Advanced R

Hadley Wickham

To Mina, the best book writing companion.

Contents

Part I

Metaprogramming

Introduction

One of the most intriguing things about R is its ability to do **metaprogramming**. This is the idea that code is data, and can be inspected and modified programmatically. This is a powerful idea; one that deeply influences much R code. At the most basic level, it allows you to do things like write `library(purrr)` instead of `library("purrr")` and enable `plot(x, sin(x))` to automatically label the axes with `x` and `sin(x)`. At a deeper level, it allows you to do things like use `y ~ x1 + x2` to represent a model that predicts the value of `y` from `x1` and `x2`, to translate `subset(df, x == y)` into `df[df$x == df$y, , drop = FALSE]`, and to use `dplyr::filter(db, is.na(x))` to generate the SQL `WHERE x IS NULL` when `db` is a remote database table.

Closely related to metaprogramming is **non-standard evaluation**, NSE for short. This term, which is commonly used to describe the behaviour of R functions, is problematic in two ways. Firstly, NSE is actually a property of the argument (or arguments) of a function, so talking about NSE functions is a little sloppy. Secondly, it's confusing to define something by what it's not (standard), so in this book I'll introduce more precise vocabulary.

Specifically, this book focuses on tidy evaluation (sometimes called tidy eval for short). Tidy evaluation is implemented in the rlang package (Henry and Wickham 2018), and I'll use rlang extensively in these chapters. This will allow you to focus on the big ideas, without being distracted by the quirks of implementation that arise from R's history. After I introduce each big idea with rlang, I'll then circle back to talk about how those ideas are expressed in base R. This approach may seem backward to some, but it's like learning how to drive using an automatic transmission rather than a stick shift: it allows you to focus on the big picture before having to learn the details. This book focusses on the theoretical side of tidy evaluation, so you can fully understand how it works from the ground up. If you are looking for a more practical introduction, I recommend the "tidy evaluation book", https://tidyeval.tidyverse.org[1].

You'll learn about metaprogramming and tidy evaluation in the following five

---

[1] As I write this chapter, the tidy evaluation book is still a work-in-progress, but by the time you read this it will hopefully be finished.

chapters:

- In **Big picture**, Chapter 1, you'll get a sense of the whole metaprogramming story, briefly learning about each major component and how they fit together to form a cohesive whole.

- In **Expressions**, Chapter 2, you'll learn that all R code can be described as a tree. You'll learn how to visualise these trees, how the rules of R's grammar convert linear sequences of characters into these trees, and how to use recursive functions to work with code trees.

- In **Quasiquotation**, Chapter 3, you'll learn to use tools from rlang to capture ("quote") unevaluated function arguments. You'll also learn about quasiquotation, which provides a set of techniques to "unquote" input to make it possible to easily generate new trees from code fragments.

- In **Evaluation**, Chapter 4, you'll learn how to evaluate captured code. Here you'll learn about an important data structure, the **quosure**, which ensures correct evaluation by capturing both the code to evaluate, and the environment in which to evaluate it. This chapter will show you how to put all the pieces together to understand how NSE works in base R, and how to write functions that work like `subset()`.

- Finally, in **Translating R code**, Chapter 5, you'll see how to combine first-class environments, lexical scoping, and metaprogramming to translate R code into other languages, namely HTML and LaTeX.

Chapter 1

Big picture

1.1 Introduction

Metaprogramming is the hardest topic in this book because it brings together many formerly unrelated topics and forces you grapple with issues that you probably haven't thought about before. You'll also need to learn a lot of new vocabulary, and at first it will seem like every new term is defined by three other terms that you haven't heard of. Even if you're an experienced programmer in another language, your existing skills are unlikely to be much help as few modern popular languages expose the level of metaprogramming that R provides. So don't be surprised if you're frustrated or confused at first; this is a natural part of the process that happens to everyone!

But I think it's easier to learn metaprogramming now than ever before. Over the last few years, the theory and practice have matured substantially, providing a strong foundation paired with tools that allow you to solve common problems. In this chapter, you'll get the big picture of all the main pieces and how they fit together.

Outline

Each section in this chapter introduces one big new idea:

- Section 1.2: Code is data and captured code is called an expression.

- Section 1.3: Code has a tree-like structure called an abstract syntax tree.

- Section 1.4: Code can create new expressions programmatically.

- Section 1.5: To "execute" code, you evaluate an expression in an environment.

- Section 1.6: You can customise evaluation by supplying custom functions in a new environment.

- Section 1.7: You can also customise evaluation by supplying a data mask, which blurs the line between environments and data frames.

- Section 1.8: All this is made simpler (and more correct) with a new data structure called the quosure.

Prerequisites

This chapter introduces the big ideas using rlang; you'll learn the base equivalents in later chapters. We'll also use the lobstr package to explore the tree structure of code.

```
library(rlang)
library(lobstr)
```

Make sure that you're also familiar with the environment (Section **??**) and data frame (Section **??**) data structures.

1.2   Code is data

The first big idea is that code is data: you can capture code and compute on it like any other type of data. The first way you can to capture code is with `rlang::expr()`. You can think of `expr()` as returning exactly what you pass in:

```
expr(mean(x, na.rm = TRUE))
#> mean(x, na.rm = TRUE)
expr(10 + 100 + 1000)
#> 10 + 100 + 1000
```

More formally, captured code is called an **expression**. An expression isn't a single type of object, but is a collective term for any of four types (call, symbol, constant, or pairlist), which you'll learn more about in Chapter 2.

`expr()` lets you capture code that you've typed. You need a different tool to capture code passed to a function because `expr()` doesn't work:

```
capture_it <- function(x) {
  expr(x)
}
capture_it(a + b + c)
#> x
```

Here you need to use a function specifically designed to capture user input in a function argument: `enexpr()`. Think of the "en" like in "enrich": `enexpr()` takes a lazily evaluted argument and turns it into an expression:

```
capture_it <- function(x) {
  enexpr(x)
}
capture_it(a + b + c)
#> a + b + c
```

Because `capture_it()` uses `enexpr()` we say that it automatically "quotes" its first argument. You'll learn more about this term in Section 3.2.1.

Once you have captured an expression, you can inspect and modify it. Complex expressions behave much like lists. That means you can modify them using `[[` and `$`:

```
f <- expr(f(x = 1, y = 2))

# Add a new argument
f$z <- 3
f
#> f(x = 1, y = 2, z = 3)

# Or remove an argument:
f[[2]] <- NULL
f
#> f(y = 2, z = 3)
```

The first element of the call is the function to be called, which means the first argument is in the second position. You'll learn the full details in Section 2.3.

1.3   Code is a tree

To do more complex manipulation with expressions, you need to fully understand their structure. Behind the scenes, almost every programming language represents code as a tree, often called the **abstract syntax tree**, or AST for short. R is unusual in that you can actually inspect and manipulate this tree.

A very convenient tool for understanding the tree-like structure is `lobstr::ast()`. Given some code, this function displays the underlying tree structure. Function calls form the branches of the tree, and are shown by rectangles. The leaves of the tree are symbols (like `a`) and constants (like `"b"`).

```
lobstr::ast(f(a, "b"))
#>  f
#>   a
#>   "b"
```

Nested function calls create more deeply branching trees:

```
lobstr::ast(f1(f2(a, b), f3(1, f4(2))))
#>  f1
#>   f2
#>    a
#>    b
#>   f3
#>    1
#>      f4
#>       2
```

Because all function forms in can be written in prefix form (Section **??**), every R expression can be displayed in this way:

```
lobstr::ast(1 + 2 * 3)
#>   `+`
#>  1
#>     `*`
#>    2
#>    3
```

Displaying the AST in this way is a useful tool for exploring R's grammar, the topic of Section 2.4.

1.4   Code can generate code

As well as seeing the tree from code typed by a human, you can also use code to create new trees. There are two main tools: `call2()` and unquoting.

`rlang::call2()` constructs a function call from its components: the function to call, and the arguments to call it with.

```
call2("f", 1, 2, 3)
#> f(1, 2, 3)
call2("+", 1, call2("*", 2, 3))
#> 1 + 2 * 3
```

call2() is often convenient to program with, but is a bit clunky for interactive use. An alternative technique is to build complex code trees by combining simpler code trees with a template. `expr()` and `enexpr()` have built-in support for this idea via `!!` (pronounced bang-bang), the **unquote operator**.

The precise details are the topic of Section 3.4, but basically `!!x` inserts the code tree stored in `x` into the expression. This makes it easy to build complex trees from simple fragments:

```
xx <- expr(x + x)
yy <- expr(y + y)

expr(!!xx / !!yy)
#> (x + x)/(y + y)
```

Notice that the output preserves the operator precedence so we get `(x + x) / (y + y)` not `x + x / y + y` (i.e. `x + (x / y) + y`). This is important, particularly if you've been wondering if it wouldn't be easier to just paste strings together.

Unquoting gets even more useful when you wrap it up into a function, first using `enexpr()` to capture the user's expression, then `expr()` and `!!` to create an new expression using a template. The example below shows how you can generate an expression that computes the coefficient of variation:

```
cv <- function(var) {
  var <- enexpr(var)
  expr(sd(!!var) / mean(!!var))
}

cv(x)
#> sd(x)/mean(x)
cv(x + y)
#> sd(x + y)/mean(x + y)
```

(This isn't very useful here, but being able to create this sort of building block is very useful when solving more complex problems.)

Importantly, this works even when given weird variable names:

```
cv(`)`)
#> sd(`)`)/mean(`)`)
```

Dealing with weird names[1] is another good reason to avoid `paste()` when generating R code. You might think this is an esoteric concern, but not worrying

---

[1]More technically, these are called non-syntactic names and are the topic of Section **??**.

about it when generating SQL code in web applications led to SQL injection attacks that have collectively cost billions of dollars.

1.5   Evaluation runs code

Inspecting and modifying code gives you one set of powerful tools. You get another set of powerful tools when you **evaluate**, i.e. execute or run, an expression. Evaluating an expression requires an environment, which tells R what the symbols in the expression mean. You'll learn the details of evaluation in Chapter 4.

The primary tool for evaluating expressions is `base::eval()`, which takes an expression and an environment:

```
eval(expr(x + y), env(x = 1, y = 10))
#> [1] 11
eval(expr(x + y), env(x = 2, y = 100))
#> [1] 102
```

If you omit the environment, `eval` uses the current environment:

```
x <- 10
y <- 100
eval(expr(x + y))
#> [1] 110
```

One of the big advantages of evaluating code manually is that you can tweak the environment. There are two main reasons to do this:

- To temporarily override functions to implement a domain specific language.
- To add a data mask so you can to refer to variables in a data frame as if they are variables in an environment.

1.6   Customising evaluation with functions

The above example used an environment that bound `x` and `y` to vectors. It's less obvious that you also bind names to functions, allowing you to override the behaviour of existing functions. This is a big idea that we'll come back to in Chapter 5 where I explore generating HTML and LaTeX from R. The example below gives you a taste of the power. Here I evelute code in a special environment where `*` and `+` have been overridden to work with strings instead of numbers:

```r
string_math <- function(x) {
  e <- env(
    caller_env(),
    `+` = function(x, y) paste0(x, y),
    `*` = function(x, y) strrep(x, y)
  )

  eval(enexpr(x), e)
}

name <- "Hadley"
string_math("Hello " + name)
#> [1] "Hello Hadley"
string_math(("x" * 2 + "-y") * 3)
#> [1] "xx-yxx-yxx-y"
```

dplyr takes this idea to the extreme, running code in an environment that generates SQL for execution in a remote database:

```r
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

con <- DBI::dbConnect(RSQLite::SQLite(), filename = ":memory:")
mtcars_db <- copy_to(con, mtcars)

mtcars_db %>%
  filter(cyl > 2) %>%
  select(mpg:hp) %>%
  head(10) %>%
  show_query()
#> <SQL>
#> SELECT `mpg`, `cyl`, `disp`, `hp`
#> FROM `mtcars`
#> WHERE (`cyl` > 2.0)
```

```
#> LIMIT 10
```

```
DBI::dbDisconnect(con)
```

1.7   Customising evaluation with data

Rebinding functions is an extremely powerful technique, but it tends to require a lot of investment. A more immediately practical application is modifying evaluation to look for variables in a data frame instead of an environment. This idea powers the base `subset()` and `transform()` functions, as well as many tidyverse functions like `ggplot2::aes()` and `dplyr::mutate()`. It's possible to use `eval()` for this, but there are a few potential pitfalls (Section 4.6), so we'll switch to `rlang::eval_tidy()` instead.

As well as expression and environment, `eval_tidy()` also takes a **data mask**, which is typically a data frame:

```
df <- data.frame(x = 1:5, y = sample(5))
eval_tidy(expr(x + y), df)
#> [1] 2 6 5 9 8
```

Evaluating with a data mask is a useful technique for interactive analysis because it allows you to write `x + y` rather than `df$x + df$y`. However, that convenience comes at a cost: ambiguity. In Section 4.4 you'll learn how to deal ambiguity using special `.data` and `.env` pronouns.

We can wrap this pattern up into a function by using `enexpr()`. This gives us a function very similar to `base::with()`:

```
with2 <- function(df, expr) {
  eval_tidy(enexpr(expr), df)
}
```

```
with2(df, x + y)
#> [1] 2 6 5 9 8
```

Unfortunately, this function has a subtle bug and we need a new data structure to help deal with it.

1.8   Quosures

To make the problem more obvious, I'm going to modify `with2()`. The basic problem still occurs without this modification but it's much harder to see.

```
with2 <- function(df, expr) {
  a <- 1000
  eval_tidy(enexpr(expr), df)
}
```

We can see the problem when we use `with2()` to refer to a variable called `a`. We want the value of `a` to come from the binding we can see (10), not the binding internal to the function (1000):

```
df <- data.frame(x = 1:3)
a <- 10
with2(df, x + a)
#> [1] 1001 1002 1003
```

The problem arises because we need to evaluate the captured expression in the environment where it was written (where `a` is 10), not the environment inside of `with2()` (where `a` is 1000).

Fortunately we call solve this problem by using a new data structure: the **quosure** which bundles an expression with an environment. `eval_tidy()` knows how to work with quosures so all we need to do is switch out `enexpr()` for `enquo()`:

```
with2 <- function(df, expr) {
  a <- 1000
  eval_tidy(enquo(expr), df)
}

with2(df, x + a)
#> [1] 11 12 13
```

Whenever you use a data mask, you must always use `enquo()` instead of `enexpr()`. This is the topic of Chapter 4.

Chapter 2

Expressions

2.1 Introduction

To compute on the language, we first need to understand its structure. That requires some new vocabulary, some new tools, and some new ways of thinking about R code. The first of these is the distinction between an operation and its result. Take the following code, which multiplies a variable `x` by 10 and saves the result to a new variable called `y`. It doesn't work because we haven't defined a variable called `x`:

```
y <- x * 10
#> Error in eval(expr, envir, enclos): object 'x' not found
```

It would be nice if we could capture the intent of the code without executing it. In other words, how can we separate our description of the action from the action itself?

One way is to use `rlang::expr()`:

```
z <- rlang::expr(y <- x * 10)
z
#> y <- x * 10
```

`expr()` returns an expression, an object that captures the structure of the code without evaluating it (i.e. running it). If you have an expression, you can evaluate it with `base::eval()`:

```
x <- 4
eval(z)
y
#> [1] 40
```

The focus of this chapter is the data structures that underlie expressions. Mastering this knowledge will allow you to inspect and modify captured code, and to generate code with code. We'll come back to `expr()` in Section 3, and to `eval()` in Chapter 4.

Outline

- Section 2.2 introduces the idea of the abstract syntax tree (AST), and reveals the tree like structure that underlies all R code.

- Section 2.3 dives into the details of the data structures that underpin the AST: constants, symbols, and calls, which are collectively known as expressions.

- Section 2.4 covers parsing, the act of converting the linear sequence of character in code into the AST, and uses that idea to explore some details of R's grammar.

- Section 2.5 shows you how you can use recursive functions to "compute on the language", writing functions that compute with expressions.

- Section 2.6 circles back to three more specialised data structures: pairlists, missing arguments, and expression vectors.

Prerequisites

Make sure you've read the metaprogramming overview in Chapter 1 to get a broad overview of the motivation and the basic vocabulary. You'll also need the rlang (https://rlang.r-lib.org) package to capture and compute on expressions, and the lobstr (https://lobstr.r-lib.org) package to visualise them.

```
library(rlang)
library(lobstr)
```

2.2   Abstract syntax trees

Expressions are also called **abstract syntax trees** (ASTs) because the structure of code is hierarchical and can be naturally represented as a tree. Understanding this tree structure is crucial for inspecting and modifying expressions (i.e. metaprogramming).

2.2.1 Drawing

We'll start by introducing some conventions for drawing ASTs, beginning with a simple call that shows their main components: `f(x, "y", 1)`. I'll draw trees in two ways[1]:

- By "hand" (i.e. with OmniGraffle):
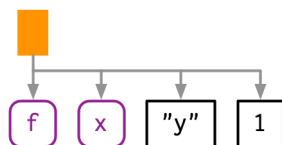
- With `lobstr::ast()`:

```
lobstr::ast(f(x, "y", 1))
#>  f
#>  x
#>  "y"
#>  1
```

Both approaches share conventions as much as possible:

- The leaves of the tree are either symbols, like `f` and `x`, or constants, like `1` or `"y"`. Symbols are drawn in purple and have rounded corners. Constants have black borders and square corners. Strings and symbols are easily confused, so strings are always surrounded in quotes.

- The branches of the tree are call objects, which represent function calls, and are drawn as orange rectangles. The first child (`f`) is the function that gets called; the second and subsequent children (`x`, `"y"`, and `1`) are the arguments to that function.

Colours will be shown when *you* call `ast()`, but do not appear in the book for complicated technical reasons.

The above example only contained one function call, making for a very shallow tree. Most expressions will contain considerably more calls, creating trees with multiple levels. For example, consider the AST for `f(g(1, 2), h(3, 4, i()))`:

---

[1]For more complex code, you can also use RStudio's tree viewer which doesn't obey quite the same graphical conventions, but allows you to interactively explore large ASTs. Try it out with `View(expr(f(x, "y", 1)))`.
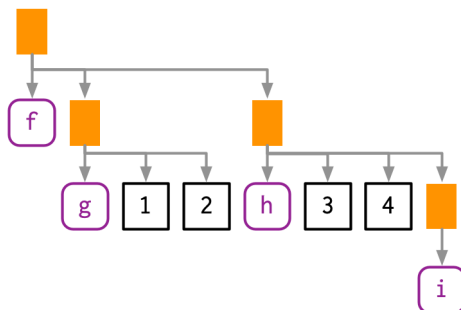
```
lobstr::ast(f(g(1, 2), h(3, 4, i()))))
#>  f
#>    g
#>    1
#>    2
#>    h
#>    3
#>    4
#>      i
```

You can read the hand-drawn diagrams from left-to-right (ignoring vertical position), and the lobstr-drawn diagrams from top-to-bottom (ignoring horizontal position). The depth within the tree is determined by the nesting of function calls. This also determines evaluation order, as evaluation generally proceeds from deepest-to-shallowest, but this is not guaranteed because of lazy evaluation (Section **??**). Also note the appearance of `i()`, a function call with no arguments; it's a branch with a single (symbol) leaf.

2.2.2   Non-code components

You might have wondered what makes these *abstract* syntax trees. They are abstract because they only capture important structural details of the code, not whitespace or comments:

```
ast(
  f(x,  y)  # important!
)
#>  f
#>  x
#>  y
```

There's only one place where whitespace affects the AST:

```
lobstr::ast(y <- x)
#>   `<-`
#>  y
#>  x
lobstr::ast(y < -x)
#>   `<`
#>  y
#>    `-`
#>     x
```

2.2.3   Infix calls

Every call in R can be written in tree form because any call can be written in prefix form (Section **??**). Take `y <- x * 10` again: what are the functions that are being called? It is not as easy to spot as `f(x, 1)` because this expression contains two infix calls: `<-` and `*`. That means that these two lines of code are equivalent:

```
y <- x * 10
`<-`(y, `*`(x, 10))
```

And they both have this AST[2]:
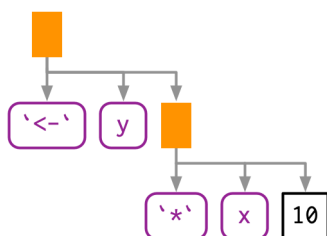

```
lobstr::ast(y <- x * 10)
#>   `<-`
#>  y
#>    `*`
#>     x
#>     10
```

There really is no difference between the ASTs, and if you generate an expression with prefix calls, R will still print it in infix form:

---

[2]The names of non-prefix functions are non-syntactic so I surround them with `` `` ``, as in Section **??**.

```
expr(`<-`(y, `*`(x, 10)))
#> y <- x * 10
```

The order in which infix operators are applied is governed by a set of rules called operator precedence, and we'll `lobstr::ast()` to explore them in Section 2.4.1.

2.2.4   Exercises

1. Reconstruct the code represented by the trees below:

```
#>  f
#>    g
#>      h
#>    `+`
#>      `+`
#>      1
#>      2
#>    3
#>    `*`
#>      `(`
#>        `+`
#>        x
#>        y
#>    z
```

2. Draw the following trees by hand then check your answers with `lobstr::ast()`.

```
f(g(h(i(1, 2, 3))))
f(1, g(2, h(3, i()))))
f(g(1, 2), h(3, i(4, 5)))
```

3. What's happening with the ASTs below? (Hint: carefully read `?"^"`)

```
lobstr::ast(`x`  + `y`)
#>   `+`
#>   x
#>   y
lobstr::ast(x ** y)
#>   `^`
#>   x
```

```
#>   y
lobstr::ast(1 -> x)
#>   `<-`
#>   x
#>   1
```

4. What is special about the AST below? (Hint: re-read Section **??**)

```
lobstr::ast(function(x = 1, y = 2) {})
#>   `function`
#>     x = 1
#>     y = 2
#>     `{`
#>   <inline srcref>
```

5. What does the call tree of an `if` statement with multiple `else if` conditions look like? Why?

2.3 Expressions

Collectively, the data structures present in the AST are called expressions. An **expression** is any member of the set of base types created by parsing code: constant scalars, symbols, call objects, and pairlists. These are the data structures used to represent captured code from `expr()`, and `is_expression(expr(...))` is always true[3]. Constants, symbols and call objects are the most important, and are discussed below. Pairlists and empty symbols are more specialised and we'll come back to them in Sections 2.6.1 and Section 2.6.2.

NB: In base R documentation "expression" is used to mean two things. As well as the definition above, expression is also used to refer to the type of object returned by `expression()` and `parse()`, which are basically lists of expressions as defined above. In this book I'll call these **expression vectors**, and I'll come back to them in Section 2.6.3.

2.3.1 Constants

Scalar constants are the simplest component of the AST. More precisely, a **constant** is either `NULL` or a length-1 atomic vector (or scalar, Section 9.4.1) like `TRUE`, `1L`, `2.5` or `"x"`. You can test for a constant with `rlang::is_syntactic_literal()`.

---

[3]It is *possible* to insert any other base object into an expression, but this is unusual and only needed in rare circumstances. We'll come back to that idea in Section 3.4.7.

Constants are "self-quoting" in the sense that the expression used to represent a constant is the constant itself:

```
identical(expr(TRUE), TRUE)
#> [1] TRUE
identical(expr(1), 1)
#> [1] TRUE
identical(expr(2L), 2L)
#> [1] TRUE
identical(expr("x"), "x")
#> [1] TRUE
```

2.3.2   Symbols

A **symbol** represents the name of an object like x, mtcars, or mean. In base R, the terms symbol and name are used interchangeably (i.e. is.name() is identical to is.symbol()), but in this book I used symbol consistently because "name" has many other meanings.

You can create a symbol in two ways: by capturing code that references an object with expr(), or turning a string into a symbol with rlang::sym():

```
expr(x)
#> x
sym("x")
#> x
```

You can turn a symbol back into a string with as.character() or rlang::as_string(). as_string() has the advantage of clearly signalling that you'll get a character vector of length 1.

```
as_string(expr(x))
#> [1] "x"
```

You can recognise a symbol because it's printed without quotes, str() tells you that it's a symbol, and is.symbol() is TRUE:

```
str(expr(x))
#>  symbol x
is.symbol(expr(x))
#> [1] TRUE
```

The symbol type is not vectorised, i.e. a symbol is always length 1. If you want multiple symbols, you'll need to put them in a list, using (e.g.) rlang::syms().

2.3.3 Calls

A **call object** represents a captured function call. Call objects are a special type of list[4] where the first component specifies the function to call (usually a symbol), and the remaining elements are the arguments for that call. Call objects create branches in the AST, because calls can be nested inside other calls.

You can identify a call object when printed because it looks just like a function call. Confusingly `typeof()` and `str()` print "language"[5] for call objects, but `is.call()` returns `TRUE`:

```
lobstr::ast(read.table("important.csv", row.names = FALSE))
#>   read.table
#>   "important.csv"
#>   row.names = FALSE
x <- expr(read.table("important.csv", row.names = FALSE))

typeof(x)
#> [1] "language"
is.call(x)
#> [1] TRUE
```

2.3.3.1 Subsetting

Calls generally behave like lists, i.e. you can use standard subsetting tools. The first element of the call object is the function to call, which is a usually a symbol:

```
x[[1]]
#> read.table
is.symbol(x[[1]])
#> [1] TRUE
```

The remainder of the elements are the arguments:

```
as.list(x[-1])
#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

You can extract individual arguments with `[[` or, if named, `$`:

---

[4]More precisely, they're pairlists, Section 2.6.1, but this distinction rarely matters.
[5]Avoid `is.language()` which returns `TRUE` for symbols, calls, and expression vectors.

```
x[[2]]
#> [1] "important.csv"
x$row.names
#> [1] FALSE
```

You can determine the number of arguments in a call object by subtracting 1 from its length:

```
length(x) - 1
#> [1] 2
```

Extracting specific arguments from calls is challenging because of R's flexible rules for argument matching: it could potentially be in any location, with the full name, with an abbreviated name, or with no name. To work around this problem, you can use `rlang::call_standardise()` which standardises all arguments to use the full name:

```
rlang::call_standardise(x)
#> read.table(file = "important.csv", row.names = FALSE)
```

(NB: If the function uses `...` it's not possible to standardise all arguments.) Calls can be modified in the same way as lists:

```
x$header <- TRUE
x
#> read.table("important.csv", row.names = FALSE, header = TRUE)
```

2.3.3.2   Function position

The first element of the call object is the **function position**. This contains the function that will be called when the object is evaluated, and is usually a symbol[6]:

```
lobstr::ast(foo())
#>  foo
```

While R allows you to surround the name of the function with quotes, the parser converts it to a symbol:

---

[6]Peculiarly, it can also be a number, as in the expression `3()`. But this call will always fail to evaluate because a number is not a function.

```
lobstr::ast("foo"())
#>  foo
```

However, sometimes the function doesn't exist in the current environment and you need to do some computation to retrieve it: for example, if the function is in another package, is a method of an R6 object, or is created by a function factory. In this case, the function position will be occupied by another call:

```
lobstr::ast(pkg::foo(1))
#>    `::`
#>   pkg
#>   foo
#>  1
lobstr::ast(obj$foo(1))
#>    `$`
#>   obj
#>   foo
#>  1
lobstr::ast(foo(1)(2))
#>   foo
#>   1
#>  2
```
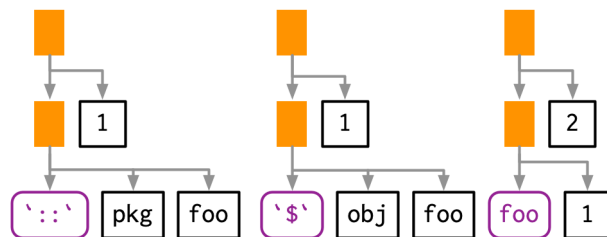

2.3.3.3 Constructing

You can construct a call object from its components using `rlang::call2()`. The first argument is the name of the function to call (either as a string, a symbol, or another call). The remaining arguments will be passed along to the call:

```
call2("mean", x = expr(x), na.rm = TRUE)
#> mean(x = x, na.rm = TRUE)
call2(expr(base::mean), x = expr(x), na.rm = TRUE)
#> base::mean(x = x, na.rm = TRUE)
```

Infix calls created in this way still print as usual.

```
call2("<-", expr(x), 10)
#> x <- 10
```

Using `call2()` to create complex expressions is a bit clunky. You'll learn another technique in Chapter 3.

2.3.4 Summary

The following table summarises the appearance of the different expression subtypes in `str()` and `typeof()`:

	`str()`	`typeof()`
Scalar constant	`logi/int/num/chr`	`logical/integer/double/character`
Symbol	`symbol`	`symbol`
Call object	`language`	`language`
Pairlist	Dotted pair list	`pairlist`
Expression vector	`expression()`	`expression`

Both base R and rlang provide functions for testing for each type of input, although the types covered are slightly different. You can easily tell them apart because all the base functions start with `is.` and the rlang functions start with `is_`.

	base	rlang
Scalar constant	—	`is_syntactic_literal()`
Symbol	`is.symbol()`	`is_symbol()`
Call object	`is.call()`	`is_call()`
Pairlist	`is.pairlist()`	`is_pairlist()`
Expression vector	`is.expression()`	—

2.3.5 Exercises

1. Which two of the six types of atomic vector can't appear in an expression? Why? Similarly, why can't you create an expression that contains an atomic vector of length greater than one?

2. What happens when you subset a call object to remove the first element? e.g. `expr(read.csv("foo.csv", header = TRUE))[-1]`. Why?

3. Describe the differences between the following call objects.

