High performance functions with Rcpp

Sometimes R code just isn't fast enough - you've used profiling to find the bottleneck, but there's simply no way to make the code any faster. This chapter is the answer to that problem. You'll learn how to rewrite key functions in C++ to get much better performance, while not taking too much longer to write. The key to this magic is Rcpp, a fantastic tool written by Dirk Eddelbuettel and Romain Francois (with key contribution by Doug Bates, John Chambers and JJ Allaire), that makes it dead simple to connect C++ to R. It is *possible* to write C or Fortran code for use in R, but it will be painful; Rcpp provides a clean, approachable API that lets you write high-performance code, insulated from R's arcane C API.

Typical bottlenecks that C++ can help with are:

- Loops that can't easily be vectorised because each iteration depends on the previous. C++ modifies objects in place, so there is little overhead when modifying a data structure many times.
- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than the overhead of calling a function in R: ~5ns compared to ~200ns.
- Problems that require advanced data structures and algorithms that R doesn't provide. Through the standard template library (STL), C++ has efficient implementations of many important data structures, from ordered maps to double ended queues.

Rewriting a function in C++ can lead to a 2-3 order of magnitude speed up, but most improvements will be more modest. While pure R code is relatively slow compared to C or C++, many bottlenecks in base R have already implemented as hand-written C functions. This means that if your function already uses vectorised operations, you are unlikely to see a large improvement in performance, perhaps around 20-50%. It's possible to get greater improvements by using C++ performance tricks, but those are beyond the scope of this chapter, and you'll need to consult other sources. Compared to base R implementations, the big advantage of C++ code is that it's much more concise: often around 10% of length of the equivalent C code in the R sources.

The aim of this chapter is to give you the absolute necessities of C++ and Rcpp. Dirk Eddelbuettel is currently working on an entire book on Rcpp, "Seamless R and C++ integration with Rcpp", which will provide much more detail than we can here. If you're serious about Rcpp, make sure to get that book when it comes out!

In this chapter you'll learn:

- How to write C++ code by seeing R functions and their C++ equivalents.
- Important Rcpp classes and methods
- How to use Rcpp "sugar" to avoid C++ loops and convert directly from vectorised R code
- How to work with missing values
- Some of the most important techniques, data structures and algorithms from standard template library (STL)

The chapter concludes with a selection of real case studies showing how others have used C++ and Rcpp to speed up their slow R code.

Getting started

All examples in this chapter need at least version 0.10.1 of the `Rcpp` package. This version includes `cppFunction` and `sourceCpp`, which makes it very easy to connect C++ to R. You'll also (obviously!) need a working C++ compiler.

`cppFunction` allows you to write C++ functions in R like this:

```
library(Rcpp)
cppFunction('
  int add(int x, int y, int z) {
    int sum = x + y + z;
    return sum;
  }'
)
formals(add)
add(1, 2, 3)
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function. If you're familiar with `inline::cxxfunction`, `cppFunction` is similar, except that you specify the function completely in the string, and it parses the C++ function arguments to figure out what the R function arguments should be:

Getting starting with C++

C++ is a large language, and there's no way to cover it exhaustively here. Our aim is to give you the basics so that you can start writing useful functions. We'll spend minimal time on advanced features like object oriented programming and templates, because our focus is not on writing big programs in C++, just small,

mostly self-contained functions that allow you to speed up slow parts of your R code.

The following section shows the basics of C++ by translating simple R functions to their R equivalents. We'll start simple with a function that has no inputs and a scalar output, and then get progressively more complicated:

- Scalar input and scalar output
- Vector input and scalar output
- Vector input and vector output
- Matrix input and vector output

No inputs, scalar output

Let's start with a very simple function. It has no arguments and always returns the the integer 1:

```
one <- function() 1L
```

The equivalent C++ function is:

```
int one() {
  return 1;
}
```

We can compile and use this from R with `cppFunction`

```
cppFunction('
  int one() {
    return 1;
  }
')
```

This small function illustrates a number of important differences between R and C++:

- The syntax to create a function looks like the syntax to call a function; you don't use assignment to create functions like in R.
- You must declare the type of output the function returns. This function returns an `int` (a scalar integer). The classes for the most common types of R vectors are: `NumericVector`, `IntegerVector`, `CharacterVector` and `LogicalVector`.

- C++ distinguishes between scalars and vectors. The scalar equivalents of numeric, integer, character and logical vectors are: `double`, `int`, `std::string` and `bool`. (Unfortunately the names of C++ scalars and R vectors are inconsistent for historical reasons)
- You must use an explicit `return` statement to return a value from the function.
- Every statement is terminated by a `;`.

Scalar input, scalar output

The next example function makes things a little more complicated by implementating a scalar version of the `sign` function which returns 1 if the input is positive, and -1 if it's negative:

