

Module 8 (Rcpp part II)

Contents

1	Rcpp: Handling non-basic types	2
1.1	Introduction	2
1.2	Lists	2
1.3	Functions	3
1.4	Attributes	4
1.5	Compound types	5
1.5.1	Factors	5
1.5.2	Matrices	6
1.5.3	Data frames	7
1.6	Summary	8
1.7	Bibliography	8
2	Rcpp: C++11	8
2.1	Introduction	8
2.2	New features in C++11	8
2.3	C++11 in Rcpp	9
2.4	Summary	10
2.5	Bibliography	10
3	Rcpp: The C++ Standard Library	10
3.1	Introduction	10
3.2	Basic classes	10
3.3	Containers	13
3.3.1	Vectors	13
3.3.2	Double-ended queues	15
3.3.3	Lists	15
3.3.4	Priority queues	16
3.3.5	Sets	16
3.3.6	Maps	17
3.4	Algorithms	17
3.5	External pointers	19
3.6	Summary	20
3.7	Bibliography	20
4	Rcpp in R packages	21
4.1	Introduction	21
4.2	Rcpp in R packages	21
4.2.1	A tutorial	21
4.3	C++11 in R packages	22
4.4	Summary	22
4.5	Bibliography	22

1 Rcpp: Handling non-basic types

1.1 Introduction

In this section, we will take a look at how to handle R lists and functions. Then we will discuss objects' attributes. Basing on that, we will show how to create and access matrices, data frames, factors, and any other S3 objects.

1.2 Lists

An R list is a sequence (vector) consisting of any kind of R objects (any SEXP in R/C API, any RObject in Rcpp).

```
Rcpp::cppFunction('
  List list_test() {
    List out(2);
    out[0] = CharacterVector::create("one", "two");
    out[1] = List::create(1, 2.0);
    return out;
  }
');


str(list_test())
## List of 2
## $ : chr [1:2] "one" "two"
## $ :List of 2
## ..$ : int 1
## ..$ : num 2
```

If a list is passed as argument, we must watch out for its elements' types.

```
Rcpp::cppFunction('
  CharacterVector listElemTypes(List x) {
    CharacterVector out(x.size());
    for (int i=0; i<x.size(); ++i) {
      if (Rf_isLogical(x[i])) out[i] = "logical";
      else if (Rf_isInteger(x[i])) out[i] = "integer";
      else if (Rf_isReal(x[i])) out[i] = "numeric";
      else if (Rf_isString(x[i])) out[i] = "character";
      else out[i] = NA_STRING;
    }
    return out;
  }
');

listElemTypes(list(TRUE, 1L, 1.5, "aaa", mean))
## [1] "logical" "integer" "numeric" "character" NA
```

Here are some functions that do SEXPTYPE checks: `Rf_isLogical()`, `Rf_isInteger()`, `Rf_isReal()`, `(Rf_isNumeric())`, `Rf_isString()`, `Rf_isNull()`, `Rf_isList()` (a list or NULL), `Rf_isVectorAtomic()`, `Rf_isFunction()`, `Rf_isLanguage()`, `Rf_isFactor()` (check before `Rf_isInteger()`, see below), `Rf_isMatrix()`, `Rf_isArray()`, `Rf_isTs()`. Note that omitting the “Rf_” prefix may sometimes be necessary.

 **Exercise:** Calculate the length of each numeric vector in a given list `x`. Return an integer vector of the same length as `x`.


Here is a solution:

```
Rcpp::cppFunction('
  IntegerVector lengthsNum(List x) {
    IntegerVector out(x.size());
```

```

    for (int i=0; i<x.size(); ++i) {
        if (Rf_isNumeric(x[i])) {
            NumericVector elem(x[i]);
            out[i] = elem.size();
        }
        else
            out[i] = NA_INTEGER;
    }
    return out;
}
')
lengthsNum(list(1:3, 1.5, mean, "mean", c(TRUE, FALSE)))

```

 **Exercise:** Check what happens if we call:

```
NumericVector elem(x[i]);
```

with no SEXTYPE check.

```

Rcpp::cppFunction('
    NumericVector list_test(List x) {
        NumericVector out(x[0]);
        return out;
    }
')

list_test(1)
## [1] 1
list_test("one")
## Error: not compatible with requested type

```

1.3 Functions


An R function acts on some RObjects and outputs an RObject.

```

Rcpp::cppFunction('
    RObject some_call(Function f, RObject x) {
        return f(x);
    }
')

some_call(runif, 5)
## [1] 0.3652096 0.2383102 0.2815522 0.3086670 0.9334224

```

 **Exercise:** Write your own implementation of `lapply()`.

A solution:

```

List lapply2(List input, Function f) {
    int n = input.size();
    List out(n);

    for(int i = 0; i < n; i++) {
        out[i] = f(input[i]);
    }

    return out;
}

```

Note that this is 10 times slower than the original `lapply()`. Rcpp produces quite a lot of code here, the

original `lapply()` is way more clever.

