error 1 occurred building shared library.
```

```
## Error: assignment of read-only location: x[i] = -x[i];
```

To make a **deep copy**, call:

```
NumericVector y = Rcpp::clone(x);
// or x = Rcpp::clone(x);
```

or:

```
NumericVector y(x.begin(), x.end());
// or x = NumericVector(x.begin(), x.end());
```

By the way, `Rcpp::Vector`'s copy constructor makes a shallow copy of an R object. In other words,

```
NumericVector y(x); // shallow copy of x
```

is **not** enough.

Theoretically, we may check if `NAMED((SEXP)x) == 0` and then it is safe to modify `x`, but this is extremely rarely used.

```
Rcpp::cppFunction('
  IntegerVector test(IntegerVector x) {
    if (NAMED((SEXP)x) > 0) {
      Rcout << "copy!" << std::endl;
      x = Rcpp::clone(x);
    }
    for (int i=0; i<x.size(); ++i) x[i] = -x[i];
    return x;
  }
')
test(c(1,2,3,7)) # conversion double->integer
## [1] -1 -2 -3 -7
test(c(1L,2L,3L,4L))
## copy!
## [1] -1 -2 -3 -4
```

3.5 NA handling

R is a language suitable for statistical computing and data analysis. Missing data (NAs) can also occur in atomic vectors. They must “fit” into existing C/C++ scalar types somehow.

```
int na_logical = NA_LOGICAL; // Note: not a bool
int na_integer = NA_INTEGER;
double na_double = NA_REAL;
String na_string = NA_STRING;
```

Note that how they are represented:

```
Rcpp::cppFunction('
void NA_info() {
  Rprintf("NA_LOGICAL %s\\t%d\\n",
    typeid(NA_LOGICAL).name(), NA_LOGICAL);
  Rprintf("NA_INTEGER %s\\t%d\\n",
    typeid(NA_INTEGER).name(), NA_INTEGER);
  Rprintf("NA_REAL %s\\t%f\\n",
    typeid(NA_REAL).name(), NA_REAL);
}
', includes="#include <typeinfo>")

NA_info()
## NA_LOGICAL i -2147483648
## NA_INTEGER i -2147483648
## NA_REAL d nan
```

Here are some ways to test for NAs:

```
LogicalVector::is_na(...)
IntegerVector::is_na(...)
NumericVector::is_na(...)
CharacterVector::is_na(...)
```

Note that in C/C++:

- a value == 0 denotes false (the FALSE constant in Rcpp)
- a value != 0 denotes true (TRUE).

We should pay special attention to NA handling. Consider the following implementation of !x.

```
Rcpp::cppFunction('
  LogicalVector negate(const LogicalVector x) {
    int n = x.size();
    LogicalVector y(n);
    for (int i=0; i<n; ++i)
      y[i] = !x[i];
    return y;
  }
')
```

```
negate(c(TRUE, FALSE, NA))
## [1] FALSE TRUE FALSE
```


The function's behavior may be explained by taking into account the fact that `NA == -2147483648 != 0 == true`.

Here is a proper solution:

```
Rcpp::cppFunction('
  LogicalVector negate2(LogicalVector x) {
    int n = x.size();
    LogicalVector y(n);
    for (int i=0; i<n; ++i) {
      if (LogicalVector::is_na(x[i]))
        y[i] = NA_LOGICAL;
      else
        y[i] = !x[i];
    }
    return y;
  }
')
```

```
negate2(c(TRUE, FALSE, NA))
## [1] FALSE TRUE NA
```

As a rule of thumb, we should test for NAs first. By the way, if there were no NAs at all, R would be more efficient. Generally, we may wish to guarantee that `op(NA) == NA`. This gives a typical R-like look-and-feel. Sometimes, however, we can just mimic the behavior of `stopifnot(!is.na(x))`.

 **Exercise:** Upgrade your vectorized + operator implementation to handle missing values correctly.

Recall that a value of type `int` represents integers from `-2147483648` to `2147483647`. However, `-2147483648 == NA_INTEGER`. Thus,

