

LOADING

LOADING

RSTUDIO::CONF 2017

# WRITING READABLE CODE WITH PIPES

Bob Rudis • [Master] Chef des Données de Sécurité @ Rapid7



Welcome to Writing Readable Code With Pipes.

I hope y'all have had a great time so far at rstudio::conf 2017!

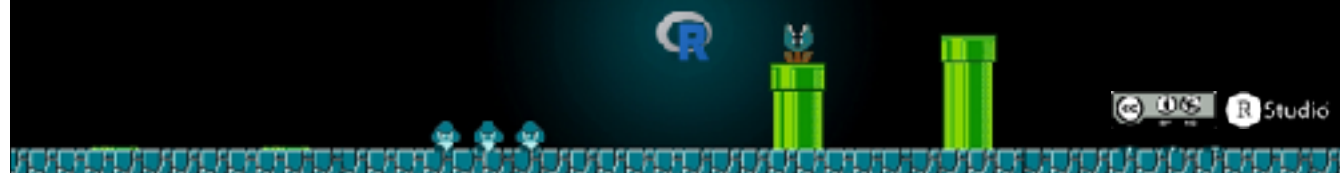
Pipes are fun! Just ask Mario. They're get you from point A to point B but — just like in the games — figuring out the best way to use the pipes will make you more productive and write better code.

RSTUDIO::CONF 2017

# WRITING READABLE CODE WITH PIPES

Bob Rudis • [Master] Chef des Données de Sécurité @ Rapid7

INSERT COIN



Welcome to Writing Readable Code With Pipes.

I hope y'all have had a great time so far at rstudio::conf 2017!

Pipes are fun! Just ask Mario. They're get you from point A to point B but — just like in the games — figuring out the best way to use the pipes will make you more productive and write better code.

RSTUDIO::CONF 2017

# WRITING READABLE CODE WITH PIPES

Bob Rudis • [Master] Chef des Données de Sécurité @ Rapid7

INSERT COIN



Welcome to Writing Readable Code With Pipes.

I hope y'all have had a great time so far at rstudio::conf 2017!

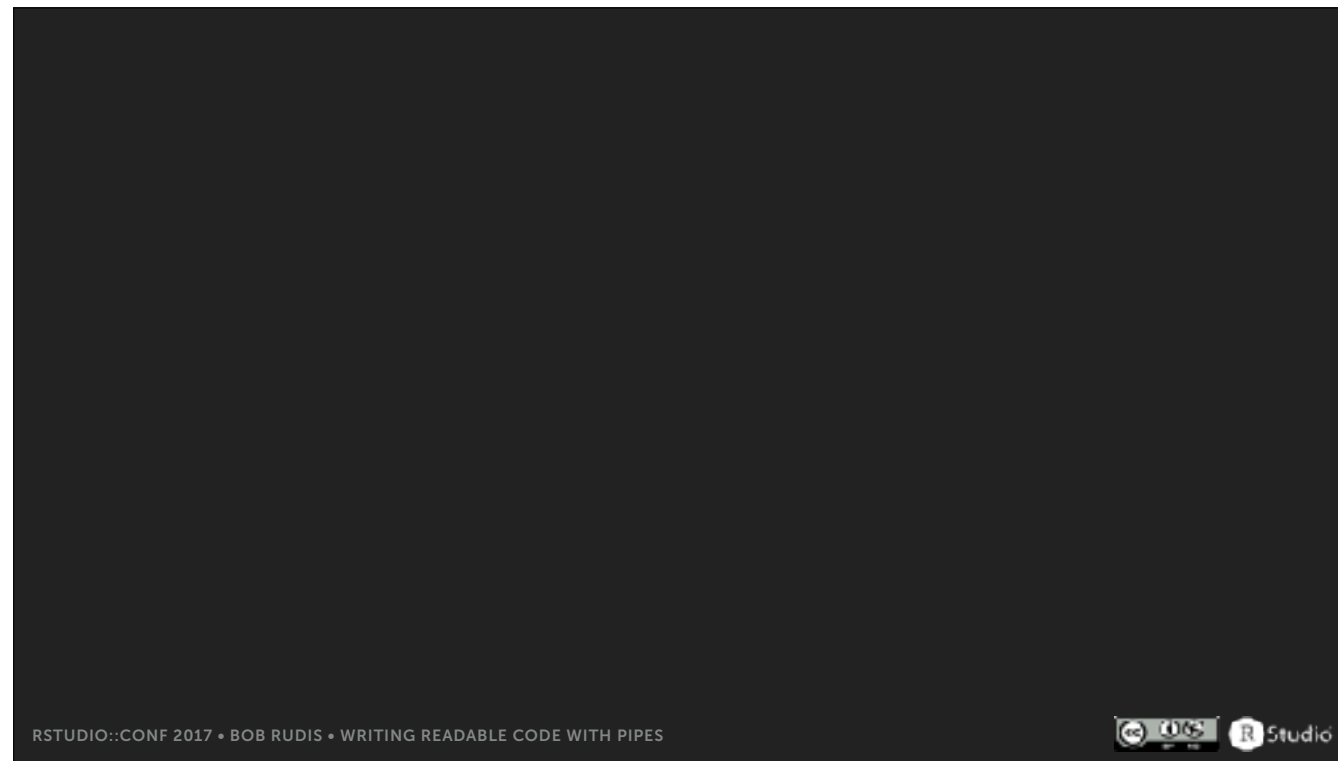
Pipes are fun! Just ask Mario. They're get you from point A to point B but — just like in the games — figuring out the best way to use the pipes will make you more productive and write better code.



Who am I?

I'm Bob Rudis : Chief Data Scientist @ Rapid7 — a firm that focuses on making our world a safer & more secure place for everyone. I'm also the co-author of Data-Driven Security, a book full of outdated R code (not even R 3.0!) but very relevant concepts for getting started in a data mindset in cybersecurity.

If the shield doesn't give you social context about me, wait for the last slide or come find me over the next day to talk about the tidyverse, cybersecurity, cooking or Captain America.



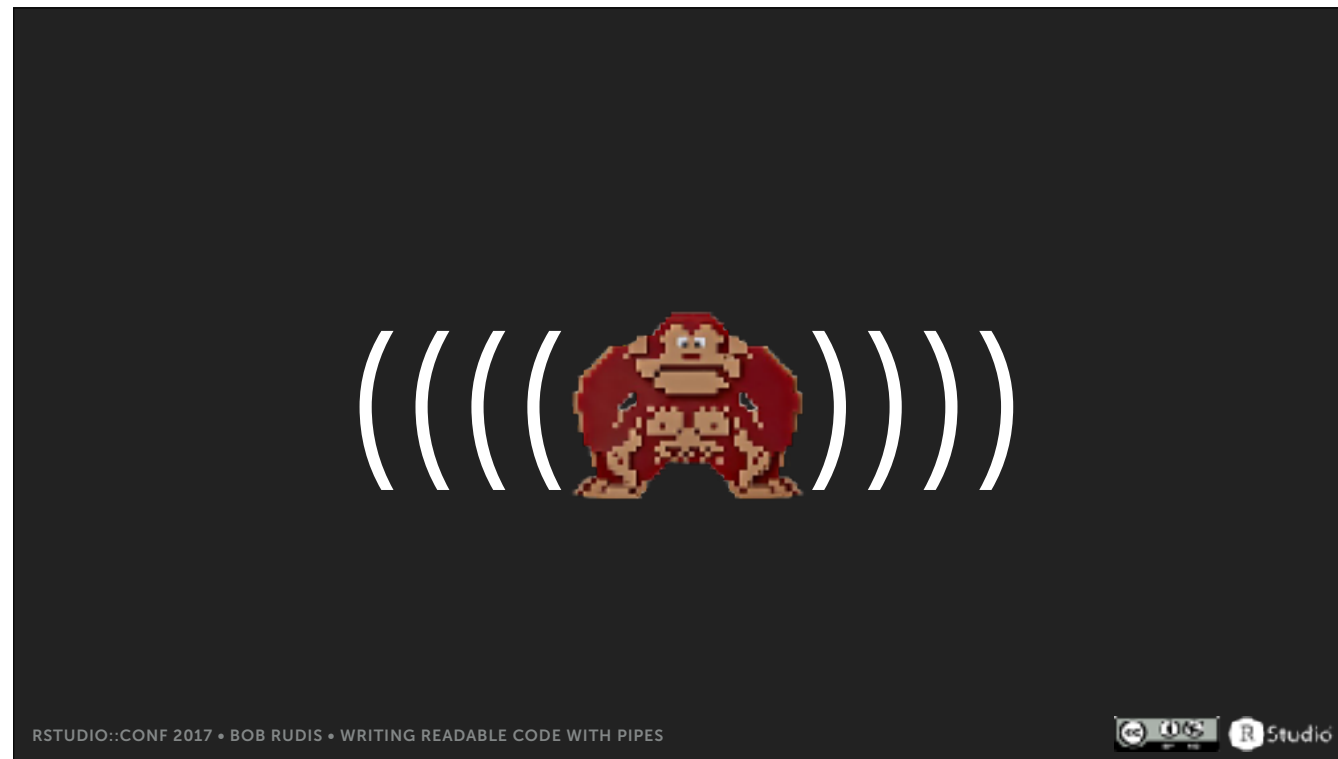
If you’ve been on the “right side” of the talk tracks today, you’ve already seen pipes. How many folks have seen / have used / know what pipes are? [REACT]

I’m here to give you some tips on how to improve your code, workflow and relationships with your co-workers and future self by making more write-able, read-able and use-able code with pipes.