```
sign1 <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}

cppFunction('
  int sign2(int x) {
    if (x > 0) {
      return 1;
    } else if (x == 0) {
      return 0;
    } else {
      return -1;
    }
  }'
)
```

In the C++ version:

- we declare the type of each input in the same way we declare the type of the output. While this makes the code a little more verbose, it also makes it very obvious what type of input the function needs.

- the `if` syntax is identical - while there are some big differences between R and C++, there are also lots of similarities! C++ also has a `while` statement that works the same way as R's. You can also use `break`, but use `continue` instead of `next`.

Vector input, scalar output

One big difference between R and C++ is that the cost of loops is much lower. For example, we could implement the `sum` function in R using a loop. If you've been programming in R a while, you'll probably have a visceral reaction to this function!

```
sum1 <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

In C++, loops have very little overhead, so it's fine to use them. However, we'll see some alternatives to for loops that more clearly express your intent; they're not faster, but they make your code easier to understand.

```
cppFunction('
  double sum2(NumericVector x) {
    int n = x.size();
    double total = 0;
    for(int i = 0; i < n; ++i) {
      total += x[i];
    }
    return total;
  }
')
```

The C++ version is similar, but:

- To find the length of the vector, we use the `size()` method, which returns an integer. Again, whenever we create a new variable we have to tell C++ what type of object it will hold. An `int` is a scalar integer, but we could have used `double` for a scalar numeric, `bool` for a scalar logical, or a `std::string` for a scalar character vector.

- The `for` statement has a different syntax: `for(intialisation; condition; increase)`. The initialise component creates a new variable called `i` and sets it equal to 0. The condition is checked in each iteration of the loop: the loop is continues while it's `true`. The increase statement is run after each loop iteration, but before the condition is checked. Here we use the special prefix operator `++` which increases the value of `i` by 1.
- Vectors in C++ start at 0. I'll say this again because it's so important: VECTORS IN C++ START AT 0! Forgetting that is probably the most common source of bugs when converting R functions to C++.
- We can't use `<-` (or `->`) for assignment, but instead use `=`.
- We can take advantage of the in-place modification operators: `total += x[i]` is equivalent to `total = total + x[i]`. Similar in-place operators are `-=`, `*=` and `/=`.

This is a good example of where C++ is much more efficient than the R equivalent. As shown by the following microbenchmark, our `sum2` function is competitive with the built-in (and highly optimised) `sum` function, while `sum1` is several orders of magnitude slower.

```
library(microbenchmark)
x <- runif(1e3)
microbenchmark(
  sum(x),
  sum1(x),
  sum2(x)
)
```

It's possible to make our version of sum even faster if we use some tricks, but those are beyond the scope of this book.

Vector input, vector output

For our next example, we'll create a function that computes the distance between one value and a vector of other values:

```
pdist1 <- function(x, ys) {
  (x - ys) ^ 2
}
```

From the function definition, it's not obvious that we want `x` to be a scalar - that's something we'd need to mention in the documentation. That's not a problem in the C++ version:

```
cppFunction('
  NumericVector pdist2(double x, NumericVector ys) {
    int n = ys.size();
    NumericVector out(n);

    for(int i = 0; i < n; ++i) {
      out[i] = pow(ys[i] - x, 2);
    }
    return out;
  }
')
```

This function introduces only a few new concepts:

- We create a new numeric vector of length `n` with a constructor: `NumericVector out(n)`. Another useful constructor takes an existing vector and makes a copy: `NumericVector zs(ys)` would create a new numeric vector called `zs` that contained a copy of `ys`.
- C++ doesn't use `^` for exponentiation, it instead uses the `pow` function.

Note that because the R function is fully vectorised, it is already going to be extremely fast. On my computer, it takes around 8 ms with a 1 million element `y` vector. The C++ function is twice as fast, ~4 ms, but assuming it took you 10 minutes to write the C++ function, you'd need to run it ~150,000 times to make it a net saver of time. The reason why the C++ function is faster is subtle, and relates to memory management. The R version needs to create an intermediate vector the same length as y (`x - ys`), and allocating memory is an expensive operation. The C++ function avoids this overhead because it uses an intermediate scalar.

In the sugar section, you'll see how to rewrite this function to take advantage of Rcpp's vectorised operations so that the C++ code is barely longer than the R code.

Matrix input, vector output

Each vector type also has a matrix equivalent: `NumericMatrix`, `IntegerMatrix`, `CharacterMatrix` and `LogicalMatrix`. Using them is straightforward. For example, we could create a function that reproduces `rowSums`:

```
cppFunction('
  NumericVector row_sums(NumericMatrix x) {
    int nrow = x.nrow(), ncol = x.ncol();
    NumericVector out(nrow);
```

```
    for (int i = 0; i < nrow; i++) {
      double total = 0;
      for (int j = 0; j < ncol; j++) {
        total += x(i, j);
      }
      out[i] = total;
    }
    return out;
  }
')
x <- matrix(sample(100), 10)
rowSums(x)
```

The main thing to notice is that when subsetting a matrix we use `()` and not `[]`, and that matrix objects have `nrow()` and `ncol()` methods.