```
x <- 1:10

call2(median, x, na.rm = TRUE)
call2(expr(median), x, na.rm = TRUE)
call2(median, expr(x), na.rm = TRUE)
call2(expr(median), expr(x), na.rm = TRUE)
```

4. `rlang::call_standardise()` doesn't work so well for the following calls. Why? What makes `mean()` special?

```
call_standardise(quote(mean(1:10, na.rm = TRUE)))
#> mean(x = 1:10, na.rm = TRUE)
call_standardise(quote(mean(n = T, 1:10)))
#> mean(x = 1:10, n = T)
call_standardise(quote(mean(x = 1:10, , TRUE)))
#> mean(x = 1:10, , TRUE)
```

5. Why does this code not make sense?

```
x <- expr(foo(x = 1))
names(x) <- c("x", "y")
```

6. Construct the expression `if(x > 1) "a" else "b"` using multiple calls to `call2()`. How does the code structure reflect the structure of the AST?

2.4   Parsing and grammar

We've talked a lot about expressions and the AST, but not about how expressions are created from code that you type (like `"x + y"`). The process by which a computer language takes a string and constructs an expression is called **parsing**, and is governed by a set of rules known as a **grammar**. In this section, we'll use `lobstr::ast()` to explore some of the details of R's grammar, and then show how you can transform back and forth between expressions and strings.

2.4.1   Operator precedence

Infix functions introduce two sources of ambiguity[7]. The first source of ambiguity arises from infix functions: what does `1 + 2 * 3` yield? Do you get 9 (i.e. (1

---

[7]This ambiguity does not exist in languages with only prefix or postfix calls. It's interesting to compare a simple arithmetic operation in Lisp (prefix) and Forth (postfix). In Lisp you'd write `(* (+ 1 2) 3))`; this avoids ambiguity by requiring parentheses everywhere. In Forth, you'd write `1 2 + 3 *`; this doesn't require any parentheses, but does require more thought when reading.

+ 2) * 3), or 7 (i.e. 1 + (2 * 3))? In other words, which of the two possible
parse trees below does R use?
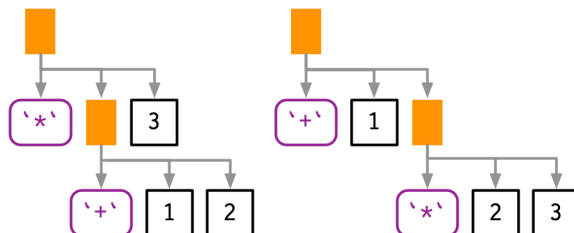
Programming languages use conventions called **operator precedence** to re-
solve this ambiguity. We can use `ast()` to see what R does:

```
lobstr::ast(1 + 2 * 3)
#>   `+`
#>   1
#>    `*`
#>     2
#>     3
```

Predicting the precedence of arithmetic operations is usually easy because it's
drilled into you in school and is consistent across the vast majority of program-
ming languages.

Predicting the precedence of other operators is harder. There's one particu-
larly surprising case in R: ! has a much lower precedence (i.e. it binds less tightly)
than you might expect. This allows you to write useful operations like:

```
lobstr::ast(!x %in% y)
#>   `!`
#>    `%in%`
#>     x
#>     y
```

R has over 30 infix operators divided into 18 precedence groups. While the
details are described in **?Syntax**, very few people have memorised the complete
ordering. If there's any confusion, use parentheses!

```
lobstr::ast((1 + 2) * 3)
#>   `*`
#>    `(`
#>     `+`
#>      1
```

```
#>     2
#>   3
```

Note the appearance of the parentheses in the AST as a call to the (function.

2.4.2 Associativity

The second source of ambiguity is introduced by repeated usage of the same infix function. For example, is `1 + 2 + 3` equivalent to `(1 + 2) + 3` or to `1 + (2 + 3)`? This normally doesn't matter because `x + (y + z) == (x + y) + z`, i.e. addition is associative, but is needed because some S3 classes define `+` in a non-associative way. For example, ggplot2 overloads `+` to build up a complex plot from simple pieces; this is non-associative because earlier layers are drawn underneath later layers (i.e. `geom_point() + geom_smooth()` does not yield the same plot as `geom_smooth() + geom_point()`).

In R, most operators are **left-associative**, i.e. the operations on the left are evaluated first:

```
lobstr::ast(1 + 2 + 3)
#>   `+`
#>    `+`
#>    1
#>    2
#>   3
```

There are two exceptions: exponentiation and assignment.

```
lobstr::ast(2^2^3)
#>   `^`
#>   2
#>    `^`
#>    2
#>    3
lobstr::ast(x <- y <- z)
#>   `<-`
#>   x
#>    `<-`
#>    y
#>    z
```

2.4.3 Parsing and deparsing

Most of the time you type code into the console, and R takes care of turning the characters you've typed into an AST. But occasionally you have code

stored in a string, and you want to parse it yourself. You can do so using
`rlang::parse_expr()`:

```
x1 <- "y <- x + 10"
x1
#> [1] "y <- x + 10"
is.call(x1)
#> [1] FALSE

x2 <- rlang::parse_expr(x1)
x2
#> y <- x + 10
is.call(x2)
#> [1] TRUE
```

`parse_expr()` always returns a single expression. If you have multiple expres-
sion separated by `;` or `\n`, you'll need to use `rlang::parse_exprs()`. It returns
a list of expressions:

```
x3 <- "a <- 1; a + 1"
rlang::parse_exprs(x3)
#> [[1]]
#> a <- 1
#>
#> [[2]]
#> a + 1
```

If you find yourself working with strings containing code very frequently, you
should reconsider your process. Read Chapter 3 and consider if you can instead
more safely generate expressions using quasiquotation.

The base equivalent to `parse_exprs()` is `parse()`. It is a little harder to use
because it's specialised for parsing R code stored in files. You need supply your
string to the `text` argument and it returns an expression vector (Section 2.6.3).
I recommend turning the output into a list:

```
as.list(parse(text = x1))
#> [[1]]
#> y <- x + 10
```

The inverse of parsing is **deparsing**: given an expression, you want the string
that would generate it. This happens automatically when you print an expression,
and you can get the string yourself with `rlang::expr_text()`:

```
z <- expr(y <- x + 10)
expr_text(z)
#> [1] "y <- x + 10"
```

Parsing and deparsing are not perfectly symmetric because parsing generates
an *abstract* syntax tree. This means we lose backticks around ordinary names,
comments, and whitespace:

```
cat(expr_text(expr({
  # This is a comment
  x <-                  `x` + 1
})))
#> {
#>     x <- x + 1
#> }
```

Be careful when using the base R equivalent, `deparse()`: it returns a character
vector with one element for each line. Whenever you use it, remember that the
length of the output might be greater than one, and plan accordingly.

2.4.4   Exercises

1. R uses parentheses in two slightly different ways as illustrated by these two
   calls:

   ```
   f((1))
   `(`(1 + 1)
   ```

   Compare and contrast the two uses by referencing the AST.

2. `=` can also be used in two ways. Construct a simple example that shows
   both uses.

3. Does `-2^2` yield 4 or -4? Why?

4. What does `!1 + !1` return? Why?

5. Why does `x1 <- x2 <- x3 <- 0` work? Describe the two reasons.

6. Compare the ASTs of `x + y %+% z` and `x ^ y %+% z`. What have you
   learned about the precedence of custom infix functions?

7. What happens if you call `parse_expr()` with a string that generates mul-
   tiple expressions? e.g. `parse_expr("x + 1; y + 1")`

8. What happens if you attempt to parse an invalid expression? e.g. `"a +"` or `"f())"`.

9. `deparse()` produces vectors when the input is long. For example, the following call produces a vector of length two:

```
expr <- expr(g(a + b + c + d + e + f + g + h + i + j + k + l +
  m + n + o + p + q + r + s + t + u + v + w + x + y + z))

deparse(expr)
```

   What does `expr_text()` do instead?

10. `pairwise.t.test()` assumes that `deparse()` always returns a length one character vector. Can you construct an input that violates this expectation? What happens?

2.5 Walking the AST with recursive functions

To conclude the chapter I'm going to use everything you've learned about ASTs to solve more complicated problems. The inspiration comes from the base codetools package, which provides two interesting functions:

- `findGlobals()` locates all global variables used by a function. This can be useful if you want to check that your function doesn't inadvertently rely on variables defined in their parent environment.

- `checkUsage()` checks for a range of common problems including unused local variables, unused parameters, and the use of partial argument matching.

Getting all of the details of these functions correct is fiddly, so we won't fully develop the ideas. Instead we'll focus on the big underlying idea: recursion on the AST. Recursive functions are a natural fit to tree-like data structures because a recursive function is made up of two parts that correspond to the two parts of the tree:

- The **recursive case** handles the nodes in the tree. Typically, you'll do something to each child of a node, usually calling the recursive function again, and then combine the results back together again. For expressions, you'll need to handle calls and pairlists (function arguments).

- The **base case** handles the leaves of the tree. The base cases ensure that the function eventually terminates, by solving the simplest cases directly. For expressions, you need to handle symbols and constants in the base case.

To make this pattern easier to see, we'll need two helper functions. First
we define `expr_type()` which will return "constant" for constant, "symbol" for
symbols, "call", for calls, "pairlist" for pairlists, and the "type" of anything else:

```
expr_type <- function(x) {
  if (rlang::is_syntactic_literal(x)) {
    "constant"
  } else if (is.symbol(x)) {
    "symbol"
  } else if (is.call(x)) {
    "call"
  } else if (is.pairlist(x)) {
    "pairlist"
  } else {
    typeof(x)
  }
}

expr_type(expr("a"))
#> [1] "constant"
expr_type(expr(x))
#> [1] "symbol"
expr_type(expr(f(1, 2)))
#> [1] "call"
```

We'll couple this with a wrapper around the switch function:

```
switch_expr <- function(x, ...) {
  switch(expr_type(x),
    ...,
    stop("Don't know how to handle type ", typeof(x), call. = FALSE)
  )
}
```

With these two functions in hand, we can write a basic template for any
function that walks the AST using `switch()` (Section **??**):

```
recurse_call <- function(x) {
  switch_expr(x,
    # Base cases
    symbol = ,
    constant = ,
```

```
    # Recursive cases
    call = ,
    pairlist =
  )
}
```

Typically, solving the base case is easy, so we'll do that first, then check the results. The recursive cases are a little more tricky, and will often require some functional programming.

2.5.1   Finding F and T

We'll start with a function that determines whether another function uses the logical abbreviations `T` and `F` because using them is often considered to be poor coding practice. Our goal is to return `TRUE` if the input contains a logical abbreviation, and `FALSE` otherwise.

Let's first the type of `T` vs. `TRUE`:

```
expr_type(expr(TRUE))
#> [1] "constant"
```

```
expr_type(expr(T))
#> [1] "symbol"
```

`TRUE` is parsed as a logical vector of length one, while `T` is parsed as a name. This tells us how to write our base cases for the recursive function: a constant is never a logical abbreviation, and a symbol is an abbreviation if it's "F" or "T":

```
logical_abbr_rec <- function(x) {
  switch_expr(x,
    constant = FALSE,
    symbol = as_string(x) %in% c("F", "T")
  )
}
```

```
logical_abbr_rec(expr(TRUE))
#> [1] FALSE
logical_abbr_rec(expr(T))
#> [1] TRUE
```

I've written `logical_abbr_rec()` function assuming that the input will be an expression as this will make the recursive operation simpler. However, when

writing a recursive function it's common to write a wrapper that provides defaults or makes the function a little easier to use. Here we'll typically make a wrapper that quotes its input (we'll learn more about that in the next chapter), so we don't need to use `expr()` every time.

```
logical_abbr <- function(x) {
  logical_abbr_rec(enexpr(x))
}

logical_abbr(T)
#> [1] TRUE
logical_abbr(FALSE)
#> [1] FALSE
```

Next we need to implement the recursive cases. Here we want to do the same thing for calls and for pairlists: recursively apply the function to each subcomponent, and return `TRUE` if any subcomponent contains a logical abbreviation. This is made easy by `purrr::some()`, which iterates over a list and returns `TRUE` if the predicate function is true for any element.

```
logical_abbr_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = FALSE,
    symbol = as_string(x) %in% c("F", "T"),

    # Recursive cases
    call = ,
    pairlist = purrr::some(x, logical_abbr_rec)
  )
}

logical_abbr(mean(x, na.rm = T))
#> [1] TRUE
logical_abbr(function(x, na.rm = T) FALSE)
#> [1] TRUE
```

2.5.2 Finding all variables created by assignment

`logical_abbr()` is relatively simple: it only returns a single `TRUE` or `FALSE`. The next task, listing all variables created by assignment, is a little more complicated. We'll start simply, and then make the function progressively more rigorous.

We start by looking at the AST for assignment:

```
ast(x <- 10)
#>   `<-`
#>   x
#>   10
```

Assignment is a call object where the first element is the symbol `<-`, the second is the name of variable, and the third is the value to be assigned.

Next, we need to decide what data structure we're going to use for the results. Here I think it will be easiest if we return a character vector. If we return symbols, we'll need to use a `list()` and that makes things a little more complicated.

With that in hand we can start by implementing the base cases and providing a helpful wrapper around the recursive function. Here the base cases are straightforward because we know that neither a symbol nor a constant represents assignment.

```
find_assign_rec <- function(x) {
  switch_expr(x,
    constant = ,
    symbol = character()
  )
}
find_assign <- function(x) find_assign_rec(enexpr(x))

find_assign("x")
#> character(0)
find_assign(x)
#> character(0)
```

Next we implement the recursive cases. This is made easier by a function that should exist in purrr, but currently doesn't. `flat_map_chr()` expects `.f` to return a character vector of arbitrary length, and flattens all results into a single character vector.

```
flat_map_chr <- function(.x, .f, ...) {
  purrr::flatten_chr(purrr::map(.x, .f, ...))
}

flat_map_chr(letters[1:3], ~ rep(., sample(3, 1)))
#> [1] "a" "b" "b" "b" "c" "c"
```

The recursive case for pairlists is straightforward: we iterate over every element of the pairlist (i.e. each function argument) and combine the results. The

case for calls is a little bit more complex - if this is a call to `<-` then we should
return the second element of the call:

```r
find_assign_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = ,
    symbol = character(),

    # Recursive cases
    pairlist = flat_map_chr(as.list(x), find_assign_rec),
    call = {
      if (is_call(x, "<-")) {
        as_string(x[[2]])
      } else {
        flat_map_chr(as.list(x), find_assign_rec)
      }
    }
  )
}

find_assign(a <- 1)
#> [1] "a"
find_assign({
  a <- 1
  {
    b <- 2
  }
})
#> [1] "a" "b"
```

Now we need to make our function more robust by coming up with examples
intended to break it. What happens when we assign to the same variable multiple
times?

```r
find_assign({
  a <- 1
  a <- 2
})
#> [1] "a" "a"
```

It's easiest to fix this at the level of the wrapper function:

```
find_assign <- function(x) unique(find_assign_rec(enexpr(x)))

find_assign({
  a <- 1
  a <- 2
})
#> [1] "a"
```

What happens if we have nested calls to <-? Currently we only return the
first. That's because when <- occurs we immediately terminate recursion.

```
find_assign({
  a <- b <- c <- 1
})
#> [1] "a"
```

Instead we need to take a more rigorous approach. I think it's best to keep
the recursive function focused on the tree structure, so I'm going to extract out
find_assign_call() into a separate function.

```
find_assign_call <- function(x) {
  if (is_call(x, "<-") && is_symbol(x[[2]])) {
    lhs <- as_string(x[[2]])
    children <- as.list(x)[-1]
  } else {
    lhs <- character()
    children <- as.list(x)
  }

  c(lhs, flat_map_chr(children, find_assign_rec))
}

find_assign_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = ,
    symbol = character(),

    # Recursive cases
    pairlist = flat_map_chr(x, find_assign_rec),
    call = find_assign_call(x)
  )
```

```
}
```

```
find_assign(a <- b <- c <- 1)
#> [1] "a" "b" "c"
find_assign(system.time(x <- print(y <- 5)))
#> [1] "x" "y"
```

The complete version of this function is quite complicated, it's important to remember we wrote it by working our way up by writing simple component parts.

2.5.3 Exercises

1. `logical_abbr()` returns `TRUE` for `T(1, 2, 3)`. How could you modify `logical_abbr_rec()` so that it ignores function calls that use `T` or `F`?

2. `logical_abbr()` works with expressions. It currently fails when you give it a function. Why not? How could you modify `logical_abbr()` to make it work? What components of a function will you need to recurse over?

   ```
   logical_abbr(function(x = TRUE) {
     g(x + T)
   })
   ```

3. Modify find assignment to also detect assignment using replacement functions, i.e. `names(x) <- y`.

4. Write a function that extracts all calls to a specified function.

2.6 Specialised data structures

There are two data structures and one special symbol that we need to cover for the sake of completeness. They are not usually important in practice.

2.6.1 Pairlists

Pairlists are a remnant of R's past and have been replaced by lists almost everywhere. The only place you are likely to see pairlists in R[8] is when working with calls to the "function" function, as the formal arguments to a function are stored in a pairlist:

---

[8]If you're working in C, you'll encounter pairlists more often. For example, call objects are also implemented using pairlists.

```
f <- expr(function(x, y = 10) x + y)

args <- f[[2]]
args
#> $x
#>
#>
#> $y
#> [1] 10
typeof(args)
#> [1] "pairlist"
```

Fortunately, whenever you encounter a pairlist, you can treat it just like a regular list:

```
pl <- pairlist(x = 1, y = 2)
length(pl)
#> [1] 2
pl$x
#> [1] 1
```

Behind the scenes pairlists are implemented using a different data structure, a linked list instead of an array. That makes subsetting a pairlist much slower than subsetting a list, but this has little practical impact.

2.6.2   Missing arguments

The special symbol that needs a little extra discussion is the empty symbol, which is used to represent missing arguments (not missing values!). You only need to care about the missing symbol if you're programmatically creating functions with missing arguments; we'll come back to that in Section 3.4.3.

You can make an empty symbol with `missing_arg()` (or `expr()`):

```
missing_arg()
typeof(missing_arg())
#> [1] "symbol"
```

An empty symbol doesn't print anything, so you can check if you have one with `rlang::is_missing()`:

```
is_missing(missing_arg())
#> [1] TRUE
```

You'll find them in the wild in function formals:

```
f <- expr(function(x, y = 10) x + y)
args <- f[[2]]
is_missing(args[[1]])
#> [1] TRUE
```

This is particularly important for ... which is always associated with an empty symbol:

```
f <- expr(function(...) list(...))
args <- f[[2]]
is_missing(args[[1]])
#> [1] TRUE
```

The empty symbol has a peculiar property: if you bind it to a variable, then access that variable, you will get an error:

```
m <- missing_arg()
m
#> Error in eval(expr, envir, enclos): argument "m" is missing, with no
#> default
```

But you won't if you store it inside another data structure!

```
ms <- list(missing_arg(), missing_arg())
ms[[1]]
```

If you need to preserve the missingness of a variable, `rlang::maybe_missing()` is often helpful. It allows you to refer to a potentially missing variable without triggering the error. See the documentation for use cases and more details.

2.6.3 Expression vectors

Finally, we need to briefly discuss the expression vector. Expression vectors are only produced by two base functions: `expression()` and `parse()`:

```
exp1 <- parse(text = c("
x <- 4
x
"))
exp2 <- expression(x <- 4, x)
```

```
typeof(exp1)
#> [1] "expression"
typeof(exp2)
#> [1] "expression"

exp1
#> expression(x <- 4, x)
exp2
#> expression(x <- 4, x)
```

Like calls and pairlists, expression vectors behave like lists:

```
length(exp1)
#> [1] 2
exp1[[1]]
#> x <- 4
```

Conceptually, an expression vector is just a list of expressions. The only difference is that calling `eval()` on an expression evaluates each individual expression. I don't believe this advantage merits introducing a new data structure, so instead of expression vectors I just use lists of expressions.

Chapter 3

Quasiquotation

3.1 Introduction

Now that you understand the tree structure of R code, it's time to return to one of the fundamental ideas that make `expr()` and `ast()` work: quotation. In tidy evaluation, all quoting functions are actually quasiquoting functions because they also support unquoting. Where quotation is the act of capturing an unevaluated expression, **unquotation** is the ability to selectively evaluate parts of an otherwise quoted expression. Together, this is called quasiquotation. Quasiquotation makes it easy to create functions that combine code written by the function's author with code written by the function's user. This helps to solve a wide variety of challenging problems.

Quasiquotation is one of the three pillars of tidy evaluation. You'll learn about the other two (quosures and the data mask) in Chapter 4. By itself, quasiquotation is most useful for programming, particularly for generating code. But when it's combined with the other techniques, tidy evaluation becomes a powerful tool for data analysis.

Outline

- Section 3.2 motivates the development of quasiquotation with a function, `cement()`, that works like `paste()` but automatically "quotes" its arguments so that you don't have to.

- Section 3.3 gives you the tools to quote expressions, whether they come from you or the user, or whether you use rlang or base R tools.

- Section 3.4 introduces the biggest difference between rlang quoting functions and base quoting function: unquoting with `!!` and `!!!`.

- Section 3.5 discusses the three main "non-quoting" techniques that base R functions uses to disable quoting behaviour.

- Section 3.6 explores another place that you can use `!!!`, functions that take `...`. It also introduces the special `:=` operator, which allows you to dynamically change argument names.

- Section 3.7 shows a few practical uses of quoting to solve problems that naturally require some code generation.

- Section 3.8 finishes up with a little history of quasiquotation for those who are interested.

Prerequisites

Make sure you've read the metaprogramming overview in Chapter 1 to get a broad overview of the motivation and the basic vocabulary, and that you're familiar with the tree structure of expressions as described in Section 2.3.

Code-wise, we'll mostly be using the tools from rlang (https://rlang.r-lib.org), but at the end of the chapter you'll also see some powerful applications in conjunction with purrr (https://purrr.tidyverse.org).

```
library(rlang)
library(purrr)
```

Related work

Quoting functions have deep connections to Lisp **macros**. But macros are usually run at compile-time, which doesn't exist in R, and they always input and output ASTs. See Lumley (2001) for one approach to implementing them in R. Quoting functions are more closely related to the more esoteric Lisp **fexprs** (http://en.wikipedia.org/wiki/Fexpr), functions where all arguments are quoted by default. These terms are useful to know when looking for related work in other programming languages.

3.2 Motivation

We'll start with a concrete example that helps motivate the need for unquoting, and hence quasiquotation. Imagine you're creating a lot of strings by joining together words:

```
paste("Good", "morning", "Hadley")
#> [1] "Good morning Hadley"
paste("Good", "afternoon", "Alice")
#> [1] "Good afternoon Alice"
```

You are sick and tired of writing all those quotes, and instead you just want to use bare words. To that end, you've written the following function. (Don't worry about the implementation for now; you'll learn about the pieces later.)

```
cement <- function(...) {
  args <- ensyms(...)
  paste(purrr::map(args, as_string), collapse = " ")
}

cement(Good, morning, Hadley)
#> [1] "Good morning Hadley"
cement(Good, afternoon, Alice)
#> [1] "Good afternoon Alice"
```

Formally, this function quotes all of its inputs. You can think of it as automatically putting quotation marks around each argument. That's not precisely true as the intermediate objects it generates are expressions, not strings, but it's a useful approximation, and the root meaning of the term "quote".

This function is nice because we no longer need to type quotation marks. The problem comes when we want to use variables. It's easy to use variables with `paste()`: just don't surround them with quotation marks.

```
name <- "Hadley"
time <- "morning"

paste("Good", time, name)
#> [1] "Good morning Hadley"
```

Obviously this doesn't work with `cement()` because every input is automatically quoted:

```
cement(Good, time, name)
#> [1] "Good time name"
```

We need some way to explicitly *unquote* the input to tell `cement()` to remove the automatic quote marks. Here we need `time` and `name` to be treated differently to `Good`. Quasiquotation gives us a standard tool to do so: `!!`, called "unquote",

and pronounced bang-bang. `!!` tells a quoting function to drop the implicit quotes:

```
cement(Good, !!time, !!name)
#> [1] "Good morning Hadley"
```

It's useful to compare `cement()` and `paste()` directly. `paste()` evaluates its arguments, so we must quote where needed; `cement()` quotes its arguments, so we must unquote where needed.

```
paste("Good", time, name)
cement(Good, !!time, !!name)
```

3.2.1   Vocabulary

The distinction between quoted and evaluated arguments is important:

- An **evaluated** argument obeys R's usual evaluation rules.

- A **quoted** argument is captured by the function, and is processed in some custom way.

`paste()` evaluates all its arguments; `cement()` quotes all its arguments.

If you're ever unsure about whether an argument is quoted or evaluated, try executing the code outside of the function. If it doesn't work or does something different, then that argument is quoted. For example, you can use this technique to determine that the first argument to `library()` is quoted:

```
# works
library(MASS)

# fails
MASS
#> Error in eval(expr, envir, enclos): object 'MASS' not found
```

Talking about whether an argument is quoted or evaluated is a more precise way of stating whether or not a function uses non-standard evaluation (NSE). I will sometimes use "quoting function" as short-hand for a "function that quotes one or more arguments", but generally, I'll talk about quoted arguments since that is the level at which the difference applies.

3.2.2   Exercises

1. For each function in the following base R code, identify which arguments are quoted and which are evaluated.

```
library(MASS)

mtcars2 <- subset(mtcars, cyl == 4)

with(mtcars2, sum(vs))
sum(mtcars2$am)

rm(mtcars2)
```

2. For each function in the following tidyverse code, identify which arguments are quoted and which are evaluated.

```
library(dplyr)
library(ggplot2)

by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))

ggplot(by_cyl, aes(cyl, mean)) + geom_point()
```

3.3 Quoting

The first part of quasiquotation is quotation: capturing an expression without evaluating it. We'll need a pair of functions because the expression can be supplied directly or indirectly, via lazily-evaluated function argument. I'll start with the rlang quoting functions, then circle back to those provided by base R.

3.3.1 Capturing expressions

There are four important quoting functions. For interactive exploration, the most important is `expr()`, which captures its argument exactly as provided:

```
expr(x + y)
#> x + y
expr(1 / 2 / 3)
#> 1/2/3
```

(Remember that white space and comments are not part of the expression, so will not be captured by a quoting function.)

`expr()` is great for interactive exploration, because it captures what you, the developer, typed. It's not so useful inside a function:

```
f1 <- function(x) expr(x)
f1(a + b + c)
#> x
```

We need another function to solve this problem: `enexpr()`. This captures
what the caller supplied to the function by looking at the internal promise object
that powers lazy evaluation (Section **??**).

```
f2 <- function(x) enexpr(x)
f2(a + b + c)
#> a + b + c
```

(It's called "en"-`expr()` by analogy to enrich. Enriching someone makes them
richer; `enexpr()`ing a argument makes it an expression.)

To capture all arguments in ..., use `enexprs()`.

```
f <- function(...) enexprs(...)
f(x = 1, y = 10 * z)
#> $x
#> [1] 1
#>
#> $y
#> 10 * z
```

Finally, `exprs()` is useful interactively to make a list of expressions:

```
exprs(x = x ^ 2, y = y ^ 3, z = z ^ 4)
# shorthand for
# list(x = expr(x ^ 2), y = expr(y ^ 3), z = expr(z ^ 4))
```

In short, use `enexpr()` and `enexprs()` to capture the expressions supplied as
arguments *by the user*. Use `expr()` and `exprs()` to capture expressions that *you*
supply.

3.3.2   Capturing symbols

Sometimes you only want to allow the user to specify a variable name, not an
arbitrary expression. In this case, you can use `ensym()` or `ensyms()`. These are
variants of `enexpr()` and `enexprs()` that check the captured expression is either
symbol or a string (which is converted to a symbol[1]). `ensym()` and `ensyms()`
throw an error if given anything else.

---

[1]This is for compatibility with base R, which allows you to provide a string instead of a
symbol in many places: `"x" <- 1`, `"foo"(x, y)`, `c("x" = 1)`.

```
f <- function(...) ensyms(...)
f(x)
#> [[1]]
#> x
f("x")
#> [[1]]
#> x
```

3.3.3 With base R

Each rlang function described above has an equivalent in base R. They primary difference is that the base equivalents do not support unquoting (which we'll talk about very soon). This make them quoting functions, rather than quasiquoting functions.

The base equivalent of `expr()` is `quote()`:

```
quote(x + y)
#> x + y
```

The base function closest to `enexpr()` is `substitute()`:

```
f3 <- function(x) substitute(x)
f3(x + y)
#> x + y
```

The base equivalent to `exprs()` is `alist()`:

```
alist(x = 1, y = x + 2)
#> $x
#> [1] 1
#>
#> $y
#> x + 2
```

The equivalent to `enexprs()` is an undocumented feature of `substitute()`[2]:

```
f <- function(...) as.list(substitute(...()))
f(x = 1, y = 10 * z)
#> $x
#> [1] 1
```

---

[2]Discovered by Peter Meilstrup and described in R-devel on 2018-08-13 (http://r.789695.n4.nabble.com/substitute-on-arguments-in-ellipsis-quot-dot-dot-dot-quot-td4751658.html).

```
#>
#> $y
#> 10 * z
```

There are two other important base quoting functions that we'll cover else-
where:

- `bquote()` provides a limited form of quasiquotation, and is discussed in
  Section 3.5.

- `~`, the formula, is a quoting function that also captures the environment. It's
  the inspiration for quosures, the topic of the next chapter, and is discussed
  in Section 4.3.4.

3.3.4   Substitution

You'll most often see `substitute()` used to capture unevaluated arguments.
However, as well as quoting, `substitute()` also does "substitution": if you give
it an expression, rather than a symbol, it will substitute in the values of symbols
defined in the current environment.