In various slides you’re going to see this guy [CLICK](without the noise). When you see him, that’s a bad sign and usually an indicator of less-than-optimal practices (I can’t say “bad” since code that works is ultimately good code — at least for one run).

You’re also going to see this guy. [CLICK] True fact:, that is the only remaining picture of Hadley with just a mustache and in coveralls.

Seriously, though, when you see Mario that’s an indication of super ++gd practice (he’ll be giving a thumbs-up, too).



If you've been on the "right side" of the talk tracks today, you've already seen pipes. How many folks have seen / have used / know what pipes are? [REACT]

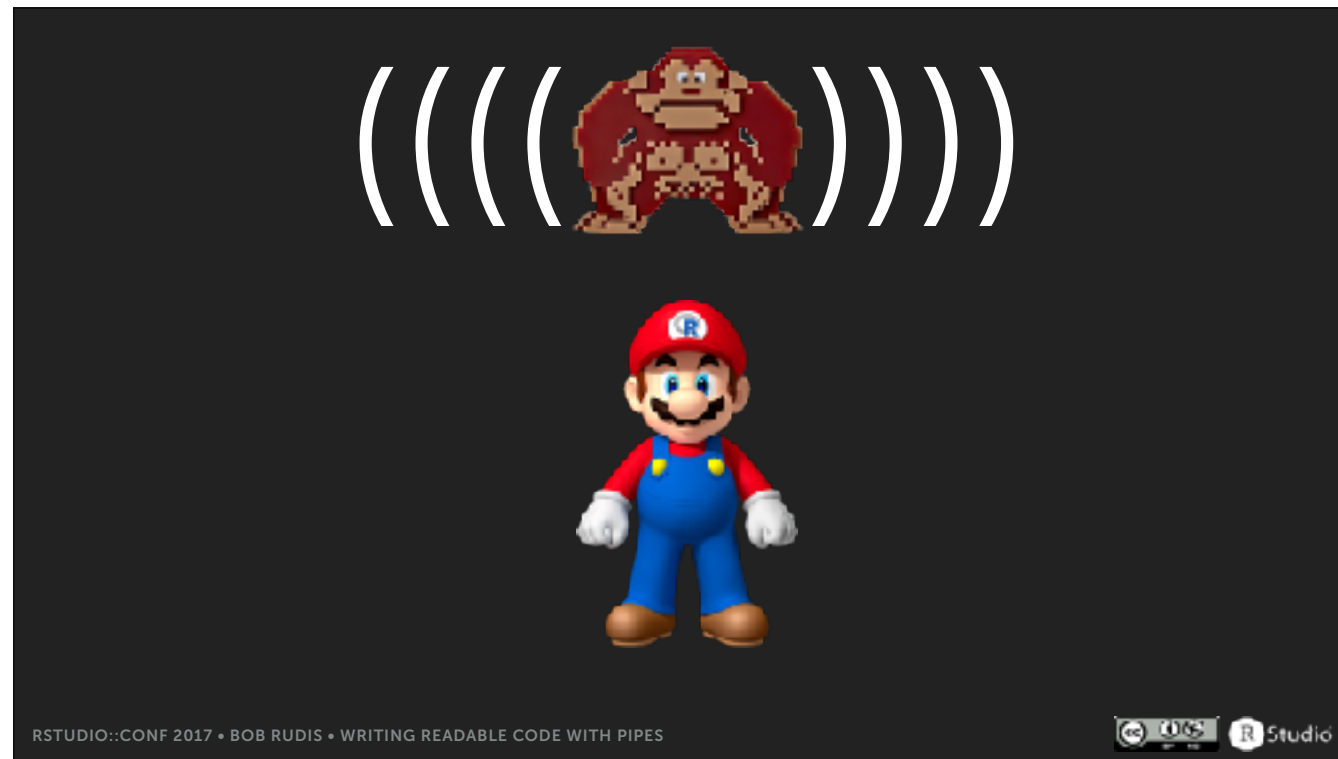
I'm here to give you some tips on how to improve your code, workflow and relationships with your co-workers and future self by making more write-able, read-able and use-able code with pipes.

In various slides you're going to see this guy [CLICK](without the noise). When you see him, that's a bad sign and usually an indicator of less-than-optimal practices (I can't say "bad" since code that works is ultimately good code — at least for one run).

You're also going to see this guy. [CLICK] True fact:, that is the only remaining picture of Hadley with just a mustache and in coveralls.

Seriously, though, when you see Mario that's an indication of super ++gd practice (he'll be giving a thumbs-up, too).





If you’ve been on the “right side” of the talk tracks today, you’ve already seen pipes. How many folks have seen / have used / know what pipes are? [REACT]

I’m here to give you some tips on how to improve your code, workflow and relationships with your co-workers and future self by making more write-able, read-able and use-able code with pipes.

In various slides you’re going to see this guy [CLICK](without the noise). When you see him, that’s a bad sign and usually an indicator of less-then-optimal practices (I can’t say “bad” since code that works is ultimately good code — at least for one run).

You’re also going to see this guy. [CLICK] True fact:, that is the only remaining picture of Hadley with just a mustache and in coveralls.

Seriously, though, when you see Mario that’s an indication of super ++gd practice (he’ll be giving a thumbs-up, too).

## PIPES : QUICK REFRESHER

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



What are these “pipes” again? [CLICK] Let’s start with a refresher course in the revolution of 2014 [CLICK] (we’re 6 days away from the anniversary!) when this new operator was introduced by Stefan in his magrittr package . We’ll just be using the pipe symbol today and not delving into the dark art behind it. You should take a look at the pkg source on github to see all the work that goes on under the covers to enable the rich functionality in this new operator. The package name comes from a famous painting by René Magritte. More on that at the end of the talk.

## PIPES : QUICK REFRESHER

`%>%`

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



What are these “pipes” again? [CLICK] Let’s start with a refresher course in the revolution of 2014 [CLICK] (we’re 6 days away from the anniversary!) when this new operator was introduced by Stefan in his `magrittr` package . We’ll just be using the pipe symbol today and not delving into the dark art behind it. You should take a look at the pkg source on github to see all the work that goes on under the covers to enable the rich functionality in this new operator. The package name comes from a famous painting by René Magritte. More on that at the end of the talk.

## PIPES : QUICK REFRESHER



Package: magrittr  
Type: Package  
Title: magrittr - a forward-pipe operator for R  
Version: 1.0.0  
Date: 2014-01-19  
Author: Stefan Milton Bache <stefan@stefanbache.dk> and  
Hadley Wickham <h.wickham@gmail.com>  
Description: Provides a mechanism for chaining commands with a  
new forward-pipe operator. Ceci n'est pas un pipe.

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



What are these “pipes” again? [CLICK] Let’s start with a refresher course in the revolution of 2014 [CLICK] (we’re 6 days away from the anniversary!) when this new operator was introduced by Stefan in his magrittr package . We’ll just be using the pipe symbol today and not delving into the dark art behind it. You should take a look at the pkg source on github to see all the work that goes on under the covers to enable the rich functionality in this new operator. The package name comes from a famous painting by René Magritte. More on that at the end of the talk.

```
DT
DiagrammeR
GerminaR
HydeNet
Momocs
SciencesPo
analogsea
archivist
binomen
blscrapeR
bpa
ckanr
curlconverter
dat
datadr
ddpcr
dendextend
diffrprojectswidget
diffrprojects
dplyr
dygraphs
emil
forcats
fulltext
gRain
gRbase
geojson
geojsonio
ggvis
gistr
gmailr
gogamer
googleAnalyticsR
highcharter
htping
igraph
jq
lawn
leaflet
leafletCN
lightsout
magrittr
mason
metacoder
metricsgraphics
modelr
multipanelfigure
nlstimedist
operators
pdp
pixiedust
plotly
poppr
ptstem
purrr
qdap
rbokeh
rdrop2
request
rex
rgbif
rhandsontable
rrr
rscorecard
rslp
rtext
rvest
saeSim
scrubr
simmer
simulator
sparklyr
srvyr
stringr
tableHTML
testthat
text2vec
tidyr
tigris
timevis
tmertools
vcfR
vegalite
vembedr
visNetwork
webshot
wellknown

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES
```

Since the introduction of the magrittr pipe there are 81 packages that import/export the operator (I went through the NAMESPACE of every pkg on CRAN). As an aside, it seems I have a pkg in almost every column. Clearly I've been slacking off a bit.