Using sourceCpp

To simplify the initial presentation the examples in this section have used inline C++ via `cppFunction`. For real problems, it's usually easier to use standalone C++ files and then source them into R using the `sourceCpp` function. This will enable you to take advantage of text editor support for C++ files (e.g. syntax highlighting) as well as make it easier to identify the line numbers of compilation errors. Standalone C++ files can also contain embedded R code in special C++ comment blocks. This is really convenient if you want to run some R test code.

Your standalone C++ file should have extension `.cpp`, and needs to start with:

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

And for each function that you want availble within R, you need to prefix it with:

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

Then using `sourceCpp("path/to/file.cpp")` will compile the C++ code, create the matching R functions and add them to your current session.

For example, running `sourceCpp` on the following file first compiles the C++ code and then compares it to native equivalent:

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

```
// [[Rcpp::export]]
double mean1(NumericVector x) {
  int n = x.size();
  double total = 0;

  for(int i = 0; i < n; ++i) {
    total =+ x[i] / n;
  }
  return total;
}

/*** R
  library(microbenchmark)
  x <- runif(1e5)
  microbenchmark(
    mean(x),
    mean1(x))
*/
```

The R code is run with `source(echo = TRUE)` so you don't need to explicitly print output.

For the remainder of this chapter C++ code will typically be presented standalone rather than wrapped in a call to `cppFunction`. If you want to try compiling and/or modifying the examples you should paste them into a C++ source file that includes the elements described above.

Exercises

With the basics of C++ in hand, now is a great time to practice by reading and writing some simple C++ functions.

For each of the following C++ functions, read the code and figure out what base R function it corresponds to. You might not understand every part of the code yet, but you should be able to figure out the basics of what the function does.

```
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y =+ x[i] / n;
  }
  return y;
```

```
}

NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out =+ out[i - 1] + x[i];
  }
  return out;
}

bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

int f4(Function pred, List x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

NumericVector f5(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x[i], y[i]);
  }

  return out;
}
```

To practice your function writing skills, convert the following functions into C++. For now, assume the inputs have no missing values.

- `all`
- `cumprod`, `cummin`, `cummax`.
- `diff`. Start by assuming lag 1, and then generalise for lag n.
- `range`.
- `var`. Read about the approaches you can take at wikipedia. Whenever implementing a numerical algorithm it's always good to check what is already known about the problem.

Rcpp classes and methods

You've already seen the basic vector classes (`IntegerVector`, `NumericVector`, LogicalVector,CharacterVector) and their scalar (int,double,bool,std::string) and matrix (IntegerMatrix,NumericMatrix, `LogicalMatrix`, `CharacterMatrix`) equivalents.

All R objects have attributes, which can be queried and modified with the `attr` method. Rcpp also provides a `names()` method for the commonly used attribute: `attr("names")`. The following code snippet illustrates these methods. Note the use of the `create()` class method to easily create an R vector from C++ scalar values.

```
// [[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;
}
```

You can use the `slot()` method in a similar way to get and set slots of S4 objects.

Rcpp also provides classes `List` and `DataFrame`. These are more useful for output than input, because lists and data frames can contain arbitrary classes, and this does not fit well with C++'s desire to have the types of all inputs known in advance. If, however, the list is an S3 object with components of

known types, you can extract the components and manually convert to their C++ equivalents with `as`.

For example, the linear model objects that `lm` produces are lists and the components are always of the same type. The following code illustrates how you might component the mean percentage error (`mpe`) of a linear model. This isn't a very good example for when you might use C++ (because it's so easily implemented in R), but it illustrates how to pull out the components of a list. Note the use of the `inherits()` method and the `stop()` function to check that the object really is a linear model.

```
// [[Rcpp::export]]
double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

  NumericVector resid = as<NumericVector>(mod["residuals"]);
  NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

  int n = resid.size();
  double err = 0;
  for(int i = 0; i < n; ++i) {
    err += resid[i] / (fitted[i] + resid[i]);
  }
  return err / n;
}

/*** R
  mod <- lm(mpg ~ wt, data = mtcars)
  mpe(mod)
*/
```

It is possible to write code that works differently depending on the type of the R input, but it is beyond the scope of this book.

