Introduction to coding

HTML Version: https://stirlingcodingclub.github.io/coding_types/

Brad Duthie

22 NOV 2023

Contents

Introduction: Objectives of these notes

The focus of the synchronous coding club meeting this week is on general computing concepts. These notes will stray a bit from that focus because I want to introduce some R code that I did not last week. Hence, these notes will include two distinct topics. The first topic will be contrasting coding and code performance in an interpreted language (R) versus a compiled language (C). The second topic will be picking up where we left off last week with the introduction to R programming. My hope is that there will be a bit of something for everyone in these notes, including novices to coding and more advanced R users. If you are just getting started, then it might make sense to skip the section contrasting interpreted versus compiled language and move right to where we left off last week with more R code to help get started.

Contrasting interpreted versus compiled language

Almost all coding is done using source code; that is, code that can be read and understood by a human. To actually run the code, we need to convert the source code into a binary format (ones and zeroes) that can be read by the computer. To do this conversion, we can either *compile* the code or *interpret* it. Technically speaking, any code *could* be compiled or interpreted, but most programming languages are associated with one or the other method.

When compiling code, the source code is translated beforehand into a form that the computer can read more easily. Only after this translation occurs is the code actually run, so the process of running code occurs in two steps (compile, then run). The benefit of compiled code is that it can generally run much faster (orders of magnitude faster); the cost is that writing compiled code is slower, more laborious, and often more frustrating. Code that takes me 2-5 minutes in an interpreted language such as R could easily take 30-60 minutes in a compiled language such as C. But if the compiled code can finish running in minutes or hours rather than days to weeks, then it might be worth the hassle.

When running interpreted code, individual chunks of code are run bit by bit through an interpreter. This interpreter breaks down the code and executes it on the fly, so everything is done in one step (e.g., in R, there is no compile then run – you just run the code in the console after you have written it). The cost of this method is that the interpreted code can be much slower. The benefit is that the actual process of writing code is generally faster and more intuitive. For many tasks, speed is also not worry, so there is little if any downside to avoiding the compiler.

In all types of code, binary instructions (compiled or interpreted) are sent to the computer's Central Processing Unit (CPU). What the CPU does with these instructions is actually quite limited; it can read and write to memory, and do some basic arithmetic. All of the instructions that you type in your source code essentially boil down to these tasks. The memory (specifically, 'random-access memory', or RAM) is separate from the CPU; it holds data that can be read and changed. The data exist as binary units (ones and zeroes), which are grouped in chunks of eight to make one 'byte'. In relatively 'high level' programming languages (e.g., R, MATLAB, python), you can more or less avoid thinking about all of this because the code is abstracted away from the nuts and bolts of the computer hardware and the management of memory is done behind the scenes. In more 'low level' programming languages (e.g., C, FORTRAN, COBOL), you will need to be explicit about how your code uses the computer's memory.

Let's start by running a very simple script of code, first in an interpreted language (R), and then in a compiled language (C). The code we write will count from one to one billion, printing every 100 millionth number. Here is what the code looks like in R.

```
count_to_1_billion <- function(){
    for(i in 1:1000000000){
        if(i %% 100000000 == 0){
          print(i);
        }
    }
    return("Done!");
}
```

You can also find the Rscript with the code above on GitHub. Note that the above code defines a function and includes a `for` loop. We will get to what these are doing in a later workshop, but for now, all that you need to do is highlight the code above and run it in the console. This will define the function. To run the function, you can then type the following line of code in the console.

```
count_to_1_billion();
```

Note, this might take a while! While you are waiting, you can create a new script for the compiled version written in C. To do this, you can either download this file from GitHub or create a new script in Rstudio and paste the following code.