In reality, [CLICK] 81 out of ~10K is a teensy tiny percentage of all the pkgs on CRAN; perhaps the equivalent, of one heart health for Mario. I'll prbly track this every couple of months this year, so watch the blog for details.

81

export("%>%")

DT  
DiagrammeR  
GerminaR  
HydeNet  
Momocs  
SciencesPo  
analogsea  
archivist  
binomen  
blscrapeR  
bpa  
ckanr  
curlconverter  
dat  
datadr  
ddpcr  
dendextend  
diffrprojectswidget  
diffrprojects  
dplyr  
dygraphs  
emil  
forcats  
fulltext  
gRain

gRbase  
geojson  
geojsonio  
ggvis  
gistr  
gmailr  
gogamer  
googleAnalyticsR  
highcharter  
htping  
igraph  
jqr  
lawn  
leaflet  
leafletCN  
lightsout  
magrittr  
mason  
metacoder  
metricsgraphics  
modelr  
multipanelfigure  
nlstimedist  
operators  
pdp

pixiedust  
plotly  
poppr  
ptstem  
purrr  
qdap  
rbokeh  
rdrop2  
request  
rex  
rgbif  
rhandsontable  
rrr  
rscorecard  
rslp  
rtext  
rvest  
saeSim  
scrubr  
simmer  
simulator  
sparklyr  
srvyr  
stringr  
tableHTML

testthat  
text2vec  
tidyr  
tigris  
timevis  
tmertools  
vcfR  
vegalite  
vembedr  
visNetwork  
webshot  
wellknown

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES




Since the introduction of the magrittr pipe there are 81 packages that import/export the operator (I went through the NAMESPACE of every pkg on CRAN). As an aside, it seems I have a pkg in almost every column. Clearly I've been slacking off a bit.

In reality, [CLICK] 81 out of ~10K is a teensy tiny percentage of all the pkgs on CRAN; perhaps the equivalent, of one heart health for Mario. I'll prbly track this every couple of months this year, so watch the blog for details.



# 81

`export("%>%")`

DT  
DiagrammeR  
GerminaR  
HydeNet  
Momocs  
SciencesPo  
analogsea  
archivist  
binomen  
blscrapeR  
bpa  
ckanr  
curlconverter  
dat  
datadr  
ddpcr  
dendextend  
diffrprojectswidget  
diffrprojects  
dplyr  
dygraphs  
emil  
forcats  
fulltext  
gRain

gRbase  
geojson  
geojsonio  
ggvis  
gistr  
gmailr  
gogamer  
googleAnalyticsR  
highcharter  
htping  
igraph  
jqr  
lawn  
leaflet  
leafletCN  
lightsout  
magrittr  
mason  
metacoder  
metricsgraphics  
modelr  
multipanelfigure  
nlstimedist  
operators  
pdp

pixiedust  
plotly  
poppr  
ptstem  
purrr  
qdap  
rbokeh  
rdrop2  
request  
rex  
rgbif  
rhandsontable  
rrr  
rscorecard  
rslp  
rtext  
rvest  
saeSim  
scrubr  
simmer  
simulator  
sparklyr  
srvyr  
stringr  
tableHTML

testthat  
text2vec  
tidyr  
tigris  
timevis  
tmertools  
vcfR  
vegalite  
vembedr  
visNetwork  
webshot  
wellknown

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Since the introduction of the magrittr pipe there are 81 packages that import/export the operator (I went through the NAMESPACE of every pkg on CRAN). As an aside, it seems I have a pkg in almost every column. Clearly I've been slacking off a bit.

In reality, [CLICK] 81 out of ~10K is a teensy tiny percentage of all the pkgs on CRAN; perhaps the equivalent, of one heart health for Mario. I'll prbly track this every couple of months this year, so watch the blog for details.

**What does “*forward chaining*” look like?**



object %>% operation() → result

`object %>% operation() → result`

I like to think of it as taking data (an R object) and

object %>% operation() → result

I like to think of it as taking data (an R object) and

object %>% operation() → result

to end up with a result (which doesn't have to look like the original data object at all)

object %>% operation() → result

DOESN'T NEED  
TO BE THE  
SAME CLASS/  
MODE/TYPE AS  
OBJECT

to end up with a result (which doesn't have to look like the original data object at all)

```
result ← operation(object)
```

just like this

```
operation ← function(data, param1, ...) {  
  data ← do_stuff_with(data)  
  data ← do_even_morestuff_with(data)  
  data  
}
```

“operation” is just an R function like you’ve written before. It takes in data and parameters, does stuff with it and spits out the new object.

```
operation ← function(data, param1, ...) {  
  data ← do_stuff_with(data)  
  data ← do_even_morestuff_with(data)  
  data  
}
```

the super important thing to remember is “data first”. Your functions must/should be written with that in mind.



"DATA FIRST"



```
operation ← function(data, param1, ...) {  
  data ← do_stuff_with(data)  
  data ← do_even_morestuff_with(data)  
  data  
}
```

the super important thing to remember is “data first”. Your functions must/should be written with that in mind.

“DATA NOT FIRST”



```
lm(formula, data, subset, weights, na.action,  
    method = "qr", model = TRUE, x = FALSE,  
    y = FALSE, qr = TRUE, singular.ok = TRUE,  
    contrasts = NULL, offset, ...)
```

You may say “duh”, but there are some seminal examples of this not being the case in R... And, there’s our friend letting us know we shouldn’t do this.

“DATA NOT FIRST”



```
lm(formula, data, subset, weights, na.action,  
    method = "qr", model = TRUE, x = FALSE,  
    y = FALSE, qr = TRUE, singular.ok = TRUE,  
    contrasts = NULL, offset, ...)
```

You may say “duh”, but there are some seminal examples of this not being the case in R... And, there’s our friend letting us know we shouldn’t do this.

```
str(mtcars)
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



A real life example of a real pipe would be to take this

```
mtcars %>% str()
```

and turn it into this



```
mtcars %>% str() # DON'T DO THIS
```

except that you really shouldn't



```
mtcars %>% summary() # OR THIS
```

really, srsly don't do this

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be > 1

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Why? It violates rule #1.

These are some of my personal piping rules/guidelines. There aren't many. I think they make sense. You need to do what works well for you.

For single operation pipes, I believe there's an expectation that a single function call that's not part of a chain (we'll see a full chain in second) has a contract to have it be called normally. This is just my own rule, so feel totally free to disagree. Whatever gets your work done and works well with your team is paramount. Let's move on to a larger example.



((()))

```
cat(crayon::red(  
  paste0(sprintf("I <3 %s",  
    map_chr(  
      c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug=="),  
      ~rawToChar(  
        openssl::base64_decode(.))))), collapse="\n"))
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Raise your hand if you've seen code like this.




Now, raise your hand if you've written code like this.

This type of snippet is one reason our 8-bit nemesis is in parentheses. I imagine him rolling down barrels with each open parenthesis.

when you wrote this code you totally knew what it was going to do.

Now, you prbly have no idea and your future self is all hating on your former self.

((()))

```
cat(crayon::red(  
  paste0(sprintf("I <3 %s",  
    map_chr(  
      c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug=="),  
      ~rawToChar(  
        openssl::base64_decode(.))))), collapse="\n"))
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Raise your hand if you've seen code like this.

Now, raise your hand if you've written code like this.

This type of snippet is one reason our 8-bit nemesis is in parentheses. I imagine him rolling down barrels with each open parenthesis.

when you wrote this code you totally knew what it was going to do.

Now, you prbly have no idea and your future self is all hating on your former self.

((()))

```
msg <- c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==")

decoded_msg <- lapply(msg, openssl::base64_decode)
decoded_msg <- sapply(decoded_msg, rawToChar,
                      USE.NAMES = FALSE)

output <- paste0(sprintf("I <3 %s", decoded_msg),
                 collapse="\n")
output <- crayon::red(output)

cat(output)
```

This is a base-y, tidy-ish way of doing things but Hadley already talked abt why this is bad.

((()))

```
msg <- c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==")
```



```
decoded_msg <- lapply(msg, openssl::base64_decode)  
decoded_msg <- sapply(decoded_msg, rawToChar,  
                      USE.NAMES = FALSE)
```

```
output <- paste0(sprintf("I <3 %s", decoded_msg),  
                 collapse="\n")  
output <- crayon::red(output)
```

```
cat(output)
```

This is a base-y, tidy-ish way of doing things but Hadley already talked abt why this is bad.

((()))

```
msg ← c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==")
```



```
decoded_msg ← lapply(msg, openssl::base64_decode)  
decoded_msg ← sapply(decoded_msg, rawToChar,  
                      USE.NAMES = FALSE)
```



```
output ← paste0(sprintf("I <3 %s", decoded_msg),  
                 collapse="\n")  
output ← crayon::red(output)  
  
cat(output)
```

This is a base-y, tidy-ish way of doing things but Hadley already talked abt why this is bad.