Note now we may set named arguments' values in functions calls:

```
Rcpp::cppFunction('
  List runif_test(IntegerVector n) {
    List out(n.size());
    Function R_runif("runif");
    for (int i=0; i<n.size(); ++i) {
      out[i] = R_runif(n[i],
        _["min"]=-1.0,      // named attrib access v1
        Named("max", 1.0)); // named attrib access v2
    }
    return out;
  }
');
runif_test(c(1, 2))
## [[1]]
## [1] -0.7449441
##
## [[2]]
## [1] -0.9609399  0.3506407
```

Recall that each *name* is bound (stored) in some *environment*. For example, if we call `sum()` in R, we in fact refer to `sum` from the base environment.

```
Environment base("package:base");
// or: Environment base = Environment::base_env()
Function R_sum = base["sum"]; // get base::sum
```

Similarly we may refer to any object in the global environment. Among the `Environment` class methods we find e.g. `get()`, `exists()`, `remove()`, `assign()`, etc.

Sometimes calling R base functions is not necessary. We have **Rcpp sugar**, which provides us with C++ rewrites of R functions made by Rcpp authors. For example, there are the `seq_len()`, `seq_along()`, `rep()`, `rep_len()`, `rep_each()`, etc. functions.

An example:

```
Rcpp::cppFunction('
  IntegerVector sugar_test() {
    IntegerVector out = seq_len(10);
    return out;
  }
');

sugar_test()
## [1] 1 2 3 4 5 6 7 8 9 10
```

Other Rcpp sugar facilities:

- All vectorized R operators
- `exp()`, `floor()`, etc.
- `dnorm()`, `runif()`, etc.
- `ifelse()`, `lapply()`, `sapply()`, etc.

Note that in some cases the R/C API may also be used directly.

1.4 Attributes

Recall that **attributes** represent some *metadata* on R objects. Generally, the `attr()` method allows for reading or setting an object's attribute.

```
out.attr("some_attribute") = "some_value";
```

```
if (Rf_isNull(in.attr("some_attribute")))  
  ; // the attribute is not set
```

To set an object's S3 *class*, use:

```
out.attr("class") = "some_class";
```

To verify if an object inherits from a given class, use:

```
in.inherits("some_class")
```

For example, this is a way to set the `names` attribute:

```
out.names() = CharacterVector::create("a", "b", "c");
```

Accessing a named element:

```
in["elem_name"]
```

Note that on an unknown name, the “index out of bounds” error will be thrown.

1.5 Compound types

R compound types:

- factor
- matrix
- data.frame


are nothing else as basic objects equipped with some specific attributes.

1.5.1 Factors

Recall how factors are represented in R:

```
x <- factor(c("a", "b", "a", "c"))  
storage.mode(x)  
## [1] "integer"  
unclass(x)  
## [1] 1 2 1 3  
## attr(,"levels")  
## [1] "a" "b" "c"  
str(attributes(x))  
## List of 2  
## $ levels: chr [1:3] "a" "b" "c"  
## $ class : chr "factor"
```


It turns out that R factors may be input to Rcpp functions both as `CharacterVectors` and `IntegerVectors`.


 **Exercise:** Implement `table(x)`, where `x` is a factor.

A solution:

```
Rcpp::cppFunction('  
  IntegerVector table2(IntegerVector x) {  
    if (!Rf_isFactor(x)) stop("expected a factor");  
    CharacterVector levels = x.attr("levels");  
    int nl = levels.size(), nx = x.size();  
    IntegerVector out(nl);  
    for (int i=0; i<nx; ++i)  
      out[x[i]-1]++; // TO DO: NA handling  
    out.names() = levels;  
    return out;  
  }  
' )  
table2(factor(c("a", "b", "b", "c", "b")))
```

```
## a b c
## 1 3 1
```

 **Exercise:** Sort a factor w.r.t. number of levels' occurrences.

 **Exercise:** Implement `split(x, f)`, where `x` – list, `f` – factor.


1.5.2 Matrices


An R matrix is an atomic vector with the `dim` attribute set. The elements are stored in a column-wise order.

```
x <- 1:6
attr(x, "dim") <- c(2, 3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Rcpp is aware of R matrices, see e.g. the `NumericMatrix` class. Among some interesting methods we find e.g. `nrow()`, `ncol()`, etc.

To access an element, we may write e.g. `x(i, j)`.

 **Exercise:** Given a square matrix, check if it's symmetric. Watch out for NAs.

 **Exercise:** Given an $n \times n$ integer matrix, $n = k^2$ for some k , determine if it represents a proper solution to the k -Sudoku problem.

Rules: the matrix consists only in values $\in \{1, \dots, n\}$; Each row and column contains unique values; Each of the $n \times k$ disjoint subsquares contains unique values.

 **Exercise:** Write a function which returns the Hilbert $n \times n$ matrix H for given n ; $h_{i,j} = 1/(i + j - 1)$.

A solution:

```
Rcpp::cppFunction('
  NumericMatrix hilbert1(int n) {
    NumericMatrix out(n, n);
    for (int c=0; c<n; ++c)
      for (int r=0; r<n; ++r)
        out(r, c) = 1.0/(r+c+1);
    return out;
  }
')
```

```
hilbert1(3)
##      [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000
```

The above may also be expressed as follows:

```
Rcpp::cppFunction('
  NumericVector hilbert2(int n) {
    NumericVector out(n*n);
    for (int c=0; c<n; ++c)
      for (int r=0; r<n; ++r)
        out[r+c*n] = 1.0/(r+c+1);
    out.attr("dim") = IntegerVector::create(n, n);
    return out;
  }
')
```

```
' )

hilbert2(3)
##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.5000000 0.3333333
## [2,] 0.5000000 0.3333333 0.2500000
## [3,] 0.3333333 0.2500000 0.2000000
```

Note that `NumericMatrix`'s performance is very good.

```
microbenchmark(hilbert1(2500), hilbert2(2500))
## Unit: milliseconds
##           expr      min       lq   median       uq      max neval
## hilbert1(2500) 89.657 89.836 89.974 90.265 153.07   100
## hilbert2(2500) 89.653 89.839 89.980 90.272 151.42   100
```

By the way, substituting `out(r, c)` for `out(c, r)` decreases speed (more CPU cache misses).

1.5.3 Data frames

An R `data.frame` is a list of atomic vectors, each having the same length. It has the `class`, `names`, and `row.names` attributes set.

```
str(unclass(data.frame(x = c("a", "b"), y = 1:2)))
## List of 2
## $ x: Factor w/ 2 levels "a","b": 1 2
## $ y: int [1:2] 1 2
## - attr(*, "row.names")= int [1:2] 1 2
attributes(data.frame(x = c("a", "b"), y = 1:2))
## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2
##
## $class
## [1] "data.frame"
```

An Rcpp example:

```
Rcpp::cppFunction('
RObject test() {
  List out(2);
  out[0] = NumericVector::create(10, 20, 30);
  out[1] = CharacterVector::create("a", "b", "c");
  out.attr("class") = "data.frame";
  out.names() = CharacterVector::create("col1", "col2");
  out.attr("row.names") =
    CharacterVector::create("row1", "row2", "row3");
  return out;
}
')
```


```
test()
##           col1 col2
## row1      10    a
## row2      20    b
## row3      30    c
```


Also, the `DataFrame` class may be used:

```
Rcpp::cppFunction('
DataFrame test() {
```

```
DataFrame out = DataFrame::create(  
  _["col1"] = NumericVector::create(10, 20, 30),  
  _["col2"] = CharacterVector::create("a", "b", "c");  
  return out;  
}  
'')
```

```
test()  
##   col1 col2  
## 1   10    a  
## 2   20    b  
## 3   30    c
```

 **Exercise:** Write an Rcpp function that does a full join of two data frames on a given factor column.

 **Exercise:** Write an Rcpp function that does a left join of two data frames on a given factor column.

1.6 Summary

An R list stores objects of any type – we have to test for the type ourselves in Rcpp. Calling R functions from Rcpp may be slow, yet it is possible. Playing with compound types is easy – we just have to pay attention to an object's attributes.

1.7 Bibliography

- Eddebuettel 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

2 Rcpp: C++11

2.1 Introduction

C++11 is the most recent version of the C++ programming language standard. It was approved by ISO on 12 August 2011, replacing C++03. It includes several enhancements in: the core language and in the standard library. You will need quite a recent version of the C++ compiler, e.g. g++ 4.8.1 or newer (at least g++ 4.7) or clang++ 3.3. Windows users should note that Rtools provides an outdated g++.

This section is mostly of interest to the readers that are already familiar with C++.

2.2 New features in C++11

auto – type deduction/inference:

```
std::deque<int> obj;  
auto i = obj.begin();  
// std::deque<int>::iterator i = obj.begin();
```

Moreover, decltype can be also used to determine the type of an expression at compile-time:

```
decltype(i) j = 2;
```

foreach-like, range-based for loops:


```
std::vector<int> x
= { 1, 2, 3 }; // an initializer list, new in C++11
for (auto xi : x) {
    std::cout << xi << std::endl;
}

for (int& xi : x) {
    xi *= 2;
}
```

Lambda (anonymous) functions:

```
std::vector<double> x = { 1, 2, 3 };
std::vector<double> y(x.size());
std::transform(x.begin(), x.end(), y.begin(),
    [](double xi) { return xi*xi; } );
std::for_each(std::begin(x), std::end(x),
    // non-member begin() and end(), new in C++11
    [](double xi) {std::cout << xi << std::endl;});
```

Recursive lambdas:

```
std::function<int(int)> fact
= [&fact](int n) {return n < 2 ? 1 : fact(n-1);};
```

New Unicode string literals:

```
// char*:
u8"This is a Unicode Codepoint: \u2018."