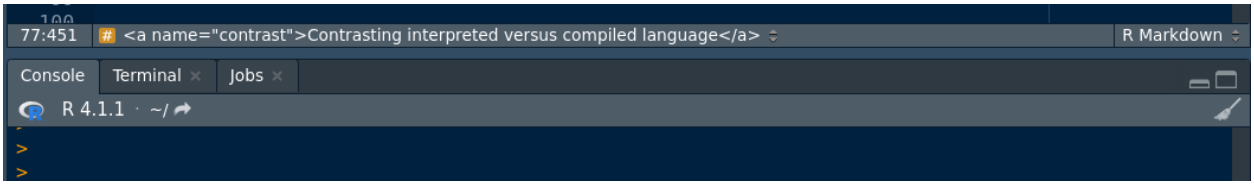
```
# include<stdio.h>

int main(void){

    long i;

    for(i = 1; i < 1000000000; i++){
       if(i % 100000000 == 0){
           printf("%lu\n", i);
       }
    }
    return 0;
}
```

Once pasted, save the file as `count_to_1_billion.c` (remember where you save it). If you get a box that pops up asking "Are you sure you want to change the type of the file so that it is no longer an R script?", then click "Yes". Note that you could have also pasted the code into a text editor such as notepad or gedit instead of Rstudio (but *not* in a word processor such as MS Word).

Now we need to compile the code. How you do this depends on the operating system that you use (Mac, Linux, or Windows). I will first show how to compile and run for Mac and Linux, then how to compile and run for Windows.

Mac and Linux Users

On Mac or Linux, you need to first open a terminal. You can do this by finding an external one on your computer (e.g., do a search for 'terminal', and one should be available), or by using the 'Terminal' tab right within Rstudio (see the middle tab below). Note that this Rstudio terminal is also available on a browser through Rstudio cloud, so you can use the Rstudio cloud to do the whole exercise (just make sure you upload the C file).



Once you are in the terminal, you need to make sure to get to the correct directory. The directory will be in the same location as where you stored your `count_to_1_billion.c` file. To navigate to the correct directory, we need to use the change directory `cd` command in the terminal. In my case, I have saved `count_to_1_billion.c` to a folder called 'loc_sim' on my computer, so I will navigate to that folder with the command below, typing `cd loc_sim` (hint `cd ..` moves you up a directory, if you ever need it) followed by the 'Enter' key.

```
brad@brad-HP:~$ cd loc_sim
```

This puts me in the loc_sim directory. If I want to view the contents of this directory, I can type `ls`.

```
brad@brad-HP:~/loc_sim$ ls
count_to_1_billion.c
```

Now I need to compile code. To do this, I will run the gcc compiler and the following command.

```
brad@brad-HP:~/loc_sim$ gcc -Wall count_to_1_billion.c -o count_to_1_billion
```