```
c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==") %>%  
  map(openssl::base64_decode) %>%  
  map_chr(rawToChar) %>%  
  sprintf(fmt = "I <3 %s") %>%  
  paste0(collapse = "\n") %>%  
  crayon::red()  
cat()
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



This would be one example of translating that into a pipe. [CLICK]

We take the data and then push it through a group of what I'll call atomic operations to achieve an end result.

[CLICK]

Even if you're not 100% sure what those purrr functions do, you can prbly get a better idea - and more quickly - that this code prints out from a character vector in red.

You're prbly wondering what that says. Rather than make you type it...



```
c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==") %>%  
  map(openssl::base64_decode) %>%  
  map_chr(rawToChar) %>%  
  sprintf(fmt = "I <3 %s") %>%  
  paste0(collapse = "\n") %>%  
  crayon::red()  
cat()
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



This would be one example of translating that into a pipe. [CLICK]

We take the data and then push it through a group of what I'll call atomic operations to achieve an end result.

[CLICK]

Even if you're not 100% sure what those purrr functions do, you can prbly get a better idea - and more quickly - that this code prints out from a character vector in red.

You're prbly wondering what that says. Rather than make you type it...



```
c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==") %>%  
  map(openssl::base64_decode) %>%  
  map_chr(rawToChar) %>%  
  sprintf(fmt = "I <3 %s") %>%  
  paste0(collapse = "\n") %>%  
  crayon::red()  
cat()
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



This would be one example of translating that into a pipe. [CLICK]

We take the data and then push it through a group of what I'll call atomic operations to achieve an end result.

[CLICK]

Even if you're not 100% sure what those purrr functions do, you can prbly get a better idea - and more quickly - that this code prints out from a character vector in red.

You're prbly wondering what that says. Rather than make you type it...



```

c("dGhpcyBhdWRpZW5jZQ==", "bXkgZmFtaWx5", "Ug==") %>%
  map(openssl::base64_decode) %>%
  map_chr(rawToChar) %>%
  sprintf(fmt = "I <3 %s") %>%
  paste0(collapse = "\n") %>%
  crayon::red() %>%
  cat()

```



RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



This would be one example of translating that into a pipe. [CLICK]

We take the data and then push it through a group of what I'll call atomic operations to achieve an end result.

[CLICK]

Even if you're not 100% sure what those purrr functions do, you can prbly get a better idea - and more quickly - that this code prints out from a character vector in red.

You're prbly wondering what that says. Rather than make you type it...



```
I <3 this audience  
I <3 my family  
I <3 R
```

***Before we go any further...***

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



So, you've got a taste of how piping can make your code more readable but we need to talk about the elephant in the talk track

```
library(tidyverse)
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



the tidyverse.

```
library(tidyverse) {  
  ggplot2 (data vis)  
  dplyr   (data manip)  
  tidyr   (data tidying)  
  readr   (data import)  
  purrr   (functional prog)  
  tibble  (data.frame++)  
}
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



It's a wrapper package that loads the pkgs that make up the core of the tidyverse. These all subscribe to the “data first” mentality, are “verb” driven (naming a function with a verb so it's clear it's performing an operation). you can totally use piping w/o the tidyverse but it's much better to be in the tidyverse than without.

## Real-world example

Read in a directory of CSV/JSON/*whatev*  
into a data frame

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Drop a note on the github repo from this talk (link at the end) if you want more examples in different contexts. We don't have much time, so we'll just go through a few.

This is one we all likely have/had/seen.

```
cowrie_2017-01-03T004539.json.gz  
cowrie_2017-01-03T044239.json.gz  
cowrie_2017-01-03T065255.json.gz  
cowrie_2017-01-03T010619.json.gz  
cowrie_2017-01-03T044746.json.gz  
cowrie_2017-01-03T073818.json.gz  
cowrie_2017-01-03T014616.json.gz  
cowrie_2017-01-03T045817.json.gz
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



We're going to use `_my_` data, too. To have with your gene strings, literature extracts, stock quotes, and animal tracking data! This is honeypot data (logs from traps we set out for attackers)

I'm also giving some of this data away in the gh repo so you can get a chance to play with real live cybersecurity honeypot data.



((()))

```
library(ndjson)
library(anytime)

do.call(rbind, lapply(list.files("data/honeypot",
pattern="cowrie.*json.gz", full.names=TRUE), function(x) {
  df <- as.data.frame(stream_in(x))
  df <- df[,c("timestamp", "src_ip", "src_port", "sensor",
"session", "dst_port", "eventid", "username", "password")]
  df$timestamp <- anytime(df$timestamp)
  df
})) -> cowrie_df
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Apart from using two fairly new pkgs, this sld be familiar code to most folks.

get a list of files, turn each one into a data frame. possibly clean up said data frame and bind all the rows together.

it's also an ugly mess, even in an RStudio editor window.

```

stream_and_filter <- function(x) {
  df <- as.data.frame(stream_in(x))
  df <- df[,c("timestamp", "src_ip", "src_port", "sensor",
             "session", "dst_port", "eventid", "username",
             "password")]
  df$timestamp <- anytime(df$timestamp)
  df
}

files <- list.files("data/honeypot",
                   pattern="cowrie.*json.gz", full.names=TRUE)
cowrie_df_list <- lapply(files, stream_and_filter)
cowrie_df <- do.call(rbind, cowrie_df_list)

```

(((👑)))

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



this is slightly better. we're not in the piping tidyverse yet but at least your future self or your co-workers won't hate you as much.

We've separated out functionality and made the code a bit more modular and easier to read. but only a bit.

Let's improve on it in the tidyverse



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

that's the same code in the piping tidyverse [CLICK] and is a good example of rule #2



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```


that's the same code in the piping tidyverse [\[CLICK\]](#) and is a good example of rule #2

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be  $> 1$
- ▶ A pipe group should be designed to accomplish a unified task

Which is



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

the unified task here is to make a data frame from a list of files

(oh yeah, Hadley hates list.files() so don't tell him I used it but dir() reminds me of DOS and also makes me think I'm asking for directions).



```
filter(mtcars, cyl==6) %>%  
  count(gear) %>%  
  ggplot(aes(gear, n)) + geom_col()
```



```
filter(mtcars, cyl==6) %>%  
  count(gear) %>%  
  ggplot(aes(gear, n)) + geom_col()
```





```
count(iris, Species, sort=TRUE) %>%  
  mutate(Species=factor(Species, levels=Species)) %>%  
  ggplot(aes(Species, n)) + geom_col()
```



```
count(iris, Species, sort=TRUE) %>%  
| mutate(Species=factor(Species, levels=Species)) %>%  
| ggplot(aes(Species, n)) + geom_col()
```



```
filter(satellites, is_active) %>%  
  count(agency_full_name, sort=TRUE) %>%  
  mutate(ct=scales::comma(n),  
         pct=scales::percent(n/sum(n))) %>%  
  select(-n) %>%  
  print(n=20)
```



```
filter(satellites, is_active) %>%  
  count(agency_full_name, sort=TRUE) %>%  
  mutate(ct=scales::comma(n),  
         pct=scales::percent(n/sum(n))) %>%  
  select(-n) %>%  
  print(n=20)
```



```
get_flu_data("national", years=2010:2016) %>%  
  mutate(week=from_yr_wk(YEAR, WEEK)) %>%  
  gather(age_group, count, starts_with("AGE")) %>%  
  ggplot(aes(week, count, group = age_group)) +  
  geom_line(aes(color = age_group)) +  
  scale_y_continuous(label=scales::comma, limits=c(0,20000)) +  
  facet_wrap(~age_group, scales="free") +
```



```
get_flu_data("national", years=2010:2016) %>%  
  mutate(week=from_yr_wk(YEAR, WEEK)) %>%  
  gather(age_group, count, starts_with("AGE")) %>%  
  ggplot(aes(week, count, group = age_group)) +  
  geom_line(aes(color = age_group)) +  
  scale_y_continuous(label=scales::comma, limits=c(0,20000)) +  
  facet_wrap(~age_group, scales="free") +
```



```
GET("https://www.federalregister.gov/articles/search.csv",
  query=list(
    `conditions[agency_ids][]`=254,
    `conditions[publication_date][gte]`="01/01/2006",
    `conditions[publication_date][lte]`="1/13/2017",
    `conditions[term]`="60396",
    `conditions[type][]`="NOTICE")) %>%
  stop_for_status() %>%
  content(as="parsed") %>%
  filter(grepl("^Quarterly", title)) → reg_df
```



```
GET("https://www.federalregister.gov/articles/search.csv",
  query=list(
    `conditions[agency_ids][]`=254,
    `conditions[publication_date][gte]`="01/01/2006",
    `conditions[publication_date][lte]`="1/13/2017",
    `conditions[term]`="60396",
    `conditions[type][]`="NOTICE")) %>%
  stop_for_status() %>%
  content(as="parsed") %>%
  filter(grepl("^Quarterly", title)) → reg_df
```





```
read_fwf(satellites, na=c("N/A"), col_types=satcat_cols,  
         fwf_positions(satcat_cols_start,  
                        satcat_cols_end, satcat_col_names)) %>%  
  mutate(multiple=(multiple=="M"),  
         payload=(payload=="*")) %>%  
  left_join(satcat_launch_sources, by="source") %>%  
  left_join(satcat_launch_sites, by="launch_site") %>%  
  left_join(satcat_op_status, by="op_status_code") %>%  
  mutate(is_active=(op_status_code %in%  
                    c("+", "p", "B", "S", "X"))) → satcat
```