```
f4 <- function(x) substitute(x * 2)
f4(a + b + c)
#> (a + b + c) * 2
```

I think this makes code hard to understand, because if it is taken out of
context, you can't tell if the goal of `substitute(x + y)` is to replace `x`, `y`, or
both. If you do want to use `substitute()` for substitution, I recommend that
you use the second argument to make your goal clear:

```
substitute(x * y * z, list(x = 10, y = quote(a + b)))
#> 10 * (a + b) * z
```

3.3.5   Summary

When quoting (i.e. capturing code), there are two important distinctions:

- Is it supplied by the developer of the code or the user of the code? I.e. is
  it fixed (supplied in the body of the function) or varying (supplied via an
  argument)?

- Do you want to capture a single expression or multiple expressions?

This leads to a 2 x 2 table of functions for rlang, Table 3.1, and base R, Table
3.2.

Table 3.1: rlang quasiquoting functions

	Developer	User
One	`expr()`	`enexpr()`
Many	`exprs()`	`enexprs()`

Table 3.2: base R quoting functions

	Developer	User
One	`quote()`	`substitute()`
Many	`alist()`	`as.list(substitute(...()))`

3.3.6 Exercises

1. How is `expr()` implemented? Look at its source code.

2. Compare and contrast the following two functions. Can you predict the output before running them?

```
f1 <- function(x, y) {
  exprs(x = x, y = y)
}
f2 <- function(x, y) {
  enexprs(x = x, y = y)
}
f1(a + b, c + d)
f2(a + b, c + d)
```

3. What happens if you try to use `enexpr()` with an expression (i.e. `enexpr(x + y)` ? What happens if `enexpr()` is passed a missing argument?

4. How are `exprs(a)` and `exprs(a = )` different? Think about both the input and the output.

5. What are other differences between `exprs()` and `alist()`? Read the documentation for the named arguments of `exprs()` to find out.

6. The documentation for `substitute()` says:

   Substitution takes place by examining each component of the parse tree as follows:

- If it is not a bound symbol in `env`, it is unchanged.
- If it is a promise object (i.e., a formal argument to a function) the expression slot of the promise replaces the symbol.
- If it is an ordinary variable, its value is substituted;
- Unless `env` is .GlobalEnv in which case the symbol is left unchanged.

Create examples that illustrate each of the four different cases.

3.4   Unquoting

So far, you've only seen relatively small advantages of the rlang quoting functions over the base R quoting functions: they have a more consistent naming scheme. The big difference is that rlang quoting functions are actually quasiquoting functions because they can also unquote.

Unquoting allows you to selectively evaluate parts of the expression that would otherwise be quoted, which effectively allows you to merge together ASTs using a template AST. Since base functions don't use unquoting, they instead use a variety of other techniques, which you'll learn about in Section 3.5.

Unquoting is one inverse of quoting. It allows you to selectively evaluate code inside `expr()`, so that `expr(!!x)` is equivalent to `x`. In Chapter 4, you'll learn about another inverse, evaluation. This happens outside `expr()`, so that `eval(expr(x))` is equivalent to `x`.

3.4.1   Unquoting one argument

Use `!!` to unquote a single argument in a function call. `!!` takes a single expression, evaluates it, and inlines the result in the AST.

```
x <- expr(a + b + c)
expr(f(!!x, y))
#> f(a + b + c, y)
```

I think this is easiest to understand with a diagram. `!!` introduces a placeholder in the AST, shown with dotted borders. Here the placeholder `x` is replaced by an AST, illustrated by a dotted connection.
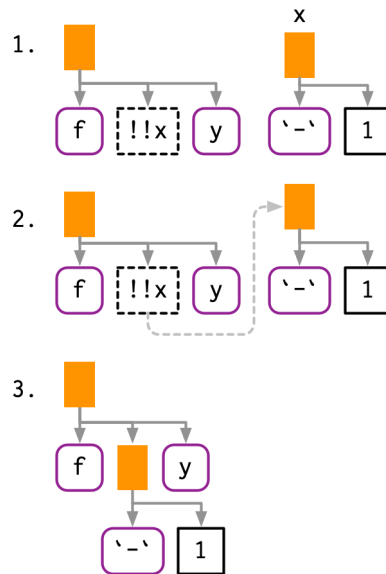
As well as call objects, `!!` also works with symbols and constants:

```
a <- sym("y")
b <- 1
expr(f(!!a, !!b))
#> f(y, 1)
```
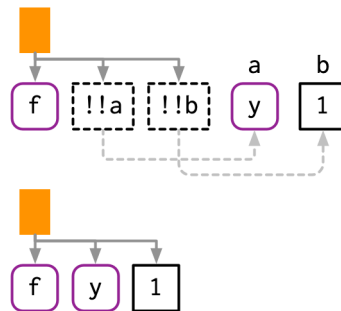

If the right-hand side of `!!` is a function call, `!!` will evaluate it and insert the results:

```
mean_rm <- function(var) {
  var <- ensym(var)
  expr(mean(!!var, na.rm = TRUE))
}
```

```
expr(!!mean_rm(x) + !!mean_rm(y))
#> mean(x, na.rm = TRUE) + mean(y, na.rm = TRUE)
```

!! preserves operator precedence because it works with expressions.

```
x1 <- expr(x + 1)
x2 <- expr(x + 2)

expr(!!x1 / !!x2)
#> (x + 1)/(x + 2)
```
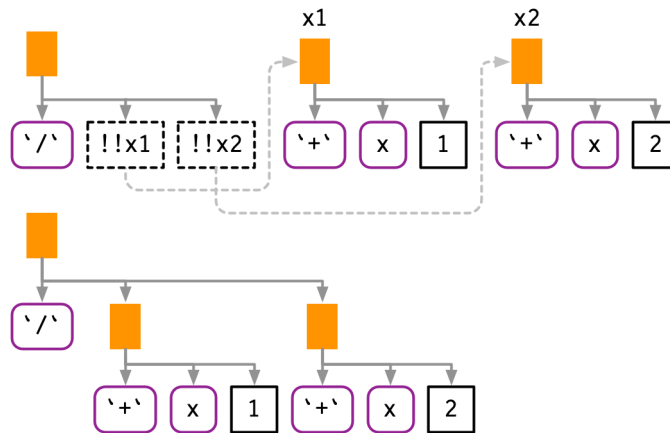

If we simply pasted the text of the expressions together, we'd end up with `x + 1 / x + 2`, which has a very different AST:
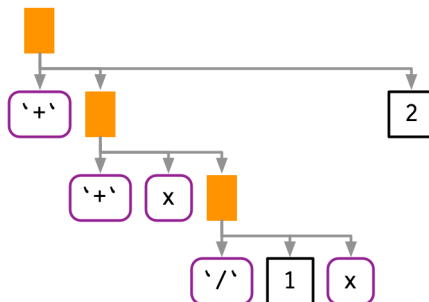
3.4.2   Unquoting a function

!! is most commonly used to replace the arguments to a function, but you can also use it to replace the function itself. The only challenge here is operator

precedence: `expr(!!f(x, y))` unquotes the result of `f(x, y)`, so you need an extra pair of parentheses.

```
f <- expr(foo)
expr((!!f)(x, y))
#> foo(x, y)
```

This also works when `f` is itself a call:

```
f <- expr(pkg::foo)
expr((!!f)(x, y))
#> pkg::foo(x, y)
```
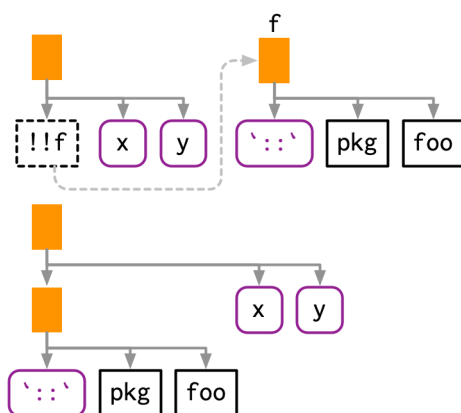

Because of the large number of parentheses involved, it can be clearer to use `rlang::call2()`:

```
f <- expr(pkg::foo)
call2(f, expr(x), expr(y))
#> pkg::foo(x, y)
```

3.4.3   Unquoting a missing argument

Very occasionally it is useful to unquote a missing argument (Section 2.6.2), but the naive approach doesn't work:

```
arg <- missing_arg()
expr(foo(!!arg, !!arg))
#> Error in enexpr(expr): argument "arg" is missing, with no default
```

You can work around this with the `rlang::maybe_missing()` helper:

```
expr(foo(!!maybe_missing(arg), !!maybe_missing(arg)))
#> foo(, )
```

3.4.4 Unquoting in special forms

There are a few special forms where unquoting is a syntax error. Take `$` for
example: it must always be followed by the name of a variable, not another
expression. This means attempting to unquote with `$` will fail with a syntax
error:

```
expr(df$!!x)
#> Error: unexpected '!' in "expr(df$!"
```

To make unquoting work, you'll need to use the prefix form (Section **??**):

```
x <- expr(x)
expr(`$`(df, !!x))
#> df$x
```

3.4.5 Unquoting many arguments
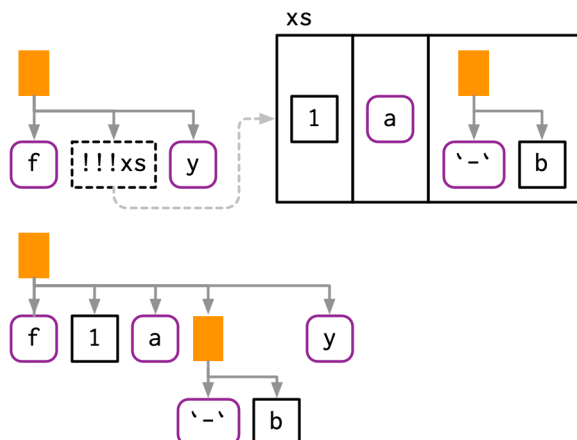
`!!` is a one-to-one replacement. `!!!` (called "unquote-splice", and pronounced
bang-bang-bang) is a one-to-many replacement. It takes a list of expressions and
inserts them at the location of the `!!!`:

```
xs <- exprs(1, a, -b)
expr(f(!!!xs, y))
#> f(1, a, -b, y)

# Or with names
ys <- set_names(xs, c("a", "b", "c"))
expr(f(!!!ys, d = 4))
#> f(a = 1, b = a, c = -b, d = 4)
```

!!! can be used in any rlang function that takes ... regardless of whether or not ... is quoted or evaluated. We'll come back to this in Section 3.6; for now note that this can be useful in `call2()`.

```
call2("f", !!!xs, expr(y))
#> f(1, a, -b, y)
```

3.4.6 The polite fiction of !!

So far we have acted as if !! and !!! are regular prefix operators like + , -, and !. They're not. From R's perspective, !! and !!! are simply the repeated application of !:

```
!!TRUE
#> [1] TRUE
!!!TRUE
#> [1] FALSE
```

!! and !!! behave specially inside all quoting functions powered by rlang, where they behave like real operators with precedence equivalent to unary + and -. This requires considerable work inside rlang, but means that you can write !!x + !!y instead of (!!x) + (!!y).

The biggest downside[3] to using a fake operator is that you might get silent errors when misusing !! outside of quasiquoting functions. Most of the time this

---

[3]Prior to R 3.5.1, there was another major downside: the R deparser treated !!x as !(!x). This is why in old versions of R you might see extra parentheses when printing expressions. The good news is that these parentheses are not real and can be safely ignored most of the time. The bad news is that they will become real if you reparse that printed output to R code. These roundtripped functions will not work as expected since !(!x) does not unquote.

is not an issue because `!!` is typically used to unquote expressions or quosures. Since expressions are not supported by the negation operator, you will get an argument type error in this case:

```
x <- quote(variable)
!!x
#> Error in !x: invalid argument type
```

But you can get silently incorrect results when working with numeric values:

```
df <- data.frame(x = 1:5)
y <- 100
with(df, x + !!y)
#> [1] 2 3 4 5 6
```

Given these drawbacks, you might wonder why we introduced new syntax instead of using regular function calls. Indeed, early versions of tidy evaluation used function calls like `UQ()` and `UQS()`. However, they're not really function calls, and pretending they are leads to a misleading mental mode. We chose `!!` and `!!!` as the least-bad solution:

- The are visually strong and don't look like existing syntax. When you see `!!x` or `!!!x` it's clear that something unusual is happening.

- They overrides a rarely used piece of syntax, as double negation is not a common pattern in R[4]. If you you do need it, you can just add parentheses `!(!x)`.

3.4.7   Non-standard ASTs

With unquoting, it's easy to create non-standard ASTs, i.e. ASTs that contain components that are not expressions. (It is also possible to create non-standard ASTs by directly manipulating the underlying objects, but it's harder to do so accidentally.) These are valid, and occasionally useful, but their correct use is beyond the scope of this book. However, it's important to learn about them, because they can be deparsed, and hence printed, in misleading ways.

For example, if you inline more complex objects, their attributes are not printed. This can lead to confusing output:

```
x1 <- expr(class(!!data.frame(x = 10)))
x1
```

---

[4]Unlike, say, javascript, where `!!x` is a commonly used shortcut to convert an integer into a logical.

```
#> class(list(x = 10))
eval(x1)
#> [1] "data.frame"
```

You have two main tools to reduce this confusion: `rlang::expr_print()` and `lobstr::ast()`:

```
expr_print(x1)
#> class(<data.frame>)
lobstr::ast(!!x1)
#>   class
#>   <inline data.frame>
```

Another confusing case arises if you inline an integer sequence:

```
x2 <- expr(f(!!c(1L, 2L, 3L, 4L, 5L)))
x2
#> f(1:5)
expr_print(x2)
#> f(<int: 1L, 2L, 3L, 4L, 5L>)
lobstr::ast(!!x2)
#>   f
#>   <inline integer>
```

It's also possible to create regular ASTs that can not be generated from code because of operator precedence. In this case, R will print parentheses that do not exist in the AST:

```
x3 <- expr(1 + !!expr(2 + 3))
x3
#> 1 + (2 + 3)

lobstr::ast(!!x3)
#>   `+`
#>   1
#>     `+`
#>     2
#>     3
```

3.4.8   Exercises

1. Given the following components:

```
xy <- expr(x + y)
xz <- expr(x + z)
yz <- expr(y + z)
abc <- exprs(a, b, c)
```

Use quasiquotation to construct the following calls:

```
(x + y) / (y + z)
-(x + z) ^ (y + z)
(x + y) + (y + z) - (x + y)
atan2(x + y, y + z)
sum(x + y, x + y, y + z)
sum(a, b, c)
mean(c(a, b, c), na.rm = TRUE)
foo(a = x + y, b = y + z)
```

2. The following two calls print the same, but are actually different:

```
(a <- expr(mean(1:10)))
#> mean(1:10)
(b <- expr(mean(!!(1:10))))
#> mean(1:10)
identical(a, b)
#> [1] FALSE
```

What's the difference? Which one is more natural?

3.5 Non-quoting

Base R has one function that implements quasiquotation: `bquote()`. It uses `.()` for unquoting:

```
xyz <- bquote((x + y + z))
bquote(-.(xyz) / 2)
#> -(x + y + z)/2
```

`bquote()` isn't used by any other function in base R, and has had relatively little impact on how R code is written. There are three challenges to effective use of `bquote()`:

- It is only easily used with your code; it is hard to apply it to arbitrary code supplied by a user.

- It does not provide an unquote-splice operator that allows you to unquote multiple expressions stored in a list.

- It lacks the ability to handle code accompanied by an environment, which is crucial for functions that evaluate code in the context of a data frame, like `subset()` and friends.

Instead functions that quote an argument use some other technique to allow indirect specification. Rather than using use unquoting all base R approaches selectively turn quoting off, so I call them **non-quoting** techniques.

There are four basic forms seen in base R:

- A pair of quoting and non-quoting functions. For example, `$` has two arguments, and the second argument is quoted. This is easier to see if you write in prefix form: `mtcars$cyl` is equivalent to `` `$`(mtcars, cyl) ``. If you want to refer to a variable indirectly, you use `[[`, as it takes the name of a variable as a string.

```
x <- list(var = 1, y = 2)
var <- "y"

x$var
#> [1] 1
x[[var]]
#> [1] 2
```

There are three other quoting functions closely related to `$`: `subset()`, `transform()`, and `with()`. These are seen as wrappers around `$` only suitable for interactive use so they all have the same non-quoting alternative: `[`

`<-`/`assign()` and `::`/`getExportedValue()` work similarly to `$`/`[`.

- A pair of quoting and non-quoting arguments. For example, `rm()` allows you to provide bare variable names in `...`, or a character vector of variable names in `list`:

```
x <- 1
rm(x)

y <- 2
vars <- c("y", "vars")
rm(list = vars)
```

`data()` and `save()` work similarly.

- An argument that controls whether a different argument is quoting or non-quoting. For example, in `library()`, the `character.only` argument controls the quoting behaviour of the first argument, `package`:

```
library(MASS)

pkg <- "MASS"
library(pkg, character.only = TRUE)
```

  `demo()`, `detach()`, `example()`, and `require()` work similarly.

- Quoting if evaluation fails. For example, the first argument to `help()` is non-quoting if it evaluates to a string; if evaluation fails, the first argument is quoted.

```
# Shows help for var
help(var)

var <- "mean"
# Shows help for mean
help(var)

var <- 10
# Shows help for var
help(var)
```

  `ls()`, `page()`, and `match.fun()` work similarly.

Another important class of quoting functions are the base modelling and plotting functions, which follow the so-called standard non-standard evaluation rules: http://developer.r-project.org/nonstandard-eval.pdf. For example, `lm()` quotes the `weight` and `subset` arguments, and when used with a formula argument, the plotting function quotes the aesthetic arguments (`col`, `cex`, etc):

```
palette(RColorBrewer::brewer.pal(3, "Set1"))
plot(
  Sepal.Length ~ Petal.Length,
  data = iris,
  col = Species,
  pch = 20,
  cex = 2
)
```
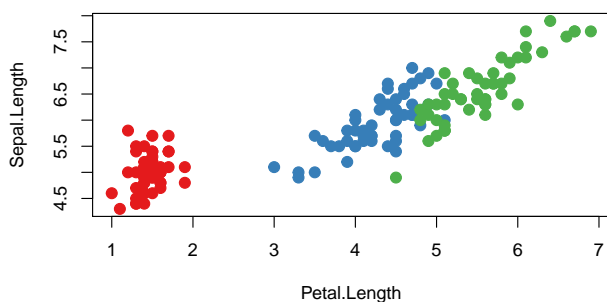
These functions have no built-in options for indirect specification, but you'll learn how to simulate unquoting in Section 3.5.

3.6 Dot-dot-dot (. . .)

`!!!` is useful because it's not uncommon to have a list of expressions that you want to insert into a call. It turns out that this pattern is common elsewhere. Take the following two motivating problems:

- What do you do if the elements you want to put in `...` are already stored in a list? For example, imagine you have a list of data frames that you want to `rbind()` together:

```
dfs <- list(
  a = data.frame(x = 1, y = 2),
  b = data.frame(x = 3, y = 4)
)
```

  You could solve this specific case with `rbind(dfs$a, dfs$b)`, but how do you generalise that solution to a list of arbitrary length?

- What do you do if you want to supply the argument name indirectly? For example, imagine you want to create a single column data frame where the name of the column is specified in a variable:

```
var <- "x"
val <- c(4, 3, 9)
```

  In this case, you could create a data frame and then change names (i.e. `setNames(data.frame(val), var)`), but this feels inelegant. How can we do better?

One way to think about these problems is to draw explicit parallels to quasiquotation:

- Row-binding multiple data frames is like unquote-splicing: we want to inline individual elements of the list into the call:

```
dplyr::bind_rows(!!!dfs)
#>   x y
#> 1 1 2
#> 2 3 4
```

  When used in this context, the behaviour of !!! is known as "spatting" in Ruby, Go, PHP, and Julia. It is closely related to *args (star-args) and **kwarg (star-star-kwargs) in Python, which are sometimes called argument unpacking.

- The second problem is like unquoting the left-hand side of =: rather than interpreting var literally, we want to use the value stored in the variable called var:

```
tibble::tibble(!!var := val)
#> # A tibble: 3 x 1
#>       x
#>   <dbl>
#> 1     4
#> 2     3
#> 3     9
```

  Note the use of := (pronounced colon-equals) rather than =. Unfortunately we need this new operation because R's grammar does not allow expressions as argument names:

```
tibble::tibble(!!var = value)
#> Error: unexpected '=' in "tibble::tibble(!!var ="
```

  := is like a vestigial organ: it's recognised by R's parser, but it doesn't have any code associated with it. It looks like an = but allows expressions on either side, making it a more flexible alternative to =. It is used in data.table for similar reasons.

Base R takes a different approach, which we'll come back to Section @ref{do-call}.

We say functions that support these tools, without quoting arguments, have **tidy dots**[5]. To gain tidy dots behaviour in your own function, all you need to do is use `list2()`.

3.6.1   Examples

One place we could use `list2()` is to create a wrapper around `attributes()` that allows us to set attributes flexibly:

```
set_attr <- function(.x, ...) {
  attr <- rlang::list2(...)
  attributes(.x) <- attr
  .x
}

attrs <- list(x = 1, y = 2)
attr_name <- "z"

1:10 %>%
  set_attr(w = 0, !!!attrs, !!attr_name := 3) %>%
  str()
#>   int [1:10] 1 2 3 4 5 6 7 8 9 10
#>   - attr(*, "w")= num 0
#>   - attr(*, "x")= num 1
#>   - attr(*, "y")= num 2
#>   - attr(*, "z")= num 3
```

3.6.2   `exec()`

What if you want to use this technique with a function that doesn't have tidy dots? One option is to use `rlang::exec()` to call a function with some arguments supplied directly (in ...) and others indirectly (in a list):

```
# Directly
exec("mean", x = 1:10, na.rm = TRUE, trim = 0.1)
#> [1] 5.5

# Indirectly
args <- list(x = 1:10, na.rm = TRUE, trim = 0.1)
exec("mean", !!!args)
```

---

[5]This is admittedly not the most creative of names, but it clearly suggests it's something that has been added to R after the fact.

```
#> [1] 5.5

# Mixed
params <- list(na.rm = TRUE, trim = 0.1)
exec("mean", x = 1:10, !!!params)
#> [1] 5.5
```

`rlang::exec()` also makes it possible to supply argument names indirectly:

```
arg_name <- "na.rm"
arg_val <- TRUE
exec("mean", 1:10, !!arg_name := arg_val)
#> [1] 5.5
```

And finally, it's useful if you have a vector of function names or a list of functions that you want to call with the same arguments:

```
x <- c(runif(10), NA)
funs <- c("mean", "median", "sd")

purrr::map_dbl(funs, exec, x, na.rm = TRUE)
#> [1] 0.444 0.482 0.298
```

`exec()` is closely related to `call2()`; where `call2()` returns an expression, `exec()` evaluates it.

3.6.3 `dots_list()`

`list2()` provides one other handy feature: by default it will ignore any empty arguments at the end. This is useful in functions like `tibble::tibble()` because it means that you can easily change the order of variables without worrying about the final comma:

```
# Can easily move x to first entry:
tibble::tibble(
  y = 1:5,
  z = 3:-1,
  x = 5:1,
)

# Need to remove comma from z and add comma to x
data.frame(
  y = 1:5,
```

```
  z = 3:-1,
  x = 5:1
)
```

list2() is a wrapper around `rlang::dots_list()` with defaults set to the most commonly used settings. You can get more control by calling `dots_list()` directly:

- `.ignore_empty` allows you to control exactly which arguments are ignored. The default ignores a single trailing argument to get the behaviour describe above, but you can choose to ignore all missing arguments, or no missing arguments.

- `.homonoyms` controls what happens if multiple arguments use the same name:

```
str(dots_list(x = 1, x = 2))
#> List of 2
#>  $ x: num 1
#>  $ x: num 2
str(dots_list(x = 1, x = 2, .homonyms = "first"))
#> List of 1
#>  $ x: num 1
str(dots_list(x = 1, x = 2, .homonyms = "last"))
#> List of 1
#>  $ x: num 2
str(dots_list(x = 1, x = 2, .homonyms = "error"))
#> Error: Arguments can't have the same name.
#> We found multiple arguments named `x` at positions 1 and 2
```

- If there are empty arguments that are not ignored, `.preserve_empty` controls what to do with them. The default throws an error; setting `.preserve_empty = TRUE` instead returns missing symbols. This is useful if you're using `dots_list()` to generate function calls.

3.6.4 With base R

Base R provides a Swiss army knife to solve these problems: `do.call()`. `do.call()` has two main arguments. The first argument, `what`, gives a function to call. The second argument, `args`, is a list of arguments to pass to that function, and so `do.call("f", list(x, y, z))` is equivalent to `f(x, y, z)`.

- `do.call()` gives a straightforward solution to `rbind()`ing together many data frames:

```
do.call("rbind", dfs)
#>   x y
#> a 1 2
#> b 3 4
```

- With a little more work, we can use `do.call()` to solve the second problem. We first create a list of arguments, then name that, then use `do.call()`:

```
args <- list(val)
names(args) <- var

do.call("data.frame", args)
#>   x
#> 1 4
#> 2 3
#> 3 9
```

Some base functions (including `interaction()`, `expand.grid()`, `options()`, and `par()`) use a trick to avoid `do.call()`: if the first component of `...` is a list, they'll take its components instead of looking at the other elements of `...`. The implementation looks something like this:

```
f <- function(...) {
  dots <- list(...)
  if (length(dots) == 1 && is.list(dots[[1]])) {
    dots <- dots[[1]]
  }

  # Do something
  ...
}
```

Another approach to avoiding `do.call()` is found in the `RCurl::getURL()` function written by Duncan Temple Lang. `getURL()` takes both `...` and `.opts` which are concatenated together. This looks something like this:

```
f <- function(..., .dots) {
  dots <- c(list(...), .dots)
  # Do something
}
```

At the time I discovered it, I found this technique particularly compelling so you can see it used throughout the tidyverse. Now, however, I prefer the approach described next.

3.6.5 Exercises

1. One way to implement `exec()` is shown below. Describe how it works. What are the key ideas?

```
exec <- function(f, ..., .env = caller_env()) {
  args <- list2(...)
  do.call(f, args, envir = .env)
}
```

2. Carefully read the source code for `interaction()`, `expand.grid()`, and `par()`. Compare and contrast the techniques they use for switching between dots and list behaviour.

3. Explain the problem with this definition of `set_attr()`

```
set_attr <- function(x, ...) {
  attr <- rlang::list2(...)
  attributes(x) <- attr
  x
}
set_attr(1:10, x = 10)
#> Error in attributes(x) <- attr: attributes must be named
```

3.7 Case studies

To make the ideas of quasiquotation concrete, this section contains a few small case studies that use it to solve real problems. Some of the case studies also use purrr: I find the combination of quasiquotation and functional programming to be particularly elegant.

3.7.1 `lobstr::ast()`

Quasiquotation allows us to solve an annoying problem with `lobstr::ast()`: what happens if we've already captured the expression?

```
z <- expr(foo(x, y))
lobstr::ast(z)
#> z
```

Because `ast()` quotes its first argument, we can use `!!`:

```
lobstr::ast(!!z)
#>  foo
#>  x
#>  y
```

3.7.2   Map-reduce to generate code

Quasiquotation gives us powerful tools for generating code, particularly when
combined with `purrr::map()` and `purr::reduce()`. For example, assume you
have a linear model specified by the following coefficients:

```
intercept <- 10
coefs <- c(x1 = 5, x2 = -4)
```

And you want to convert it into an expression like `10 + (x1 * 5) + (x2 *
-4)`. The first thing we need to do is turn the character names vector into a list
of symbols. `rlang::syms()` is designed precisely for this case:

```
coef_sym <- syms(names(coefs))
coef_sym
#> [[1]]
#> x1
#>
#> [[2]]
#> x2
```

Next we need to combine each variable name with its coefficient. We can do
this by combining `rlang::expr()` with `purrr::map2()`:

```
summands <- map2(coef_sym, coefs, ~ expr((!!.x * !!.y)))
summands
#> [[1]]
#> (x1 * 5)
#>
#> [[2]]
#> (x2 * -4)
```

In this case, the intercept is also a part of the sum, although it doesn't involve
a multiplication. We can just add it to the start of the `summands` vector:

```
summands <- c(intercept, summands)
summands
```

```
#> [[1]]
#> [1] 10
#>
#> [[2]]
#> (x1 * 5)
#>
#> [[3]]
#> (x2 * -4)
```

Finally, we need to reduce (Section **??**) the individual terms into a single sum by adding the pieces together:

```
eq <- reduce(summands, ~ expr(!!.x + !!.y))
eq
#> 10 + (x1 * 5) + (x2 * -4)
```

We could make this even more general by allowing the user to supply the name of the coefficient, and instead of assuming many different variables, index into a single one.