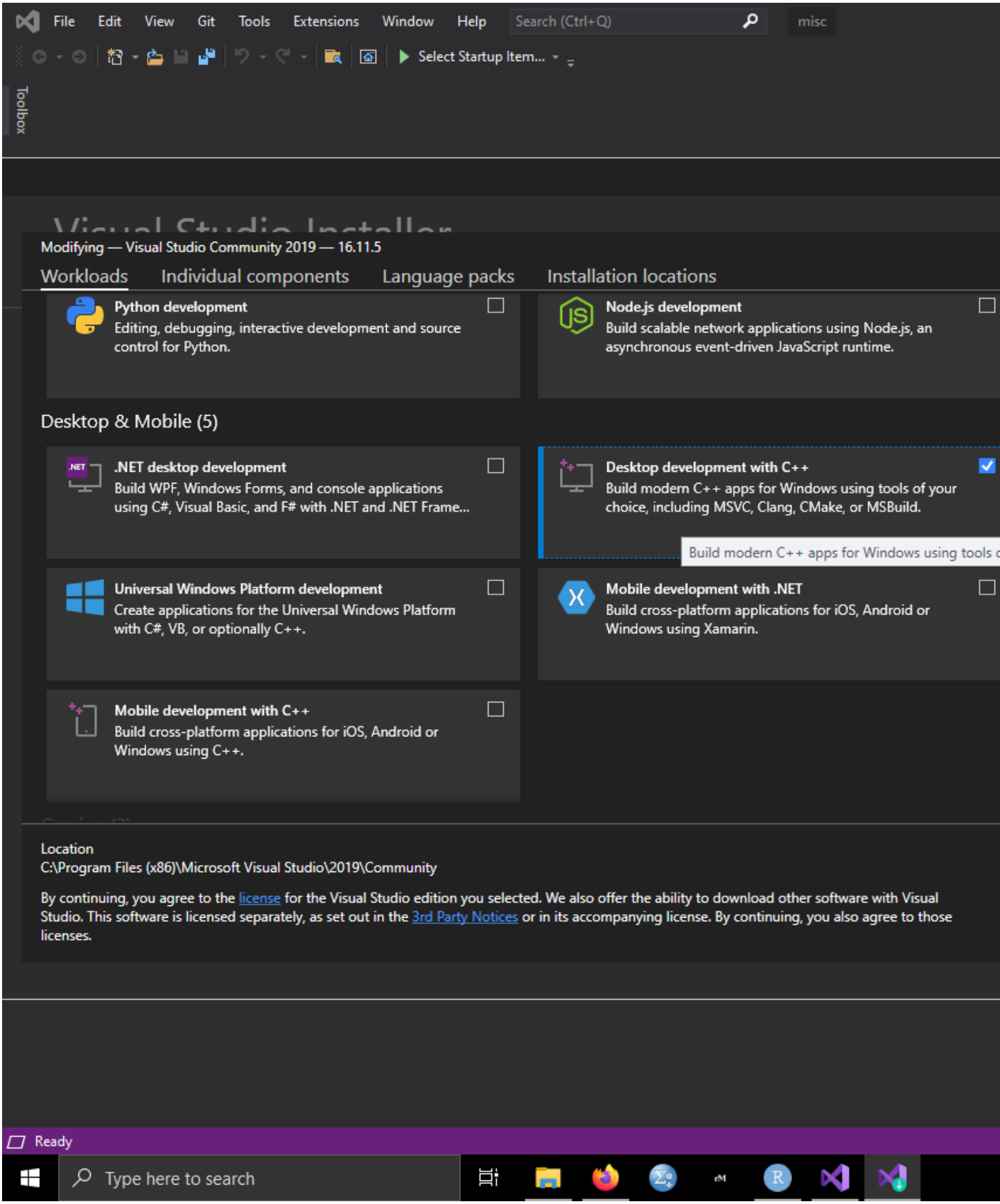
The gcc compiler should be on most Linux and Mac computers. The `-Wall` above tells the compiler to give us warnings if necessary, and the rest of the code tells us to use the `count_to_1_billion.c` file to build the program `count_to_1_billion`. If there are no warnings, then the compiler should execute, and the new program should show up in my 'loc_sim' folder. To run it, I can use the following code.

```
./count_to_1_billion
```