You can put R functions in an object of type `Function`. Calling an R function from C++ is straightforward. The string constructor of the function object will look for a function of that name in the global environment.

```
Function assign("assign");
```

You can call functions with arguments specified by position:

```
assign("y", 1);
```

Or by name, with a special syntax:

```
assign(_["x"] = "y", _["value"] = 1);
```

The challenge is storing the output. If you don't know in advance what the output will be, store it in an `RObject` or in components of a `List`. For example, the following code is a basic implementation of `lapply` in C++:

```
// [[Rcpp::export]]
List lapply1(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;
}

/*** R
  lapply1(1:10, sqrt)
  lapply1(list("a", 1, F), class)
*/
```

There are also classes for many more specialised language objects: `Environment`, ComplexVector, RawVector, DottedPair, Language, Promise, Symbol, WeakReference‘ and so on. These are beyond the scope of this chapter and won't be discussed further.

Rcpp sugar

Rcpp provides a lot of "sugar", C++ functions that work very similarly to their R equivalents. (The main difference is that they don't recycle their inputs - you need to do that yourself). Rcpp sugar makes it possible to write efficient C++ code that looks almost identical to the R equivalent. If a sugar version of the function you're interested exists, you should use it: it's expressive and well tested. Sugar functions aren't always faster than your hand-written equivalent, but they will get faster as more time is spend on optimising Rcpp.

Sugar functions can be roughly broken down into

- arithmetic and logical operators
- logical summary functions
- vector views
- other useful functions

Arithmetic and logical operators

All the basic arithmetic and logical operators are vectorised: `+` `*`, `-`, `/`, `pow`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`. For example, we could use sugar to considerably simply the implementation of our `pdist2` function:

```
pdist1 <- function(x, ys) {
  (x - ys) ^ 2
}
```

```
// [[Rcpp::export]]
NumericVector pdist3(double x, NumericVector ys) {
  return pow((x - ys), 2);
}
```

Logical summary functions

The sugar function `any` and `all` are fully lazy, so that e.g `any(x == 0)` might only need to evaluate one element of the value, and return a special type that can be converted into a `bool` using `is_true`, `is_false`, or `is_na`.

For example, we could use this to write an efficient funtion to determine whether or not a numeric vector contains any missing values. In R we could do `any(is.na(x))`:

```
any_na1 <- function(x) any(is.na(x))
```

However that will do almost the same amount of work whether there's a missing value in the first position or the last. Here's the C++ implementation:

```
// [[Rcpp::export]]
bool any_na2(NumericVector x) {
  return is_true(any(is_na(x)));
}
```

Our C++ `any_na2` function is slightly slower `any_na1` when there are no missing values, or the missing value is at the end, but it's much faster when the first value is missing.

```
library(microbenchmark)
x0 <- runif(1e5)
x1 <- c(x0, NA)
x2 <- c(NA, x0)
```

```
microbenchmark(
  any_na1(x0), any_na2(x0),
  any_na1(x1), any_na2(x1),
  any_na1(x2), any_na2(x2))
```

Vector views

A number of helpful functions provide a "view" of a vector: `head`, `tail`, `rep_each`, `rep_len`, `rev`, `seq_along`, `seq_len`. In R these would all produce copies of the vector, but in Rcpp they simply point to the existing vector and override the subsetting operator (`[`) to implement special behaviour. This makes them very efficient: `rep_len(x, 1e6)` does not have to make a million copies of x.

Other useful functions

Finally, there are a grab bag of sugar functions that mimic frequently used R functions:

- Math functions: `abs`, `acos`, `asin`, `atan`, `beta`, `ceil`, `ceiling`, `choose`, `cos`, `cosh`, `digamma`, `exp`, `expm1`, `factorial`, `floor`, `gamma`, `lbeta`, `lchoose`, `lfactorial`, `lgamma`, `log`, `log10`, `log1p`, `pentagamma`, `psigamma`, `round`, `signif`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `tetragamma`, `trigamma`, `trunc`,
- Scalar summaries: `mean`, `min`, `max`, `range`, `sum`, `sd` and `var`.
- Vector summaries: `cumsum`, `diff`, `pmin`, and `pmax`
- Finding values: `match`, `self_match`, `which_max`, `which_min`
- `duplicated`, `unique`
- `d/q/p/r` for all standard distributions in R.
- `noNA(x)`: this asserts that the vector `x` does not contain any missing values, and allows optimisation of some mathematical operations.