// char16_t*:
u"This is a bigger Unicode Codepoint: \u2018."

// char32_t*:
U"This is a Unicode Codepoint: \U00002018."
```

`nullptr` should be used instead of `NULL`. This is because C-like `NULL` commonly expands to either `((void*)0)` or `0`, which interacts poorly with function overloading. On the other hand, `nullptr` denotes a value of type `std::nullptr_t`.

In C++03, enumerations are not type-safe:

Strongly-typed enums:

```
enum class Numbers {One, Two, Three};
Numbers o = Numbers::Three;
```

For more information, read more at isocpp.org.

In the next section, we will discuss the C++ Standard Library (a.k.a. STL). Readers who are already familiar with STL may note its new features:

- Threading facilities (Be careful with randomness)
- Tuple types
- Hash tables (unordered containers)
- Regular expressions
- Pseudorandom number generators

New containers are also available, for example: `forward_list`, `array`, `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`. Also, constant iterators have been defined: `cbegin()`, `cend()`.

2.3 C++11 in Rcpp

To force the `gcc/clang` compiler to use C++11, we call:

```
Sys.setenv(PKG_CXXFLAGS = "-std=c++11")
```

Starting with version 0.10.3 of Rcpp, we can also enable this via the `cpp11` plugin:

```
// [[Rcpp::plugins("cpp11")]]
```

From now on, we'll rely on C++11.

2.4 Summary

Make sure you set up Rcpp properly to compile C++11 code. If you know C++ already (C++98 or C++03), spend some time on exploring new features. You might also be interested in taking a look at the draft of the next C++ standard.

2.5 Bibliography

- en.cppreference.com – C++ reference

3 Rcpp: The C++ Standard Library

3.1 Introduction

The C++ Standard Library (a.k.a. the Standard Template Library¹) is a collection of classes and functions written in the core language. It is part of the C++ ISO standard. STL provides:

- basic data structures (containers),
- implementation of useful algorithms (sorting, searching, pattern matching, etc.),
- I/O stream handling,
- threading support,
-

See en.cppreference.com/w/ for a comprehensive manual.

Note that C++ Standard Library classes and functions are defined in the `std` namespace. For example, to access the `vector` class, we write `std::vector`. If we find that tedious, we may put the following line at the beginning of the source file.

```
using namespace std;
```

3.2 Basic classes

`std::string` stores and manipulates sequences of 8-bit char-based code points.

```
#include <string>
```

As far as ASCII strings are concerned, it is OK to convert between `Rcpp::String` and `std::string`. In fact, `Rcpp::String` has a very similar interface to that of `std::string`. However, unless you are a power user, I generally discourage you from working with strings in Rcpp. If you really have to do so, play with UTF-8:

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

// [[Rcpp::depends(stringi)]]
// [[Rcpp::export]]
String concat(CharacterVector x) {
  Function stri_enc_toutf8("stri_enc_toutf8");
  x = stri_enc_toutf8(x); // now x is surely in UTF-8
}
```

¹These aren't synonyms, but for simplicity we use these terms interchangeably.

```
String out;
for (const auto& s: x)
    out += s;
return out;
}
```

See also `stri_enc_toascii(x)` and `stri_trans_general(x, "LATIN-ASCII")`.

Among “safe” string operations on UTF-8 we find:

- string concatenation
- string searching (fixed pattern matching)

On the other hand, those operations are considered “unsafe”:

- getting length of a string
- string subsetting

This is because one UTF-8 code point not necessarily occupies 1 byte.

We may use Rcpp’s `Rcout` and `Rcerr` to output messages to the R’s console. These two objects are instances of a class inheriting from `std::ostream`.

```
Rcpp::cppFunction('
void ostream_test() {
    Rcout << "test_cout" << std::endl;
    String x = String("R string");
    Rcout << x.get_cstring() << std::endl;
}
')

ostream_test()
## test_cout
## R string
```

Moreover, `std::pair` may be used to encapsulate 2 objects in one, e.g.:

```
#include <utility>
// ...
std::pair<double, int> obj;
Rcout << obj.first; // access to the double-type field
Rcout << obj.second; // the second field
```

By the way, the same may be done by defining a C-style struct.

More generally, `std::tuple` represents an ordered tuple (a fixed-size collection of heterogeneous values).

```
#include <tuple>
// ...
std::tuple<int, int, int> obj
    = std::make_tuple(1, 2, 3);
Rcout << std::get<0>(obj);

int o1, o2, o3;
std::tie(o1, o2, o3) = obj;
Rcout << o1;
```

C++ provides an exception raising and handling mechanism. By default, exception classes derive from `std::exception`. Rcpp catches C++ exceptions and transforms them to R errors.

```
Rcpp::cppFunction('
void test() {
    throw Rcpp::exception(":-(");
}
')

test()
```

```
## Error: :-("
```

In Rcpp it is not important what kind of `std::exception` you throw, you may always use e.g. `Rcpp::exception`. There is even a shortcut for `throw Rcpp::exception("msg")`, `stop("msg")`.