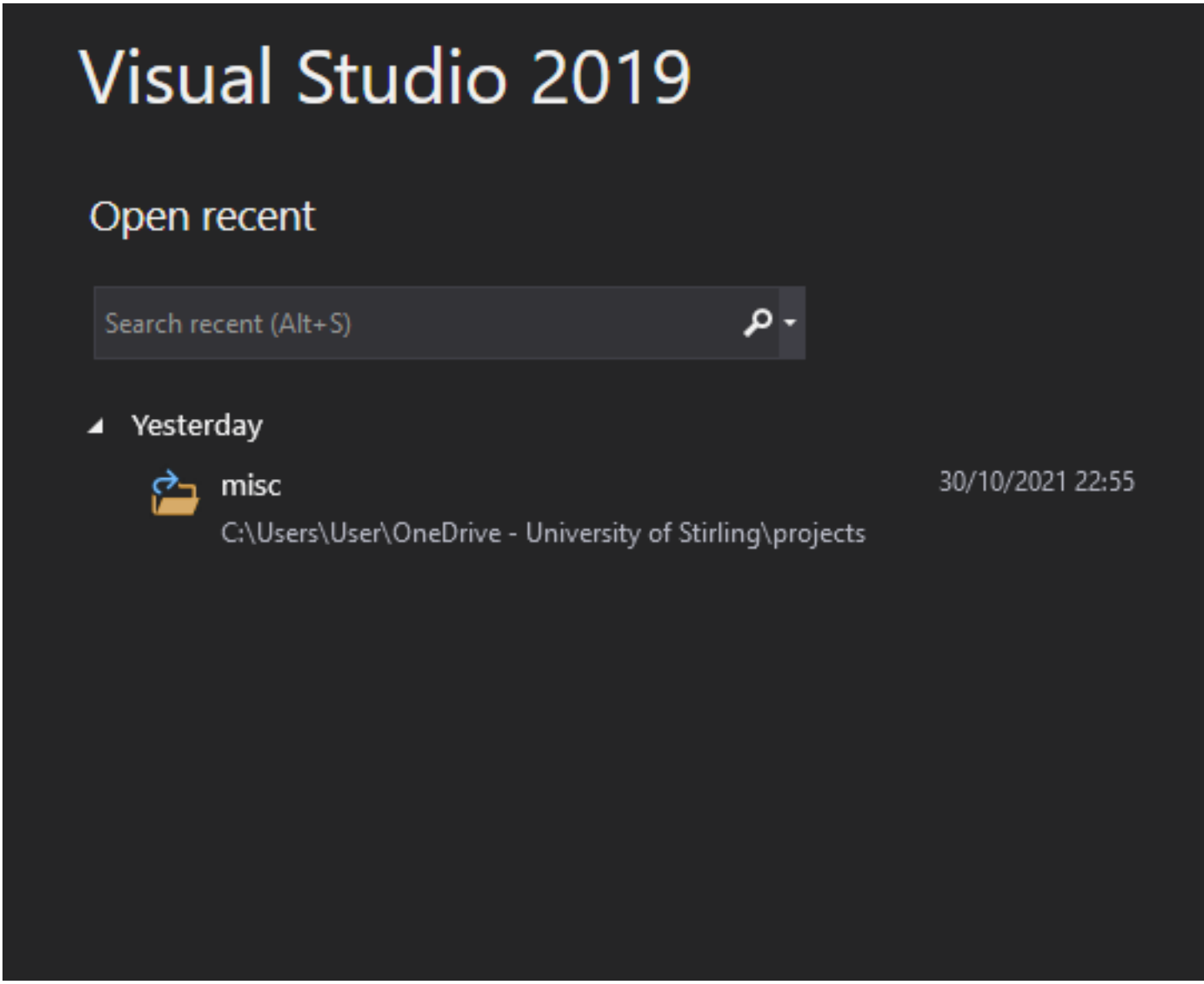
Notice the amount of time that took when compared to the equivalent R code.