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be > 1
- ▶ A pipe group should be designed to accomplish a unified task
- ▶ It's OK to change object class/type/mode



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

we started off with a character vector and ended up with a data frame. this pipe chain only had two object type changes. I'll freely admit that this tripped me up when the pipe iterator first came out. a very common munging pipe chain is to have ugly character vectors being turned into lists which get turned into data frames. Thanks to the broom package (a pocket universe attached to the tidyverse), you can even take numeric vectors and turn them into lists of fitted models and turn them into data frames via piping.



```
🥇 list.files("data/honeypot", pattern = "cowrie.*json.gz",  
             full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

we started off with a character vector and ended up with a data frame. this pipe chain only had two object type changes. I'll freely admit that this tripped me up when the pipe iterator first came out. a very common munging pipe chain is to have ugly character vectors being turned into lists which get turned into data frames. Thanks to the broom package (a pocket universe attached to the tidyverse), you can even take numeric vectors and turn them into lists of fitted models and turn them into data frames via piping.



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

we started off with a character vector and ended up with a data frame. this pipe chain only had two object type changes. I'll freely admit that this tripped me up when the pipe iterator first came out. a very common munging pipe chain is to have ugly character vectors being turned into lists which get turned into data frames. Thanks to the broom package (a pocket universe attached to the tidyverse), you can even take numeric vectors and turn them into lists of fitted models and turn them into data frames via piping.

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be > 1
- ▶ A pipe group should be designed to accomplish a unified task
- ▶ It's OK to change object class/type/mode
- ▶ Be data-source aware
- ▶ Pipe operations should be "atomic"

But, before you go all crazy with piping there are two more rules to consider.



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

in reality, this is an expensive operation for me since I (well the awesome Rapid7 Labs engineering team does) generate over 20,000 of these JSON files a day.



```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```

in reality, this is an expensive operation for me since I (well the awesome Rapid7 Labs engineering team does) generate over 20,000 of these JSON files a day.





```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_ip, dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp))
```



in reality, this is an expensive operation for me since I (well the awesome Rapid7 Labs engineering team does) generate over 20,000 of these JSON files a day.

```
list.files("data/honeypot", pattern = "cowrie.*json.gz",
           full.names = TRUE) %>%
  map_df(stream_in) %>%
  select(timestamp, src_ip, src_port, sensor, session,
         dst_port, eventid, username, password) %>%
  mutate(timestamp = anytime(timestamp),
         date = as.Date(timestamp)) %>%
  perform_something_else() %>%
  count(date, username, password, sort = TRUE) %>%
  mutate(creds=sprintf("%s:%s", username, password)) %>%
  ggplot2(aes(creds, n) + geom_col())
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



that means if I have a pipe chain like this where I read them all in, so some more stuff to the resultant YUGE data frame and ultimately plot top credential use frequencies i've got two problem pitfalls I should have avoided.

```
list.files("data/honeypot", pattern = "cowrie.*json.gz",
           full.names = TRUE) %>%
  map_df(stream_in) %>%
  select(timestamp, src_ip, src_port, sensor, session,
         dst_port, eventid, username, password) %>%
  mutate(timestamp = anytime(timestamp),
         date = as.Date(timestamp)) %>%
  perform_something_else() %>%
  count(date, username, password, sort = TRUE) %>%
  mutate(creds=sprintf("%s:%s", username, password)) %>%
  ggplot2(aes(creds, n) + geom_col())
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



first is “what if something goes wrong in the post-data-frame creation function? this is often a general criticism of piping, especially within pkg source code. Here, though, it’s a real concern since if it breaks, I’ve lost quite a bit of time. It may seem like another “duh”, but I’m going to bet you’ll remember this slide when you do it yourself in the future. the other gotcha is getting a nice plot but losing the data behind it. pretty much the same problem, but it’s an easy trap to fall into.

```

list.files("data/honeypot", pattern = "cowrie.*json.gz",
           full.names = TRUE) %>%
  map_df(stream_in) %>%
  select(timestamp, src_ip, src_port, sensor, session,
         dst_port, eventid, username, password) %>%
  mutate(timestamp = anytime(timestamp),
         date = as.Date(timestamp)) %>%
  ✨ perform_something_else() %>%
  count(date, username, password, sort = TRUE) %>%
  mutate(creds=sprintf("%s:%s", username, password)) %>%
  ggplot2(aes(creds, n) + geom_col())

```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



first is “what if something goes wrong in the post-data-frame creation function? this is often a general criticism of piping, especially within pkg source code. Here, though, it’s a real concern since if it breaks, I’ve lost quite a bit of time. It may seem like another “duh”, but I’m going to bet you’ll remember this slide when you do it yourself in the future. the other gotcha is getting a nice plot but losing the data behind it. pretty much the same problem, but it’s an easy trap to fall into.

```

list.files("data/honeypot", pattern = "cowrie.*json.gz",
          full.names = TRUE) %>%
  map_df(stream_in) %>%
  select(timestamp, src_ip, src_port, sensor, session,
         dst_port, eventid, username, password) %>%
  mutate(timestamp = anytime(timestamp),
         date = as.Date(timestamp)) %>%
  ✨ perform_something_else() %>%
  count(date, username, password, sort = TRUE) %>%
  mutate(creds=sprintf("%s:%s", username, password)) %>%
  🇮🇹 ggplot2(aes(creds, n) + geom_col()

```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



first is “what if something goes wrong in the post-data-frame creation function? this is often a general criticism of piping, especially within pkg source code. Here, though, it’s a real concern since if it breaks, I’ve lost quite a bit of time. It may seem like another “duh”, but I’m going to bet you’ll remember this slide when you do it yourself in the future. the other gotcha is getting a nice plot but losing the data behind it. pretty much the same problem, but it’s an easy trap to fall into.

```
list.files("data/honeypot", pattern = "cowrie.*json.gz",
          full.names = TRUE) %>%
  map_df(stream_in) %>%
  select(timestamp, src_ip, src_port, sensor, session,
         dst_port, eventid, username, password) %>%
  mutate(timestamp = anytime(timestamp),
         date = as.Date(timestamp)) %>%
  perform_something_else() %>%
  count(date, username, password, sort = TRUE) %>%
  mutate(creds=sprintf("%s:%s", username, password)) %>%
  ggplot2(aes(creds, n) + geom_col())
```

this is what I'd consider the atomic group sequence.

```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp),  
         date = as.Date(timestamp))
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



Make a data frame. [CLICK]

do other stuff with the data frame after it's safely in the R environment. It's just being aware of your data source.

```
list.files("data/honeypot", pattern = "cowrie.*json.gz",  
          full.names = TRUE) %>%  
  map_df(stream_in) %>%  
  select(timestamp, src_ip, src_port, sensor, session,  
         dst_port, eventid, username, password) %>%  
  mutate(timestamp = anytime(timestamp),  
         date = as.Date(timestamp)) → df
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



