We caught this post on the [The Surprising Number Of Programmers Who Can’t Program](https://letterstoanewdeveloper.com/2019/08/23/the-surprising-number-of-programmers-who-cant-program/) from the Hacker News RSS feed. Said post links to another, classic post on the same subject and you should read both before continuing.

Back? Great! Let’s dig in.

**Why does hrbrmstr care about this?**

Offspring #3 completed his Freshman year at UMaine Orono last year but wanted to stay academically active over the summer (he’s majoring in astrophysics and knows he’ll need some programming skills to excel in his field) and took an introductory C++ course from UMaine that was held virtually, with 1 lecture per week (14 weeks IIRC) and 1 assignment due per week with no other grading.

After seeing what passes for a standard (UMaine is not exactly on the top list of institutions to attend if one wants to be a computer scientist) intro C++ course, I’m not really surprised “Johnny can’t code”. Thirteen weeks in the the class finally started covering OO concepts, and the course is ending with a scant intro to polymorphism. Prior to this, most of the assignments were just variations on each other (read from stdin, loop with conditionals, print output) with no program going over 100 LoC (that includes comments and spacing). This wasn’t a “compsci for non-compsci majors” course, either. Anyone majoring in an area of study that requires programming could have taken this course to fulfill one of the requirements, and they’d be set on a path of forever using StackOverflow copypasta to try to get their future work done.

I’m fairly certain most of #3’s classmates could not program fizzbuzz without googling and even more certain most have no idea they weren’t really “coding in C++” most of the course.

If this is how most other middling colleges are teaching the basics of computer programming, it’s no wonder employers are having a difficult time finding qualified talent.

**You have an “R” tag — actually, a few language tags — on this post, so where’s the code?**

After the article triggered the lament in the previous section, a crazy, [@coolbutuseless](http://twitter.com/coolbutuseless)-esque thought came into my head: “I wonder how many different language FizzBuz solutions can be created from within R?”.

The criteria for that notion is/was that there needed to be some Rcpp::cppFunction(), reticulate::py\_run\_string(), V8 context eval()-type way to have the code in-R but then run through those far-super-to-any-other-language’s polyglot extensibility constructs.

Before getting lost in the weeds, there were some other thoughts on language inclusion:

* Should Java be included? I :heart: {rJava}, but cat()-ing Java code out and running system() to compile it first seemed like cheating (even though that’s kinda just what cppFunction() does). Toss a note into a comment if you think a Java example should be added (or add said Java example in a comment or link to it in one!).
* I think Julia should be in this example list but do not care enough about it to load [{JuliaCall}](https://github.com/Non-Contradiction/JuliaCall) and craft an example (again, link or post one if you can crank it out quickly).
* I think Lua could be in this example given the existence of [{luar}](https://github.com/noamross/luar). If you agree, give it a go!
* Go & Rust compiled code can also be called in R (thanks to Romain & Jeroen) once they’re turned into C-compatible libraries. Should this polyglot example show this as well?
* What other languages am I missing?

**The aforementioned “weeds”**

One criteria for each language fizzbuzz example is that they need to be readable, not hacky-cool. That doesn’t mean the solutions still can’t be a bit creative. We’ll lightly go through each one I managed to code up. First we’ll need some helpers:

suppressPackageStartupMessages({

library(purrr)

library(dplyr)

library(reticulate)

library(V8)

library(Rcpp)

})

The R, JavaScript, and Python implementations are all in the microbenchmark() call way down below. Up here are C and C++ versions. The C implementation is boring and straightforward, but we’re using Rprintf() so we can capture the output vs have any output buffering woes impact the timings.

cppFunction('

void cbuzz() {

// super fast plain C

for (unsigned int i=1; i<=100; i++) {

if (i % 15 == 0) Rprintf("FizzBuzz\\n");

else if (i % 3 == 0) Rprintf("Fizz\\n");

else if (i % 5 == 0) Rprintf("Buzz\\n");

else Rprintf("%d\\n", i);

}

}

')

The cbuzz() example is just fine even in C++ land, but we can take advantage of some C++11 vectorization features to stay formally in C++-land and play with some fun features like lambdas. This will be a bit slower than the C version plus consume more memory, but shows off some features some folks might not be familiar with:

cppFunction('

void cppbuzz() {

std::vector numbers(100); // will eventually be 1:100

std::iota(numbers.begin(), numbers.end(), 1); // kinda sorta equiva of our R 1:100 but not exactly true

std::vector fb(100); // fizzbuzz strings holder

// transform said 1..100 into fizbuzz strings

std::transform(

numbers.begin(), numbers.end(),

fb.begin(),

[](int i) -> std::string { // lambda expression are cool like a fez

if (i % 15 == 0) return("FizzBuzz");

else if (i % 3 == 0) return("Fizz");

else if (i % 5 == 0) return("Buzz");

else return(std::to\_string(i));

}

);

// round it out with use of for\_each and another lambda

// this turns out to be slightly faster than range-based for-loop

// collection iteration syntax.

std::for\_each(

fb.begin(), fb.end(),

[](std::string s) { Rcout << s << std::endl; }

);

}

',

plugins = c('cpp11'))

Both of those functions are now available to R.