Missing values

If you're working with missing values, you need to know two things:

- what happens when you put missing values in scalars (e.g. `double`)
- how to get and set missing values in vectors (e.g. `NumericVector`)

Scalars

The following code explores what happens when you take one of R's missing values, coerce it into a scalar, and then coerce back to an R vector. This is a generally useful technique: if you don't know what an operation will do, design an experiment to figure it out.

```
// [[Rcpp::export]]
List scalar_missings() {
  CharacterVector chr(1);
  chr[0] = NA_STRING;

  int int_s = NA_INTEGER;
  std::string chr_s = std::string(chr[0]);
  bool lgl_s = NA_LOGICAL;
  double num_s = NA_REAL;

  return(List::create(int_s, chr_s, lgl_s, num_s));
}

/*** R
  str(scalar_missings())
*/
```

So

- `IntegerVector` -> `int`: stored as the smallest integer. If you leave as is, it will be preserved, but no C++ operations are aware of the missingness: `evalCpp('NA_INTEGER + 1')` gives -2147483647.
- `CharacterVector` -> `std::string`: the string "NA"
- `LogicalVector` -> `bool`: TRUE. To work with missing values in logical vectors, use an `int` instead of a `bool`.
- `NumericVector` -> `double`: stored as an NaN, and preserved. Most numerical operations will behave as you expect, but logical comparisons will not. See below for more details.

If you're working with doubles, you may be able to get away with ignoring missing values and working with NaN (not a number). R's missing values are a special type of the IEEE 754 floating point number NaN. That means if you coerce them to `double` in your C++ code, they will behave like regular NaN's. That means, in a logical context they always evaluate to FALSE:

```
evalCpp("NAN == 1")
evalCpp("NAN < 1")
evalCpp("NAN > 1")
evalCpp("NAN == NAN")
```

But be careful when combining then with boolean values:

```
evalCpp("NAN && TRUE")
evalCpp("NAN || FALSE")
```

In numeric contexts, they propagate similarly to NA in R:

```
evalCpp("NAN + 1")
evalCpp("NAN - 1")
evalCpp("NAN / 1")
evalCpp("NAN * 1")
```

Vectors

To set a missing value in a vector, you need to use a missing value specific to the type of vector. Unfortunately these are not named terribly consistently:

```
// [[Rcpp::export]]
List missing_sampler() {
  return(List::create(
    NumericVector::create(NA_REAL),
    IntegerVector::create(NA_INTEGER),
    LogicalVector::create(NA_LOGICAL),
    CharacterVector::create(NA_STRING)));
}

/*** R
  str(missing_sampler())
*/
```

To check if a value in a vector is missing, use the class method `is_na`:

```
// [[Rcpp::export]]
LogicalVector is_na2(NumericVector x) {
  int n = x.size();
  LogicalVector out(n);

  for (int i = 0; i < n; ++i) {
```

```
    out[i] = NumericVector::is_na(x[i]);
  }
  return out;
}

/*** R
  is_na2(c(NA, 5.4, 3.2, NA))
*/
```

Another alternative is the similarly named sugar function `is_na`: it takes a vector and returns a logical vector.

Exercises

- Rewrite any of the functions from the first exercises to correctly deal with missing values. If `na.rm` is true, ignore the missing values. If `na.rm` is false, return missing values the first time a missing value is encountered. Some functions you can practice with are: `min`, `max`, `range`, `mean`, `var`
- `cumsum` and `diff` diff need slightly more complicated behaviour for missing values.

The STL

The real strength of C++ shows itself when you need to implement more complex algorithms. The standard template library (STL) provides set of extremely useful data structures and algorithms. This section will explain the most important algorithms and data structures and point you in the right direction to learn more. We can't teach you everything you need to know about the STL, but hopefully the examples will at least show you the power of the STL, and persuade that it's useful to learn more.

If you need an algorithm or data strucutre that isn't implemented in STL, a good place to look is boost. Installing boost on to your computer is beyond the scope of this chapter, but once you have it installed, you can use `boost` data structures and algorithms by including the appropriate header file with (e.g.) `#include <boost/array.h>`.

Using iterators

Iterators are used extensively in the STL: many functions either accept or return iterators. They are the next step up from basic loops, abstracting away the details of the underlying data structure. Iterators have three main operators: they can be advanced with `++`, dereferenced (to get the value they refer to) with

`*` and compared using `==`. For example we could re-write our sum function using iterators:

```
// [[Rcpp::export]]
double sum3(NumericVector x) {
  double total = 0;

  for(NumericVector::iterator it = x.begin(); it != x.end(); ++it) {
    total += *it;
  }
  return total;
}
```

The main changes are in the for loop:

- we start at `x.begin()` and loop until we get to `x.end()`. A small optimisiation is to store the value of the end iterator so we don't need to look it up each time. This only saves about 2 ns per iteration, so it's only important when the calculations in the loop are very simple.
- instead of indexing into x, we use the dereference operator to get its current value: `*it`.
- notice the type of the iterator: `NumericVector::iterator`. Each vector type has its own iterator type: `LogicalVector::iterator`, `CharacterVector::iterator` etc.