Windows users

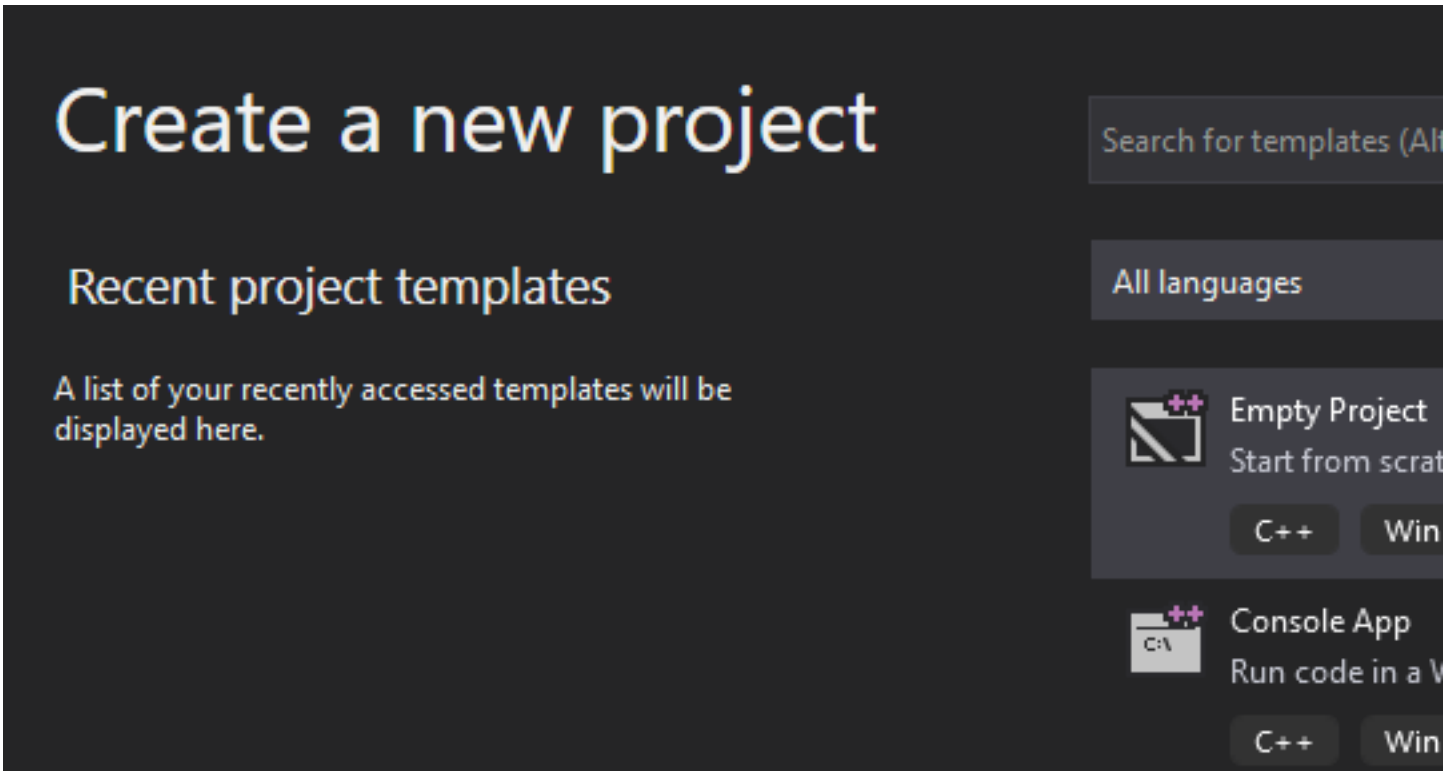
As with Mac and Linux users, you first need to open a terminal. Unfortunately, to access a compiler, you will need to download some new tools. Microsoft windows does not actually come with a pre-installed compiler for its command prompt, so we first need to download one form Visual Studio. Use the link to go to their page and download the 'Visual Studio' Community version (free for students, open-source contributors, and individuals). The most recent one at the time of writing is 'Visual Studio 2019'. You will need to download and install this onto your machine. After Visual Studio is successfully installed, then you will also need to install a Workload called 'Desktop development with C++' (see below).


File Edit View Git Tools Extensions Window Help
Search (Ctrl+Q)
misc
Select Startup Item...
Toolbox
Modifying — Visual Studio Community 2019 — 16.11.5
Workloads
Individual components
Language packs
Installation locations
Python development
Editing, debugging, interactive development and source control for Python.
Node.js development
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.
Desktop & Mobile (5)
.NET desktop development
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...
Desktop development with C++
Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
Build modern C++ apps for Windows using tools
Universal Windows Platform development
Create applications for the Universal Windows Platform with C#, VB, or optionally C++.
Mobile development with .NET
Build cross-platform applications for iOS, Android or Windows using Xamarin.
Mobile development with C++
Build cross-platform applications for iOS, Android or Windows using C++.
Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community
By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.
Ready
Type here to search