C++ exceptions may be caught (cf. R's `tryCatch()`):

```
Rcpp::cppFunction('
void test() {
  try {
    stop(":-(");
  }
  catch (...) {
    Rcout << "an error occurred";
  }
}
')

test()
## an error occurred
```

Errors raised by called R functions are converted to C++ exceptions:

```
Rcpp::cppFunction('
void test() {
  try {
    Function R_fun("runif");
    R_fun(CharacterVector::create(
      "some incorrect input value"));
  }
  catch (...) {
    Rcout << "an error occurred\\n";
  }
}
')

test()
## Warning: NAs introduced by coercion
## an error occurred
```

Never call R's `Rf_error()` and `Rf_warning()` in your Rcpp functions. Both functions may do a `longjmp` out of the Rcpp's `try...catch` block, which can lead to a memory leak. Surprisingly, this also involves `Rf_warning()`, as a user may wish to convert all warnings to R errors. Thus, `Rf_error()`s and `Rf_warning()`s are not compatible with C++ exceptions.

```
Rcpp::sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;
class Test { public:
  Test() { Rcout << "start\\n"; }
  ~Test() { Rcout << "end\\n"; }
};
// [[Rcpp::export]]
void test() {
  Test t;
  Rf_warning("test");
} ')
options(warn=10); test()

## start
## Error: (converted from warning) test
```

```
Rcpp::sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;
class Test { public:
    Test() { Rcout << "start\\n"; }
    ~Test() { Rcout << "end\\n"; }
};
// [[Rcpp::export]]
void test() {
    Test t;
    Function warning("warning");
    warning("test");
} '
options(warn=10); test()

## start
## end
## Error: (converted from warning) test
```

3.3 Containers

We are going to present a brief overview of STL containers. Just note what container types are available and what are their strengths. You can always consult the STL manual for more detailed information on their usage.

Generally, it will be difficult to implement more effective R built-in operations by our own means. STL is useful when implementing complex, time-consuming algorithms. A typical scenario involves the following scheme:

1. Input data from R (Rcpp lists, atomic vectors, etc.)
2. Play with STL's containers & algorithms
3. Generate R output objects (Rcpp lists, atomic vectors, etc.)

3.3.1 Vectors

`std::vector` is a container encapsulating dynamic size arrays.

```
#include <vector>
```

The elements are stored contiguously. Thus, a `std::vector` is like an atomic vector in R.

Here are time complexities of common operations:

- Random access – $O(1)$
- Insertion or removal of elements at the end – amortized $O(1)$
- Insertion or removal of elements – $O(n)$

`std::vector` is a generic container. To define the types of elements it stores, we write `std::vector<TYPE>` e.g. `std::vector<double>`.

A first example:

```
#include <Rcpp.h>
#include <vector>
// [[Rcpp::plugins("cpp11")]]

double my_sum(const std::vector<double>& v) {
    double res = 0.0;
    for (size_t i=0; i<v.size(); ++i)
        res += v[i];
    return res;
}
```

```
// [[Rcpp::export]]
double vector_test() {
  std::vector<double> x(10); // 10 doubles
  for (size_t i=0; i<x.size(); ++i)
    x[i] = Rf_runif(0, 1);
  return my_sum(x);
}
```

The above is equivalent to:

```
#include <Rcpp.h>
#include <vector>
// [[Rcpp::plugins("cpp11")]]

double my_sum(const std::vector<double>& v) {
  double res = 0.0;
  for (auto it = v.cbegin(); it != v.cend(); ++it)
    res += *it;
  return res;
}

// [[Rcpp::export]]
double vector_test() {
  std::vector<double> x(10); // 10 doubles
  for (auto it = x.begin(); it != x.end(); ++it)
    *it = Rf_runif(0, 1);
  return my_sum(x);
}
```

Which in turn is equivalent to:

```
#include <Rcpp.h>
#include <vector>
// [[Rcpp::plugins("cpp11")]]

double my_sum(const std::vector<double>& v) {
  double res = 0.0;
  for (auto it = v.cbegin(); it != v.cend(); ++it)
    res += *it;
  return res;
}

// [[Rcpp::export]]
double vector_test() {
  std::vector<double> x(10); // 10 doubles
  for (auto it = x.begin(); it != x.end(); ++it)
    *it = Rf_runif(0, 1);
  return my_sum(x);
}
```

Note that Rcpp's Vector's interface is inspired by `std::vector`'s one.

```
#include <Rcpp.h>
// [[Rcpp::plugins("cpp11")]]

double my_sum(const Rcpp::NumericVector& v) {
  double res = 0.0;
  for (auto it = v.begin(); it != v.end(); ++it)
    res += *it;
  return res;
}
```

```
// [[Rcpp::export]]
double vector_test(Rcpp::NumericVector x) {
    return my_sum(x);
}
```

Moreover:

```
#include <Rcpp.h>
// [[Rcpp::plugins("cpp11")]]

double sum(const Rcpp::NumericVector& v) {
    double res = 0.0;
    for (auto elem: v)
        res += elem;
    return res;
}

// [[Rcpp::export]]
double vector_test(Rcpp::NumericVector x) {
    return sum(x);
}
```

3.3.2 Double-ended queues

`std::deque` is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end.


```
#include <deque>
```

As opposed to `std::vector`, the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays.