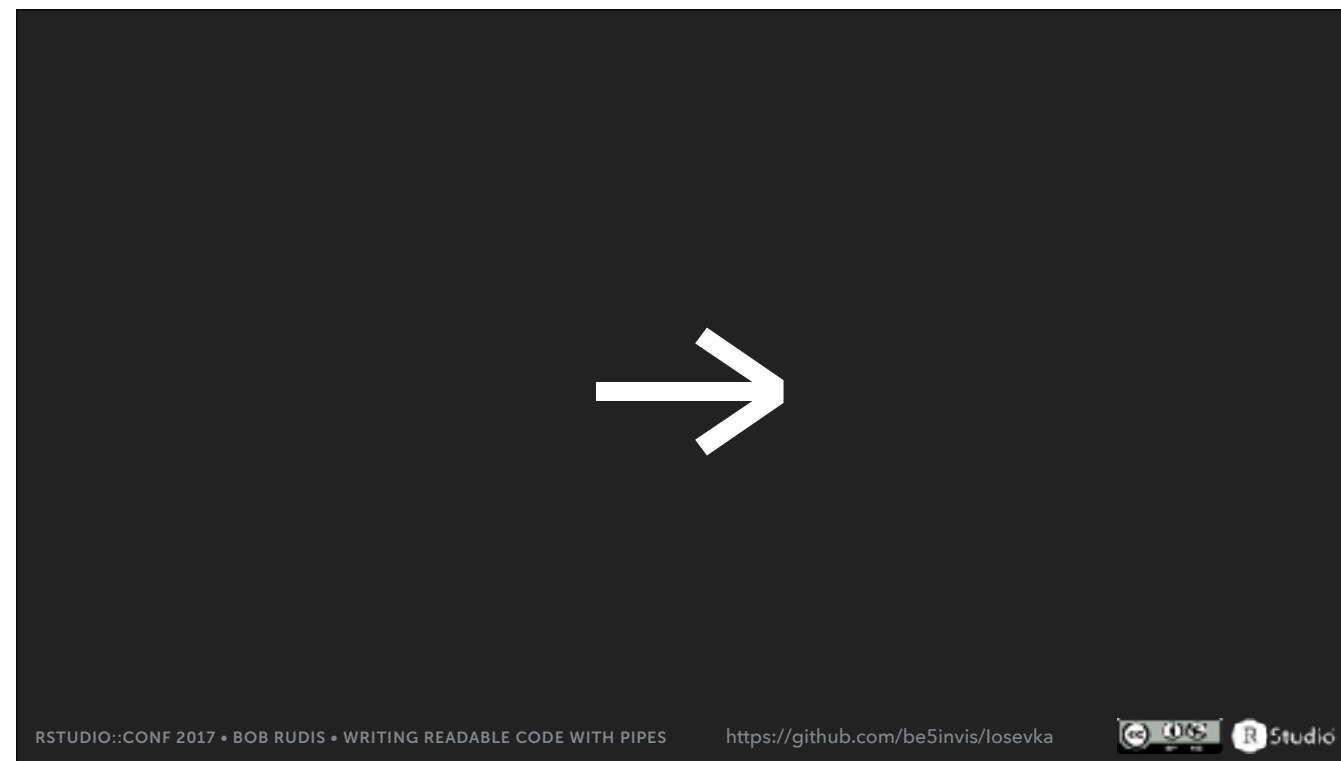
Make a data frame. [CLICK]

do other stuff with the data frame after it's safely in the R environment. It's just being aware of your data source.

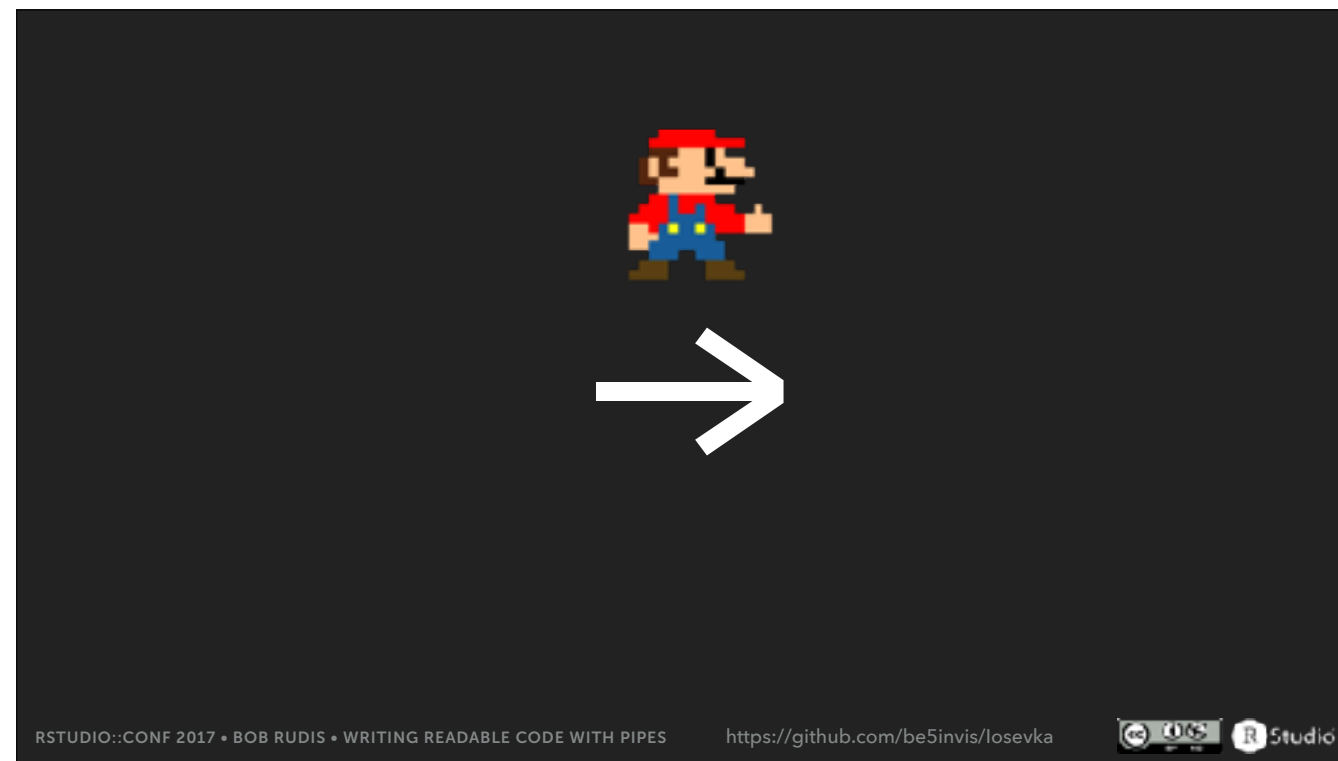




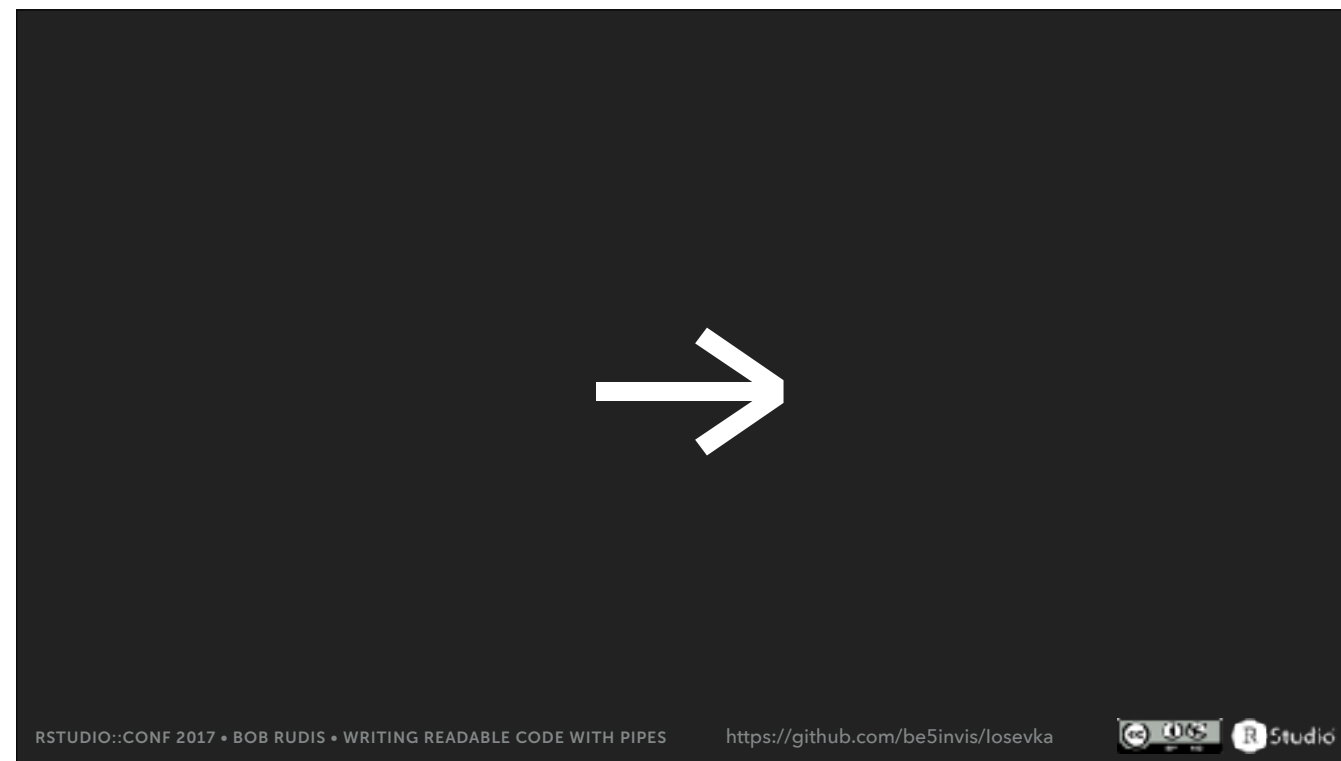
Oh, yeah. Hadley's super wrong about the forward assignment operator.