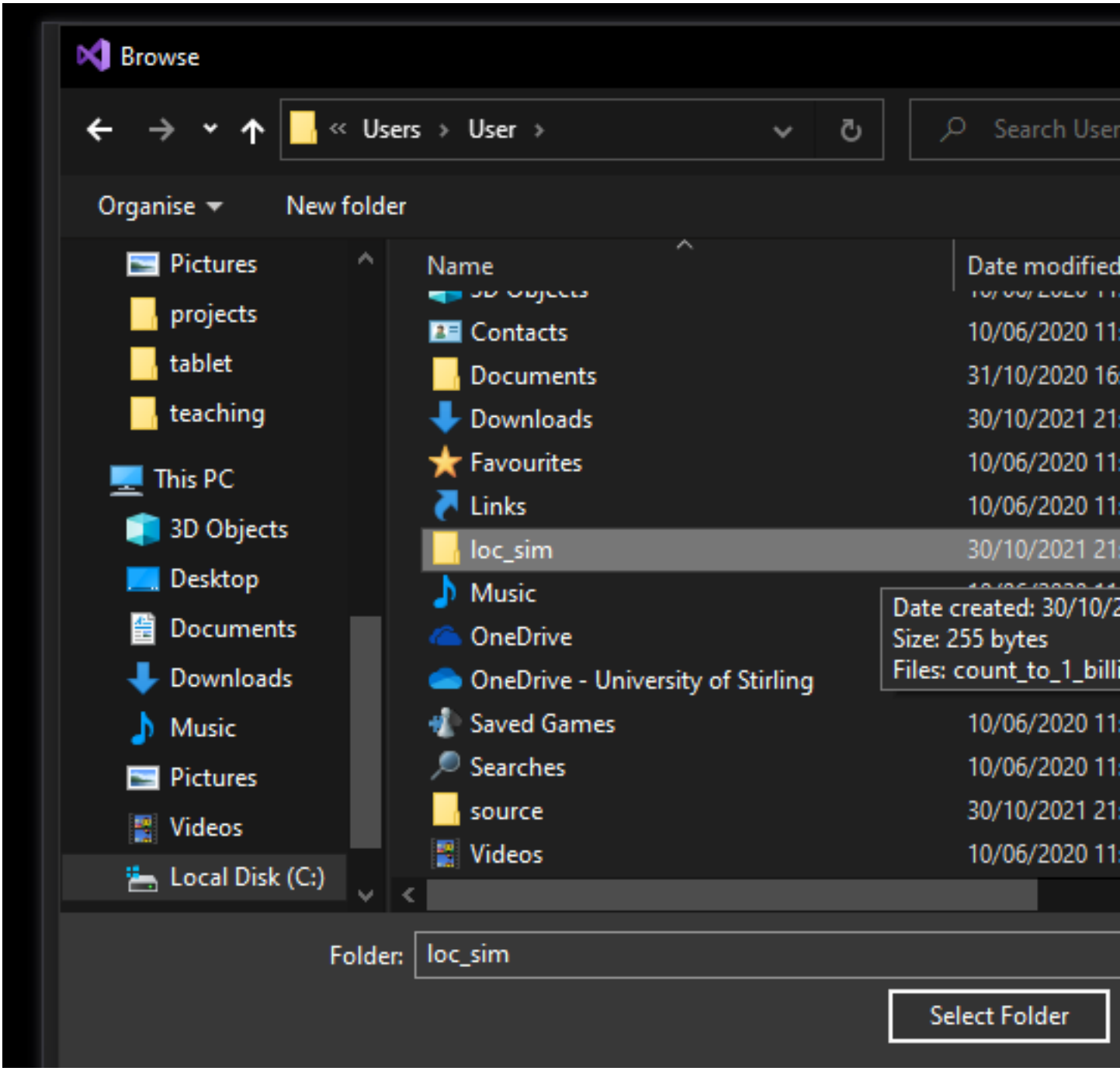
I might also recommend installing the Linux tools as well (scroll down to find these). Once these are downloaded, then you can go into Visual Studio and create a new project.