```
var <- expr(y)
coef_sym <- map(seq_along(coefs), ~ expr((!!var)[[!!.x]]))
coef_sym
#> [[1]]
#> y[[1L]]
#>
#> [[2]]
#> y[[2L]]
```

And finish by wrapping this up in a function:

```
linear <- function(var, val) {
  var <- ensym(var)
  coef_name <- map(seq_along(val[-1]), ~ expr((!!var)[[!!.x]]))

  summands <- map2(val[-1], coef_name, ~ expr((!!.x * !!.y)))
  summands <- c(val[[1]], summands)

  reduce(summands, ~ expr(!!.x + !!.y))
}

linear(x, c(10, 5, -4))
#> 10 + (5 * x[[1L]]) + (-4 * x[[2L]])
```

Note the use of `ensym()`: we want the user to supply the name of a single variable, not a more complex expression.

3.7.3   Slicing an array

An occasionally useful tool missing from base R is the ability to extract a slice of an array given a dimension and an index. For example, we'd like to write `slice(x, 2, 1)` to extract the first slice along the second dimension, i.e. `x[, 1, ]`. This is a moderately challenging problem because it requires working with missing arguments.

We'll need to generate a call with multiple missing arguments. We first generate a list of missing arguments with `rep()` and `missing_arg()`, then unquote-splice them into a call:

```
indices <- rep(list(missing_arg()), 3)
expr(x[!!!indices])
#> x[, , ]
```

Then we use subset-assignment to insert the index in the desired position:

```
indices[[2]] <- 1
expr(x[!!!indices])
#> x[, 1, ]
```

We then wrap this into a function, using a couple of `stopifnot()`s to make the interface clear:

```
slice <- function(x, along, index) {
  stopifnot(length(along) == 1)
  stopifnot(length(index) == 1)

  nd <- length(dim(x))
  indices <- rep(list(missing_arg()), nd)
  indices[[along]] <- index

  expr(x[!!!indices])
}

x <- array(sample(30), c(5, 2, 3))
slice(x, 1, 3)
#> x[3, , ]
slice(x, 2, 2)
#> x[, 2, ]
```

```
slice(x, 3, 1)
#> x[, , 1]
```

A real `slice()` would evaluate the generated call (Chapter 4), but here I think it's more illuminating to see the code that's generated, as that's the hard part of the challenge.

3.7.4 Creating functions

Another powerful application of quotation is creating functions "by hand", using `rlang::new_function()`. It's a function that create a function from its three components (Section **??**) arguments, body, and (optionally) an environment:

```
new_function(
  exprs(x = , y = ),
  expr({x + y})
)
#> function (x, y)
#> {
#>     x + y
#> }
```

NB: the empty arguments in `exprs()` generates arguments with no defaults.

One use of `new_function()` is as an alternative to function factories with scalar or symbol arguments. For example, we could write a function that generates functions that raise a function to the power of a number.

```
power <- function(exponent) {
  new_function(
    exprs(x = ),
    expr({
      x ^ !!exponent
    }),
    caller_env()
  )
}
power(0.5)
#> function (x)
#> {
#>     x^0.5
#> }
```

Another application of `new_function()` is for functions that work like `graphics::curve()`, which allows you to plot a mathematical expression without creating a function:

```
curve(sin(exp(4 * x)), n = 1000)
```
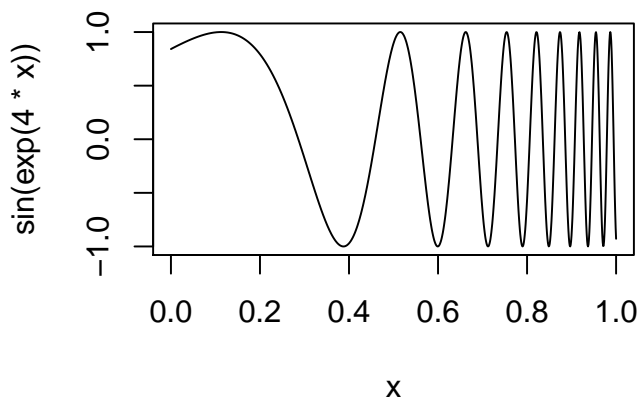

Here `x` is a pronoun: it doesn't represent a single concrete value, but is instead a placeholder that varies over the range of the plot. One way to implement `curve()` is to turn that expression into a function with a single argument, `x`, then call that function:

```
curve2 <- function(expr, xlim = c(0, 1), n = 100) {
  expr <- enexpr(expr)
  f <- new_function(exprs(x = ), expr)

  x <- seq(xlim[1], xlim[2], length = n)
  y <- f(x)

  plot(x, y, type = "l", ylab = expr_text(expr))
}
curve2(sin(exp(4 * x)), n = 1000)
```

Functions like `curve()` that use an expression containing a pronoun are known as **anaphoric** functions[6].

3.7.5 Exercises

1. In the linear-model example, we could replace the `expr()` in `reduce(summands, ~ expr(!!.x + !!.y))` with `call2()`: `reduce(summands, call2, "+")`.

---

[6]Anaphoric comes from the linguistics term "anaphora", an expression that is context dependent. Anaphoric functions are found in Arc (http://www.arcfn.com/doc/anaphoric.html) (a LISP like language), Perl (http://www.perlmonks.org/index.pl?node_id=666047), and Clojure (http://amalloy.hubpages.com/hub/Unhygenic-anaphoric-Clojure-macros-for-fun-and-profit).

Compare and contrast the two approaches. Which do you think is easier to read?

2. Re-implement the Box-Cox transform defined below using unquoting and `new_function()`:

```
bc <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
  }
}
```

3. Re-implement the simple `compose()` defined below using quasiquotation and `new_function()`:

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

3.8   History

The idea of quasiquotation is an old one. It was first developed by the philosopher Willard van Orman Quine[7] in the early 1940s. It's needed in philosophy[8] because it helps when precisely delineating the use and mention of words, i.e. distinguishing between the object and the words we use to refer to that object.

Quasiquotation was first used in a programming language, LISP, in the mid-1970s (Bawden 1999). LISP has one quoting function `` ` ``, and uses `,` for unquoting. Most languages with a LISP heritage behave similarly. For example, racket (`` ` `` and `@`), Clojure (`` ` `` and `~`), and Julia (`:` and `@`) all have quasiquotation tools that differ only slightly from LISP. These languages have a single quoting function and you must call it explicitly.

In R, however, many functions quote one or more inputs. This introduces ambiguity (because you need to read the documentation to determine if an argument is quoted or not), but allows for concise and elegant data exploration code. In base R, only one function supports quasiquotation: `bquote()`, written

---

[7]You might be familiar with the name Quine from "quines", computer programs that return a copy of their own source when run.

[8]A fun connection between philosophy and R is in https://johnmacfarlane.net/142/substitutional-quantifiers.pdf; this article is written by philosophy professor John MacFarlane, the author of pandoc, which powers RMarkdown.

in 2003 by Thomas Lumley. However, `bquote()` has some major limitations which prevented it from having a wide impact on R code (Section 3.5).

My attempt to resolve these limitations lead to the lazyeval package (2014-2015). Unfortunately, my analysis of the problem was incomplete and while lazyeval solves some problems, it create others. It was not until I started working with Lionel Henry on the problem that all the pieces finally fell into place and we create the full tidy evaluation framework (2017). Despite the newness of tidy evaluation, I teach it here because it is a rich and powerful theory that, once mastered, makes many hard problems much easier.

Chapter 4

Evaluation

4.1 Introduction

The user-facing inverse of quotation is unquotation: it gives the *user* the ability
to selectively evaluate parts of an otherwise quoted argument. The developer-
facing complement of quotation is evaluation: this gives the *developer* the ability
to evaluate quoted expressions in custom environments to achieve specific goals.

This chapter begins with a discussion of evaluation in its purest form. You'll
learn how `eval()` evaluates an expression in an environment, and then how it
can be used to implement a number of important base R functions. Once you
have the basics under your belt, you'll learn extensions to evaluation that are
needed for robustness. There are two big new ideas:

- The quosure: a data structure that captures an expression along with its
  associated environment, as found in function arguments.

- The data mask, which makes it easier to evaluate an expression in the
  context of a data frame. This introduces potential evaluation ambiguity
  which we'll then resolve with data pronouns.

Together, quasiquotation, quosures, and data masks form what we call **tidy
evaluation**, or tidy eval for short. Tidy eval provides a principled approach
to non-standard evaluation that makes it possible to use such functions both
interactively and embedded with other functions. Tidy evaluation is the most
important practical implication of all this theory so we'll spend a little time
exploring the implications. The chapter finishes off with a discussion of the
closest related approaches in base R, and how you can program around their
drawbacks.

Outline

- Section 4.2 discusses the basics of evaluation using `eval()`, and shows how you can use it to implement key functions like `local()` and `source()`.

- Section 4.3 introduces a new data structure, the quosure, which combines an expression with an environment. You'll learn how to capture them from promises, and evaluate them using `rlang::eval_tidy()`.

- Section 4.4 extends evaluation with the "data mask", which makes it trivial to intermingle symbols bound in an environment with variables found in a data frame.

- Section 4.5 shows how to use tidy evaluation in practice, focussing on the common pattern of quoting and unquoting, and how to handle ambiguity with pronouns.

- Section 4.6 circles back to evaluation in base R, discusses some of the downsides, and shows how to use quasiquotation and evaluation to wrap functions that use NSE.

Prerequisites

You'll need to be familiar with the content of Chapter 2 and Chapter 3, as well as the environment data structure (Section **??**) and the caller environment (Section **??**).

We'll continue to use rlang (https://rlang.r-lib.org) and purrr (https://purrr.tidyverse.org).

```
library(rlang)
library(purrr)
```

4.2   Evaluation basics

Here we'll explore the details of `eval()` which we briefly mentioned in the last chapter. It has two key arguments: `expr` and `envir`. The first argument, `expr`, is the object to evaluate, typically a symbol or expression[1]. None of the evaluation functions quote their inputs, so you'll usually use them with `expr()` or similar:

```
x <- 10
eval(expr(x))
#> [1] 10
```

---

[1] All other objects yield themselves when evaluated; i.e. `eval_bare(x)` yields `x`, except when `x` is a symbol or expression.

```
y <- 2
eval(expr(x + y))
#> [1] 12
```

The second argument, `env`, gives the environment in which the expression should be evaluated, i.e. where to look for the values of `x`, `y`, and `+`. By default, this is the current environment, i.e. the calling environment of `eval()`, but you can override it if you want:

```
eval(expr(x + y), env(x = 1000))
#> [1] 1002
```

The first argument is evaluated, not quoted, which can lead to confusing results once if you use a custom environment and forget to manually quote:

```
eval(print(x + 1), env(x = 1000))
#> [1] 11
#> [1] 11

eval(expr(print(x + 1)), env(x = 1000))
#> [1] 1001
```

Expression vectors

`base::eval()` has special behaviour for expression *vectors*, evaluating each component in turn. This makes for a very compact implementation of `source2()` because `base::parse()` also returns an expression object:

```
source3 <- function(file, env = parent.frame()) {
  lines <- parse(file)
  res <- eval(lines, envir = env)
  invisible(res)
}
```

While `source3()` is considerably more concise than `source2()`, this is the only advantage to expression vectors. Overall I don't believe this benefit outweighs the cost of introducing a new data structure, and hence this book avoids the use of expression vectors.

Now that you've seen the basics, let's explore some applications. We'll focus primarily on base R functions that you might have used before, reimplementing the underlying principles using rlang.

4.2.1   Application: `local()`

Sometimes you want to perform a chunk of calculation that creates some intermediate variables. The intermediate variables have no long-term use and could be quite large, so you'd rather not keep them around. One approach is to clean up after yourself using `rm()`; another is to wrap the code in a function and just call it once. A more elegant approach is to use `local()`:

```
# Clean up variables created earlier
rm(x, y)

foo <- local({
  x <- 10
  y <- 200
  x + y
})

foo
#> [1] 210
x
#> Error in eval(expr, envir, enclos): object 'x' not found
y
#> Error in eval(expr, envir, enclos): object 'y' not found
```

The essence of `local()` is quite simple and re-implemented below. We capture the input expression, and create a new environment in which to evaluate it. This is a new environment (so assignment doesn't affect the existing environment) with the caller environment as parent (so that `expr` can still access variables in that environment). This effectively emulates running `expr` as if it was inside a function (i.e. it's lexically scoped, Section **??**).

```
local2 <- function(expr) {
  env <- env(caller_env())
  eval_bare(enexpr(expr), env)
}

foo <- local2({
  x <- 10
  y <- 200
  x + y
})

foo
```

```
#> [1] 210
x
#> Error in eval(expr, envir, enclos): object 'x' not found
y
#> Error in eval(expr, envir, enclos): object 'y' not found
```

Understanding how `base::local()` works is harder, as it uses `eval()` and `substitute()` together in rather complicated ways. Figuring out exactly what's going on is good practice if you really want to understand the subtleties of `substitute()` and the base `eval()` functions, so is included in the exercises below.

4.2.2 Application: `source()`

We can create a simple version of `source()` by combining `eval()` with `parse_expr()` from Section 2.4.3. We read in the file from disk, use `parse_expr()` to parse the string into a list of expressions, and then use `eval_bare()` to evaluate each element in turn. This version evaluates the code in the caller environment, and invisibly returns the result of the last expression in the file just like `base::source()`.

```r
source2 <- function(path, env = caller_env()) {
  file <- paste(readLines(path, warn = FALSE), collapse = "\n")
  exprs <- parse_exprs(file)

  res <- NULL
  for (i in seq_along(exprs)) {
    res <- eval(exprs[[i]], env)
  }

  invisible(res)
}
```

The real `source()` is considerably more complicated because it can `echo` input and output, and has many other settings that control its behaviour.

4.2.3 Gotcha: `function()`

There's one small gotcha that you should be aware of if you're using `eval_bare()` and `expr()` to generate functions:

```
x <- 10
y <- 20
f <- eval(expr(function(x, y) !!x + !!y))
f
#> function(x, y) !!x + !!y
```

This function doesn't look like it will work, but it does:

```
f()
#> [1] 30
```

This is because, if available, functions print their `srcref` attribute (Section **??**), and because `srcref` is a base R feature it's unaware of quasiquotation.

To work around this problem, either use `new_function()` (Section 3.7.4) or remove the `srcref` attribute:

```
attr(f, "srcref") <- NULL
f
#> function (x, y)
#> 10 + 20
```

4.2.4   Exercises

1. Carefully read the documentation for `source()`. What environment does it use by default? What if you supply `local = TRUE`? How do you provide a custom environment?

2. Predict the results of the following lines of code:

   ```
   eval(expr(eval(expr(eval(expr(2 + 2))))))
   eval(eval(expr(eval(expr(eval(expr(2 + 2)))))))
   expr(eval(expr(eval(expr(eval(expr(2 + 2)))))))
   ```

3. Fill in the function bodies below to re-implement `get()` using `sym()` and `eval()`, and`assign()` using `sym()`, `expr()`, and `eval()`. Don't worry about the multiple ways of choosing an environment that `get()` and `assign()` support; assume that the user supplies it explicitly

   ```
   # name is a string
   get2 <- function(name, env) {}
   assign2 <- function(name, value, env) {}
   ```

4. Modify `source2()` so it returns the result of *every* expression, not just the last one. Can you eliminate the for loop?

5. We can make `base::local()` slightly easier to understand by spreading out over multiple lines:

```
local3 <- function(expr, envir = new.env()) {
  call <- substitute(eval(quote(expr), envir))
  eval(call, envir = parent.frame())
}
```

Explain how `local()` works in words. (Hint: you might want to `print(call)` to help understand what `substitute()` is doing, and read the documentation to remind yourself what environment `new.env()` will inherit from.)

4.3 Quosures

Almost every use of `eval()` involves both an expression and environment. This coupling is so important that we need a data structure that can hold both pieces. Base R does not have such a structure[2] so rlang fills the gap with the **quosure**, an object that contains an expression and an environment. The name is a portmanteau of quoting and closure, because a quosure both quotes the expression and encloses the environment. Quosures reify the internal promise object (Section **??**) into something that you can program with.

In this section, you'll learn how to create and manipulate quosures, and a little about how they are implemented.

4.3.1 Creating

There are three ways to create quosures:

- Use `enquo()` and `enquos()` to capture user-supplied expressions. The vast majority of quosures should be created this way.

```
foo <- function(x) enquo(x)
foo(a + b)
#> <quosure>
#> expr: ^a + b
#> env:  global
```

- `quo()` and `quos()` exist to match to `expr()` and `exprs()`, but they are included only for the sake of completeness and are needed very rarely. If you

---

[2]Technically a formula combines an expression and environment, but formulas are tightly coupled to modelling so a new data structure makes sense.

find yourself using them, think carefully if `expr()` and careful unquoting can eliminate the need to capture the environment.

```
quo(x + y + z)
#> <quosure>
#> expr: ^x + y + z
#> env:  global
```

- `new_quosure()` create a quosure from its components: an expression and an environment. This is rarely needed in practice, but is useful for learning, so is used a lot in this chapter.

```
new_quosure(expr(x + y), env(x = 1, y = 10))
#> <quosure>
#> expr: ^x + y
#> env:  0x48dbfd8
```

4.3.2 Evaluating

Quosures are paired with a new evaluation function `eval_tidy()` that takes a single quosure instead of a expression-environment pair. It is straightforward to use:

```
q1 <- new_quosure(expr(x + y), env(x = 1, y = 10))
eval_tidy(q1)
#> [1] 11
```

For this simple case, `eval_tidy(q1)` is basically a shortcut for `eval_bare(get_expr(q1), get_env(q2))`. However, it has two important features that you'll learn about later in the chapter: it supports nested quosures (Section 4.3.5) and pronouns (Section 4.4.2).

4.3.3 Dots

Quosures are typically just a convenience: they make code cleaner because you only have one object to pass around, instead of two. They are, however, essential when it comes to working with ... because it's possible for each argument passed to … to be associated with a different environment. In the following example note that both quosures have the same expression, x, but a different environment:

```
f <- function(...) {
  x <- 1
```

```
  g(..., f = x)
}
g <- function(...) {
  enquos(...)
}


x <- 0
qs <- f(global = x)
qs
#> <list_of<quosure>>
#>
#> $global
#> <quosure>
#> expr: ^x
#> env:  global
#>
#> $f
#> <quosure>
#> expr: ^x
#> env:  0x50f4868
```

That means that when you evaluate them, you get the correct results:

```
map_dbl(qs, eval_tidy)
#> global      f
#>      0      1
```

Correctly evaluating the elements of `...` was one of the original motivations for the development of quosures.

4.3.4 Under the hood

Quosures were inspired by R's formulas, because formulas capture an expression and an environment:

```
f <- ~runif(3)
str(f)
#> Class 'formula'  language ~runif(3)
#>   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

An early version of tidy evaluation used formulas instead of quosures, as an attractive feature of `~` is that it provides quoting with a single keystroke. Unfortunately, however, there is no clean way to make `~` a quasiquoting function.

Quosures are a subclass of formulas:

```
q4 <- new_quosure(expr(x + y + z))
class(q4)
#> [1] "quosure" "formula"
```

Which means that under the hood, quosures, like formulas, are a call object:

```
is_call(q4)
#> [1] TRUE

q4[[1]]
#>  `~`
q4[[2]]
#> x + y + z
```

With an attribute that stores the environment:

```
attr(q4, ".Environment")
#> <environment: R_GlobalEnv>
```

If you need to extract the expression or environment, don't rely on these implementation details. Instead use `get_expr()` and `get_env()`:

```
get_expr(q4)
#> x + y + z
get_env(q4)
#> <environment: R_GlobalEnv>
```

4.3.5   Nested quosures

It's possible to use quasiquotation to embed a quosure in an expression. This is an advanced tool, and most of the time you don't need to think about it because it just works, but I talk about it here so you can spot nested quosures in the wild and not be confused. Take this example, which inlines two quosures into an expression:

```
q2 <- new_quosure(expr(x), env(x = 1))
q3 <- new_quosure(expr(x), env(x = 10))

x <- expr(!!q2 + !!q3)
```

It evaluates correctly with `eval_tidy()`:

```
eval_tidy(x)
#> [1] 11
```

However, if you print it, you only see the xs, with their formula heritage leaking through:

```
x
#> (~x) + ~x
```

You can get a better display with `rlang::expr_print()` (Section 3.4.7):

```
expr_print(x)
#> (^x) + (^x)
```

When you use `expr_print()` in the console, quosures are coloured according to their environment, making it easier to spot when symbols are bound to different variables.

4.3.6 Exercises

1. Predict what evaluating each of the following quosures will return if evaluated.

    ```
    q1 <- new_quosure(expr(x), env(x = 1))
    q1
    #> <quosure>
    #> expr:  ^x
    #> env:   0x47c4bb0

    q2 <- new_quosure(expr(x + !!q1), env(x = 10))
    q2
    #> <quosure>
    #> expr:  ^x + (^x)
    #> env:   0x4962500

    q3 <- new_quosure(expr(x + !!q2), env(x = 100))
    q3
    #> <quosure>
    #> expr:  ^x + (^x + (^x))
    #> env:   0x4bea800
    ```

2. Write an `enenv()` function that captures the environment associated with an argument. (Hint: this should only require two function calls.)

4.4   Data masks

In this section, you'll learn about the **data mask**, a data frame where the evaluated code will look first for variable definitions. The data mask is the key idea that powers base functions like `with()`, `subset()` and `transform()`, and is used throughout the tidyverse in packages like dplyr and ggplot2.

4.4.1   Basics

The data mask allows you to mingle variables from an environment and and data frame in a single expression. You supply the data mask as the second argument to `eval_tidy()`:

```
q1 <- new_quosure(expr(x * y), env(x = 100))
df <- data.frame(y = 1:10)

eval_tidy(q1, df)
#>  [1]  100  200  300  400  500  600  700  800  900 1000
```

This code is a little hard to follow because there's so much syntax as we're creating every object from from scratch. It's easier to see what's going on if we make a little wrapper. I call this `with2()` because it's equivalent to `base::with()`.

```
with2 <- function(data, expr) {
  expr <- enquo(expr)
  eval_tidy(expr, data)
}
```

We can now rewrite the code above as below:

```
x <- 100
with2(df, x * y)
#>  [1]  100  200  300  400  500  600  700  800  900 1000
```

`base::eval()` has similar functionality, although it doesn't call it a data mask. Instead you can supply a data frame to the second argument and an environment to the third. That gives the following implementation of `with()`:

```
with3 <- function(data, expr) {
  expr <- substitute(expr)
  eval(expr, data, caller_env())
}
```

4.4.2 Pronouns

Using a data mask introduces ambiguity. For example, in the following code you can't know whether `x` will come from the data mask or the environment, unless you know what variables are found in `df`.

```
with2(df, x)
```

That makes code harder to reason about (because you need to know more context), which can introduce bugs. To resolve that issue, the data mask provides two pronouns: `.data` and `.env`.

- `.data$x` always refers to `x` in the data mask.
- `.env$x` always refers to `x` in the environment.

```
x <- 1
df <- data.frame(x = 2)

with2(df, .data$x)
#> [1] 2
with2(df, .env$x)
#> [1] 1
```

You can also subset `.data` and `.env` using `[[`, e.g. `.data[["x"]]`. Otherwise the pronouns are special objects and you shouldn't expect them to behave like data frames or environments. In particular, they throw error if the object isn't found:

```
with2(df, .data$y)
#> Error: Column `y` not found in `.data`
```

4.4.3 Application: `subset()`

We'll explore tidy evaluation in the context of `base::subset()`, because it's a simple yet powerful function that makes a common data manipulation challenge easier. If you haven't used it before, `subset()`, like `dplyr::filter()`, provides a convenient way of selecting rows of a data frame. You give it some data, along with an expression that is evaluated in the context of that data. This considerably reduces the number of times you need to type the name of the data frame:

```
sample_df <- data.frame(a = 1:5, b = 5:1, c = c(5, 3, 1, 4, 1))

# Shorthand for sample_df[sample_df$a >= 4, ]
```

```
subset(sample_df, a >= 4)
#>    a b c
#> 4 4 2 4
#> 5 5 1 1


# Shorthand for sample_df[sample_df$b == sample_df$c, ]
subset(sample_df, b == c)
#>    a b c
#> 1 1 5 5
#> 5 5 1 1
```

The core of our version of `subset()`, `subset2()`, is quite simple. It takes two arguments: a data frame, `data`, and an expression, `rows`. We evaluate `rows` using `df` as a data mask, then use the results to subset the data frame with `[`. I've included a very simple check to ensure the result is a logical vector; real code would do more to create an informative error.

```
subset2 <- function(data, rows) {
  rows <- enquo(rows)
  rows_val <- eval_tidy(rows, data)
  stopifnot(is.logical(rows_val))

  data[rows_val, , drop = FALSE]
}

subset2(sample_df, b == c)
#>    a b c
#> 1 1 5 5
#> 5 5 1 1
```

4.4.4   Application: transform

A more complicated situation is `base::transform()` which allows you to add new variables to a data frame, evaluating their expressions in the context of the existing variables:

```
df <- data.frame(x = c(2, 3, 1), y = runif(3))
transform(df, x = -x, y2 = 2 * y)
#>    x      y     y2
#> 1 -2 0.0808 0.162
#> 2 -3 0.8343 1.669
#> 3 -1 0.6008 1.202
```

Again, our own `transform2()` requires little code. We capture the unevaluated `...` with `enquos(...)`, and then evaluate each expression using a for loop. Real code would do more error checking to ensure that each input is named and evaluates to a vector the same length as `data`.

```
transform2 <- function(.data, ...) {
  dots <- enquos(...)

  for (i in seq_along(dots)) {
    name <- names(dots)[[i]]
    dot <- dots[[i]]

    .data[[name]] <- eval_tidy(dot, .data)
  }

  .data
}

transform2(df, x2 = x * 2, y = -y)
#>   x        y x2
#> 1 2 -0.0808  4
#> 2 3 -0.8343  6
#> 3 1 -0.6008  2
```

NB: I named the first argument `.data` to avoid problems if the user tried to create a variable called `data`. They will still have problems if they attempt to create a variable called `.data`, but this is much less likely. This is the same reasoning that leads to the `.x` and `.f` arguments to `map()` (Section **??**).

4.4.5 Application: `select()`

A data mask will typically be a data frame, but it's sometimes useful to provide a list filled with more exotic contents. This is basically how the `select` argument in `base::subset()` works. It allows you to refer to variables as if they were numbers:

```
df <- data.frame(a = 1, b = 2, c = 3, d = 4, e = 5)
subset(df, select = b:d)
#>   b c d
#> 1 2 3 4
```

The key idea is to create a named list where each component gives the position of the corresponding variable:

```
vars <- as.list(set_names(seq_along(df), names(df)))
str(vars)
#> List of 5
#>  $ a: int 1
#>  $ b: int 2
#>  $ c: int 3
#>  $ d: int 4
#>  $ e: int 5
```

Then implementation is again only a few lines of code:

```
select2 <- function(data, ...) {
  dots <- enquos(...)

  vars <- as.list(set_names(seq_along(data), names(data)))
  cols <- unlist(map(dots, eval_tidy, vars))

  df[, cols, drop = FALSE]
}
select2(df, b:d)
#>   b c d
#> 1 2 3 4
```

dplyr::select() takes this idea and runs with it, providing a number of helpers that allow you to select variables based on their names (e.g. starts_with("x") or ends_with("_a")).

4.4.6   Exercises

1. Why did I use a for loop in transform2() instead of map()? Consider transform2(df, x = x * 2, x = x * 2).

2. Here's an alternative implementation of subset2():

   ```
   subset3 <- function(data, rows) {
     rows <- enquo(rows)
     eval_tidy(expr(data[!!rows, , drop = FALSE]), data = data)
   }

   df <- data.frame(x = 1:3)
   subset3(df, x == 1)
   ```

   Compare and contrast subset3() to subset2(). What are its advantages and disadvantages?

3. The following function implements the basics of `dplyr::arrange()`. Annotate each line with a comment explaining what it does. Can you explain why `!!.na.last` is strictly correct, but omitting the `!!` is unlikely to cause problems?

```
arrange2 <- function(.df, ..., .na.last = TRUE) {
  args <- enquos(...)

  order_call <- expr(order(!!!args, na.last = !!.na.last))

  ord <- eval_tidy(order_call, .df)
  stopifnot(length(ord) == nrow(.df))

  .df[ord, , drop = FALSE]
}
```

4.5 Using tidy evaluation

While it's important to understand how `eval_tidy()` works, most of the time you won't call it directly. Instead, you'll usually use it indirectly by calling a function that uses `eval_tidy()`. This section will give a few practical examples of wrapping functions that use tidy evaluation.

4.5.1 Quoting and unquoting

Imagine we have written a function that resamples a dataset:

```
resample <- function(df, n) {
  idx <- sample(nrow(df), n, replace = TRUE)
  df[idx, , drop = FALSE]
}
```

We want to create a new function that allows us to resample and subset in a single step. Our naive approach doesn't work:

```
subsample <- function(df, cond, n = nrow(df)) {
  df <- subset2(df, cond)
  resample(df, n)
}
```

```
df <- data.frame(x = c(1, 1, 1, 2, 2), y = 1:5)
```

```
subsample(df, x == 1)
#> Error in eval_tidy(rows, data): object 'x' not found
```

subsample() doesn't quote any arguments so cond is evaluated normally (not in a data mask), and we get an error when it tries to find a binding for x. To fix this problem we need to quote cond, and then unquote it when we pass it on ot subset2():

```
subsample <- function(df, cond, n = nrow(df)) {
  cond <- enquo(cond)

  df <- subset2(df, !!cond)
  resample(df, n)
}

subsample(df, x == 1)
#>     x y
#> 1   1 1
#> 1.1 1 1
#> 2   1 2
```

This is a very common pattern; whenever you call a quoting function with arguments from the user, you need to quote them yourself and then unquote.