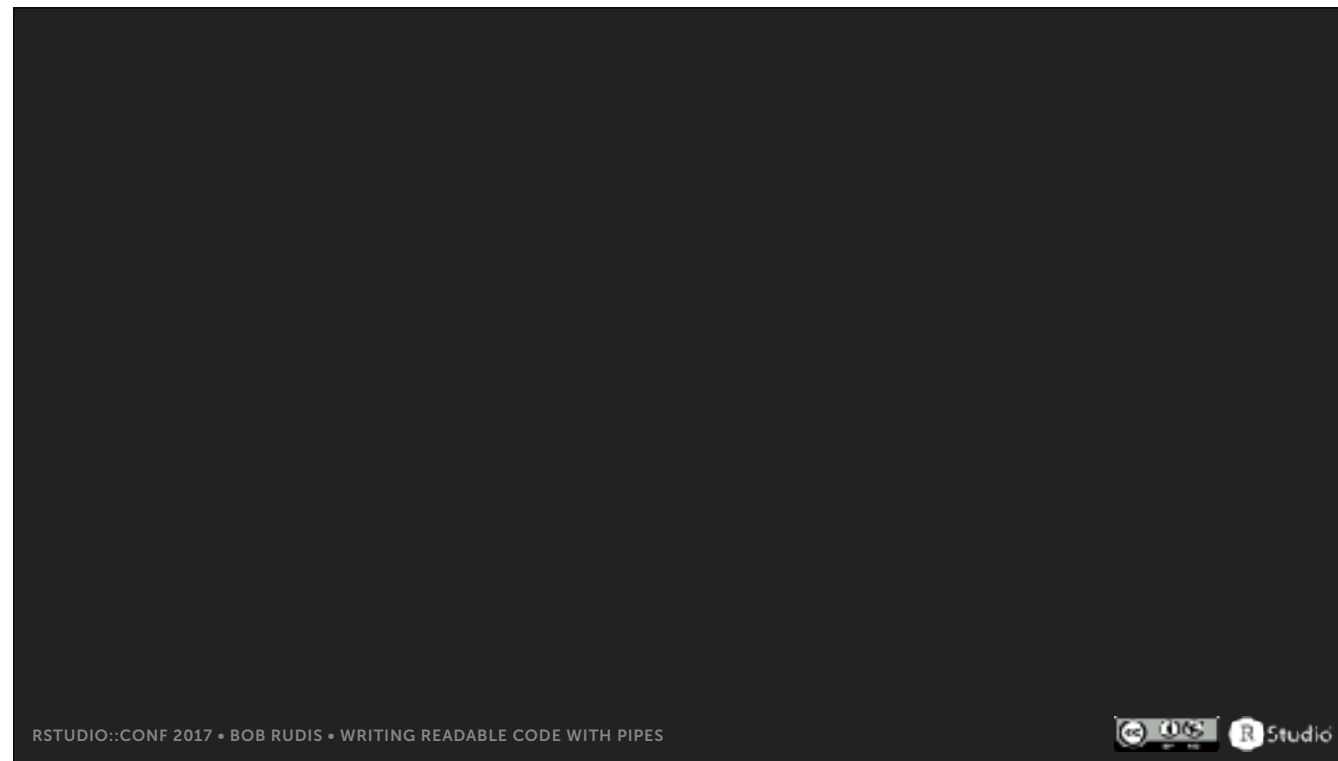
Oh, yeah. Hadley's super wrong about the forward assignment operator.



Oh, yeah. Hadley's super wrong about the forward assignment operator.



Oh, yeah. Hadley's super wrong about the forward assignment operator.



these are two other examples of not being data source aware. Bandwidth isn't free (Rapid7's S3 bill is living proof). [\[CLICK\]](#)

Write code around those readers to check to see if you really do need to load it anew or if you can use a local cache.

We fall into this trap since it's hard to make quick blog posts or book chapters with full production caching code, but only use "live" remote data in a pipe if you absolutely need to (there are use-cases but I'd wager most folks here prbly don't have one).

```
read_html("https://example.com/page.html") %>%  
  html_nodes("p") %>%  
  html_text() → some_text
```

```
read_csv("https://example.com/data.csv") %>%  
  mutate(...) %>%  
  count(...) → df
```

these are two other examples of not being data source aware. Bandwidth isn't free (Rapid7's S3 bill is living proof). [\[CLICK\]](#)

Write code around those readers to check to see if you really do need to load it anew or if you can use a local cache.

We fall into this trap since it's hard to make quick blog posts or book chapters with full production caching code, but only use "live" remote data in a pipe if you absolutely need to (there are use-cases but I'd wager most folks here prbly don't have one).

```
✦ read_html("https://example.com/page.html") %>%  
  html_nodes("p") %>%  
  html_text() → some_text
```

```
✦ read_csv("https://example.com/data.csv") %>%  
  mutate(...) %>%  
  count(...) → df
```

these are two other examples of not being data source aware. Bandwidth isn't free (Rapid7's S3 bill is living proof). [\[CLICK\]](#)

Write code around those readers to check to see if you really do need to load it anew or if you can use a local cache.

We fall into this trap since it's hard to make quick blog posts or book chapters with full production caching code, but only use "live" remote data in a pipe if you absolutely need to (there are use-cases but I'd wager most folks here prbly don't have one).

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be > 1
- ▶ A pipe group should be designed to accomplish a unified task
- ▶ It's OK to change object class/type/mode
- ▶ Be data-source aware
- ▶ Pipe operations should be "atomic"
- ▶ Pipe (briefly) in pipes



## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be > 1
- ▶ A pipe group should be designed to accomplish a unified task
- ▶ It's OK to change object class/type/mode
- ▶ Be data-source aware
- ▶ Pipe operations should be "atomic"
- ▶ Pipe (briefly) in pipes

this is really a correlation to the previous ones, but you should consider using pipes within pipes vs make extra long anonymous functions.

```

income <- c("$130,000 - $134,999", "$55,000 - $59,999",
            "$90,000 - $94,999", "$70,000 - 74,999",
            "$10,000 - $14,999", "$25,000 - $29,999",
            "$20,000 - $24,999", "$25,000 - $29,999",
            "$40,000 - $44,999")

sapply(strsplit(income, ' - '),
       function(x) mean(as.numeric(gsub('[,\\$]', '', x))))

## [1] 132499.5  57499.5  92499.5  72499.5  12499.5
## [6]  27499.5  22499.5  27499.5  42499.5

```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



this takes survey data results of answers to salary ranges and ultimately finds the mean for the range. there are similar string manipulation tasks like this in many workflows.



```
income %>%  
  strsplit(" - ") %>%  
  map(gsub, pattern="[\\$]", replacement="") %>%  
  map(as.numeric) %>%  
  map_dbl(mean)
```

that's much more readable here



```
income_tmp <- strsplit(" - ")
income_tmp <- map(income_tmp,
                  stri_replace_all_regex, "[,\\$]", "")
income_tmp <- map(income_tmp, as.numeric)
mean_incomes <- map_dbl(mean)
```

and much better than this mess in named base operations



```
strsplit(income, " - ") %>%  
  map_dbl(~gsub("[,\\$]", "", .) %>%  
    as.numeric() %>%  
    mean())
```

i think this representation is much more compact but still readable (provided you grok the tilde syntax of purrr).

the tilde creates an implicit anonymous function.

this can be more performant than looping for each operation. if you have more than a million rows i'd consider doing this for short operations

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be  $> 1$
- ▶ A pipe group should be designed to accomplish a unified task
- ▶ It's OK to change object class/type/mode
- ▶ Pipe operations should be "atomic"
- ▶ Be data-source aware
- ▶ Pipe (briefly) in pipes
- ▶ Don't be reticent to create new verbs

speaking of which



```
dollar_to_numeric <- function(x) {  
  map(x, ~as.numeric(gsub("[,\\$]", "", .)))  
}  
  
compute_means <- function(x) { map_dbl(x, mean) }  
  
strsplit(income, " - ") %>%  
  dollar_to_numeric() %>%  
  compute_means()
```

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



ultimately, you're communicating with at least your future self if not co-workers if not the internet . You absolutely know what that pipe chain does (i cld/shld have wrapped the string spitting, too). it's self-documenting and each component is also easy to unit test (you do test your code, right?). it also lends itself to making support pkgs for your analyses or developing your own personal r pkg.

there are times you may need to combine things into larger functions (i.e. if you have to iterate over a million+ rows) vs make each iterative transformation distinctly.