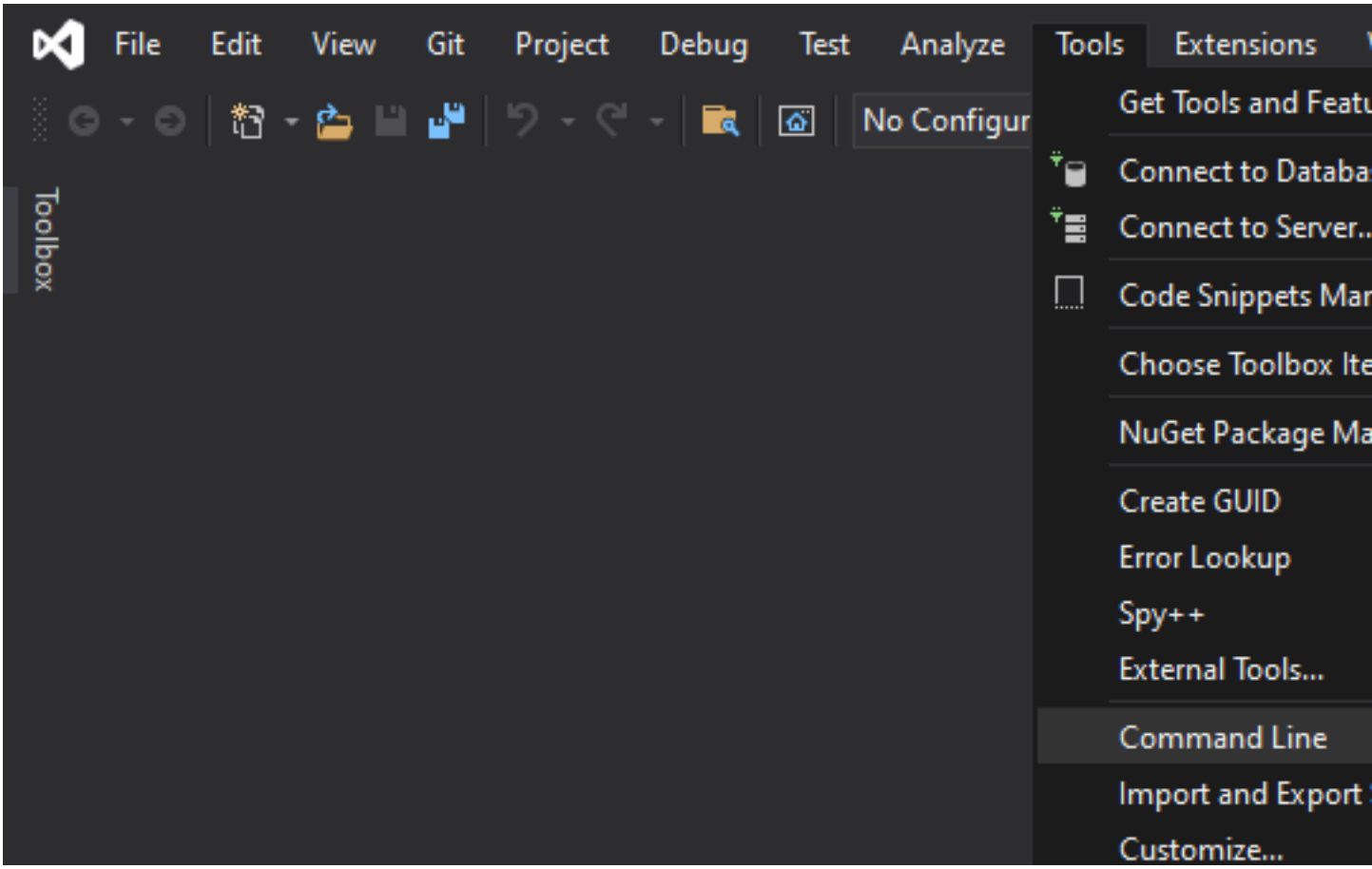
From that new project, you can choose the 'Empty Project' option to start from scratch.



Browse through your files and find the repository where the `count_to_1_billion.c` file was saved (mine is in 'loc_sim', as shown below). Select the folder.



After this folder is selected, we can open a command line within Visual Studio by going to 'Tools > Command Line > Developer Command Prompt'.