Iterators also allow us to use the C++ equivalents of the apply family of functions. For example, we could again rewrite `sum` to use the `accumulate` function, which takes an starting and ending iterator and adds all the values in between. The third argument to accumulate gives the initial value: it's particularly important because this also determines the data type that accumulate uses (here we use `0.0` and not `0` so that accumulate uses a `double`, not an `int`.). To use `accumulate` we need to include the `<numeric>` header.

```
#include <numeric>

// [[Rcpp::export]]
double sum4(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 0.0);
}
```

`accumulate` (along with the other functions in `<numeric>`, `adjacent_difference`, `inner_product` and `partial_sum`) are not that important in Rcpp because Rcpp sugar provides equivalents.

Algorithms

The `<algorithm>` header provides a large number of algorithms that work with iterators. For example, we could write a basic Rcpp version of `findInterval` that takes two arguments a vector of values and a vector of breaks - the aim is to find the bin that each x falls into. This shows off a few more advanced iterator features. Read the code below and see if you can figure out how it works.

```
#include <algorithm>

// [[Rcpp::export]]
IntegerVector findInterval2(NumericVector x, NumericVector breaks) {
  IntegerVector out(x.size());

  NumericVector::iterator it, pos;
  IntegerVector::iterator out_it;

  for(it = x.begin(), out_it = out.begin(); it != x.end(); ++it, ++out_it) {
    pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
    *out_it = std::distance(pos, breaks.begin());
  }

  return out;
}
```

The key points are:

- We step through two iterators (input and output) simultaneously.
- We can assign into an dereferenced iterator (`out_it`) to change the values in `out`.
- `upper_bound` returns an iterator. If we wanted the value of the `upper_bound` we could dereference it; to figure out its location, we use the `distance` function.
- Small note: if we want this function to be as fast as `findInterval` in R (which uses hand-written C code), we need to cache the calls to `.begin()` and `.end()`. This is easy, but it distracts from this example so it has been omitted. Making this change yields a function that's slightly faster than R's `findInterval` function, but is about 1/10 of the code.

It's generally better to use algorithms from the STL than hand rolled loops. In "Effective STL", Scott Meyer gives three reasons: efficiency, correctness and maintainability. Algorithms from the STL are written by C++ experts to be extremely efficient, and they have been around for a long time so they are well

tested. Using standard algorithms also makes the intent of your code more clear, helping to make it more readable and more maintainable.

A good reference guide for algorithms is http://www.cplusplus.com/reference/algorithm/

Data structures

The STL provides a large set of data structures: `array`, `bitset`, `list`, `forward_list`, `map`, `multimap`, `multiset`, `priority_queue`, `queue`, `dequeue`, `set`, `stack`, `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`, and `vector`. The most important of these datastructures are the `vector`, the `unordered_set`, and the `unordered_map`. We'll focus on these three in this section, but using the others is similar: they just have different performance tradeoffs. For example, the `deque` (pronounced "deck") has a very similar interface to vectors but a different underlying implementation that has different performance trade-offs. You may want to try them for your problem. A good reference for STL data structures is http://www.cplusplus.com/reference/stl/ - I recommend you keep it open while working with the STL.

Rcpp knows how to convert from many STL data structures to their R equivalents, so you can return them from your functions without explicitly converting to R data structures.

Vectors

A stl vector is very similar to an R vector, except that it expands efficiently. This makes vectors appropriate to use when you don't know in advance how big the output will be. Vectors are templated, which means that you need to specify the type of object the vector will contain when you create it: `vector<int>`, `vector<bool>`, `vector<double>`, `vector<std::string>`. You can access individual elements of a vector using the standard `[]` notation, and you can add a new element to the end of the vector using `.push_back()`. If you have some idea in advance how big the vector will be, you can use `.reserve()` to allocate sufficient storage.

The following code implements run length encoding (`rle`). It produces two vectors of output: a vector of values, and a vector `lengths` giving how many times each element is repeated. It works by looping through the input vector `x` comparing each value to the previous: if it's the same, then it increments the last value in `lengths`; if it's different, it adds the value to the end of `values`, and sets the corresponding length to 1.

```
// [[Rcpp::export]]
List rle2(NumericVector x) {
  std::vector<int> lengths;
```

```
  std::vector<double> values;

  // Initialise first value
  int i = 0;
  double prev = x[0];
  values.push_back(prev);
  lengths.push_back(1);

  for(NumericVector::iterator it = x.begin() + 1; it != x.end(); ++it) {
    if (prev == *it) {
      lengths[i]++;
    } else {
      values.push_back(*it);
      lengths.push_back(1);

      i++;
      prev = *it;
    }
  }

  return List::create(_["lengths"] = lengths, _["values"] = values);
}
```

(An alternative implementation would be to replace `i` with the iterator `lengths.rbegin()` which always points to the last element of the vector - you might want to try implementing that yourself.)

Other methods of a vector are described at http://www.cplusplus.com/reference/vector/vector/.