4.5.2 Handling ambiguity

In the case above, we needed to think about tidy evaluation because of quasiquotation. We also need to think about tidy evaluation even when the wrapper doesn't need to quote any arguments. Take this wrapper around subset2():

```
threshold_x <- function(df, val) {
  subset2(df, x >= val)
}
```

This function can silently return an incorrect result in two situations:

- When x exists in the calling environment, but not in df:

  ```
  x <- 10
  no_x <- data.frame(y = 1:3)
  threshold_x(no_x, 2)
  #>   y
  #> 1 1
  ```

```
#> 2 2
#> 3 3
```

- When `val` exists in `df`:

```
has_val <- data.frame(x = 1:3, val = 9:11)
threshold_x(has_val, 2)
#> [1] x    val
#> <0 rows> (or 0-length row.names)
```

These failure modes arise because tidy evaluation is ambiguous: each variable can be found in **either** the data mask **or** the environment. To make this function safe we need to remove the ambiguity using the `.data` and `.env` pronouns:

```
threshold_x <- function(df, val) {
  subset2(df, .data$x >= .env$val)
}
```

```
x <- 10
threshold_x(no_x, 2)
#> Error: Column `x` not found in `.data`
threshold_x(has_val, 2)
#>   x val
#> 2 2  10
#> 3 3  11
```

Generally, whenever you use the `.env` pronoun, you can use unquoting instead:

```
threshold_x <- function(df, val) {
  subset2(df, .data$x >= !!val)
}
```

There are subtle differences in when `val` is evaluated. If you unquote, `val` will be early evaluated by `enquo()`; if you use a pronoun, `val` will be lazily evaluated by `eval_tidy()`. These differences are usually unimportant, so pick the form that looks most natural.

4.5.3 Quoting and ambiguity

To finish our discussion let's consider the case where we have both quoting and potential ambiguity. I'll generalise `threshold_x()` slightly so that the user can pick the variable used for thresholding. Here I used `.data[[var]]` because it

makes the code a little simpler; in the exercises you'll have a chance to explore
how you might use `$` instead.

```r
threshold_var <- function(df, var, val) {
  var <- as_string(ensym(var))
  subset2(df, .data[[var]] >= !!val)
}


df <- data.frame(x = 1:10)
threshold_var(df, x, 8)
#>       x
#> 8     8
#> 9     9
#> 10   10
```

It is not always the responsibility of the function author to avoid ambiguity.
Imagine we generalise further to allow thresholding based on any expression:

```r
threshold_expr <- function(df, expr, val) {
  expr <- enquo(expr)
  subset2(df, !!expr >= !!val)
}
```

It's not possible to evaluate `expr` only the data mask, because the data mask
doesn't include any functions like `+` or `==`. Here, it's the user's responsibility
to avoid ambiguity. As a general rule of thumb, as a function author it's your
responsibility to avoid ambiguity with any expressions that you create; it's the
user's responsibility to avoid ambiguity in expressions that they create.

4.5.4   Exercises

1. I've included an alternative implementation of `threshold_var()` below.
   What makes it different to the approach I used above? What make it
   harder?

   ```r
   threshold_var <- function(df, var, val) {
     var <- ensym(var)
     subset2(df, `$`(.data, !!var) >= !!val)
   }
   ```

4.6   Base evaluation

Now that you understand tidy evaluation, it's time to come back to the alternative approaches taken by base R. Here I'll explore the two most common uses in base R:

- `substitute()` and evaluation in the caller environment, as used by `subset()`. I'll use this technique to motivate why this technique is not programming friendly, as warned about in the `subset()` documentation.

- `match.call()`, call manipulation, and evaluation in the caller environment, as used by `write.csv()` and `lm()`. I'll use this technique to motivate how quasiquotation and (regular) evaluation can help you write wrappers such functions.

These two approaches are common forms of non-standard evaluation (NSE).

4.6.1   `substitute()`

The most common form of NSE in base R is `substitute()` + `eval()`. The following code shows how you might write the core of `subset()` in this style using `substitute()` and `eval()` rather than `enquo()` and `eval_tidy()`. I repeat the code introduced in Section 4.4.3 so you can compare easily. The main difference is the evaluation environment: in `subset_base()` the argument is evaluated in the caller environment, while in `subset_tidy()`, it's evaluated in the environment where it was defined.

```
subset_base <- function(data, rows) {
  rows <- substitute(rows)
  rows_val <- eval(rows, data, caller_env())
  stopifnot(is.logical(rows_val))

  data[rows_val, , drop = FALSE]
}

subset_tidy <- function(data, rows) {
  rows <- enquo(rows)
  rows_val <- eval_tidy(rows, data)
  stopifnot(is.logical(rows_val))

  data[rows_val, , drop = FALSE]
}
```

4.6.1.1   Programming with `subset()`

The documentation of `subset()` includes the following warning:

> This is a convenience function intended for use interactively. For
> programming it is better to use the standard subsetting functions like
> [, and in particular the non-standard evaluation of argument `subset`
> can have unanticipated consequences.

There are main three problems:

- `base::subset()` always evaluates `rows` in the calling environment, but
  if `...` has been used, then the expression might need to be evaluated
  elsewhere:

```
f1 <- function(df, ...) {
  xval <- 3
  subset_base(df, ...)
}

my_df <- data.frame(x = 1:3, y = 3:1)
xval <- 1
f1(my_df, x == xval)
#>   x y
#> 3 3 1
```

  This may seems like an esoteric concern, but it means that `subset_base()`
  cannot reliably work with functionals like `map()` or `lapply()`:

```
local({
  zzz <- 2
  dfs <- list(data.frame(x = 1:3), data.frame(x = 4:6))
  lapply(dfs, subset_base, x == zzz)
})
#> Error in eval(rows, data, caller_env()): object 'zzz' not found
```

- Calling `subset()` from another function requires some care: you have to use
  `substitute()` to capture a call to `subset()` complete expression, and then
  evaluate. I think this code is hard to understand because `substitute()`
  doesn't use a syntactic marker for unquoting. Here I print the generated
  call to make it a little easier to see what's happening.

```
f2 <- function(df1, expr) {
  call <- substitute(subset_base(df1, expr))
  expr_print(call)
  eval(call, caller_env())
}

my_df <- data.frame(x = 1:3, y = 3:1)
f2(my_df, x == 1)
#> subset_base(my_df, x == 1)
#>   x y
#> 1 1 3
```

- `eval()` doesn't provide any pronouns so there's no way to require part of the expression to come from the data. As far as I can tell, there's no way to make the following function safe except by manually checking for the presence of `z` variable in `df`.

```
f3 <- function(df) {
  call <- substitute(subset_base(df, z > 0))
  expr_print(call)
  eval(call, caller_env())
}

my_df <- data.frame(x = 1:3, y = 3:1)
z <- -1
f3(my_df)
#> subset_base(my_df, z > 0)
#> [1] x y
#> <0 rows> (or 0-length row.names)
```

4.6.1.2   What about [?

Given that tidy evaluation is quite complex, why not simply use `[` as `?subset` recommends? Primarily, it seems unappealing to me to have functions that can only be used interactively, and never inside another function.

Additionally, even the simple `subset()` function provides two useful features compared to `[`:

- It sets `drop = FALSE` by default, so it's guaranteed to return a data frame.

- It drops rows where the condition evaluates to `NA`.

That means `subset(df, x == y)` is not equivalent to `df[x == y,]` as you might expect. Instead, it is equivalent to `df[x == y & !is.na(x == y), ,` `drop = FALSE]`: that's a lot more typing! Real-life alternatives to `subset()`, like `dplyr::filter()`, do even more. For example, `dplyr::filter()` can translate R expressions to SQL so that they can be executed in a database. This makes programming with `filter()` relatively more important.

4.6.2 `match.call()`

Another common form of NSE is to capture the complete call with `match.call()`, modify it, and evaluate the result. `match.call()` is similar to `substitute()`, but instead of capturing a single argument, it captures the complete call. It doesn't have an equivalent in rlang.

```
g <- function(x, y, z) {
  match.call()
}
g(1, 2, z = 3)
#> g(x = 1, y = 2, z = 3)
```

One prominent user of `match.call()` is `write.csv()`, which basically works by transforming the call into a call to `write.table()` with the appropriate arguments set. The following code shows the heart of `write.csv()`:

```
write.csv <- function(...) {
  call <- match.call(write.table, expand.dots = TRUE)

  call[[1]] <- quote(write.table)
  call$sep <- ","
  call$dec <- "."

  eval(call, parent.frame())
}
```

I don't think this technique is a good idea because you can achieve the same result without NSE:

```
write.csv <- function(...) {
  write.table(..., sep = ",", dec = ".")
}
```

Nevertheless, it's important to understand this technique because it's commonly used in modelling functions. These functions also prominently print the captured call, which poses some special challenges, as you'll see next.

4.6.2.1 Wrapping modelling functions

To begin, consider the simplest possible wrapper around `lm()`:

```
lm2 <- function(formula, data) {
  lm(formula, data)
}
```

This wrapper works, but is suboptimal because `lm()` captures its call and displays it when printing.

```
lm2(mpg ~ disp, mtcars)
#>
#> Call:
#> lm(formula = formula, data = data)
#>
#> Coefficients:
#> (Intercept)          disp
#>     29.5999       -0.0412
```

Fixing this is important because this call is the chief way that you see the model specification when printing the model. To overcome this problem, we need to capture the arguments, create the call to `lm()` using unquoting, then evaluate that call. To make it easier to see what's going on, I'll also print the expression we generate. This will become more useful as the calls get more complicated.

```
lm3 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  data <- enexpr(data)

  lm_call <- expr(lm(!!formula, data = !!data))
  expr_print(lm_call)
  eval(lm_call, env)
}

lm3(mpg ~ disp, mtcars)
#> lm(mpg ~ disp, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ disp, data = mtcars)
#>
#> Coefficients:
```

```
#> (Intercept)         disp
#>     29.5999      -0.0412
```

There are three pieces that you'll use whenever wrapping a base NSE function in this way:

- You capture the unevaluated arguments using `enexpr()`, and capture the caller environment using `caller_env()`.

- You generate a new expression using `expr()` and unquoting.

- You evaluate that expression in the caller environment. You have to accept that the function will not work correctly if the arguments are not defined in the caller environment. Providing the `env` argument at least provides a hook that experts can use if the default environment isn't correct.

The use of `enexpr()` has a nice side-effect: we can use unquoting to generate formulas dynamically:

```
resp <- expr(mpg)
disp1 <- expr(vs)
disp2 <- expr(wt)
lm3(!!resp ~ !!disp1 + !!disp2, mtcars)
#> lm(mpg ~ vs + wt, data = mtcars)
#>
#> Call:
#> lm(formula = mpg ~ vs + wt, data = mtcars)
#>
#> Coefficients:
#> (Intercept)           vs           wt
#>       33.00         3.15        -4.44
```

4.6.2.2   The evaluation environment

What if you want to mingle objects supplied by the user with objects that you create in the function? For example, imagine you want to make an auto-resampling version of `lm()`. You might write it like this:

```
resample_lm0 <- function(formula, data, env = caller_env()) {
  formula <- enexpr(formula)
  resample_data <- resample(data, n = nrow(data))

  lm_call <- expr(lm(!!formula, data = resample_data))
```

```
  expr_print(lm_call)
  eval(lm_call, env)
}


df <- data.frame(x = 1:10, y = 5 + 3 * (1:10) + round(rnorm(10), 2))
resample_lm0(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#> Error in is.data.frame(data): object 'resample_data' not found
```

Why doesn't this code work? We're evaluating `lm_call` in the caller environment, but `resample_data` exists in the execution environment. We could instead evaluate in the execution environment of `resample_lm0()`, but there's no guarantee that `formula` could be evaluated in that environment.

There are two basic ways to overcome this challenge:

1. Unquote the data frame into the call. This means that no lookup has to occur, but has all the problems of inlining expressions (Section 3.4.7). For modelling functions this means that the captured call is suboptimal:

   ```
   resample_lm1 <- function(formula, data, env = caller_env()) {
     formula <- enexpr(formula)
     resample_data <- resample(data, n = nrow(data))

     lm_call <- expr(lm(!!formula, data = !!resample_data))
     expr_print(lm_call)
     eval(lm_call, env)
   }
   resample_lm1(y ~ x, data = df)$call
   #> lm(y ~ x, data = <data.frame>)
   #> lm(formula = y ~ x, data = list(x = c(8L, 10L, 2L, 3L, 7L, 5L,
   #> 7L, 7L, 1L, 8L), y = c(28.45, 37.07, 11.62, 15.15, 25.72, 19.75,
   #> 25.72, 25.72, 7.99, 28.45)))
   ```

2. Alternatively you can create a new environment that inherits from the caller, and bind variables that you've created inside the function to that environment.

   ```
   resample_lm2 <- function(formula, data, env = caller_env()) {
     formula <- enexpr(formula)
     resample_data <- resample(data, n = nrow(data))

     lm_env <- env(env, resample_data = resample_data)
   ```

```
  lm_call <- expr(lm(!!formula, data = resample_data))
  expr_print(lm_call)
  eval(lm_call, lm_env)
}
resample_lm2(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#>
#> Call:
#> lm(formula = y ~ x, data = resample_data)
#>
#> Coefficients:
#> (Intercept)              x
#>        3.06           3.30
```

This is more work, but gives the cleanest specification.

4.6.3   Exercises

1. Why does this function fail?

```
lm3a <- function(formula, data) {
  formula <- enexpr(formula)

  lm_call <- expr(lm(!!formula, data = data))
  eval(lm_call, caller_env())
}
lm3a(mpg ~ disp, mtcars)$call
#> Error in as.data.frame.default(data, optional = TRUE): cannot coerce
#> class '"function"' to a data.frame
```

2. When model building, typically the response and data are relatively constant while you rapidly experiment with different predictors. Write a small wrapper that allows you to reduce duplication in the code below.

```
lm(mpg ~ disp, data = mtcars)
lm(mpg ~ I(1 / disp), data = mtcars)
lm(mpg ~ disp * cyl, data = mtcars)
```

3. Another way to way to write `resample_lm()` would be to include the resmaple expression (`data[sample(nrow(data), replace = TRUE), , drop = FALSE]`) in the data argument. Implement that approach. What are the advantages? What are the disadvantages?

Chapter 5

Translating R code

5.1 Introduction

The combination of first-class environments, lexical scoping, and metaprogramming gives us a powerful toolkit for translating R code into other languages. One fully-fledged example of this idea is dbplyr, which powers the database backends for dplyr, allowing you to express data manipulation in R and automatically translate it into SQL. You can see the key idea in `translate_sql()` which takes R code and returns the equivalent SQL:

```
library(dbplyr)
translate_sql(x ^ 2)
#> <SQL> POWER("x", 2.0)
translate_sql(x < 5 & !is.na(x))
#> <SQL> "x" < 5.0 AND NOT((("x") IS NULL))
translate_sql(!first %in% c("John", "Roger", "Robert"))
#> <SQL> NOT("first" IN ('John', 'Roger', 'Robert'))
translate_sql(select == 7)
#> <SQL> "select" = 7.0
```

Translating R to SQL is complex because of the many idiosyncrasies of SQL dialects, so here I'll develop two simple, but useful, domain specific languages (DSL): one to generate HTML, and the other to generate mathematical equations in LaTeX.

If you're interested in learning more about domain specific languages in general, I highly recommend *Domain Specific Languages* (Fowler 2010). It discusses many options for creating a DSL and provides many examples of different languages.

Outline

- Section 5.2 creates a DSL for generating HTML, using quasiquotation and purrr to generate a function for each HTML tag, then tidy evaluation to easily access them.

- Section 5.3 transforms mathematically R code into its LaTeX equivalent using a combination of tidy evaluation and expression walking.

Prerequisites

This chapter pulls together many techniques discussed elsewhere in the book. In particular, you'll need to understand environments, expressions, tidy evaluation, and a little functional programming and S3. We'll use rlang (https://rlang.r-lib.org) for metaprogramming tools, and purrr (https://purrr.tidyverse.org) for functional programming.

```
library(rlang)
library(purrr)
```

5.2   HTML

HTML (HyperText Markup Language) is the language that underlies the majority of the web. It's a special case of SGML (Standard Generalised Markup Language), and it's similar but not identical to XML (eXtensible Markup Language). HTML looks like this:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Even if you've never looked at HTML before, you can still see that the key component of its coding structure is tags, which look like `<tag></tag>` or `<tag />`. Tags can be nested within other tags and intermingled with text. There are over 100 HTML tags, but in this chapter we'll focus on just a handful:

- `<body>` is the top-level tag that contains all content.
- `<h1>` defines a top level heading.
- `<p>` defines a paragraph.
- `<b>` emboldens text.
- `<img>` embeds an image.

Tags can have named **attributes** which look like `<tag name1='value1' name2='value2'></tag>`. Two of the most important attributes are `id` and `class`, which are used in conjunction with CSS (Cascading Style Sheets) to control the visual appearance of the page.

Void tags, like `<img>`, don't have any children, and are written `<img />`, not `<img></img>`. Since they have no content, attributes are more important, and `img` has three that are used with almost every image: `src` (where the image lives), `width`, and `height`.

Because `<` and `>` have special meanings in HTML, you can't write them directly. Instead you have to use the HTML **escapes**: `>` and `<`. And since those escapes use `&`, if you want a literal ampersand you have to escape it as `&`.

5.2.1 Goal

Our goal is to make it easy to generate HTML from R. To give a concrete example, we want to generate the following HTML:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Using the following code that matches the structure of the HTML as closely as possible:

```
with_html(
  body(
    h1("A heading", id = "first"),
    p("Some text &", b("some bold text.")),
    img(src = "myimg.png", width = 100, height = 100)
  )
)
```

This DSL has the following three properties:

- The nesting of function calls matches the nesting of tags.

- Unnamed arguments become the content of the tag, and named arguments become their attributes.

- `&` and other special characters are automatically escaped.

5.2.2   Escaping

Escaping is so fundamental to translation that it'll be our first topic. There are
two related challenges:

- In user input, we need to automatically escape &, < and >.

- At the same time we need to make sure that the &, < and > we generate
  are not double-escaped (i.e. that we don't accidentally generate &amp;,
  &lt; and &gt;).

The easiest way to do this is to create an S3 class (Section **??**) that distin-
guishes between regular text (that needs escaping) and HTML (that doesn't).

```r
html <- function(x) structure(x, class = "advr_html")

print.advr_html <- function(x, ...) {
  out <- paste0("<HTML> ", x)
  cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}
```

We then write an escape generic. It has two important methods:

- `escape.character()` takes a regular character vector and returns an
  HTML vector with special characters (&, <, >) escaped.

- `escape.advr_html()` leaves already escaped HTML alone.

```r
escape <- function(x) UseMethod("escape")

escape.character <- function(x) {
  x <- gsub("&", "&amp;", x)
  x <- gsub("<", "&lt;", x)
  x <- gsub(">", "&gt;", x)

  html(x)
}

escape.advr_html <- function(x) x
```

Now we check that it works

```r
escape("This is some text.")
#> <HTML> This is some text.
```

```
escape("x > 1 & y < 2")
#> <HTML> x &gt; 1 &amp; y &lt; 2

# Double escaping is not a problem
escape(escape("This is some text. 1 > 2"))
#> <HTML> This is some text. 1 &gt; 2

# And text we know is HTML doesn't get escaped.
escape(html("<hr />"))
#> <HTML> <hr />
```

Conveniently, this also allows a user to opt out of our escaping if they know the content is already escaped.

5.2.3 Basic tag functions

Next, we'll write a one-tag function by hand, then figure out how to generalise it so we can generate a function for every tag with code.

Let's start with `<p>`. HTML tags can have both attributes (e.g., id or class) and children (like `<b>` or `<i>`). We need some way of separating these in the function call. Given that attributes are named and children are not, it seems natural to use named and unnamed arguments for them respectively. For example, a call to `p()` might look like:

```
p("Some text. ", b(i("some bold italic text")), class = "mypara")
```

We could list all the possible attributes of the `<p>` tag in the function definition, but that's hard because there are many attributes, and because it's possible to use custom attributes (http://html5doctor.com/html5-custom-data-attributes/). Instead, we'll use `...` and separate the components based on whether or not they are named. With this in mind, we create a helper function that wraps around `rlang::list2()` (Section 3.6) and returns named and unnamed components separately:

```
dots_partition <- function(...) {
  dots <- list2(...)

  is_named <- names(dots) != ""
  list(
    named = dots[is_named],
    unnamed = dots[!is_named]
  )
```

```
}
```

```
str(dots_partition(a = 1, 2, b = 3, 4))
#> List of 2
#>  $ named  :List of 2
#>   ..$ a: num 1
#>   ..$ b: num 3
#>  $ unnamed:List of 2
#>   ..$ : num 2
#>   ..$ : num 4
```

We can now create our `p()` function. Notice that there's one new function here: `html_attributes()`. It takes a named list and returns the HTML attribute specification as a string. It's a little complicated (in part, because it deals with some idiosyncrasies of HTML that I haven't mentioned here), but it's not that important and doesn't introduce any new programming ideas, so I won't discuss it in detail. You can find the source online (https://github.com/hadley/adv-r/blob/master/dsl-html-attributes.r) if you want to work through it yourself.

```
source("dsl-html-attributes.r")
p <- function(...) {
  dots <- dots_partition(...)
  attribs <- html_attributes(dots$named)
  children <- map_chr(dots$unnamed, escape)

  html(paste0(
    "<p", attribs, ">",
    paste(children, collapse = ""),
    "</p>"
  ))
}

p("Some text")
#> <HTML> <p></p>
p("Some text", id = "myid")
#> <HTML> <p id='myid'>Some text</p>
p("Some text", class = "important", `data-value` = 10)
#> <HTML> <p class='important' data-value='10'>Some text</p>
```

5.2.4 Tag functions

It's straightforward to adapt `p()` to other tags: we just need to replace `"p"` with the name of the tag. One elegant way to do that is to create a function with `rlang::new_function()` (Section 3.7.4), using unquoting and `paste0()` to generate the starting and ending tags.

```r
tag <- function(tag) {
  new_function(
    exprs(... = ),
    expr({
      dots <- dots_partition(...)
      attribs <- html_attributes(dots$named)
      children <- map_chr(dots$unnamed, escape)

      html(paste0(
        !!paste0("<", tag), attribs, ">",
        paste(children, collapse = ""),
        !!paste0("</", tag, ">")
      ))
    }),
    caller_env()
  )
}
tag("b")
#> function (...)
#> {
#>     dots <- dots_partition(...)
#>     attribs <- html_attributes(dots$named)
#>     children <- map_chr(dots$unnamed, escape)
#>     html(paste0("<b", attribs, ">", paste(children, collapse = ""),
#>         "</b>"))
#> }
```

We need the weird `exprs(... = )` syntax to generate the empty `...` argument in the tag function. See Section 2.6.2 for more details.

Now we can run our earlier example:

```r
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text. ", b(i("some bold italic text")), class = "mypara")
#> <HTML> <p class='mypara'>Some text. <b></b></p>
```

Before we generate functions for every possible HTML tag, we need to create a variant that handles void tags. `void_tag()` is quite similar to `tag()`, but it throws an error if there are any unnamed tags, and the tag itself looks a little different.

```r
void_tag <- function(tag) {
  new_function(
    exprs(... = ),
    expr({
      dots <- dots_partition(...)
      if (length(dots$unnamed) > 0) {
        abort(!!paste0("<", tag, "> must not have unnamed arguments"))
      }
      attribs <- html_attributes(dots$named)

      html(paste0(!!paste0("<", tag), attribs, " />"))
    }),
    caller_env()
  )
}

img <- void_tag("img")
img
#> function (...)
#> {
#>     dots <- dots_partition(...)
#>     if (length(dots$unnamed) > 0) {
#>         abort("<img> must not have unnamed arguments")
#>     }
#>     attribs <- html_attributes(dots$named)
#>     html(paste0("<img", attribs, " />"))
#> }
img(src = "myimage.png", width = 100, height = 100)
#> <HTML> <img src='myimage.png' width='100' height='100' />
```

5.2.5   Processing all tags

Next we need to generate these functions for every tag. We'll start with a list of all HTML tags:

```r
tags <- c("a", "abbr", "address", "article", "aside", "audio",
  "b","bdi", "bdo", "blockquote", "body", "button", "canvas",
  "caption","cite", "code", "colgroup", "data", "datalist",
  "dd", "del","details", "dfn", "div", "dl", "dt", "em",
  "eventsource","fieldset", "figcaption", "figure", "footer",
  "form", "h1", "h2", "h3", "h4", "h5", "h6", "head", "header",
  "hgroup", "html", "i","iframe", "ins", "kbd", "label",
  "legend", "li", "mark", "map","menu", "meter", "nav",
  "noscript", "object", "ol", "optgroup", "option", "output",
  "p", "pre", "progress", "q", "ruby", "rp","rt", "s", "samp",
  "script", "section", "select", "small", "span", "strong",
  "style", "sub", "summary", "sup", "table", "tbody", "td",
  "textarea", "tfoot", "th", "thead", "time", "title", "tr",
  "u", "ul", "var", "video"
)

void_tags <- c("area", "base", "br", "col", "command", "embed",
  "hr", "img", "input", "keygen", "link", "meta", "param",
  "source", "track", "wbr"
)
```

If you look at this list carefully, you'll see there are quite a few tags that have the same name as base R functions (`body`, `col`, `q`, `source`, `sub`, `summary`, `table`). This means we don't want to make all the functions available by default, either in the global environment or in a package. Instead, we'll put them in a list (like in Section **??**) and then provide a helper to make it easy to use them when desired. First, we make a named list containing all the tag functions:

```r
html_tags <- c(
  tags %>% set_names() %>% map(tag),
  void_tags %>% set_names() %>% map(void_tag)
)
```

This gives us an explicit (but verbose) way to create HTML:

```r
html_tags$p(
  "Some text. ",
  html_tags$b(html_tags$i("some bold italic text")),
  class = "mypara"
)
#> <HTML> <p class='mypara'>Some text. <b></b></p>
```

We can then finish off our HTML DSL with a function that allows us to evaluate code in the context of that list. Here we slightly abuse the data mask, passing it a list of functions rather than a data frame. This is quick hack to mingle the execution environment of `code` with the functions in `html_tags`.

```
with_html <- function(code) {
  code <- enquo(code)
  eval_tidy(code, html_tags)
}
```

This gives us a succinct API which allows us to write HTML when we need it but doesn't clutter up the namespace when we don't.

```
with_html(
  body(
    h1("A heading", id = "first"),
    p("Some text &", b("some bold text.")),
    img(src = "myimg.png", width = 100, height = 100)
  )
)
#> <HTML> <body></body>
```

If you want to access the R function overridden by an HTML tag with the same name inside `with_html()`, you can use the full `package::function` specification.

5.2.6   Exercises

1. The escaping rules for `<script>` tags are different because they contain JavaScript, not HTML. Instead of escaping angle brackets or ampersands, you need to escape `</script>` so that the tag isn't closed too early. For example, `script("'</script>'")`, shouldn't generate this:

   ```
   <script>'</script>'</script>
   ```

   But

   ```
   <script>'<\/script>'</script>
   ```

   Adapt the `escape()` to follow these rules when a new argument `script` is set to `TRUE`.

2. The use of `...` for all functions has some big downsides. There's no input validation and there will be little information in the documentation or autocomplete about how they are used in the function. Create a new function

that, when given a named list of tags and their attribute names (like below), creates tag functions with named arguments.

```
list(
  a = c("href"),
  img = c("src", "width", "height")
)
```

All tags should get `class` and `id` attributes.

3. Reason about the following code that calls `with_html()` referencing objects from the environment. Will it work or fail? Why? Run the code to verify your predictions.

```
greeting <- "Hello!"
with_html(p(greeting))

p <- function() "p"
address <- "123 anywhere street"
with_html(p(address))
```

4. Currently the HTML doesn't look terribly pretty, and it's hard to see the structure. How could you adapt `tag()` to do indenting and formatting? (You may need to do some research into block vs inline tags.)

5.3 LaTeX

The next DSL will convert R expressions into their LaTeX math equivalents. (This is a bit like `?plotmath`, but for text instead of plots.) LaTeX is the lingua franca of mathematicians and statisticians: it's common to use LaTeX notation whenever you want to express an equation in text, like in email. Since many reports are produced using both R and LaTeX, it might be useful to be able to automatically convert mathematical expressions from one language to the other.

Because we need to convert both functions and names, this mathematical DSL will be more complicated than the HTML DSL. We'll also need to create a "default" conversion, so that symbols that we don't know about get a standard conversion. This means that we can no longer use just evaluation: we also need to walk the abstract syntax tree (AST).

5.3.1 LaTeX mathematics

Before we begin, let's quickly cover how formulas are expressed in LaTeX. The full standard is very complex, but fortunately is well documented

(http://en.wikibooks.org/wiki/LaTeX/Mathematics), and the most common commands have a fairly simple structure:

- Most simple mathematical equations are written in the same way you'd type them in R: `x * y, z ^ 5`. Subscripts are written using `_` (e.g., `x_1`).

- Special characters start with a `\`: `\pi =` , `\pm = ±`, and so on. There are a huge number of symbols available in LaTeX: searching online for `latex math symbols` returns many lists (http://www.sunilpatel.co.uk/latex-type/latex-math-symbols/). There's even a service (http://detexify.kirelabs.org/classify.html) that will look up the symbol you sketch in the browser.

- More complicated functions look like `\name{arg1}{arg2}`. For example, to write a fraction you'd use `\frac{a}{b}`. To write a square root, you'd use `\sqrt{a}`.