Time complexity of common operations:

- Random access – $O(1)$
- Insertion or removal of elements at the end or beginning – amortized $O(1)$
- Insertion or removal of elements – $O(n)$

Expansion of a deque is cheaper than the expansion of a `std::vector` because it does not involve copying of the existing elements to a new memory location.

 **Exercise:** Implement R's `rle()` using `std::deque`.

Here, `std::deque` is useful, as we do not know the number of output elements in advance.

An exemplary implementation gives:

```
x <- sample(1:5, replace=TRUE, 100000)
microbenchmark(rle(x), rle2(x), rle3(x))
## Unit: milliseconds
##      expr      min       lq      median        uq       max neval
##  rle(x) 18.766212 19.210545 21.091808 21.530202 91.71746   100
## rle2(x)  2.412978  2.559377  2.832134  4.829493 73.00219   100
```

Hint: Use `std::pair` while writing your implementation.

3.3.3 Lists


`std::list` is a container that supports constant time insertion and removal of elements from anywhere in the container.

```
#include <list>
```

Time complexity of common operations:

- Random access – N/A
- Insertion or removal of elements at the end or beginning – $O(1)$
- Insertion or removal of elements – $O(1)$

See also: `std::stack`, `std::queue`.

 **Exercise:** *Implement the Graham scan – finding the convex hull of a set of points in 2D; input: numeric matrix with 2 columns.*

3.3.4 Priority queues

`std::priority_queue` is a container that provides $O(1)$ time element extraction and logarithmic-time insertion.

```
#include <queue>
```

Any *comparer* may be provided. This data structure is useful for implementing certain graph algorithms, like Dijkstra's shortest path search.

3.3.5 Sets

`std::set` and `std::unordered_set` contain a set of unique objects. Search, insertion, and removal have average logarithmic- and average constant-time complexity, respectively.

```
#include <set>
#include <unordered_set>
```

By the way, multisets are also available.

 **Exercise:** *Implement `unique()` using `std::unordered_set`.*

A solution:

```
Sys.setenv("PKG_CXXFLAGS"="-std=c++11")
Rcpp::cppFunction('
    NumericVector unique2(NumericVector x) {
        std::unordered_set<double> set;
        for (const auto& e : x)
            set.insert(e);

        NumericVector ret(set.size());
        int i=0;
        for (const auto& item : set)
            ret[i++] = item;
        return ret;
    }
', includes="#include <unordered_set>")


x <- as.double(sample(1:100, replace=TRUE, 100000))
microbenchmark(u2=unique2(x), uR=unique(x))
## Unit: milliseconds
## expr    min      lq median      uq    max neval
##  u2 3.3793 3.3968 3.4223 3.4620 4.6164   100
##  uR 3.9105 3.9355 4.0555 4.5415 5.7944   100
x <- as.double(sample(1:10000, replace=TRUE, 100000))
microbenchmark(u2=unique2(x), uR=unique(x))
## Unit: milliseconds
## expr    min      lq median      uq    max neval
##  u2 8.1123 8.2253 8.3623 8.4573 11.454   100
##  uR 5.3490 5.3928 5.4790 5.9702 11.450   100
```


3.3.6 Maps

`std::map` and `std::unordered_map` are associative containers consisting of key-value pairs with unique keys. Search, removal, and insertion operations have logarithmic- and average constant-time complexity, respectively.

```
#include <map>
#include <unordered_map>
```

By the way, multimaps are also available.

 **Exercise:** Implement R's `table()` for character vectors (word count) using `std::unordered_map`. Return a named integer vector.

My exemplary implementation gives::

```
set.seed(123)
x <- stringi::stri_rand_strings(10000, 5, "[a-z]")
microbenchmark(wordcount(x), table(x))
## Unit: milliseconds
##      expr      min       lq     median       uq      max neval
## wordcount(x) 6.12595  6.75791  7.07298  7.48410 12.3634   100
##      table(x) 60.08283 61.54373 63.03329 64.86751 134.6654   100
```

Hint: Use `std::unordered_map<std::string, int>`. `Rcpp::String` is “compatible” with `std::string`.

 **Exercise:** What about the `table()` function for factors?


3.4 Algorithms

```
#include <algorithms>
```


provides us with a set of useful algorithms.

Since Rcpp Vectors implement `RandomAccessIterators` (see `begin()` and `end()` methods), most of the algorithms may be directly applied on them.