Next, we need to prepare to run JavaScript and Python code, so we’ll initialize both of those environments:

ctx <- v8()

py\_config() # not 100% necessary but I keep my needed {reticulate} options in env vars for reproducibility

Then, we tell R to capture all the output. Using sink() is a bit better than capture.output() in this use-case since to avoid nesting calls, and we need to handle Python stdout the same way py\_capture\_output() does to be fair in our measurements:

output\_tools <- import("rpytools.output")

restore\_stdout <- output\_tools$start\_stdout\_capture()

cap <- rawConnection(raw(0), "r+")

sink(cap)

There are a few implementations below across the tidy and base R multiverse. Some use vectorization; some do not. This will let us compare overall “speed” of solution. If you have another suggestion for a *readable* solution in R, drop a note in the comments:

microbenchmark::microbenchmark(

# tidy\_vectors\_case() is slowest but you get all sorts of type safety

# for free along with very readable idioms.

tidy\_vectors\_case = map\_chr(1:100, ~{

case\_when(

(.x %% 15 == 0) ~ "FizzBuzz",

(.x %% 3 == 0) ~ "Fizz",

(.x %% 5 == 0) ~ "Buzz",

TRUE ~ as.character(.x)

)

}) %>%

cat(sep="\n"),

# tidy\_vectors\_if() has old-school if/else syntax but still

# forces us to ensure type safety which is cool.

tidy\_vectors\_if = map\_chr(1:100, ~{

if (.x %% 15 == 0) return("FizzBuzz")

if (.x %% 3 == 0) return("Fizz")

if (.x %% 5 == 0) return("Buzz")

return(as.character(.x))

}) %>%

cat(sep="\n"),

# walk() just replaces `for` but stays in vector-land which is cool

tidy\_walk = walk(1:100, ~{

if (.x %% 15 == 0) cat("FizzBuzz\n")

if (.x %% 3 == 0) cat("Fizz\n")

if (.x %% 5 == 0) cat("Buzz\n")

cat(.x, "\n", sep="")

}),

# vapply() gets us some similiar type assurance, albeit with arcane syntax

base\_proper = vapply(1:100, function(.x) {

if (.x %% 15 == 0) return("FizzBuzz")

if (.x %% 3 == 0) return("Fizz")

if (.x %% 5 == 0) return("Buzz")

return(as.character(.x))

}, character(1), USE.NAMES = FALSE) %>%

cat(sep="\n"),

# sapply() is def lazy but this can outperform vapply() in some

# circumstances (like this one) and is a bit less arcane.

base\_lazy = sapply(1:100, function(.x) {

if (.x %% 15 == 0) return("FizzBuzz")

if (.x %% 3 == 0) return("Fizz")

if (.x %% 5 == 0) return("Buzz")

return(.x)

}, USE.NAMES = FALSE) %>%

cat(sep="\n"),

# for loops...ugh. might as well just use C

base\_for = for(.x in 1:100) {

if (.x %% 15 == 0) cat("FizzBuzz\n")

else if (.x %% 3 == 0) cat("Fizz\n")

else if (.x %% 5 == 0) cat("Buzz\n")

else cat(.x, "\n", sep="")

},

# ok, we'll just use C!

c\_buzz = cbuzz(),

# we can go back to vector-land in C++

cpp\_buzz = cppbuzz(),

# some <3 for javascript

js\_readable = ctx$eval('

for (var i=1; i <101; i++){

if (i % 15 == 0) console.log("FizzBuzz")

else if (i % 3 == 0) console.log("Fizz")

else if (i % 5 == 0) console.log("Buzz")

else console.log(i)

}

'),

# icky readable, non-vectorized python

python = reticulate::py\_run\_string('

for x in range([1, 101](https://www.google.com/maps/search/1,+101?entry=gmail&source=g)):

if (x % 15 == 0):

print("Fizz Buzz")

elif (x % 5 == 0):

print("Buzz")

elif (x % 3 == 0):

print("Fizz")

else:

print(x)

')

) -> res

Turn off output capturing:

sink()

if (!is.null(restore\_stdout)) invisible(output\_tools$end\_stdout\_capture(restore\_stdout))

We used microbenchmark(), so here are the results:

res

## Unit: microseconds

## expr min lq mean median uq max neval cld

## tidy\_vectors\_case 20290.749 21266.3680 22717.80292 22231.5960 23044.5690 33005.960 100 e

## tidy\_vectors\_if 457.426 493.6270 540.68182 518.8785 577.1195 797.869 100 b

## tidy\_walk 970.455 1026.2725 1150.77797 1065.4805 1109.9705 8392.916 100 c

## base\_proper 357.385 375.3910 554.13973 406.8050 450.7490 13907.581 100 b

## base\_lazy 365.553 395.5790 422.93719 418.1790 444.8225 587.718 100 ab

## base\_for 521.674 545.9155 576.79214 559.0185 584.5250 968.814 100 b

## c\_buzz 13.538 16.3335 18.18795 17.6010 19.4340 33.134 100 a

## cpp\_buzz 39.405 45.1505 63.29352 49.1280 52.9605 1265.359 100 a

## js\_readable 107.015 123.7015 162.32442 174.7860 187.1215 270.012 100 ab

## python 1581.661 1743.4490 2072.04777 1884.1585 1985.8100 12092.325 100 d

Said results are since this is a toy example, but We wanted to show that Jeroen’s {V8} can be super fast, especially when there’s no value marshaling to be done and that some things you may have thought should be faster, aren’t.

**FIN**

Definitely add links or code for changes or additions (especially the aforementioned other languages). Hopefully my lament about the computer science program at UMaine is not universally true for all the programming courses there.