Sets

Sets maintain a unique set of values, and can efficiently tell if you've seen a value before. They are useful for problems that involve duplicates or unique values (like `unique`, `duplicated`, or `in`). C++ provides both ordered (`std::set`) and unordered sets (`std::tr1::unorded_set`), depending on whether or not order matters for you. Unordered sets tend to be much faster (because they use a hash table internally rather than a tree), so even if you need an ordered set, you should consider using an unordered set and then sorting the output. Like vectors, sets are templated, so you need to request the appropriate type of set for your purpose: `unordered_set<int>`, `unordered_set<bool>`, etc.

http://www.cplusplus.com/reference/set/set/ and http://www.cplusplus.com/reference/unordered_set/unord provide complete documentation for sets structures.

The following function uses an unordered set to implement an equivalent to `duplicated` for integer vectors. Note the use of `seen.insert(x[i]).second` -

`insert` returns a pair, the `first` value is an iterator that points to element and the `second` value is a boolean that's true if the value was a new addition to the set.

```
// [[Rcpp::export]]
LogicalVector duplicated(IntegerVector x) {
  std::tr1::unordered_set<int> seen;
  int n = x.size();
  LogicalVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = seen.insert(x[i]).second;
  }

  return out;
}
```

Map

A map is similar to a set, but instead of storing presence or absence, it can store additional data. It's useful for functions like `table` or `match` that need to look up a value. As with sets, there are ordered (`std::map`) and unordered (`std::tr1::unorded_map`) versions, but if output order matters it's usually faster to use an unordered map and sort the results.

Since maps have a value and a key, you need to specify both when initialising a map: `map<double, int>`, `unordered_map<int, double>`, and so on.

XXX: Insert example implementation of match when `String` complete

Exercises

To practive using the STL algorithms and data structures, implement the following using R functions in C++, using the hints provided:

- `median.default` using `partial_sort`
- `%in%` using `unordered_set` and the `find` or `count` methods
- `unique` using an `unordered_set` (challenge: do it in one line!)
- `min` using `std::min`, or `max` using `std::max`
- `which.min` using `min_element`, or `which.max` using `max_element`
- `setdiff`, `union` and `intersect` using sorted ranges and `set_union`, `set_intersection` and `set_difference`

Case studies

The following case studies illustrate some real life uses of C++ to replace slow R code.

Gibbs sampler

The following case study updates an example blogged about by Dirk Eddelbuettel, illustrating the conversion of a gibbs sampler in R to C++. The R and C++ code shown below is very similar (it only took a few minutes to convert the R version to the C++ version), but runs about 20 times faster on my computer. Dirk's blog post also shows another way to make it even faster: using the faster (but presumably less numerically accurate) random number generator functions in GSL (easily accessible from R through RcppGSL) can eke out another 2-3x speed improvement.

The R code is as follows:

```
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0

  for (i in 1:N) {
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

This is straightforward to convert to C++. We:

- add type declarations to all variables
- use `(` instead of `[` to index into the matrix
- subscript the results of `rgamma` and `rnorm` to convert from a vector into a scalar

```
// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;
```

```
  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3.0, y * y + 4)[0];
      y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }

  return(mat);
}
```

Benchmarking the two implementations yields:

```
library(microbenchmark)
microbenchmark(
  gibbs_r(100, 10),
  gibbs_cpp(100, 10)
)
```

R vectorisation vs. C++ vectorisation

This example is adapted from Rcpp is smoking fast for agent-based models in data frames. The challenge is to predict a model response from three inputs. The basic R version looks like:

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

We want to be able to apply this function to many inputs, so we might write a vectorised version using a for loop.

```
vacc1 <- function(age, female, ily) {
  n <- length(age)
  out <- numeric(n)
  for (i in seq_len(n)) {
    out[i] <- vacc1a(age[i], female[i], ily[i])
  }
```

```
  out
}
```

If you're familiar with R, you'll have a gut feeling that this will be slow, and indeed it is. There are two ways we could attack this problem. If you have a good R vocabulary, you might immediately see how to vectorise the function (using `ifelse`, `pmin` and `pmax`). Alternatively, we could rewrite `vacc1a` and `vacc1` in C++, using our knowledge that loops and function calls have much lower overhead in C++.

Either approach is fairly straighforward. In R:

```
vacc2 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

(If you've worked R a lot you might recongise some potential bottlenecks in this code: `ifelse`, `pmin`, and `pmax` are known to be slow, and could be replaced with `p + 0.75 + 0.5 * female`, `p[p < 0] <- 0`, `p[p > 1] <- 1`. You might want to try timing that function yourself, but since it relies on rather esoteric knowledge I haven't included it in this case study.)