For example, `std::sort()` implements a $O(n \log n)$ sorting algorithm. Moreover, `std::stable_sort()` provides a $O(n \log^2 n)$ worst-case stable sort. By default, the elements are compared using `operator<`. However, other comparers may be provided.

 **Exercise:** Implement `sort()` for numeric vectors.

 **Exercise:** Implement `sort()` for numeric vectors (stable).

 **Exercise:** Implement `order()` for numeric vectors (stable).


A solution:

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

struct Comparer {
    NumericVector& x;
    Comparer(NumericVector& _x) : x(_x) { }
    bool operator()(int i, int j) {
        return x[i-1] <= x[j-1]; // 1-based indices => 0-based
    }
};
```

```
// [[Rcpp::export]]
IntegerVector order2(NumericVector x) {
  int n = x.size();
  IntegerVector y(n);
  for (int i=0; i<n; ++i) y[i] = i+1;
  Comparer comp(x);
  std::stable_sort(y.begin(), y.end(), comp);
  return y;
}
```

This is ca. 1.5x faster than R's `order()`.

 **Exercise:** For a given permutation (an integer sequence), generate next lexicographically greater permutation of $\{1, \dots, n\}$.


A solution:


```
Rcpp::cppFunction('
RObject nextperm(IntegerVector x) {
  IntegerVector out = Rcpp::clone(x);
  if (!std::next_permutation(out.begin(), out.end()))
    return R_NilValue;
  return out;
}')
x <- 1:3
repeat { print(x); x <- nextperm(x); if (is.null(x)) break }
## [1] 1 2 3
## [1] 1 3 2
## [1] 2 1 3
## [1] 2 3 1
## [1] 3 1 2
## [1] 3 2 1
```


Another example: Using `std::any_of()` and anonymous functions:

```
Sys.setenv("PKG_CXXFLAGS"="-std=c++11")
Rcpp::cppFunction('
double sum_if_not_na(NumericVector v) {
  if (std::any_of(v.begin(), v.end(),
    [](double x) -> bool{return NumericVector::is_na(x);}))
    stop("NA found");
  return std::accumulate(v.begin(), v.end(), 0.0);
}
', includes=c("#include <algorithm>"))

sum_if_not_na(c(1, 2, 3))
## [1] 6
sum_if_not_na(c(1, 2, NA))
## Error: NA found
```

 **Exercise:** Two equal-sized vectors are given: x_0, y_0 ; x_0 is increasingly sorted. Given a numeric vector x , calculate $f(x)$, where f is a piecewise linear function interpolating (x_0, y_0) , cf. `approx()`.

 **Exercise:** Implement `match(x, y)`, where x, y – integer vectors; y – sorted.

 **Exercise:** Implement `findInterval(x, y)`, x, y – numeric vector; y – sorted

3.5 External pointers

STL containers are mostly used *within* R functions. It turns out that any C/C++ objects may be alive between R functions calls. This may lead to a very effective R code, especially when you are not able to implement everything purely in C++. However, such objects **are not serialized**. In other words, they become invalidated after you quit R. Restoring R session will not resurrect them.

An example:

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

// [[Rcpp::export]]
XPtr< std::queue<double> > queue_create() {
    std::queue<double>* queue = new std::queue<double>();
    return XPtr< std::queue<double> >(queue, true);
}

// [[Rcpp::export]]
bool queue_empty(XPtr< std::queue<double> > queue) {
    return (*queue).empty();
}

// [[Rcpp::export]]
void queue_push(XPtr< std::queue<double> > queue, double obj) {
    (*queue).push(obj);
}

// [[Rcpp::export]]
double queue_pop(XPtr< std::queue<double> > queue) {
    if ((*queue).empty()) stop("empty_queue");
    double obj = (*queue).front();
    (*queue).pop();
    return obj;
}
```

Usage:

```
q <- queue_create()
queue_push(q, 1)
queue_push(q, 2)
queue_pop(q)
## [1] 1
queue_push(q, 3)
while (!queue_empty(q)) print(queue_pop(q))
## [1] 2
## [1] 3
```

What if we would wish to **store R objects in an STL container**? We should be careful: R uses a **garbage collector**. Objects in the container may be automatically deleted. In such a case, we may use a call to `R_PreserveObject()`; it adds an R object to an internal list of objects not to be garbage collected. It is the user's responsibility to release the preserved objects when they are no longer needed. A subsequent call to `R_ReleaseObject()` removes it from that list.

Let us implement a queue of arbitrary R objects.