```
Rcpp::cppFunction('
  IntegerVector overflow_test() {
    int x = 2147483647;
    return IntegerVector::create(x, x+1, x+2);
  }
')
```



```
overflow_test()  
## [1] 2147483647 NA -2147483647
```

3.6 NULL

Atomic vectors and the NULL type are all basic atomic types. In Rcpp, the NULL object is referred to as `R_NilValue`. We may use `Rf_isNull()` to check if a given RObject is NULL.

3.7 Summary

We should pay special attention to missing values handling. Moreover, we should never modify vectors passed to Rcpp functions as arguments.

3.8 Bibliography

- Eddelbuettel D., *Seamless R and C++ Integration with Rcpp*, Springer, 2013
- en.cppreference.com – C++ reference
- Stroustrup B., *A Tour of C++*, Addison-Wesley, 2013
- Stroustrup B., *The C++ Programming Language*, Addison-Wesley, 2000
- Josuttis N.M., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 2012

4 Rcpp: Functions from the R/C API

4.1 Introduction

Let us make a brief review of the R/C API (see Writing R Extensions, Sec. 6). The C++ library (a.k.a. STL) – a comprehensive source of interesting algorithms and data structures – will be discussed later on.

4.2 cmath

C++ inherited a very handy C library of widely-used functions. For example, `<cmath>`, see en.cppreference.com/w/c/numeric, includes a wide range of common mathematical functions. However, most of them are available also via R/C API – its usage is recommended.

For example, The R manual claims that `<cmath>`'s `pow()` might not make R-specific NA/NaN/Inf checks. Such are implemented in R/C API's `R_pow()`.

4.3 cstring

Moreover, `<cstring>` consists of some “popular” string functions. Recall that a string is a 0-terminated byte array. A character vector is a vector of byte vectors. Unless you are a very advanced programmer, do not rely on them. For example, some R strings may use an 8-bit encoding, and the other ones may be in UTF-8.

4.4 R/C API

4.4.1 Numerical and mathematical functions

Here are some of the R/C API's numerical utilities:

```
double R_pow (double x, double y); //  $x^y$   
double log1p (double x); //  $\log(1+x)$  [basicstyle=tt]  
double log1pexp (double x); //  $\log(1 + \exp(x))$   
double lgamma1p (double x); //  $\log(\text{gamma}(x + 1))$ 
```

```
double expm1 (double x); //  $\exp(x) - 1$ 
double sinpi (double x); //  $\sin(\pi * x)$ 
double fmax2 (double x, double y); //  $\max(x, y)$ 
double fround (double x, double digits);
```

Functions like `log1p()` are provided for better numeric accuracy. Moreover:

```
double Rf_gammafn (double x);
double Rf_lgammafn (double x);
double Rf_digamma (double x);
double Rf_trigamma (double x);
double Rf_tetragamma (double x);
double Rf_pentagamma (double x);
double Rf_psigamma (double x, double deriv);
double Rf_beta (double a, double b);
double Rf_lbeta (double a, double b);
double Rf_choose (double n, double k);
double Rf_lchoose (double n, double k);
double Rf_bessel_i (double x, double nu, double expo);
double Rf_bessel_j (double x, double nu);
double Rf_bessel_k (double x, double nu, double expo);
double Rf_bessel_y (double x, double nu);
```

Mathematical constants:

M_E	e	2.7182818
M_LOG2E	$\log_2(e)$	1.4426950
M_LOG10E	$\log_{10}(e)$	0.4342945
M_LN2	$\ln(2)$	0.6931472
M_LN10	$\ln(10)$	2.3025851
M_PI	π	3.1415927
M_PI_2	$\pi/2$	1.5707963
M_PI_4	$\pi/4$	0.7853982
M_1_PI	$1/\pi$	0.3183099
M_2_PI	$2/\pi$	0.6366198
M_2_SQRTPI	$2/\sqrt{\pi}$	1.1283792
M_SQRT2	$\sqrt{2}$	1.4142136
M_SQRT1_2	$1/\sqrt{2}$	0.7071068
M_SQRT_3	$\sqrt{3}$	1.7320508
M_SQRT_32	$\sqrt{32}$	5.6568542
M_LOG10_2	$\log_{10}(2)$	0.3010300
M_2PI	2π	6.2831853
M_SQRT_PI	$\sqrt{\pi}$	1.7724539
M_1_SQRT_2PI	$1/\sqrt{2\pi}$	0.3989423
M_SQRT_2dPI	$\sqrt{2/\pi}$	0.7978846
M_LN_SQRT_PI	$\ln(\sqrt{\pi})$	0.5723649
M_LN_SQRT_2PI	$\ln(\sqrt{2\pi})$	0.9189385
M_LN_SQRT_PId2	$\ln(\sqrt{\pi/2})$	0.2257914

4.4.2 Utility functions

The following utility functions were defined:

```
void R_isort (int* x, int n);
void R_rsort (double* x, int n);
int findInterval (double* xt, int n, double x,
                 Rboolean rightmost_closed, Rboolean all_inside,
                 int ilo, int* mflag);
```

An example:

```
Rcpp::cppFunction('
  NumericVector sort2(NumericVector x) {
    NumericVector ret = Rcpp::clone(x);
    R_rsort(REAL((SEXP)ret), ret.size());
    return ret;
  }
')

sort2(c(5, 2, 3, 1, 4))
## [1] 1 2 3 4 5
```

4.4.3 Distribution functions

All the d/p/q/r functions are available via the R/C API. They are **not vectorized**. For example:

```
double Rf_dnorm(double x, double mu, double sigma,
  int give_log);
double Rf_pnorm(double x, double mu, double sigma,
  int lower_tail, int give_log);
double Rf_qnorm(double p, double mu, double sigma,
  int lower_tail, int log_p);
double Rf_rnorm(double mu, double sigma);
```


Below we list available probability distributions and their parameters:

beta	beta	a, b
non-central beta	nbeta	a, b, ncp
binomial	binom	n, p
Cauchy	cauchy	location, scale
chi-squared	chisq	df
non-central chi-squared	nchisq	df, ncp
exponential	exp	scale
F	f	n1, n2
non-central F	nf	n1, n2, ncp
gamma	gamma	shape, scale
geometric	geom	p
hypergeometric	hyper	NR, NB, n
logistic	logis	location, scale
lognormal	lnorm	logmean, logsd
negative binomial	nbinom	size, prob
normal	norm	mu, sigma
Poisson	pois	lambda
Student's t	t	n
non-central t	nt	df, delta
uniform	unif	a, b
Weibull	weibull	shape, scale

What is most important, exported Rcpp functions take care of the random number generator's seed automatically (that is not obvious). If you use randomness in non-exported functions, add the following declaration at the beginning of your routine:

```
RNGScope scope;
```

This calls R's `GetRNGstate()` and `PutRNGstate()`.

 **Exercise:** Assume that (for some reasons) we would like to estimate the standard deviation of means of (X_1, \dots, X_n) i.i.d. $U[0, 1]$.

In an exemplary solution, we will use $n = 100$ and $m = 1000$ MC iterations.