- To group elements together use `{}`: i.e., `x ^ a + b` vs. `x ^ {a + b}`.

- In good math typesetting, a distinction is made between variables and functions. But without extra information, LaTeX doesn't know whether `f(a * b)` represents calling the function `f` with input `a * b`, or is shorthand for `f * (a * b)`. If `f` is a function, you can tell LaTeX to typeset it using an upright font with `\textrm{f}(a * b)`. (The `rm` stands for "Roman", the opposite of italics.)

5.3.2 Goal

Our goal is to use these rules to automatically convert an R expression to its appropriate LaTeX representation. We'll tackle this in four stages:

- Convert known symbols: `pi` → `\pi`

- Leave other symbols unchanged: `x` → `x`, `y` → `y`

- Convert known functions to their special forms: `sqrt(frac(a, b))` → `\sqrt{\frac{a, b}}`

- Wrap unknown functions with `\textrm`: `f(a)` → `\textrm{f}(a)`

We'll code this translation in the opposite direction of what we did with the HTML DSL. We'll start with infrastructure, because that makes it easy to experiment with our DSL, and then work our way back down to generate the desired output.

5.3.3 `to_math()`

To begin, we need a wrapper function that will convert R expressions into LaTeX math expressions. This will work like `to_html()` by capturing the unevaluated expression and evaluating it in a special environment. There are two main differences:

- The evaluation environment is no longer constant, as it has to vary depending on the input. This is necessary to handle unknown symbols and functions.

- We never evaluate in the argument environment because we're translating every function to a LaTeX expression. The user will need to use explicitly `!!` in order to evaluate normally.

This gives us:

```r
to_math <- function(x) {
  expr <- enexpr(x)
  out <- eval_bare(expr, latex_env(expr))

  latex(out)
}

latex <- function(x) structure(x, class = "advr_latex")
print.advr_latex <- function(x) {
  cat("<LATEX> ", x, "\n", sep = "")
}
```

Next we'll build up `latex_env()`, starting simply and getting progressively more complex.

5.3.4 Known symbols

Our first step is to create an environment that will convert the special LaTeX symbols used for Greek character, e.g., `pi` to `\pi`. We'll use the trick from Section 4.4.3 to bind the symbol `pi` to the value `"\pi"`.

```r
greek <- c(
  "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
  "gamma", "varpi", "phi", "delta", "kappa", "rho",
  "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
  "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
  "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi",
```

```
  "Upsilon", "Omega", "Theta", "Pi", "Phi"
)
greek_list <- set_names(paste0("\\", greek), greek)
greek_env <- as_environment(greek_list)
```

We can then check it:

```
latex_env <- function(expr) {
  greek_env
}
```

```
to_math(pi)
#> <LATEX> \pi
to_math(beta)
#> <LATEX> \beta
```

Looks good so far!

5.3.5   Unknown symbols

If a symbol isn't Greek, we want to leave it as is. This is tricky because we don't know in advance what symbols will be used, and we can't possibly generate them all. Instead, we'll use the approach described in Section 2.5: walking the AST and to find all symbols. This gives us `all_names_rec()` and helper `all_names()`:

```
all_names_rec <- function(x) {
  switch_expr(x,
    constant = character(),
    symbol =   as.character(x),
    call =     flat_map_chr(as.list(x[-1]), all_names)
  )
}
```

```
all_names <- function(x) {
  unique(all_names_rec(x))
}
```

```
all_names(expr(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
```

We now want to take that list of symbols and convert it to an environment so that each symbol is mapped to its corresponding string representation (e.g.,

so `eval(quote(x), env)` yields `"x"`). We again use the pattern of converting a named character vector to a list, then converting the list to an environment.

```r
latex_env <- function(expr) {
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names))

  symbol_env
}


to_math(x)
#> <LATEX> x
to_math(longvariablename)
#> <LATEX> longvariablename
to_math(pi)
#> <LATEX> pi
```

This works, but we need to combine it with the Greek symbols environment. Since we want to give preference to Greek over defaults (e.g., `to_math(pi)` should give `"\\pi"`, not `"pi"`), `symbol_env` needs to be the parent of `greek_env`. To do that, we need to make a copy of `greek_env` with a new parent. This gives us a function that can convert both known (Greek) and unknown symbols.

```r
latex_env <- function(expr) {
  # Unknown symbols
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names))

  # Known symbols
  env_clone(greek_env, parent = symbol_env)
}


to_math(x)
#> <LATEX> x
to_math(longvariablename)
#> <LATEX> longvariablename
to_math(pi)
#> <LATEX> \pi
```

5.3.6   Known functions

Next we'll add functions to our DSL. We'll start with a couple of helpers that make it easy to add new unary and binary operators. These functions are very

simple: they only assemble strings.

```r
unary_op <- function(left, right) {
  new_function(
    exprs(e1 = ),
    expr(
      paste0(!!left, e1, !!right)
    ),
    caller_env()
  )
}

binary_op <- function(sep) {
  new_function(
    exprs(e1 = , e2 = ),
    expr(
      paste0(e1, !!sep, e2)
    ),
    caller_env()
  )
}

unary_op("\\sqrt{", "}")
#> function (e1)
#> paste0("\\sqrt{", e1, "}")
binary_op("+")
#> function (e1, e2)
#> paste0(e1, "+", e2)
```

Using these helpers, we can map a few illustrative examples of converting R to LaTeX. Note that with R's lexical scoping rules helping us, we can easily provide new meanings for standard functions like +, -, and *, and even (and {.

```r
# Binary operators
f_env <- child_env(
  .parent = empty_env(),
  `+` = binary_op(" + "),
  `-` = binary_op(" - "),
  `*` = binary_op(" * "),
  `/` = binary_op(" / "),
  `^` = binary_op("^"),
  `[` = binary_op("_"),
```

```
  # Grouping
  `{` = unary_op("\\left{ ", " \\right}"),
  `(` = unary_op("\\left( ", " \\right)"),
  paste = paste,

  # Other math functions
  sqrt = unary_op("\\sqrt{", "}"),
  sin =  unary_op("\\sin(", ")"),
  log =  unary_op("\\log(", ")"),
  abs =  unary_op("\\left| ", "\\right| "),
  frac = function(a, b) {
    paste0("\\frac{", a, "}{", b, "}")
  },

  # Labelling
  hat =   unary_op("\\hat{", "}"),
  tilde = unary_op("\\tilde{", "}")
)
```

We again modify `latex_env()` to include this environment. It should be the last environment R looks for names in so that expressions like `sin(sin)` will work.

```
latex_env <- function(expr) {
  # Known functions
  f_env

  # Default symbols
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names), parent = f_env)

  # Known symbols
  greek_env <- env_clone(greek_env, parent = symbol_env)

  greek_env
}

to_math(sin(x + pi))
#> <LATEX> \sin(x + \pi)
to_math(log(x[i]^2))
#> <LATEX> \log(x_i^2)
```

```
to_math(sin(sin))
#> <LATEX> \sin(sin)
```

5.3.7   Unknown functions

Finally, we'll add a default for functions that we don't yet know about. Like the unknown names, we can't know in advance what these will be, so we again walk the AST to find them:

```
all_calls_rec <- function(x) {
  switch_expr(x,
    constant = ,
    symbol =   character(),
    call = {
      fname <- as.character(x[[1]])
      children <- flat_map_chr(as.list(x[-1]), all_calls)
      c(fname, children)
    }
  )
}
all_calls <- function(x) {
  unique(all_calls_rec(x))
}


all_calls(expr(f(g + b, c, d(a))))
#> [1] "f" "+" "d"
```

We need a closure that will generate the functions for each unknown call:

```
unknown_op <- function(op) {
  new_function(
    exprs(... = ),
    expr({
      contents <- paste(..., collapse = ", ")
      paste0(!!paste0("\\mathrm{", op, "}("), contents, ")")
    })
  )
}
unknown_op("foo")
#> function (...)
#> {
#>     contents <- paste(..., collapse = ", ")
```

```
#>      paste0("\\mathrm{foo}(", contents, ")")
#> }
#> <environment: 0x63b4aa8>
```

And again we update `latex_env()`:

```
latex_env <- function(expr) {
  calls <- all_calls(expr)
  call_list <- map(set_names(calls), unknown_op)
  call_env <- as_environment(call_list)

  # Known functions
  f_env <- env_clone(f_env, call_env)

  # Default symbols
  names <- all_names(expr)
  symbol_env <- as_environment(set_names(names), parent = f_env)

  # Known symbols
  greek_env <- env_clone(greek_env, parent = symbol_env)
  greek_env
}
```

This completes our original requirements:

```
to_math(sin(pi) + f(a))
#> <LATEX> \sin(\pi) + \mathrm{f}(a)
```

You could certainly take this idea further and translate types of mathematical expression, but you should not need any additional metaprogramming tools.

5.3.8   Exercises

1. Add escaping. The special symbols that should be escaped by adding a backslash in front of them are \, $, and %. Just as with HTML, you'll need to make sure you don't end up double-escaping. So you'll need to create a small S3 class and then use that in function operators. That will also allow you to embed arbitrary LaTeX if needed.

2. Complete the DSL to support all the functions that `plotmath` supports.

Part II

Techniques

Introduction

The final four chapters cover two general programming techniques: finding and fixing bugs, and finding and fixing performance issues.

In Chapter 6, we'll first talking debugging, because finding the root cause of error can be extremely frustrating. Fortunately R has some great tools for debugging, and when they're coupled with a solid strategy, you should be able to find the root cause for most problems rapidly and relatively painlessly.

The remaining three chapters focus on performance, first measuring it (Chapter 7) and then improving it (Chapters 8 and 9). Tools to measuring and improve performance are particularly important because R is not a fast language. This is not an accident: R was purposely designed to make interactive data analysis easier for humans, not to make computers as fast as possible. While R is slow compared to other programming languages, for most purposes, it's fast enough. These chapters help you handle the cases where is no longer fast enough, either by improving the performance of your R code, or by switching to a language, C++, that is designed for performance.

Chapter 6

Debugging

6.1 Introduction

What do you do when R code throws an unexpected error? What tools do you have to find and fix the problem? This chapter will teach you the art and science of debugging, starting with a general strategy, then following up with specific tools.

I'll show the tools provided both by R itself, as well as the RStudio IDE. I recommend using RStudio's tools if possible, but I'll also show you the equivalents that work everywhere. You may also want to refer to the official RStudio debugging documentation (https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio) which always reflects the latest version of RStudio.

NB: You shouldn't need to use these tools when writing *new* functions. If you find yourself using them frequently with new code, reconsider your approach. Instead of trying to write one big function all at once, work interactively on small pieces. If you start small, you can quickly identify why something doesn't work, and don't need sophisticated debugging tools.

Outline

- Section 6.2 outlines a general strategy for finding and fixing errors.

- Section 6.3 introduces you to the `traceback()` function which helps you locate exactly where an error occurred.

- Section 6.4 shows you how to pause the execution of a function and launch environment where you can interactively explore what's happening.

- Section 6.5 discusses the challenging problem of debugging when you're running code non-interactively.

- Section 6.6 discusses a handful of non-error problems that occassionally also need debugging.

6.2   Overall approach

"Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true."

—Norm Matloff

Finding the root cause of a problem is always challenging. Most bugs are subtle and hard to find because if they were obvious, you would've avoided them in the first place. A good strategy helps. Below I outline a four step process that I have found useful:

1. **Google!**

   Whenever you see an error message, start by googling it. If you're lucky, you'll discover that it's a common error with a known solution. When googling, improve your chances of a good match by removing any variable names or values that are specific to your problem.

   You can automate this process with the errorist (Balamuta 2018a) and searcher (Balamuta 2018b) packages. See their websites for more details.

2. **Make it repeatable**

   To find the root cause of an error, you're going to need to execute the code many times as you consider and reject hypotheses. To make that iteration as quick possible, it's worth some upfront investment to make the problem both easy and fast to reproduce.

   Start by creating a reproducible example (Section **??**). Next, make the example minimal by removing code and simplifying data. As you do this, you may discover inputs that don't trigger the error. Make note of them: they will be helpful when diagnosing the root cause.

   If you're using automated testing, this is also a good time to create an automated test case. If your existing test coverage is low, take the opportunity to add some nearby tests to ensure that existing good behaviour is preserved. This reduces the chances of creating a new bug.

3. **Figure out where it is**

   If you're lucky, one of the tools in the following section will help you to quickly identify the line of code that's causing the bug. Usually, however, you'll have to think a bit more about the problem. It's a great idea to adopt the scientific method. Generate hypotheses, design experiments to

test them, and record your results. This may seem like a lot of work, but a systematic approach will end up saving you time. I often waste a lot of time relying on my intuition to solve a bug ("oh, it must be an off-by-one error, so I'll just subtract 1 here"), when I would have been better off taking a systematic approach.

If this fails, you might need to ask help from someone else. If you've followed the previous step, you'll have a small example that's easy to share with others. That makes it much easier for other people to look at the problem, and more likely to help you find a solution.

4. **Fix it and test it**

Once you've found the bug, you need to figure out how to fix it and to check that the fix actually worked. Again, it's very useful to have automated tests in place. Not only does this help to ensure that you've actually fixed the bug, it also helps to ensure you haven't introduced any new bugs in the process. In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.

6.3 Locate the error

Once you've made the error repeatable, the next step is to figure out where it comes from. The most important tool for this part of the process is `traceback()`, which shows you the sequence of calls (aka the call stack, Section **??**) that lead to the error.

Here's a simple example: you can see that `f()` calls `g()` calls `h()` calls `i()` that checks its argument is numeric:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) {
    stop("`d` must be numeric", call. = FALSE)
  }
  d + 10
}
```

When we run `f("a")` code in RStudio we see:

```
> f("a")
```

```
Error: `d` must be numeric                              ⬆ Show Traceback
                                                        ✹ Rerun with Debug
```

Two options appear to the right of the error message: "Show Traceback" and "Rerun with Debug". If you click "Show traceback" you see:

```
> f("a")
```

```
Error: `d` must be numeric                              ⬆ Hide Traceback
                                                        ✹ Rerun with Debug

  5. stop("`d` must be numeric", call. = FALSE) at debugging.R#6
  4. i(c) at debugging.R#3
  3. h(b) at debugging.R#2
  2. g(a) at debugging.R#1
  1. f("a")
```

If you're not using RStudio, you can use `traceback()` to get the same information (sans pretty formatting):

```
traceback()
#> 5: stop("`d` must be numeric", call. = FALSE) at debugging.R#6
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f("a")
```

NB: You read the `traceback()` output from bottom to top: the initial call is `f()`, which calls `g()`, then `h()`, then `i()`, which triggers the error. If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`. These are clickable in RStudio, and will take you to the corresponding line of code in the editor.

6.3.1   Lazy evaluation

One drawback to `traceback()` is that it always linearises the call tree, which can be confusing if there is much lazy evaluation involved (Section **??**). For example, take the following example where the error happens when evaluating the first argument to `f()`:

```
j <- function() k()
k <- function() stop("Oops!", call. = FALSE)
f(j())
#> Error: Oops!
```

```
traceback()
#> 7: stop("Oops!") at #1
#> 6: k() at #1
#> 5: j() at debugging.R#1
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f(j())
```

You can using `rlang::with_abort()` and `rlang::last_trace()` to see the call tree. Here, I think it makes it much easier to see the source of the problem. Look at the last branch of of the call tree so see that the error comes from `j()` calling `k()`.

```
rlang::with_abort(f(j()))
#> Error: Oops!
rlang::last_trace()
#>
#>   1.   rlang::with_abort(f(j()))
#>   2.     base::withCallingHandlers(...)
#>   3.   global::f(j())
#>   4.     global::g(a)
#>   5.       global::h(b)
#>   6.         global::i(c)
#>   7.   global::j()
#>   8.     global::k()
```

NB: `rlang::last_trace()` is ordered in the opposite way to `traceback()`. We'll come back to that issue in Section 6.4.2.4.

6.4   The interactive debugger

Sometimes, the precise location of the error is enough to let you track it down and fix it. Frequently, however, you need more information, and the easiest way to get it is with the interactive debugger which allows you to pause execution of a function and interactively explore its state.

If you're using RStudio, the easiest way to enter the interactive debugger is through RStudio's "Rerun with Debug" tool. This reruns the command that created the error, pausing execution where the error occurred. Otherwise, you can insert a call to `browser()` where you want to pause, and re-run the function yourself. For example, we could insert a call `browser()` in `g()`:

Figure 6.1: RStudio debugging toolbar

```
g <- function(b) {
  browser()
  h(b)
}
f(10)
```

`browser()` is just a regular function call which means that you can run it conditionally by wrapping it in an `if` statement:

```
g <- function(b) {
  if (b < 0) {
    browser()
  }
  h(b)
}
```

In either case, you'll end up in an interactive environment *inside* the function where you can run arbitrary R code to explore the current state. You'll know when you're in the interactive debugger because you get a special prompt:

```
Browse[1]>
```

In RStudio, you'll see the corresponding code in the editor (with the statement that will be run next highlighted), objects in the current environment in the "Environment" pane, and the call stack in the "Traceback" pane.

6.4.1 `browser()` commands

As well as allowing you to run regular R code, `browser()` provides a few special commands. You can use them by either typing short text commands, or by clicking a button in the RStudio toolbar, Figure 6.1:

- Next, `n`: executes the next step in the function. If you have a variable named `n`, you'll need `print(n)` to display its value.

- Step into, ⟳ or `s`: works like next, but if the next step is a function, it will step into that function so you can explore it interactively.

- Finish, ⬅ or `f`: finishes execution of the current loop or function.

- Continue, `c`: leaves interactive debugging and continues regular execution of the function. This is useful if you've fixed the bad state and want to check that the function proceeds correctly.

- Stop, `Q`: stops debugging, terminates the function, and returns to the global workspace. Use this once you've figured out where the problem is, and you're ready to fix it and reload the code.

There are two other slightly less useful commands that aren't available in the toolbar:

- Enter: repeats the previous command. I find this too easy to activate accidentally, so I turn it off using `options(browserNLdisabled = TRUE)`.

- `where`: prints stack trace of active calls (the interactive equivalent of `traceback`).

6.4.2 Alternatives

There are three alternatives to using `browser()`: setting breakpoints in RStudio, `option(error = recover)`, and `debug()` and friends.

6.4.2.1 Breakpoints

In RStudio, you can set a breakpoint by clicking to the left of the line number, or pressing `Shift + F9`. Breakpoints behave similarly to `browser()` but they are easier to set (one click instead of nine key presses), and you don't run the risk of accidentally including a `browser()` statement in your source code. There are two small downsides to breakpoints:

- There are a few unusual situations in which breakpoints will not work. Read breakpoint troubleshooting (http://www.rstudio.com/ide/docs/debugging/breakpoint-troubleshooting) for more details.

- RStudio currently does not support conditional breakpoints.

6.4.2.2 `recover()`

Another way to activate `browser()` is to use `options(error = recover)`. Now when you get an error, you'll get an interactive prompt that displays the traceback and gives you the ability to interactively debug inside any of the frames:

```
options(error = recover)
f("x")
#> Error: `d` must be numeric
```

```
#>
#> Enter a frame number, or 0 to exit
#>
#> 1: f("x")
#> 2: debugging.R#1: g(a)
#> 3: debugging.R#2: h(b)
#> 4: debugging.R#3: i(c)
#>
#> Selection:
```

You can return to default error handling with `options(error = NULL)`.

6.4.2.3 `debug()`

Another approach is to call a function that inserts the `browser()` call for you:

- `debug()` inserts a browser statement in the first line of the specified function. `undebug()` removes it. Alternatively, you can use `debugonce()` to browse only on the next run.

- `utils::setBreakpoint()` works similarly, but instead of taking a function name, it takes a file name and line number and finds the appropriate function for you.

These two functions are both special cases of `trace()`, which inserts arbitrary code at any position in an existing function. `trace()` is occasionally useful when you're debugging code that you don't have the source for. To remove tracing from a function, use `untrace()`. You can only perform one trace per function, but that one trace can call multiple functions.

6.4.2.4 The call stack

Unfortunately, the call stacks printed by `traceback()`, `browser()` & where, and `recover()` are not consistent. The following table shows how the call stacks from a simple nested set of calls are displayed by the three tools. The numbering is different between `traceback()` and where, and `recover()` displays calls in the opposite order.

traceback()	where	recover()	rlang functions
5: stop("...")			
4: i(c)	where 1: i(c)	1: f()	1. global::f(10)
3: h(b)	where 2: h(b)	2: g(a)	2. global::g(a)
2: g(a)	where 3: g(a)	3: h(b)	3. global::h(b)
1: f("a")	where 4: f("a")	4: i("a")	4. global::i("a")

RStudio displays calls in the same order as `traceback()`. rlang functions use the same ordering and numbering as `recover()`, but also use indenting to reinforce the hierarchy of calls.

6.4.3 Compiled code

It is also possible to use an interactive debugger (gdb or lldb) for compiled code (like C or C++). Unfortunately that's beyond the scope of this book, but there are a few resources that you might find useful:

- http://r-pkgs.had.co.nz/src.html#src-debugging
- https://github.com/wch/r-debug/blob/master/debugging-r.md
- http://kevinushey.github.io/blog/2015/04/05/debugging-with-valgrind/
- https://www.jimhester.com/2018/08/22/debugging-rstudio/

6.5 Non-interactive debugging

Debugging is most challenging when you can't run code interactively, typically because it's part of some pipeline run automatically (possibly on another computer), or because the error doesn't occur when you run same code interactively. This can be extremely frustrating!

This section will give you some useful tools, but don't forget the general strategy in Section 6.2. When you can't explore interactively, it's particularly important to spend some time making the problem as small as possible so you can iterate quickly. Sometimes `callr::r(f, list(1, 2))` can useful; this calls `f(1, 2)` in a fresh session, and can be a useful way to reproduce the problem.

You might also want to double check for these common issues:

- Is the global environment different? Have you loaded different packages? Are there objects left around from previous sessions that are causing differences?

- Is the working directory different?

- Is the `PATH` environment variable, which determines where external commands (like `git`) are found, different?

- Is the `R_LIBS` environment variable, which determines where `library()` looks for packages, different?

6.5.1 `dump.frames()`

`dump.frames()` is the equivalent to `recover()` for non-interactive code; it saves a `last.dump.rda` file in the working directory. Later, an interactive session, you

can later `load("last.dump.rda"); debugger()` to enter an interactive debugger with the same interface as `recover()`. This lets you "cheat", interactively debugging code that was run non-interactively.

```
# In batch R process ----
dump_and_quit <- function() {
  # Save debugging info to file last.dump.rda
  dump.frames(to.file = TRUE)
  # Quit R with error status
  q(status = 1)
}
options(error = dump_and_quit)

# In a later interactive session ----
load("last.dump.rda")
debugger()
```

6.5.2 Print debugging

If `dump.frames()` doesn't help, a good fallback is **print debugging**, where you insert numerous print statements to precisely locate the problem, and see the values of important variables. Print debugging is slow and primitive, but it always works, so it's particularly useful if you can't get a good traceback. Start by inserting coarse-grained markers, and then make them progressively more fine-grained as you determine exactly where the problem is.

```
f <- function(a) {
  cat("f()\n")
  g(a)
}
g <- function(b) {
  cat("g()\n")
  cat("b =", b, "\n")
  h(b)
}
h <- function(c) {
  cat("i()\n")
  i(c)
}

f(10)
#> f()
```

```
#> g()
#> b = 10
#> i()
#> [1] 20
```

Print debugging is particularly useful for compiled code because it's not un-common for the compiler to modify your code to such an extent you can't figure out the root problem even when inside an interactive debugger.

6.5.3 RMarkdown

Debugging code inside RMarkdown files requires some special tools. First, if you're knitting the file using RStudio, switch to calling `rmarkdown::render("path/to/file.Rmd")` instead. This runs the code in the current session, which makes it easier to debug. If doing this makes the problem go away, you'll need to figure out what makes the environments different.

If the problem persists, you'll need to use your interactive debugging skills. Whatever method you use, you'll need an extra step: in the error handler, you'll need to call `sink()`. This removes the default "sink" that knitr uses to capture all output, and ensures that you can see the results in the console. For example, to use `recover()` with RMarkdown, you'd put the following code in your setup block:

```
options(error = function() {
  sink()
  recover()
})
```

This will generate a warning about "no sink to remove" when knitr completes; you can safely ignore this warning.

If you simply want to a traceback, the easiest option is to use `rlang::trace_back()`, taking advantage of the `rlang_trace_top_env` option. This ensures that you only see the traceback from your code, not all of the functions called by RMarkdown and knitr.

```
options(rlang_trace_top_env = rlang::current_env())
options(error = function() {
  sink()
  print(rlang::trace_back(bottom = sys.frame(-1)), simplify = "none")
})
```

6.6   Non-error failures

There are other ways for a function to fail apart from throwing an error:

- A function may generate an unexpected warning. The easiest way to track
  down warnings is to convert them into errors with `options(warn = 2)` and
  use the in the call stack, like `doWithOneRestart()`, `withOneRestart()`,
  regular debugging tools. When you do this you'll see some extra calls
  `withRestarts()`, and `.signalSimpleWarning()`. Ignore these: they are
  internal functions used to turn warnings into errors.

- A function may generate an unexpected message.   You can use
  `rlang::with_abort()` to turn these messages into errors:

```
f <- function() g()
g <- function() message("Hi!")
f()
#> Hi!


rlang::with_abort(f(), "message")
#> Error: Hi!
rlang::last_trace()
#>
#> 1.  rlang::with_abort(f(), "message")
#> 2.   base::withCallingHandlers(...)
#> 3.  global::f()
#> 4.    global::g()
```

- A function might never return. This is particularly hard to debug au-
  tomatically, but sometimes terminating the function and looking at the
  `traceback()` is informative. Otherwise, use use print debugging, as in
  Section 6.5.2.

- The worst scenario is that your code might crash R completely, leaving
  you with no way to interactively debug your code. This indicates a bug in
  compiled (C or C++) code.

  If it's in your compiled code, you'll need to follow the links in Section
  6.4.3 and learn how to use an interactive C debugger (or insert many print
  statements).

  If it's in a package or base R, you'll need to contact the package maintainer.
  In either case, work on making the smallest possible reprex (Section **??**) to
  help the developer help you.

Chapter 7

Measuring performance

7.1 Introduction

> "Programmers waste enormous amounts of time thinking about, or
> worrying about, the speed of noncritical parts of their programs, and
> these attempts at efficiency actually have a strong negative impact
> when debugging and maintenance are considered."

> — Donald Knuth.

Before you can make your code faster, you first need to figure out what's
making it slow. This sounds easy, but it's not. Even experienced programmers
have a hard time identifying bottlenecks in their code. So instead of relying on
your intuition, you should **profile** your code: measure the run-time of each line
of code using realistic inputs.

Once you've identified bottlenecks you'll need to carefully experiment with
alternatives to find faster code that is still equivalent. In Chapter 8 you'll learn a
bunch of ways to speed up code, but first you need to learn how to **microbench-
mark** so that you can precisely measure the difference in performance.

Outline

- Section 7.2 shows you how to use profiling tools to dig into exactly what is
  making code slow.

- Section 7.3 shows how to use microbenchmarking to explore alternative
  implementations and figure out exactly which one is fastest.

Prerequisites

We'll use profvis (https://rstudio.github.io/profvis/) for profiling, and bench (https://bench.r-lib.org/) for microbenchmarking.

```
library(profvis)
library(bench)
```

7.2   Profiling

Across programming languages, the primary tool used to understand code performance is the profiler. There are a number of different types of profilers ,but R uses a fairly simple type called a sampling or statistical profiler. A sampling profiler stops the execution of code every few milliseconds and records the call stack (i.e. which function is currently executing, and the funcion that function, and so on). For example, consider `f()`, below:

```
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

(I use `profvis::pause()` instead of `Sys.sleep()` because `Sys.sleep()` does not appear in profiling outputs because as far as R can tell, it doesn't use up any computing time.)

If we profiled the execution of `f()`, stopping the execution of code every 0.1 s, we'd see a profile like this:

```
"pause" "f"
"pause" "f" "g"
"pause" "f" "g" "h"
"pause" "f" "h"
```

Each line represents one "tick" of the profiler (0.1 s in this case), and function calls are recorded from right to left: the first line shows `f()` calling `pause()`. It

shows that the code spends 0.1 s running `f()`, then 0.2 s running `g()`, then 0.1 s running `h()`.

If we actually profile `f()`, using `utils::Rprof()` as in the code below, we're unlikely to get such a clear result.

