7. Using the `readr` Package

The `readr` package is recently developed by Hadley Wickham to deal with reading in large flat files quickly. The package provides replacements for functions like `read.table()` and `read.csv()`. The analogous functions in `readr` are `read_table()` and `read_csv()`. These functions are often *much* faster than their base R analogues and provide a few other nice features such as progress meters.

For the most part, you can read use `read_table()` and `read_csv()` pretty much anywhere you might use `read.table()` and `read.csv()`. In addition, if there are non-fatal problems that occur while reading in the data, you will get a warning and the returned data frame will have some information about which rows/observations triggered the warning. This can be very helpful for "debugging" problems with your data before you get neck deep in data analysis.

The importance of the `read_csv` function is perhaps better understood from an historical perspective. R's built in `read.csv` function similarly reads CSV files, but the `read_csv` function in `readr` builds on that by removing some of the quirks and "gotchas" of `read.csv` as well as dramatically optimizing the speed with which it can read data into R. The `read_csv` function also adds some nice user-oriented features like a progress meter and a compact method for specifying column types.

A typical call to `read_csv` will look as follows.

```
> library(readr)
> teams <- read_csv("data/team_standings.csv")
Parsed with column specification:
cols(
  Standing = col_integer(),
  Team = col_character()
)
> teams
# A tibble: 32 × 2
   Standing        Team
      <int>       <chr>
1         1       Spain
2         2 Netherlands
3         3     Germany
4         4     Uruguay
5         5   Argentina
6         6      Brazil
7         7       Ghana
8         8    Paraguay
9         9       Japan
10       10       Chile
# ... with 22 more rows
```

By default, `read_csv` will open a CSV file and read it in line-by-line. It will also (by default), read in the first few rows of the table in order to figure out the type of each column (i.e. integer, character, etc.). From the `read_csv` help page:

> If 'NULL', all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to supply the correct types yourself.

You can specify the type of each column with the `col_types` argument.

In general, it's a good idea to specify the column types explicitly. This rules out any possible guessing errors on the part of `read_csv`. Also, specifying the column types explicitly provides a useful safety check in case anything about the dataset should change without you knowing about it.

```
> teams <- read_csv("data/team_standings.csv", col_types = "cc")
```

Note that the `col_types` argument accepts a compact representation. Here `"cc"` indicates that the first column is `character` and the second column is `character` (there are only two columns). Using the `col_types` argument is useful because often it is not easy to automatically figure out the type of a column by looking at a few rows (especially if a column has many missing values).

The `read_csv` function will also read compressed files automatically. There is no need to decompress the file first or use the `gzfile` connection function. The following call reads a gzip-compressed CSV file containing download logs from the RStudio CRAN mirror.

```
> logs <- read_csv("data/2016-07-19.csv.bz2", n_max = 10)
Parsed with column specification:
cols(
  date = col_date(format = ""),
  time = col_time(format = ""),
  size = col_integer(),
  r_version = col_character(),
  r_arch = col_character(),
  r_os = col_character(),
  package = col_character(),
  version = col_character(),
  country = col_character(),
  ip_id = col_integer()
)
```

Note that the warnings indicate that `read_csv` may have had some difficulty identifying the type of each column. This can be solved by using the `col_types` argument.

```
> logs <- read_csv("data/2016-07-19.csv.bz2", col_types = "ccicccccci", n_max = 10)
> logs
# A tibble: 10 × 10
         date     time     size r_version r_arch         r_os    package
        <chr>    <chr>    <int>     <chr>  <chr>        <chr>      <chr>
1  2016-07-19 22:00:00  1887881     3.3.0 x86_64      mingw32 data.table
2  2016-07-19 22:00:05    45436     3.3.1 x86_64      mingw32 assertthat
3  2016-07-19 22:00:03 14259016     3.3.1 x86_64      mingw32    stringi
4  2016-07-19 22:00:05  1887881     3.3.1 x86_64      mingw32 data.table
5  2016-07-19 22:00:06   389615     3.3.1 x86_64      mingw32    foreach
6  2016-07-19 22:00:08    48842     3.3.1 x86_64    linux-gnu       tree
7  2016-07-19 22:00:12      525     3.3.1 x86_64 darwin13.4.0   survival
8  2016-07-19 22:00:08  3225980     3.3.1 x86_64      mingw32       Rcpp
9  2016-07-19 22:00:09   556091     3.3.1 x86_64      mingw32     tibble
10 2016-07-19 22:00:10   151527     3.3.1 x86_64      mingw32   magrittr
# ... with 3 more variables: version <chr>, country <chr>, ip_id <int>
```

You can specify the column type in a more detailed fashion by using the various `col_*` functions. For example, in the log data above, the first column is actually a date, so it might make more sense to read it in as a Date variable. If we wanted to just read in that first column, we could do

```
> logdates <- read_csv("data/2016-07-19.csv.bz2",
+                      col_types = cols_only(date = col_date()),
+                      n_max = 10)
> logdates
# A tibble: 10 × 1
         date
       <date>
1  2016-07-19
2  2016-07-19
3  2016-07-19
4  2016-07-19
5  2016-07-19
6  2016-07-19
7  2016-07-19
8  2016-07-19
9  2016-07-19
10 2016-07-19
```

Now the `date` column is stored as a `Date` object which can be used for relevant date-related computations (for example, see the `lubridate` package).

The `read_csv` function has a `progress` option that defaults to TRUE. This options provides a nice progress meter while the CSV file is being read. However, if you are using `read_csv` in a function, or perhaps embedding it in a loop, it’s probably best to set `progress = FALSE`.