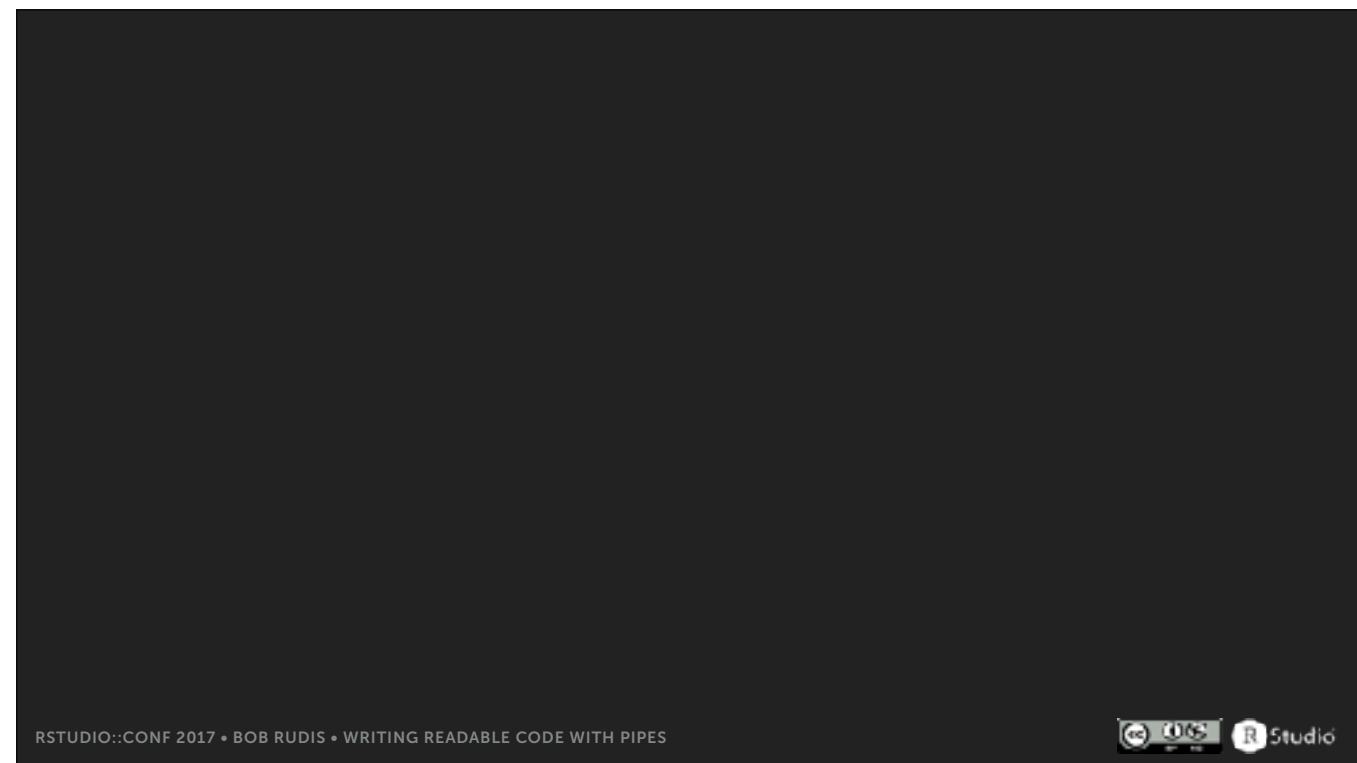
“

*Programs are meant to be read  
by humans and only incidentally  
for computers to execute.*

—Knuth

why go through all this? because Knuth said so (he's a personal hero of mine from when I got my first copy of the art of computer programming).





remember, you're coding for this happy guy, not some 8-bit gorilla.



remember, you're coding for this happy guy, not some 8-bit gorilla.

## LES RÈGLES DE LA TUYAUTERIE DE hrbrmstr

(hrbrmstr's Rules of Piping)

- ▶ The chain should be  $> 1$
- ▶ A pipe group should be designed to accomplish a unified task
- ▶ It's OK to change object class/type/mode
- ▶ Be data-source aware
- ▶ Pipe operations should be "atomic"
- ▶ Pipe (briefly) in pipes
- ▶ Don't be reticent to create new verbs
- ▶ Keep them **short**

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES

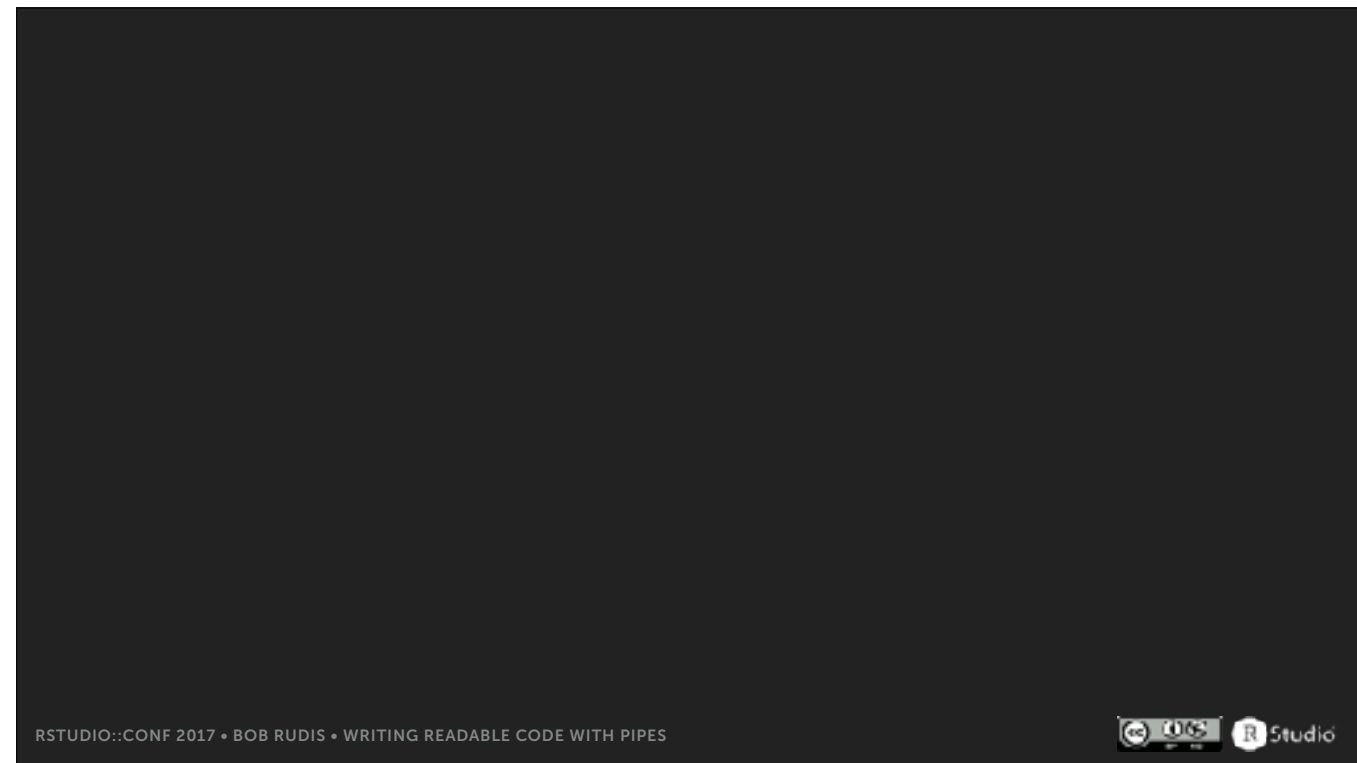


As Columbo would say, there's just one more thing.

This last one shld go without saying but I'll say it anyway without examples. It's in line with other bullets but long pipelines are a recipe for disaster. it prbly means you haven't bucketed your operations well.

Think of pipes in the same way as a data viz. The code bits around a given line provide context.

long pipes are hard to test and actually hard to read for the same reasons many visualizations are (needing to rely on memory or having to reference critical information far away from where your eyes are now).



why go through all this? because Knuth said so (he's a personal hero of mine from when I got my first copy of the art of computer programming).

# W W M D

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



why go through all this? because Knuth said so (he's a personal hero of mine from when I got my first copy of the art of computer programming).

# WHAT WOULD Mario DO?

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



why go through all this? because Knuth said so (he's a personal hero of mine from when I got my first copy of the art of computer programming).









*Ceci n'est pas une pipe.*

google the phrase to give this context ;)

HRBRMSTR	RSTUDIO::CONF	SLIDES	TIME
20170113	x 5	66	20M

TALK OVER

BOB RUDIS  
[MASTER] CHIEF DATA SCIENTIST @ RAPID7  
EMAIL: BOBRUD.IS  
BLOG: RUD.IS/B  
TWITTER: **CHRBMRSTR**  
GITHUB.COM/HRBRMSTR/RSTUDIOCONF2017

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



thank you!

HRBRMSTR	RSTUDIO::CONF	SLIDES	TIME
20170113	0 x 5	66	20M

TALK OVER

BOB RUDIS  
[MASTER] CHIEF DATA SCIENTIST @ RAPID7  
EMAIL: BOBRUD.IS  
BLOG: RUD.IS/B  
TWITTER: **CHRBMRSTR**  
GITHUB.COM/HRBRMSTR/RSTUDIOCONF2017

RSTUDIO::CONF 2017 • BOB RUDIS • WRITING READABLE CODE WITH PIPES



thank you!