```
n <- 100
m <- 1000

experiment1 <- compiler::cmpfun(function(n, m) {
  sd(replicate(m, {
    x <- runif(n)
    mean(x)
  })))
})

set.seed(123)
experiment1(n, m)
## [1] 0.0285281
```

Another solution:

```
Rcpp::cppFunction('
void regen_runif(NumericVector x) {
  int n = x.size();
  for (int i=0; i<n; ++i) {
    x[i] = Rf_runif(0.0, 1.0);
  }
}' )
experiment2 <- compiler::cmpfun(function(n, m) {
  x <- numeric(n)
  sd(replicate(m, {
    regen_runif(x)
    mean(x)
  })))
})

set.seed(123)
experiment2(n, m)
## [1] 0.0285281
```

A third solution:

```
Rcpp::cppFunction('
double experiment_iter(int n) {
  double sum = 0.0;
  for (int i=0; i<n; ++i) {
    sum += Rf_runif(0.0, 1.0);
  }
  return sum / (double)n;
}' )

set.seed(123)
experiment3 <- function(n, m) {
  sd(replicate(m, experiment_iter(n)))
}
experiment3(n, m)
## [1] 0.0285281
```

Yet another solution:

```
Rcpp::cppFunction('
double experiment4(int n, int m) {
  double mean_all = 0.0, var_all = 0.0;
  for (int j=0; j<m; ++j) {
    double sum = 0.0;
    for (int i=0; i<n; ++i)
      sum += Rf_runif(0.0, 1.0);
  }
}
```

```

    double mean = sum / (double)n;
    double mean_old = mean_all, var_old = var_all;
    mean_all = mean_old + (mean - mean_old)/(j+1);
    var_all = var_old + mean_old*mean_old -
mean_all*mean_all + (mean*mean-var_old-mean_old*mean_old)/(j+1);
  }
  return sqrt(var_all*(double)m/(double)(m-1));
} ' )
set.seed(123)
experiment4(n, m)


```

Some benchmarks:

```

microbenchmark(
  e1=experiment1(n, m), e2=experiment2(n, m),
  e3=experiment3(n, m), e4=experiment4(n, m)
)
## Unit: milliseconds
##   expr      min       lq     median       uq      max neval
##   e1 22.121196 22.732515 24.787439 27.646985 36.782307   100
##   e2 17.882525 18.596295 19.526100 21.953807 32.409233   100
##   e3  5.622066  5.801422  5.962219  6.573058 11.082139   100
##   e4  2.245388  2.259036  2.268131  2.294507  3.473138   100

```

 **Exercise:** For given n , generate a random permutation of $\{1, \dots, n\}$.

4.4.4 Integration

Let us approximate π by calculating:

```

integrate(function(x) sqrt(1 - x^2), 0, 1)$val * 4
## [1] 3.141593

```

Our code will be ca. 7x faster.

```

#include <Rcpp.h>
#include <R_ext/Applic.h>

void quartcirc(double *x, int n, void*/*not used*/) {
  // x - input & output
  // n points at a time
  for (int i=0; i<n; ++i) {
    x[i] = sqrt(1-x[i]*x[i]);
  }
}

// [[Rcpp::export]]
double pi() {
  double a = 0.0, b = 1.0;
  double epsabs = 1e-9, epsrel = 1e-9;
  int limit = 100;
  double result, abserr; // out
  int neval, ier, last; // out
  int lenw = 4*limit;
  int* iwork = (int*)R_alloc(limit, sizeof(int));
  double* work = (double*)R_alloc(lenw, sizeof(double));
  Rdqags(quartcirc, NULL,
    &a, &b, &epsabs, &epsrel, &result, &abserr,
    &neval, &ier, &limit, &lenw, &last, iwork, work);
  return result*4.0;
}

```

By the way, R's optimization routines are also available.

4.5 Summary

All the mathematical R functions are available via the R/C API. By using them cleverly in Rcpp you can obtain a significant speedup.

4.6 Bibliography

- R Core Team, *Writing R Extensions*, 2014, Sec. 6
- Eddelbuettel D., *Seamless R and C++ Integration with Rcpp*, Springer, 2013
- en.cppreference.com – C++ reference
- Stroustrup B., *A Tour of C++*, Addison-Wesley, 2013
- Stroustrup B., *The C++ Programming Language*, Addison-Wesley, 2000
- Josuttis N.M., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 2012