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
writeLines(readLines(tmp))
#> sample.interval=100000
#> "pause" "g" "f"
#> "pause" "h" "g" "f"
#> "pause" "h" "f"
```

That's because all profilers must make a fundamental trade-off between accuracy and performance. The compromise that makes, using a sampling profiler, only has minimal impact on performance, but is fundamentally stochastic because there's some variability in both the accuracy of the timer and in the time taken by each operation. That means each time that you profile you'll get a slightly different answer. Fortunately, the variability most affects functions that take very little time ot run, which are also the functions that we're least interested in.

7.2.1 Visualising profiles

The default profiling resolution is quite small, so if your function takes even a few seconds it will generate hundreds of samples. That quickly grows beyond our ability to look at directly, so instead of using `uils::Rprof()` we'll use the profvis package to visualise aggregates. profvis also connects profiling data back to the underlying source code, making it easier to build up a mental model of what you need to change. If you find profvis doesn't help for your code, you might try one of the other options like `utils::summaryRprof()` or the proftools package (Tierney and Jarjour 2016).

There are two ways to use profvis:

- From the "Profile" menu in RStudio.

- With `profvis::profvis()`. I recommend storing your code in a separate file and `source()`ing it in; this will ensure you get the best connection between profiling data and source code.
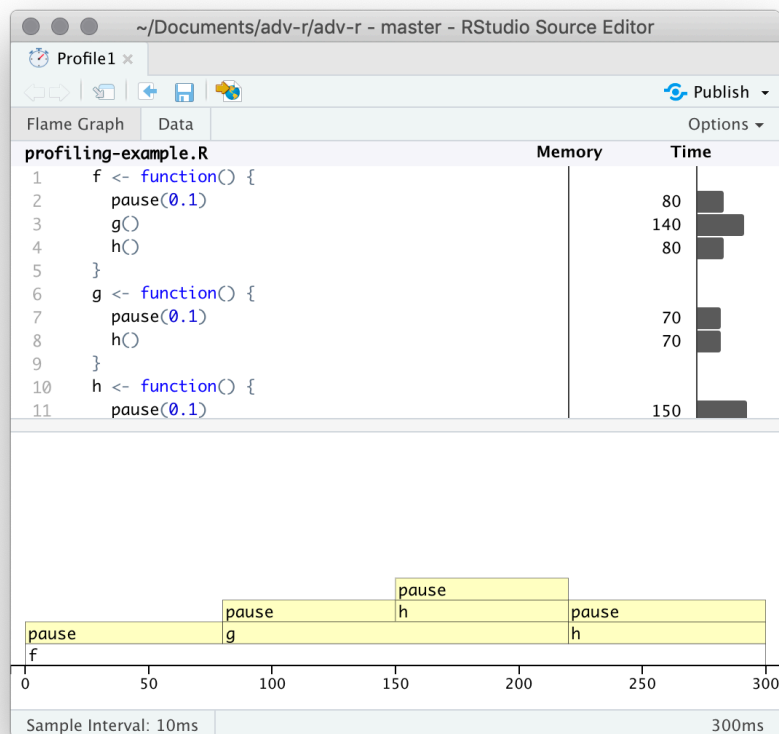
Figure 7.1: profvis output showing source on top and flame graph below.

```
source("profiling-example.R")
profvis(f())
```

After profiling is complete, profvis will open an interactive HTML document that allows you to explore the results. There are two panes, as shown in Figure 7.1.

The top pane shows the source code, overlaid with bar graphs for memory and execution time for each line of code. Here I'll focus on time, and we'll come back to memory shortly. This display gives you a good overall feel for the bottlenecks but doesn't always help you precisely identify the cause. Here, for example, you can see that h() takes 150ms, twice as long as g(); that's not because the function itself is slower, but because it's called twice as often.

The bottom pane displays a **flame graph** showing the full call stack. This allows you to see the full sequence of calls leading to each function, allowing you
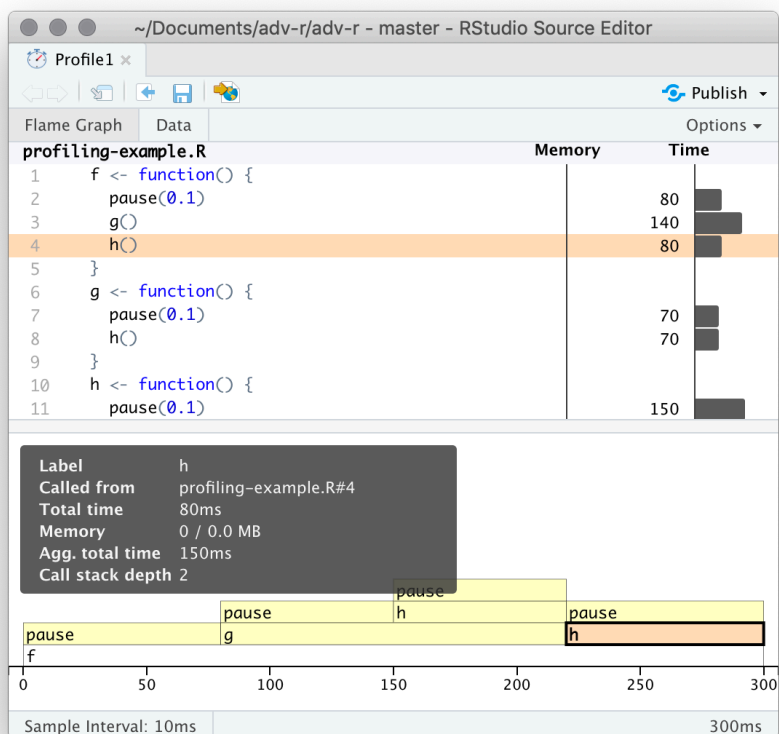
Figure 7.2: Hovering over a call in the flamegraph highlights the corresponding line of code, and displays additional information about performance.

to see that `h()` is called from two different places. In this display you can mouse over individual calls to get more information, and see the corresponding line of source code, as in Figure 7.2.

Alternatively, you can use the **data tab**, Figure 7.3 lets you interactively dive into the tree of performance data. This is basically the same display as the flame graph (rotated 90°), but it's more useful when you have very large or deeply nested call stacks because you can choose to interactively zoom into only selected components.

7.2.2   Memory profiling

There is a special entry in the flame graph that doesn't correspond to your code: `<GC>`, which indicates that the garbage collector is running. If `<GC>` is taking a lot

Figure 7.3: The data gives an interactive tree that allows you to selectively zoom into key components

of time, it's usually an indicating that you're creating many short-lived objects. For example, take this small snippet of code:

```r
x <- integer()
for (i in 1:1e4) {
  x <- c(x, i)
}
```

If you profile it, you'll see that most of the time is spent in the garbage collector, Figure 7.4.

When you see the garbage collector taking up a lot of time in your own code, you can often figure out the source of the problem by looking at the memory column: you'll see a line where large amounts of memory are being allocated (the bar on the right) and freed (the bar on the left). Here the problem arises because

Figure 7.4: Profiling a loop that modifies an existing variable reveals that most time is spent in the garbage collector ('<GC>')

of copy-on-modify (Section **??**): each iteration of the loop creates another copy of x. You'll learn strategies to resolve this type of problem in Section 8.6.

7.2.3   Limitations

There are some other limitations to profiling:

- Profiling does not extend to C code. You can see if your R code calls C/C++ code but not what functions are called inside of your C/C++ code. Unfortunately, tools for profiling compiled code are beyond the scope of this book; start by looking at https://github.com/r-prof/jointprof.

- If you're doing a lot of functional programming with anonymous functions, it can be hard to figure out exactly which function is being called. The easiest way to work around this is to name your functions.

- Lazy evaluation means that arguments are often evaluated inside another function, and this complicates the call stack (Section **??**). Unfortunately R's profiler doesn't store enough information to disentangle lazy evaluation so that in the following code, profiling would make it seem like i() was called by j() because the argument isn't evaluated until it's needed by j().

```
i <- function() {
  pause(0.1)
  10
}
j <- function(x) {
  x + 10
}
j(i())
```

If this is confusing, use `force()` (Section @ref{forcing-evaluation}) to force computation to happen earlier.

7.2.4 Exercises

1. Profile the following function with `torture = TRUE`. What is surprising? Read the source code of `rm()` to figure out what's going on.

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
```

7.3 Microbenchmarking

A **microbenchmark** is a measurement of the performance of a very small piece of code, something that might take milliseconds (ms), microseconds (µs), or nanoseconds (ns) to run. Microbenchmarks are useful for comparing small snippets of code for specific tasks. Be very wary of generalising the results of microbenchmarks to real code: the observed differences in microbenchmarks will typically be dominated by higher-order effects in real code; a deep understanding of subatomic physics is not very helpful when baking.

A great tool for microbenchmarking in R is the bench package (Hester 2018). bench uses a a high precision timer, making it possible to compare operations that only take a tiny amount of time. For example, the following code compares the speed of two approaches to computing a square root.

```
x <- runif(100)
(lb <- bench::mark(
  sqrt(x),
  x ^ 0.5
))
```

```
#> # A tibble: 2 x 10
#>   expression      min     mean   median      max `itr/sec` mem_alloc
#>   <chr>       <bch:tm> <bch:t> <bch:tm> <bch:t>     <dbl> <bch:byt>
#> 1 sqrt(x)     589.06ns 790.7ns 669.04ns  30.2µs  1264693.      848B
#> 2 x^0.5         9.04µs  10.1µs   9.44µs 785.8µs    98952.      848B
#> # … with 3 more variables: n_gc <dbl>, n_itr <int>,
#> #   total_time <bch:tm>
```

By default, `bench::mark()` runs each expression at least once (`min_iterations = 1`), and at most enough times to take 0.5s (`min_time = 0.5`). It checks that each run returns the same value which is typically what you when microbenchmarking; if you want to compare the speed of expressions that return different values, set `check = FALSE`.

7.3.1 `bench::mark()` results

`bench::mark()` returns the results as a tibble, with one row for each input expression, and the following columns:

- `min`, `mean`, `median`, `max`, and `itr/sec` summarise the time taken by the expression. Focus on the minimum (the best possible running time) and the median (the typical time). In this example, you can see that using the special purpose `sqrt()` function is faster than the general exponentiation operator.

You can visualise the distribution of the individual timings with `plot()`:

```
plot(lb)
```
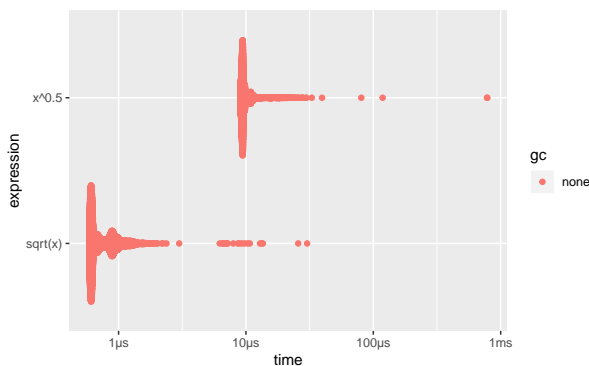

The distribution tends to be heavily right-skewed (note that the x-axis is already on a log scale!), which is why you should avoid comparing means. You'll also often see multimodality because your computer is running something else in the background.

- `mem_alloc` tells you the amount of memory allocated by the first run, and
  `n_gc()` tells you the total number of garbage collections over all runs. These
  are useful for assessing the memory usage of the expression.

- `n_itr` and `total_time` tells you how many times the expression was eval-
  uated and how long that took in total. `n_itr` will always be greater than
  the `min_iteration` parameter, and `total_time` will always be greater than
  the `min_time` parameter.

- `result`, `memory`, `time`, and `gc` are list-columns that store the raw underlying
  data.

Because the result is a special type of tibble, you can use `[` to select just the
most important columns. I'll do that frequently in the next chapter.

```
lb[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression      min    median `itr/sec`  n_gc
#>   <chr>       <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 sqrt(x)     589.06ns 669.04ns  1264693.     0
#> 2 x^0.5         9.04µs   9.44µs    98952.     0
```

7.3.2 Interpreting results

As with all microbenchmarks, pay careful attention to the units: here, each
computation takes about 590 ns, 590 billionths of a second. To help calibrate the
impact of a microbenchmark on run time, it's useful to think about how many
times a function needs to run before it takes a second. If a microbenchmark
takes:

- 1 ms, then one thousand calls takes a second.
- 1 µs, then one million calls takes a second.
- 1 ns, then one billion calls takes a second.

The `sqrt()` function takes about 590 ns, or 0.59 µs, to compute the square
root of 100 numbers. That means if you repeated the operation a million times,
it would take 0.59 s, and hence changing the way you compute the square root is
unlikely to significantly affect real code. This is the reason you need to exercise
care when generalising microbenchmarking results.

7.3.3 Exercises

1. Instead of using `bench::mark()`, you could use the built-in function
   `system.time()`. But `system.time()` is much less precise, so you'll need

to repeat each operation many times with a loop, and then divide to find the average time of each operation, as in the code below.

```
n <- 1e6
system.time(for (i in 1:n) sqrt(x)) / n
system.time(for (i in 1:n) x ^ 0.5) / n
```

How do the estimates from `system.time()` compare to those from `bench::mark()`? Why are they different?

2. Here are two other ways to compute the square root of a vector. Which do you think will be fastest? Which will be slowest? Use microbenchmarking to test your answers.

```
x ^ (1 / 2)
exp(log(x) / 2)
```

Chapter 8

Improving performance

8.1 Introduction

> "We should forget about small efficiencies, say about 97% of the time:
> premature optimization is the root of all evil. Yet we should not pass
> up our opportunities in that critical 3%. A good programmer will
> not be lulled into complacency by such reasoning, he will be wise to
> look carefully at the critical code; but only after that code has been
> identified."
>
> — Donald Knuth.

Once you've used profiling to identify a bottleneck, you need to make it faster. It's difficult to provide general advice on improving performance, but I try my best with four techniques that can be applied in many situations. I'll also suggest a general strategy for performance optimisation that helps ensure that your faster code is still correct.

It's easy to get caught up in trying to remove all bottlenecks. Don't! Your time is valuable and is better spent analysing your data, not eliminating possible inefficiencies in your code. Be pragmatic: don't spend hours of your time to save seconds of computer time. To enforce this advice, you should set a goal time for your code and optimise only up to that goal. This means you will not eliminate all bottlenecks. Some you will not get to because you've met your goal. Others you may need to pass over and accept either because there is no quick and easy solution or because the code is already well optimised and no significant improvement is possible. Accept these possibilities and move on to the next candidate.

If you'd like to learn more about the performance characteristics of the R language itself, I'd highly recommend "Evaluating the Design of the R Language"

(Morandat et al. 2012). It draws conclusions by combining a modified R inter-preter with a wide set of code found in the wild.

Outline

- Section 8.2 teaches you how to organise your code to make optimisation as easy, and bug free, as possible.

- Section 8.3 reminds you to look for existing solutions.

- Section 8.4 emphasises the importance of being lazy: often the easiest way to make a function faster is to let it to do less work.

- Section 8.5 concisely defines vectorisation, and shows you how to make the most of built-in functions.

- Section 8.6 discusses the performance perils of copying data.

- Section 8.7 pulls all the pieces together into a case study showing how to speed up repeated t-tests by ~1000x.

- Section 8.8 finishes the chapter with pointers to more resources that will help you write fast code.

Prerequisites

We'll use bench (https://bench.r-lib.org/) to precisely compare the performance of small self-contained code chunks.

```
library(bench)
```

8.2   Code organisation

There are two traps that are easy to fall into when trying to make your code faster:

1. Writing faster but incorrect code.
2. Writing code that you think is faster, but is actually no better.

The strategy outlined below will help you avoid these pitfalls.

When tackling a bottleneck, you're likely to come up with multiple approaches. Write a function for each approach, encapsulating all relevant behaviour. This makes it easier to check that each approach returns the correct result and to time how long it takes to run. To demonstrate the strategy, I'll compare two approaches for computing the mean:

```
mean1 <- function(x) mean(x)
mean2 <- function(x) sum(x) / length(x)
```

I recommend that you keep a record of everything you try, even the failures. If a similar problem occurs in the future, it'll be useful to see everything you've tried. To do this I recommend RMarkdown, which makes it easy to intermingle code with detailed comments and notes.

Next, generate a representative test case. The case should be big enough to capture the essence of your problem but small enough that it only a few seconds at most. You don't want it to take too long because you'll need to run the test case many times to compare approaches. On the other hand, you don't want the case to be too small because then results might not scale up to the real problem. Hre I'm going to use 100,000 numbers:

```
x <- runif(1e5)
```

Now use `bench::mark()` to precisely compare the variations. `bench::mark()` automatically checks that all calls return the same values. This doesn't guarantee that the function behaves the same for all inputs, so in an ideal world you'll also have unit tests to make sure you don't accidentally change the behaviour of the function.

```
bench::mark(
  mean1(x),
  mean2(x)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec`  n_gc
#>   <chr>        <bch:tm> <bch:tm>    <dbl> <dbl>
#> 1 mean1(x)        226µs    235µs     4226.     0
#> 2 mean2(x)        112µs    113µs     8436.     0
```

(You might be surprised by the results: `mean(x)` is considerably slower than `sum(x) / length(x)`. This is because, among other reasons, `mean(x)` makes two passes over the vector to be more numerically accurate.)

If you'd like to see this strategy in action, I've used it a few times on stackoverflow:

- http://stackoverflow.com/questions/22515525#22518603
- http://stackoverflow.com/questions/22515175#22515856
- http://stackoverflow.com/questions/3476015#22511936

8.3   Check for existing solutions

Once you've organised your code and captured all the variations you can think of, it's natural to see what others have done. You are part of a large community, and it's quite possible that someone has already tackled the same problem. Two good places to start are:

- CRAN task views (http://cran.rstudio.com/web/views/). If there's a CRAN task view related to your problem domain, it's worth looking at the packages listed there.

- Reverse dependencies of Rcpp, as listed on its CRAN page (http://cran. r-project.org/web/packages/Rcpp). Since these packages use C++, they're likely to be fast.

Otherwise, the challenge is describing your bottleneck in a way that helps you find related problems and solutions. Knowing the name of the problem or its synonyms will make this search much easier. But because you don't know what it's called, it's hard to search for it! The best way to solve this problem is to read widely so that you can build up your own vocabulary over time. Alternatively, ask others. Talk to your colleagues and brainstorm some possible names, then search on Google and StackOverflow. It's often helpful to restrict your search to R related pages. For Google, try rseek (http://www.rseek.org/). For stackoverflow, restrict your search by including the R tag, `[R]`, in your search.

Record all solutions that you find, not just those that immediately appear to be faster. Some solutions might be slower initially, but end up being faster because they're easier to optimise. You may also be able to combine the fastest parts from different approaches. If you've found a solution that's fast enough, congratulations! Otherwise, read on.

8.3.1   Exercises

1. What are faster alternatives to `lm()`? Which are specifically designed to work with larger datasets?

2. What package implements a version of `match()` that's faster for repeated lookups? How much faster is it?

3. List four functions (not just those in base R) that convert a string into a date time object. What are their strengths and weaknesses?

4. Which packages provide the ability to compute a rolling mean?

5. What are the alternatives to `optim()`?

8.4 Do as little as possible

The easiest way to make a function faster is to let it do less work. One way to do that is use a function tailored to a more specific type of input or output, or to a more specific problem. For example:

- `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()` are faster than equivalent invocations that use `apply()` because they are vectorised (Section 8.5).

- `vapply()` is faster than `sapply()` because it pre-specifies the output type.

- If you want to see if a vector contains a single value, `any(x == 10)` is much faster than `10 %in% x` because testing equality is simpler than testing set inclusion.

Having this knowledge at your fingertips requires knowing that alternative functions exist: you need to have a good vocabulary. Expand your vocab by regularly reading R code. Good places to read code are the R-help mailing list (https://stat.ethz.ch/mailman/listinfo/r-help) and StackOverflow (http://stackoverflow.com/questions/tagged/r).

Some functions coerce their inputs into a specific type. If your input is not the right type, the function has to do extra work. Instead, look for a function that works with your data as it is, or consider changing the way you store your data. The most common example of this problem is using `apply()` on a data frame. `apply()` always turns its input into a matrix. Not only is this error prone (because a data frame is more general than a matrix), it is also slower.

Other functions will do less work if you give them more information about the problem. It's always worthwhile to carefully read the documentation and experiment with different arguments. Some examples that I've discovered in the past include:

- `read.csv()`: specify known column types with `colClasses`. (Also consider switching to `readr::read_csv()` or `data.table::fread()` which are considerably faster than `read.csv()`.)

- `factor()`: specify known levels with `levels`.

- `cut()`: don't generate labels with `labels = FALSE` if you don't need them, or, even better, use `findInterval()` as mentioned in the "see also" section of the documentation.

- `unlist(x, use.names = FALSE)` is much faster than `unlist(x)`.

- `interaction()`: if you only need combinations that exist in the data, use `drop = TRUE`.

Below, I explore how you might improve apply this strategy to improve the performance of `mean()` and `as.data.frame()`.

8.4.1 `mean()`

Sometimes you can make a function faster by avoiding method dispatch. If you're calling a method in a tight loop, you can avoid some of the costs by doing the method lookup only once:

- For S3, you can do this by calling `generic.class()` instead of `generic()`.

- For S4, you can do this by using `getMethod()` to find the method, saving it to a variable, and then calling that function.

For example, calling `mean.default()` is quite a bit faster than calling `mean()` for small vectors:

```
x <- runif(1e2)

bench::mark(
  mean(x),
  mean.default(x)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression        min    median `itr/sec`  n_gc
#>   <chr>         <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean(x)         2.57µs   2.98µs   298248.     1
#> 2 mean.default(x)  1.22µs    1.4µs   642430.     0
```

This optimisation is a little risky. While `mean.default()` is almost twice as fast for 100 values, it will fail in surprising ways if `x` is not a numeric vector.

An even riskier optimisation is to directly call the underlying `.Internal` function. This is faster because it doesn't do any input checking or handle NA's, so you are buying speed at the cost of safety.

```
x <- runif(1e2)
bench::mark(
  mean(x),
  mean.default(x),
  .Internal(mean(x))
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 x 5
#>   expression            min   median `itr/sec`  n_gc
```

```
#>     <chr>                   <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean(x)                     2.59µs   2.95µs   296758.     1
#> 2 mean.default(x)             1.22µs   1.43µs   627231.     0
#> 3 .Internal(mean(x)) 340.17ns 364.15ns  2439698.     0
```

NB: most of these differences arise because x is small. If you increase the size the differences basically disappear, because most of the time is now spent computing the mean, not finding the underlying implementation. This is a good reminder that the size of the input matters, and you should motivate your optimisations based on realistic data.

```
x <- runif(1e4)
bench::mark(
  mean(x),
  mean.default(x),
  .Internal(mean(x))
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 x 5
#>   expression              min    median `itr/sec`  n_gc
#>   <chr>               <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 mean(x)               24.8µs   25.2µs   37956.     1
#> 2 mean.default(x)       23.4µs   23.6µs   41069.     0
#> 3 .Internal(mean(x))    22.4µs   22.5µs   43282.     0
```

8.4.2 as.data.frame()

Knowing that you're dealing with a specific type of input can be another way to write faster code. For example, as.data.frame() is quite slow because it coerces each element into a data frame and then rbind()s them together. If you have a named list with vectors of equal length, you can directly transform it into a data frame. In this case, if you can make strong assumptions about your input, you can write a method that's considerably faster than the default.

```
quickdf <- function(l) {
  class(l) <- "data.frame"
  attr(l, "row.names") <- .set_row_names(length(l[[1]]))
  l
}

l <- lapply(1:26, function(i) runif(1e3))
names(l) <- letters
```

```
bench::mark(
  as.data.frame = as.data.frame(l),
  quick_df      = quickdf(l)
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 2 x 5
#>   expression          min    median `itr/sec`  n_gc
#>   <chr>           <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 as.data.frame    1.07ms   1.12ms      882.     7
#> 2 quick_df         6.13µs   7.21µs   124100.     2
```

Again, note the trade-off. This method is fast because it's dangerous. If you give it bad inputs, you'll get a corrupt data frame:

```
quickdf(list(x = 1, y = 1:2))
#> Warning in format.data.frame(if (omit) x[seq_len(n0), , drop = FALSE]
#> else x, : corrupt data frame: columns will be truncated or padded
#> with NAs
#>   x y
#> 1 1 1
```

To come up with this minimal method, I carefully read through and then rewrote the source code for `as.data.frame.list()` and `data.frame()`. I made many small changes, each time checking that I hadn't broken existing behaviour. After several hours work, I was able to isolate the minimal code shown above. This is a very useful technique. Most base R functions are written for flexibility and functionality, not performance. Thus, rewriting for your specific need can often yield substantial improvements. To do this, you'll need to read the source code. It can be complex and confusing, but don't give up!

8.4.3   Exercises

1. What's the difference between `rowSums()` and `.rowSums()`?

2. Make a faster version of `chisq.test()` that only computes the chi-square test statistic when the input is two numeric vectors with no missing values. You can try simplifying `chisq.test()` or by coding from the mathematical definition (http://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test).

3. Can you make a faster version of `table()` for the case of an input of two integer vectors with no missing values? Can you use it to speed up your chi-square test?

8.5 Vectorise

If you've used R for any length of time, you've probably heard the admonishment to "vectorise your code". But what does that actually mean? Vectorising your code is not just about avoiding for loops, although that's often a step. Vectorising is about taking a "whole object" approach to a problem, thinking about vectors, not scalars. There are two key attributes of a vectorised function:

- It makes many problems simpler. Instead of having to think about the components of a vector, you only think about entire vectors.

- The loops in a vectorised function are written in C instead of R. Loops in C are much faster because they have much less overhead.

Chapter **??** stressed the importance of vectorised code as a higher level abstraction. Vectorisation is also important for writing fast R code. This doesn't mean simply using `map()` or `lapply()`. Instead, vectorisation means finding the existing R function that is implemented in C and most closely applies to your problem.

Vectorised functions that apply to many common performance bottlenecks include:

- `rowSums()`, `colSums()`, `rowMeans()`, and `colMeans()`. These vectorised matrix functions will always be faster than using `apply()`. You can sometimes use these functions to build other vectorised functions.

  ```
  rowAny <- function(x) rowSums(x) > 0
  rowAll <- function(x) rowSums(x) == ncol(x)
  ```

- Vectorised subsetting can lead to big improvements in speed. Remember the techniques behind lookup tables (Section **??**) and matching and merging by hand (Section **??**). Also remember that you can use subsetting assignment to replace multiple values in a single step. If `x` is a vector, matrix or data frame then `x[is.na(x)] <- 0` will replace all missing values with 0.

- If you're extracting or replacing values in scattered locations in a matrix or data frame, subset with an integer matrix. See Section **??** for more details.

- If you're converting continuous values to categorical make sure you know how to use `cut()` and `findInterval()`.

- Be aware of vectorised functions like `cumsum()` and `diff()`.

Matrix algebra is a general example of vectorisation. There loops are executed by highly tuned external libraries like BLAS. If you can figure out a way to use

matrix algebra to solve your problem, you'll often get a very fast solution. The ability to solve problems with matrix algebra is a product of experience. A good place to start is to ask people with experience in your domain.

The downside of vectorisation is that it makes it harder to predict how operations will scale. The following example measures how long it takes to use character subsetting to lookup 1, 10, and 100 elements from a list. You might expect that looking up 10 elements would take 10x as long as looking up 1, and that looking up 100 elements would take 10x longer again. In fact, the following example shows that it only takes about ~10x longer to look up 100 elements than it does to look up 1. That happens because once you get to a certain size, the internal implementation switches to a strategy that has a higher set up cost, but scales better.

```r
lookup <- setNames(as.list(sample(100, 26)), letters)

x1 <- "j"
x10 <- sample(letters, 10)
x100 <- sample(letters, 100, replace = TRUE)

bench::mark(
  lookup[x1],
  lookup[x10],
  lookup[x100],
  check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 3 x 5
#>   expression         min    median `itr/sec`  n_gc
#>   <chr>          <bch:tm> <bch:tm>     <dbl> <dbl>
#> 1 lookup[x1]     493.02ns 603.03ns  1397596.     0
#> 2 lookup[x10]      1.54µs   1.68µs   530069.     1
#> 3 lookup[x100]     4.22µs   4.93µs   184860.     0
```

Vectorisation won't solve every problem, and rather than torturing an existing algorithm into one that uses a vectorised approach, you're often better off writing your own vectorised function in C++. You'll learn how to do so in Chapter 9.

8.5.1 Exercises

1. The density functions, e.g., `dnorm()`, have a common interface. Which arguments are vectorised over? What does `rnorm(10, mean = 10:1)` do?

2. Compare the speed of `apply(x, 1, sum)` with `rowSums(x)` for varying sizes of `x`.

3. How can you use `crossprod()` to compute a weighted sum? How much faster is it than the naive `sum(x * w)`?

8.6 Avoid copies

A pernicious source of slow R code is growing an object with a loop. Whenever you use `c()`, `append()`, `cbind()`, `rbind()`, or `paste()` to create a bigger object, R must first allocate space for the new object and then copy the old object to its new home. If you're repeating this many times, like in a for loop, this can be quite expensive. You've entered Circle 2 of the "R inferno" (http://www.burns-stat. com/pages/Tutor/R_inferno.pdf).

You saw one example of this type of problem in Section 7.2.2, so here I'll show a slightly more complex example of the same basic issue. We first generate some random strings, and then combine them either iteratively with a loop using `collapse()`, or in a single pass using `paste()`. Note that the performance of `collapse()` gets relatively worse as the number of strings grows: combining 100 strings takes almost 30 times longer than combining 10 strings.

```r
random_string <- function() {
  paste(sample(letters, 50, replace = TRUE), collapse = "")
}
strings10 <- replicate(10, random_string())
strings100 <- replicate(100, random_string())

collapse <- function(xs) {
  out <- ""
  for (x in xs) {
    out <- paste0(out, x)
  }
  out
}

bench::mark(
  loop10  = collapse(strings10),
  loop100 = collapse(strings100),
  vec10   = paste(strings10, collapse = ""),
  vec100  = paste(strings100, collapse = ""),
  check = FALSE
)[c("expression", "min", "median", "itr/sec", "n_gc")]
#> # A tibble: 4 x 5
#>   expression      min    median `itr/sec`  n_gc
```

```
#>    <chr>        <bch:tm> <bch:tm>      <dbl> <dbl>
#> 1 loop10       19.87µs  21.89µs     43059.     3
#> 2 loop100     799.37µs 828.22µs      1192.     2
#> 3 vec10         4.89µs   5.13µs    186616.     0
#> 4 vec100       39.16µs  40.27µs     23770.     0
```

Modifying an object in a loop, e.g., `x[i] <- y`, can also create a copy, depending on the class of `x`. Section **??** discusses this issue in more depth and gives you some tools to determine when you're making copies.