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

class Q : public std::queue<SEXP> {
public: ~Q() {
    while (!this->empty()) {
        SEXP obj = this->front();
        this->pop();
    }
}
```

```

        R_ReleaseObject(obj);
    }
}
};

/* *** Queue of arbitrary R objects (cont'd) *** */
// [[Rcpp::export]]
XPtr< Q > queue_create() {
    Q* queue = new Q();
    return XPtr< Q >(queue, true);
}
// [[Rcpp::export]]
bool queue_empty(XPtr< Q > queue) {
    return (*queue).empty();
}
// [[Rcpp::export]]
void queue_push(XPtr< Q > queue, SEXP obj) {
    R_PreserveObject(obj);
    (*queue).push(obj);
}
// [[Rcpp::export]]
SEXP queue_pop(XPtr< Q > queue) {
    if ((*queue).empty()) stop("empty_queue");
    SEXP obj = (*queue).front();
    (*queue).pop();
    R_ReleaseObject(obj);
    return obj;
}

```

Test:

```

q <- queue_create()
x1 <- 1:10
x2 <- c("a", "b")
queue_push(q, x1)
queue_push(q, x2)
queue_pop(q)
## [1] 1 2 3 4 5 6 7 8 9 10
queue_pop(q)
## [1] "a" "b"

```

3.6 Summary

The C++ Standard Library (and also the boost libraries) provides a set of very important algorithms and data structures. They are well-tested and highly efficient. They are here to serve you in writing your own Rcpp functions.

3.7 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 in R packages

4.1 Introduction

Rcpp may be easily incorporated into R packages. Recall that R packages provide us the method of choice when it comes to developing “large” R software.

4.2 Rcpp in R packages

In order to use Rcpp in R packages:

- a) the DESCRIPTION file must include:

```
LinkingTo: Rcpp
Imports: Rcpp
```

- b) the NAMESPAC file must include:

```
useDynLib(mypackage)
importFrom(Rcpp, sourceCpp)
```

Including compiled code in R packages is quite difficult, see *Writing R Extensions*, Sec. 5. Luckily, we have the roxygen2 package, the `Rcpp::compileAttributes()` and other functions. These are automatically called by RStudio on a package’s build (CTRL+SHIFT+B).

4.2.1 A tutorial

1. Start by using Rcpp’s default package skeleton:

```
library("Rcpp")
setwd("your_favorite_directory")
Rcpp.package.skeleton("mypackage")
```

2. Create a new RStudio project (associating it with *Existing Directory*).
3. Edit Project’s options (*Tools* menu). In the Build Tools tab, click *Generate documentation with Roxygen*. Check all the fields under *Roxygen Options*.
4. Edit the DESCRIPTION file accordingly.
5. Remove the Read-and-delete-me, man/*, and src/rcpp_hello_world.cpp files.
6. Create a file R/mypackage-package.R.

```
#' @title package title...
# '
# ' @description
# ' package description....
# '
# ' @useDynLib mypackage
# ' @name mypackage-package
# ' @docType package
# ' @importFrom Rcpp sourceCpp
invisible(NULL)
```

This will be used to generate the package’s main man page (in man/mypackage-package.Rd). Moreover, it assures us that useDynLib and importFrom are included in the NAMESPAC file.

7. Add a C++ source file, e.g. src/test.cpp.

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

//' @title ...
//' @description ...
//'
//' @param min ...
//' @param max ...
//' @return ...
//'
//' @export
// [[Rcpp::export]]
double test(double min=0.0, double max=1.0) {
  return Rf_runif(min, max);
}
```

The `//'`-comments will be inserted in `src/test.R` as roxygen2 comments.

8. Build the package (CTRL+SHIFT+B). roxygen2 and `Rcpp::compileAttributes()` are called (see the *Build* tab).

Note the `R/RcppExports.R` and `src/RcppExports.cpp` files. They are automatically generated by `Rcpp::compileAttributes()`.

9. Type `test()` in the R console.

4.3 C++11 in R packages

From version 3.1.0, R provides support for C++11 in packages, in addition to C++98. In order to use C++11 code in a package, the package's `src/Makevars` file should include the line `CXX_STD = CXX11`.

For older versions of R, we may put `PKG_CPPFLAGS = -std=c++11` into `src/Makevars`. However, this solution is not portable: it works with the `gcc/clang` compilers only.

Note that including:

```
// [[Rcpp::plugins(cpp11)]]
```

in a `.cpp` file does not change the compiler flags.

Packages without a `Makevars` file may specify that they require C++11 by including C++11 in the `SystemRequirements` field of the `DESCRIPTION` file. However, the use of C++11 in packages developed with CRAN submission in mind is discouraged (as of July 2014).

By the way, the `Rcpp11` package provides us with a complete redesign of `Rcpp`. The `DESCRIPTION` file should include the following lines:

```
LinkingTo: Rcpp11
SystemRequirements: C++11
```

Unfortunately, the package is still in its early days (as of July, 2014). I encourage you to study the package's facilities on your own and keep track of its new versions. This is because the package's design looks very promising.

4.4 Summary

By using `Rcpp` code in R packages you can quite easily provide all their users with facilities that are efficient w.r.t. to CPU time and memory consumption.

4.5 Bibliography

- R Core Team, *Writing R Extensions*, 2014