Or in C++:

```
double vacc3a(double age, bool female, bool ily){
  double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
  p = p * (female ? 1.25 : 0.75);
  p = std::max(p, 0.0);
  p = std::min(p, 1.0);
  return p;
}

// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female, LogicalVector ily) {
  int n = age.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = vacc3a(age[i], female[i], ily[i]);
  }

  return out;
}
```

We next generate some sample data, and check that all three versions return the same values:

```
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)

stopifnot(
  all.equal(vacc1(age, female, ily), vacc2(age, female, ily)),
  all.equal(vacc1(age, female, ily), vacc3(age, female, ily))
)
```

The original blog post forgot to do this, and hence introduced a bug in the C++ version: it used `0.004` instead of `0.04`. Finally, we can benchmark our three approaches:

```
microbenchmark(
  vacc1(age, female, ily),
  vacc2(age, female, ily),
  vacc3(age, female, ily)
```

Not surprisingly, our original approach with loops is very slow. Vectorising in R gives a huge speedup, and we can eke out even more performance (~10x) with the C++ loop. I was a little surprised that the C++ was so much faster, but it is because the R version has to create 11 vectors to store intermediate results, where the C++ code only needs to create 1.

Using Rcpp in a Package

The same C++ code that is used with `sourceCpp` can also be bundled into a package. There are several benefits of moving code from a standalone C++ source file to a package:

1. Your code can be made available to users without C++ development tools (at least on Windows or Mac OS X where binary packages are common)
2. Multiple source files and their dependencies are handled automatically by the R package build system
3. Packages provide additional infrastructure for testing, documentation and consistency

To generate a new Rcpp package that includes a simple hello, world function you can use the `Rcpp.package.skeleton` function as follows:

```
> Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

To generate a package based on C++ files that you've been using with `sourceCpp` you can use the `cpp_files` parameter:

```
> Rcpp.package.skeleton("NewPackage", example_code = FALSE,
                        cpp_files = c("convolve.cpp"))
```

To add `Rcpp` to an existing package, your put your C++ files in the `src/` directory and modify/create the following configuration files:

- In `DESCRIPTION` add

  ```
  Depends: Rcpp (>= 0.10.1)
  LinkingTo: Rcpp
  ```

- Make sure your `NAMESPACE` includes:

  ```
  useDynLib(mypackage)
  ```

- You need `src/Makevars` which contains:

  ```
  PKG_LIBS = `$(R_HOME)/bin/Rscript -e "Rcpp:::LdFlags()"`
  ```

And `src/Makevars.win` that contains:

```
PKG_LIBS = $(shell "${R_HOME}/bin${R_ARCH_BIN}/Rscript.exe" -e "Rcpp:::LdFlags()")
```

For more details see the Writing a package that uses Rcpp vignette.

If your packages uses the `Rcpp::export` attribute then one additional step in the package build process is requried. The `compileAttributes` function scans the source files within a package for `Rcpp::export` attributes and generates the code required to export the functions to R.

You should re-run `compileAttributes` whenever functions are added, removed, or have their signatures changed. Note that if you build your package using RStudio or `devtools` then this step occurs automatically.

Learning more

More Rcpp

This chapter has only touched on a small part of Rcpp, giving you the basic tools to rewrite poorly performing R code in C++. Rcpp has many other capabilities that make it easy to interface R to existing C++ code, including:

- Additional features of attributes including specifying default arguments, linking in external C++ dependencies, and exporting C++ interfaces from packages. These features and more are covered in the Rcpp attributes vignette.
- Automatically creating wrappers between C++ data structures and R data structures, including mapping C++ classes to reference classes. A good introduction to this topic is the vignette of Rcpp modules
- The Rcpp quick reference guide contains a useful summary of Rcpp classes and common programming idioms.

I strongly recommend keeping an eye on the Rcpp homepage and signing up for the Rcpp mailing list. Rcpp is still under active development, and is getting better with every release.

More C++

Writing performant code may also require you to rethink your basic approach: a solid understand of basic data structures and algorithms is very helpful here. That's beyond the scope of this book, but I'd suggest the "algorithm design manual" or MIT's Introduction to Algorithms.

Other resources I've found helpful in learning C++ are:

- Effective C++ and Effective STL by Scott Meyers.
- C++ Annotations, aimed at" knowledgeable users of C (or any other language using a C-like grammar, like Perl or Java) who would like to know more about, or make the transition to, C++"
- Algorithm Libraries, which provides a more technical, but still precise, description of important STL concepts. (Follow the links under notes)

Acknowledgements

I'd like to thank the Rcpp-mailing list for many helpful conversations, particularly Romain Francois and Dirk Eddelbuettel who have not only provided detailed answers to many of my questions, but have been incredibly responsive at improving Rcpp. This chapter would not have been possible without JJ Allaire; he encouraged me to learn C++ and then answered many of my dumb questions along the way.