8.7   Case study: t-test

The following case study shows how to make t-tests faster using some of the techniques described above. It's based on an example in "Computing thousands of test statistics simultaneously in R" (http://stat-computing.org/newsletter/issues/scgn-18-1.pdf) by Holger Schwender and Tina Müller. I thoroughly recommend reading the paper in full to see the same idea applied to other tests.

Imagine we have run 1000 experiments (rows), each of which collects data on 50 individuals (columns). The first 25 individuals in each experiment are assigned to group 1 and the rest to group 2. We'll first generate some random data to represent this problem:

```
m <- 1000
n <- 50
X <- matrix(rnorm(m * n, mean = 10, sd = 3), nrow = m)
grp <- rep(1:2, each = n / 2)
```

For data in this form, there are two ways to use `t.test()`. We can either use the formula interface or provide two vectors, one for each group. Timing reveals that the formula interface is considerably slower.

```
system.time(
  for (i in 1:m) {
    t.test(X[i, ] ~ grp)$statistic
  }
)
#>    user  system elapsed
#>   0.840   0.000   0.838
system.time(
  for (i in 1:m) {
    t.test(X[i, grp == 1], X[i, grp == 2])$statistic
```

```
 }
)
#>    user  system elapsed
#>   0.192   0.000   0.195
```

Of course, a for loop computes, but doesn't save the values. We can `map_dbl()`
(Section **??**) to do that. This adds a little overhead:

```
compT <- function(i){
  t.test(X[i, grp == 1], X[i, grp == 2])$statistic
}
system.time(t1 <- purrr::map_dbl(1:m, compT))
#>    user  system elapsed
#>   0.200   0.000   0.199
```

How can we make this faster? First, we could try doing less work. If you look
at the source code of `stats:::t.test.default()`, you'll see that it does a lot
more than just compute the t-statistic. It also computes the p-value and formats
the output for printing. We can try to make our code faster by stripping out
those pieces.

```
my_t <- function(x, grp) {
  t_stat <- function(x) {
    m <- mean(x)
    n <- length(x)
    var <- sum((x - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }

  g1 <- t_stat(x[grp == 1])
  g2 <- t_stat(x[grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}

system.time(t2 <- purrr::map_dbl(1:m, ~ my_t(X[.,], grp)))
#>    user  system elapsed
#>   0.028   0.000   0.028
stopifnot(all.equal(t1, t2))
```

This gives us about a 6x speed improvement.

Now that we have a fairly simple function, we can make it faster still by vec-
torising it. Instead of looping over the array outside the function, we will modify
`t_stat()` to work with a matrix of values. Thus, `mean()` becomes `rowMeans()`,
`length()` becomes `ncol()`, and `sum()` becomes `rowSums()`. The rest of the code
stays the same.

```r
rowtstat <- function(X, grp){
  t_stat <- function(X) {
    m <- rowMeans(X)
    n <- ncol(X)
    var <- rowSums((X - m) ^ 2) / (n - 1)

    list(m = m, n = n, var = var)
  }

  g1 <- t_stat(X[, grp == 1])
  g2 <- t_stat(X[, grp == 2])

  se_total <- sqrt(g1$var / g1$n + g2$var / g2$n)
  (g1$m - g2$m) / se_total
}
system.time(t3 <- rowtstat(X, grp))
#>    user  system elapsed
#>   0.012   0.000   0.012
stopifnot(all.equal(t1, t3))
```

That's much faster! It's at least 40x faster than our previous effort, and
around 1000x faster than where we started.

8.8   Other techniques

Being able to write fast R code is part of being a good R programmer. Beyond
the specific hints in this chapter, if you want to write fast R code, you'll need to
improve your general programming skills. Some ways to do this are to:

- Read R blogs (http://www.r-bloggers.com/) to see what performance prob-
  lems other people have struggled with, and how they have made their code
  faster.

- Read other R programming books, like "The Art of R Programming"
  (Matloff 2011) or Patrick Burns' *R Inferno* (http://www.burns-stat.com/
  documents/books/the-r-inferno/) to learn about common traps.

- Take an algorithms and data structure course to learn some well known ways of tackling certain classes of problems. I have heard good things about Princeton's Algorithms course (https://www.coursera.org/course/algs4partI) offered on Coursera.

- Learn how to parallelise your code. Two places to start are "Parallel R" (McCallum and Weston 2011) and "Parallel Computing for Data Science" (Matloff 2015).

- Read general books about optimisation like "Mature optimisation" (Bueno 2013) or the "Pragmatic Programmer" (Hunt and Thomas 1990).

You can also reach out to the community for help. StackOverflow can be a useful resource. You'll need to put some effort into creating an easily digestible example that also captures the salient features of your problem. If your example is too complex, few people will have the time and motivation to attempt a solution. If it's too simple, you'll get answers that solve the toy problem but not the real problem. If you also try to answer questions on StackOverflow, you'll quickly get a feel for what makes a good question.

Chapter 9

Rewriting R code in C++

9.1 Introduction

Sometimes R code just isn't fast enough. You've used profiling to figure out where your bottlenecks are, and you've done everything you can in R, but your code still isn't fast enough. In this chapter you'll learn how to improve performance by rewriting key functions in C++. This magic comes by way of the Rcpp (http://www.rcpp.org/) package (Eddelbuettel and François 2011) (with key contributions by Doug Bates, John Chambers, and JJ Allaire).

Rcpp makes it very simple to connect C++ to R. While it is *possible* to write C or Fortran code for use in R, it will be painful by comparison. Rcpp provides a clean, approachable API that lets you write high-performance code, insulated from R's complex C API.

Typical bottlenecks that C++ can address include:

- Loops that can't be easily vectorised because subsequent iterations depend on previous ones.

- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than in R.

- 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.

The aim of this chapter is to discuss only those aspects of C++ and Rcpp that are absolutely necessary to help you eliminate bottlenecks in your code. We won't spend much time on advanced features like object oriented programming or templates because the focus is on writing small, self-contained functions, not big

programs. A working knowledge of C++ is helpful, but not essential. Many good tutorials and references are freely available, including http://www.learncpp.com/ and https://en.cppreference.com/w/cpp. For more advanced topics, the *Effective C++* series by Scott Meyers is a popular choice.

Outline

- Section 9.2 teaches you how to write C++ by converting simple R functions to their C++ equivalents. You'll learn how C++ differs from R, and what the key scalar, vector, and matrix classes are called.

- Section 9.2.5 shows you how to use `sourceCpp()` to load a C++ file from disk in the same way you use `source()` to load a file of R code.

- Section 9.3 discusses how to modify attributes from Rcpp, and mentions some of the other important classes.

- Section 9.4 teaches you how to work with R's missing values in C++.

- Section 9.5 shows you how to use some of the most important data structures and algorithms from the standard template library, or STL, built-in to C++.

- Section 9.6 shows two real case studies where Rcpp was used to get considerable performance improvements.

- Section 9.7 teaches you how to add C++ code to a package.

- Section 9.8 concludes the chapter with pointers to more resources to help you learn Rcpp and C++.

Prerequisites

We'll use Rcpp (http://www.rcpp.org/) to call C++ from R:

```
library(Rcpp)
```

You'll also need a working C++ compiler. To get it:

- On Windows, install Rtools (http://cran.r-project.org/bin/windows/Rtools/).
- On Mac, install Xcode from the app store.
- On Linux, `sudo apt-get install r-base-dev` or similar.

9.2 Getting started with C++

`cppFunction()` allows you to write C++ functions in R:

```
cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
# add works like a regular R function
add
#> function (x, y, z)
#> .Call(<pointer: 0x7f909b3fbf60>, x, y, z)
add(1, 2, 3)
#> [1] 6
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function. There's a lot going on underneath the hood but Rcpp takes care of all the details so you don't need to worry about it.

The following sections will teach you the basics by translating simple R functions to their C++ 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

9.2.1 No inputs, scalar output

Let's start with a very simple function. It has no arguments and always returns 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 as you do 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`.

- Scalars and vectors are different. The scalar equivalents of numeric, integer,
  character, and logical vectors are: `double`, `int`, `String`, and `bool`.

- You must use an explicit `return` statement to return a value from a func-
  tion.

- Every statement is terminated by a `;`.

9.2.2   Scalar input, scalar output

The next example function implements a scalar version of the `sign()` function
which returns 1 if the input is positive, and -1 if it's negative:

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

cppFunction('int signC(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. As in R you can use `break` to exit the loop, but to skip one iteration you need to use `continue` instead of `next`.

9.2.3   Vector input, scalar output

One big difference between R and C++ is that the cost of loops is much lower in C++. 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!

```
sumR <- 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. In Section 9.5, you'll see alternatives to `for` loops that more clearly express your intent; they're not faster, but they can make your code easier to understand.

```
cppFunction('double sumC(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. C++ methods are called with `.` (i.e., a full stop).

- The `for` statement has a different syntax: `for(init; check; increment)`. This loop is initialised by creating a new variable called `i` with value 0. Before each iteration we check that `i < n`, and terminate the loop if it's not. After each iteration, we increment the value of `i` by one, using the special prefix operator `++` which increases the value of `i` by 1.

- In C++, vector indices start at 0, which means that the last element is at position `n - 1`. I'll say this again because it's so important: **IN C++, VECTOR INDICES START AT 0**! This is a very common source of bugs when converting R functions to C++.

- Use `=` for assignment, not `<-`.

- C++ provides operators that modify in-place: `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 R. As shown by the following microbenchmark, `sumC()` is competitive with the built-in (and highly optimised) `sum()`, while `sumR()` is several orders of magnitude slower.

```
x <- runif(1e3)
bench::mark(
  sum(x),
  sumC(x),
  sumR(x)
)[1:6]
#> # A tibble: 3 x 6
#>   expression       min     mean    median       max `itr/sec`
#>   <chr>        <bch:tm> <bch:tm> <bch:tm> <bch:tm>      <dbl>
#> 1 sum(x)         1.31µs   1.52µs   1.34µs  88.83µs    659189.
#> 2 sumC(x)        3.27µs   4.62µs   4.73µs   1.06ms    216666.
#> 3 sumR(x)       38.32µs  40.83µs  39.16µs   3.08ms     24492.
```

9.2.4 Vector input, vector output

Next we'll create a function that computes the Euclidean distance between a value and a vector of values:

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

In R, it's not obvious that we want `x` to be a scalar from the function definition, and we'd need to make that clear in the documentation. That's not a problem in the C++ version because we have to be explicit about types:

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

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  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 way of making a vector is to copy an existing one: `NumericVector zs = clone(ys)`.

- C++ uses `pow()`, not ^, for exponentiation.

Note that because the R version is fully vectorised, it's already going to be fast.

```
y <- runif(1e6)
bench::mark(
  pdistR(0.5, y),
  pdistC(0.5, y)
)[1:6]
#> # A tibble: 2 x 6
#>   expression          min     mean   median      max `itr/sec`
#>   <chr>           <bch:tm> <bch:tm> <bch:tm> <bch:tm>     <dbl>
#> 1 pdistR(0.5, y)    8.58ms   8.92ms   8.86ms  11.31ms      112.
#> 2 pdistC(0.5, y)    4.69ms   4.85ms   4.79ms   5.93ms      206.
```

On my computer, it takes around 5 ms with a 1 million element `y` vector. The C++ function is about 2.5x faster, ~2 ms, but assuming it took you 10 minutes to write the C++ function, you'd need to run it ~200,000 times to make rewriting

worthwhile. 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.

9.2.5   Using sourceCpp

So far, we've used inline C++ with `cppFunction()`. This makes presentation simpler, but for real problems, it's usually easier to use stand-alone C++ files and then source them into R using `sourceCpp()`. This lets you take advantage of text editor support for C++ files (e.g., syntax highlighting) as well as making it easier to identify the line numbers in compilation errors.

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

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

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

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

If you're familiar with roxygen2, you might wonder how this relates to `@export`. `Rcpp::export` controls whether a function is exported from C++ to R; `@export` controls whether a function is exported from a package and made available to the user.

You can embed R code in special C++ comment blocks. This is really convenient if you want to run some test code:

```
/*** R
# This is R code
*/
```

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

To compile the C++ code, use `sourceCpp("path/to/file.cpp")`. This will create the matching R functions and add them to your current session. Note that these functions can not be saved in a `.Rdata` file and reloaded in a later session; they must be recreated each time you restart R.

For example, running `sourceCpp()` on the following file implements mean in C++ and then compares it to the built-in `mean()`:

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

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

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

/*** R
x <- runif(1e5)
bench::mark(
  mean(x),
  meanC(x)
)
*/
```

NB: if you run this code yourself, you'll notice that `meanC()` is much faster than the built-in `mean()`. This is because it trades numerical accuracy for speed.

For the remainder of this chapter C++ code will be presented stand-alone 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. This is easy to do in RMarkdown: all you need to do is specify `engine = "Rcpp"`.

9.2.6 Exercises

1. With the basics of C++ in hand, it's now a great time to practice by reading and writing some simple C++ functions. For each of the following functions, read the code and figure out what the corresponding base R function is. 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;
```

```cpp
  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[i] = 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(x1[i], y1[i]);
  }

  return out;
}
```

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

   1. `all()`

   2. `cumprod()`, `cummin()`, `cummax()`.

   3. `diff()`. Start by assuming lag 1, and then generalise for lag `n`.

   4. `range()`.

   5. `var()`. Read about the approaches you can take on wikipedia (http:
      //en.wikipedia.org/wiki/Algorithms_for_calculating_variance).
      Whenever implementing a numerical algorithm, it's always good to
      check what is already known about the problem.

9.3 Other classes

You've already seen the basic vector classes (`IntegerVector`, `NumericVector`,
`LogicalVector`, `CharacterVector`) and their scalar (`int`, `double`, `bool`, `String`)
equivalents. Rcpp also provides wrappers for all other base data types. The most
important are for lists and data frames, functions, and attributes, as described
below. Rcpp also provides classes for more types like `Environment`, `DottedPair`,
`Language`, `Symbol`, etc, but these are beyond the scope of this chapter.

9.3.1 Lists and data frames

Rcpp also provides `List` and `DataFrame` classes, but they are more useful for
output than input. This is because lists and data frames can contain arbitrary
classes but C++ needs to know their classes in advance. If the list has known
structure (e.g., it's an S3 object), you can extract the components and manually
convert them to their C++ equivalents with `as()`. For example, the object
created by `lm()`, the function that fits a linear model, is a list whose components
are always of the same type. The following code illustrates how you might extract
the mean percentage error (`mpe()`) of a linear model. This isn't a good example

of when to use C++, because it's so easily implemented in R, but it shows how to
work with an important S3 class. Note the use of `.inherits()` and the `stop()`
to check that the object really is a linear model.

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

// [[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)
#> [1] -0.0154
```

9.3.2 Functions

You can put R functions in an object of type `Function`. This makes calling an
R function from C++ straightforward. The only challenge is that we don't know
what type of output the function will return, so we use the catchall type `RObject`.

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

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

```
callWithOne(function(x) x + 1)
#> [1] 2
callWithOne(paste)
#> [1] "1"
```

Calling R functions with positional arguments is obvious:

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

But you need a special syntax for named arguments:

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

9.3.3 Attributes

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

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

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

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

  return out;
}
```

For S4 objects, `.slot()` plays a similar role to `.attr()`.

9.4 Missing values

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

- How R's missing values behave in C++'s scalars (e.g., `double`).
- How to get and set missing values in vectors (e.g., `NumericVector`).

9.4.1   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. Note that this kind of experimentation is a useful way to figure out what any operation does.

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

// [[Rcpp::export]]
List scalar_missings() {
  int int_s = NA_INTEGER;
  String chr_s = NA_STRING;
  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())
#> List of 4
#>  $ : int NA
#>  $ : chr NA
#>  $ : logi TRUE
#>  $ : num NA
```

With the exception of `bool`, things look pretty good here: all of the missing values have been preserved. However, as we'll see in the following sections, things are not quite as straightforward as they seem.

9.4.1.1   Integers

With integers, missing values are stored as the smallest integer. If you don't do anything to them, they'll be preserved. But, since C++ doesn't know that the smallest integer has this special behaviour, if you do anything to it you're likely to get an incorrect value: for example, `evalCpp('NA_INTEGER + 1')` gives -2147483647.

So if you want to work with missing values in integers, either use a length one `IntegerVector` or be very careful with your code.

9.4.1.2 Doubles

With doubles, you may be able to get away with ignoring missing values and working with NaNs (not a number). This is because R's NA is a special type of IEEE 754 floating point number NaN. So any logical expression that involves a NaN (or in C++, NAN) always evaluates as FALSE:

```
evalCpp("NAN == 1")
#> [1] FALSE
evalCpp("NAN < 1")
#> [1] FALSE
evalCpp("NAN > 1")
#> [1] FALSE
evalCpp("NAN == NAN")
#> [1] FALSE
```

(Here I'm using `evalCpp()` which allows you to see the result of running a single C++ expression, making it excellent for this sort of interactive experimentation.)

But be careful when combining them with boolean values:

```
evalCpp("NAN && TRUE")
#> [1] TRUE
evalCpp("NAN || FALSE")
#> [1] TRUE
```

However, in numeric contexts NaNs will propagate NAs:

```
evalCpp("NAN + 1")
#> [1] NaN
evalCpp("NAN - 1")
#> [1] NaN
evalCpp("NAN / 1")
#> [1] NaN
evalCpp("NAN * 1")
#> [1] NaN
```

9.4.2 Strings

`String` is a scalar string class introduced by Rcpp, so it knows how to deal with missing values.

9.4.3   Boolean

While C++'s `bool` has two possible values (`true` or `false`), a logical vector in
R has three (`TRUE`, `FALSE`, and `NA`). If you coerce a length 1 logical vector, make
sure it doesn't contain any missing values otherwise they will be converted to
TRUE. An easy fix is to use `int` instead, as this can represent `TRUE`, `FALSE`, and
`NA`.

9.4.4   Vectors

With vectors, you need to use a missing value specific to the type of vector,
`NA_REAL`, `NA_INTEGER`, `NA_LOGICAL`, `NA_STRING`:

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

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

```
str(missing_sampler())
#> List of 4
#>  $ : num NA
#>  $ : int NA
#>  $ : logi NA
#>  $ : chr NA
```

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

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

// [[Rcpp::export]]
LogicalVector is_naC(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;
}
```

```
is_naC(c(NA, 5.4, 3.2, NA))
#> [1]  TRUE FALSE FALSE  TRUE
```

Another alternative is the sugar function `is_na()`, which takes a vector and returns a logical vector.

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

// [[Rcpp::export]]
LogicalVector is_naC2(NumericVector x) {
  return is_na(x);
}
```

```
is_naC2(c(NA, 5.4, 3.2, NA))
#> [1]  TRUE FALSE FALSE  TRUE
```

9.4.5 Exercises

1. Rewrite any of the functions from the first exercise to deal with missing values. If `na.rm` is true, ignore the missing values. If `na.rm` is false, return a missing value if the input contains any missing values. Some good functions to practice with are `min()`, `max()`, `range()`, `mean()`, and `var()`.

2. Rewrite `cumsum()` and `diff()` so they can handle missing values. Note that these functions have slightly more complicated behaviour.

9.5 The STL

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

If you need an algorithm or data structure that isn't implemented in STL, a good place to look is boost (http://www.boost.org/doc/). Installing boost on 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.hpp>`.

9.5.1   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:

1. Advance with `++`.
2. Get the value they refer to, or **dereference**, with `*`.
3. Compare with `==`.

For example we could re-write our sum function using iterators:

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

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

  NumericVector::iterator it;
  for(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 optimization 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.

This code can be simplified still further through the use of a C++11 feature: range-based for loops. C++11 is widely available, and can easily be activated for use with Rcpp by adding `[[Rcpp::plugins(cpp11)]]`.

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

// [[Rcpp::export]]
double sum4(NumericVector xs) {
  double total = 0;

  for(const auto &x : xs) {
    total += x;
  }
  return total;
}
```

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 a starting and an ending iterator, and adds up all the values in the vector. The third argument to accumulate gives the initial value: it's particularly important because this also determines the data type that accumulate uses (so 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>
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum5(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 0.0);
}
```

9.5.2 Algorithms

The `<algorithm>` header provides a large number of algorithms that work with iterators. A good reference is available at https://en.cppreference.com/w/cpp/algorithm. For example, we could write a basic Rcpp version of `findInterval()` that takes two arguments a vector of values and a vector of breaks, and locates 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>
#include <Rcpp.h>
using namespace Rcpp;

// [[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(breaks.begin(), pos);
  }

  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 handwritten C code), we need to compute the calls to `.begin()`
  and `.end()` once and save the results. 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 Meyers 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.

9.5.3 Data structures

The STL provides a large set of data structures: `array`, `bitset`, `list`, `forward_list`, `map`, `multimap`, `multiset`, `priority_queue`, `queue`, `deque`, `set`, `stack`, `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`, and `vector`. The most important of these data structures 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 trade-offs. 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 https://en.cppreference.com/w/cpp/container — 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.

9.5.4 Vectors

An STL vector is very similar to an R vector, except that it grows 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<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.

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

// [[Rcpp::export]]
List rleC(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);

NumericVector::iterator it;
for(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 https://en.cppreference.com/w/
cpp/container/vector.

9.5.5   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::unordered_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. More details are available at https://en.cppreference.com/w/cpp/container/set and https://en.cppreference.com/w/cpp/container/unordered_set.

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::plugins(cpp11)]]
#include <Rcpp.h>
#include <unordered_set>
using namespace Rcpp;

// [[Rcpp::export]]
LogicalVector duplicatedC(IntegerVector x) {
  std::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;
}
```

9.5.6   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::unordered_map`) versions. Since maps have a value and a key, you need to specify both types when initialising a map: `map<double, int>`, `unordered_map<int, double>`, and so on. The following example shows how you could use a `map` to implement `table()` for numeric vectors:

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

// [[Rcpp::export]]
std::map<double, int> tableC(NumericVector x) {
```

```cpp
  std::map<double, int> counts;

  int n = x.size();
  for (int i = 0; i < n; i++) {
    counts[x[i]]++;
  }

  return counts;
}
```

9.5.7   Exercises

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

1. `median.default()` using `partial_sort`.

2. `%in%` using `unordered_set` and the `find()` or `count()` methods.

3. `unique()` using an `unordered_set` (challenge: do it in one line!).

4. `min()` using `std::min()`, or `max()` using `std::max()`.

5. `which.min()` using `min_element`, or `which.max()` using `max_element`.

6. `setdiff()`, `union()`, and `intersect()` for integers using sorted ranges and `set_union`, `set_intersection` and `set_difference`.

9.6   Case studies

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

9.6.1   Gibbs sampler

The following case study updates an example blogged about (http://dirk.eddelbuettel.com/blog/2011/07/14/) 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 random number generator functions in GSL (easily accessible from R through the RcppGSL package) can make it another 2–3x faster.

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

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

// [[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, 1 / (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:

```
bench::mark(
  gibbs_r(100, 10),
  gibbs_cpp(100, 10),
  check = FALSE
)
#> # A tibble: 2 x 10
#>   expression      min     mean   median     max `itr/sec` mem_alloc
#>   <chr>       <bch:tm> <bch:tm> <bch:tm> <bch:>     <dbl> <bch:byt>
#> 1 gibbs_r(1…   5.64ms   5.83ms   5.74ms 6.81ms      172.    4.97MB
#> 2 gibbs_cpp… 306.05µs 350.71µs 344.11µs 1.49ms     2851.     4.1KB
#> # … with 3 more variables: n_gc <dbl>, n_itr <int>,
#> #   total_time <bch:tm>
```

9.6.2   R vectorisation vs. C++ vectorisation

This example is adapted from "Rcpp is smoking fast for agent-based models
in data frames" (https://gweissman.github.io/babelgraph/blog/2017/06/15/
rcpp-is-smoking-fast-for-agent-based-models-in-data-frames.html).   The chal-
lenge is to predict a model response from three inputs. The basic R version of
the predictor 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 vector-input 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 straightforward. In R:

```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 recognise some potential bottlenecks in this code: `ifelse`, `pmin`, and `pmax` are known to be slow, and could be replaced with `p * 0.75 + p * 0.5 * female`, `p[p < 0] <- 0`, `p[p > 1] <- 1`. You might want to try timing those variations yourself.)

Or in C++:

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

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 introduced a bug in the C++ version: it used `0.004` instead of `0.04`. Finally, we can benchmark our three approaches:

```
bench::mark(
  vacc1 = vacc1(age, female, ily),
  vacc2 = vacc2(age, female, ily),
  vacc3 = vacc3(age, female, ily)
)
#> # A tibble: 3 x 10
#>   expression      min     mean    median        max `itr/sec` mem_alloc
#>   <chr>       <bch:tm> <bch:tm> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 vacc1         1.83ms   1.97ms   1.93ms   3.57ms      508.    7.86KB
#> 2 vacc2        100.5µs 120.14µs 108.88µs 444.87µs     8323.  223.98KB
#> 3 vacc3         29.2µs  31.57µs  30.42µs 185.43µs    31676.   14.48KB
#> # … with 3 more variables: n_gc <dbl>, n_itr <int>,
#> #   total_time <bch:tm>
```

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.

9.7 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 stand-alone C++ source file to a package:

1. Your code can be made available to users without C++ development tools.

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

- In `DESCRIPTION` add

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

- Make sure your `NAMESPACE` includes:

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

  We need to import something (anything) from Rcpp so that internal Rcpp code is properly loaded. This is a bug in R and hopefully will be fixed in the future.

The easiest way to set this up automatically is to call `usethis::use_rcpp()`. Before building the package, you'll need to run `Rcpp::compileAttributes()`. This function scans the C++ files for `Rcpp::export` attributes and generates the code required to make the functions available in R. Re-run `compileAttributes()` whenever functions are added, removed, or have their signatures changed. This is done automatically by the devtools package and by Rstudio.

For more details see the Rcpp package vignette, `vignette("Rcpp-package")`.

9.8 Learning more

This chapter has only touched on a small part of Rcpp, giving you the basic tools to rewrite poorly performing R code in C++. As noted, 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, `vignette("Rcpp-attributes")`.

- 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 Rcpp modules vignette, `vignette("Rcpp-modules")`

- The Rcpp quick reference guide, `vignette("Rcpp-quickref")`, contains a useful summary of Rcpp classes and common programming idioms.

I strongly recommend keeping an eye on the Rcpp homepage (http://www.rcpp.org) and signing up for the Rcpp mailing list (http://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/rcpp-devel).

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

- "Effective C++" (Meyers 2005) and "Effective STL" (Meyers 2001).

- *C++ Annotations* (http://www.icce.rug.nl/documents/cplusplus/cplusplus.html), 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* (http://www.cs.helsinki.fi/u/tpkarkka/alglib/k06/), which provides a more technical, but still concise, description of important STL concepts. (Follow the links under notes).

Writing performance code may also require you to rethink your basic approach: a solid understanding 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" (Skiena 1998), MIT's *Introduction to Algorithms* (http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/), *Algorithms* by Robert Sedgewick and Kevin Wayne which has a free online textbook (http://algs4.cs.princeton.edu/home/) and a matching Coursera course (https://www.coursera.org/learn/algorithms-part1).

9.9   Acknowledgments

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.

References

Balamuta, James. 2018a. *Errorist: Automatically Search Errors or Warnings*. https://github.com/coatless/errorist.

———. 2018b. *Searcher: Query Search Interfaces*. https://github.com/coatless/searcher.

Bawden, Alan. 1999. "Quasiquotation in Lisp." In *PEPM '99*, 4–12. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.309.227.

Bueno, Carlos. 2013. *Mature Optimization Handbook*. http://carlos.bueno.org/optimization/.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. https://doi.org/10.18637/jss.v040.i08.

Fowler, Martin. 2010. *Domain-Specific Languages*. Pearson Education. http://amzn.com/0321712943.

Henry, Lionel, and Hadley Wickham. 2018. *Rlang: Tools for Low-Level R Programming*. https://rlang.r-lib.org.

Hester, Jim. 2018. *Bench: High Precision Timing of R Expressions*. http://bench.r-lib.org/.

Hunt, Andrew, and David Thomas. 1990. *The Pragmatic Programmer*. Addison Wesley.

Lumley, Thomas. 2001. "Programmer's Niche: Macros in R." *R News* 1 (3): 11–13. https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf.

Matloff, Norman. 2011. *The Art of R Programming*. No Starch Press.

———. 2015. *Parallel Computing for Data Science*. Chapman & Hall/CRC. http://amzn.com/1466587016.

McCallum, Q. Ethan, and Steve Weston. 2011. *Parallel R*. O'Reilly. http://amzn.com/B005Z29QT4.

Meyers, Scott. 2001. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Pearson Education. http://amzn.com/0201749629.

———. 2005. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Pearson Education. http://amzn.com/0321334876.

Morandat, Floréal, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. "Evaluating the Design of the R Language." In *European Conference on Object-Oriented Programming*, 104–31. Springer. http://r.cs.purdue.edu/pub/ecoop12.pdf.

Skiena, Steven S. 1998. *The Algorithm Design Manual*. Springer Science & Business Media. http://amzn.com/0387948600.

Tierney, Luke, and Riad Jarjour. 2016. *Proftools: Profile Output Processing Tools for R*. https://CRAN.R-project.org/package=proftools.

Index