Once in the command prompt, we can use the `DIR` command to confirm that we are in the correct directory with the `count_to_1_billion.c` file

```
C:\Users\User\loc_sim>DIR
```

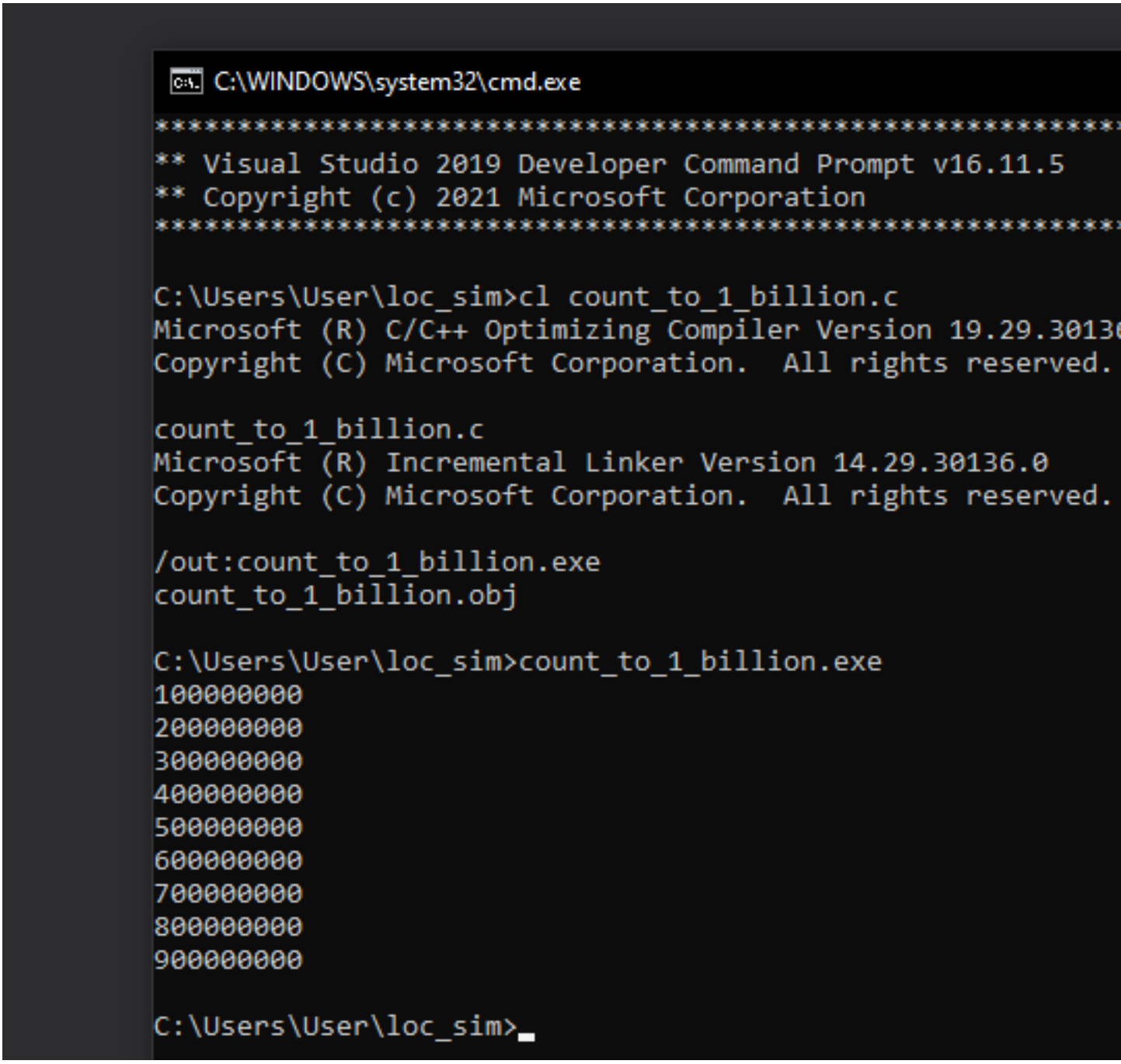
We can now finally type the command to compile the code.

```
C:\Users\User\loc_sim>cl -Wall count_to_1_billion.c
```

Note that the command here is different from the Mac and Linux command. Once compiled (ignore any notes), we can run the program 'count_to_1_billion.exe' with the command below.

```
C:\Users\User\loc_sim>count_to_1_billion.exe
```

This should produce an output like the one shown below.

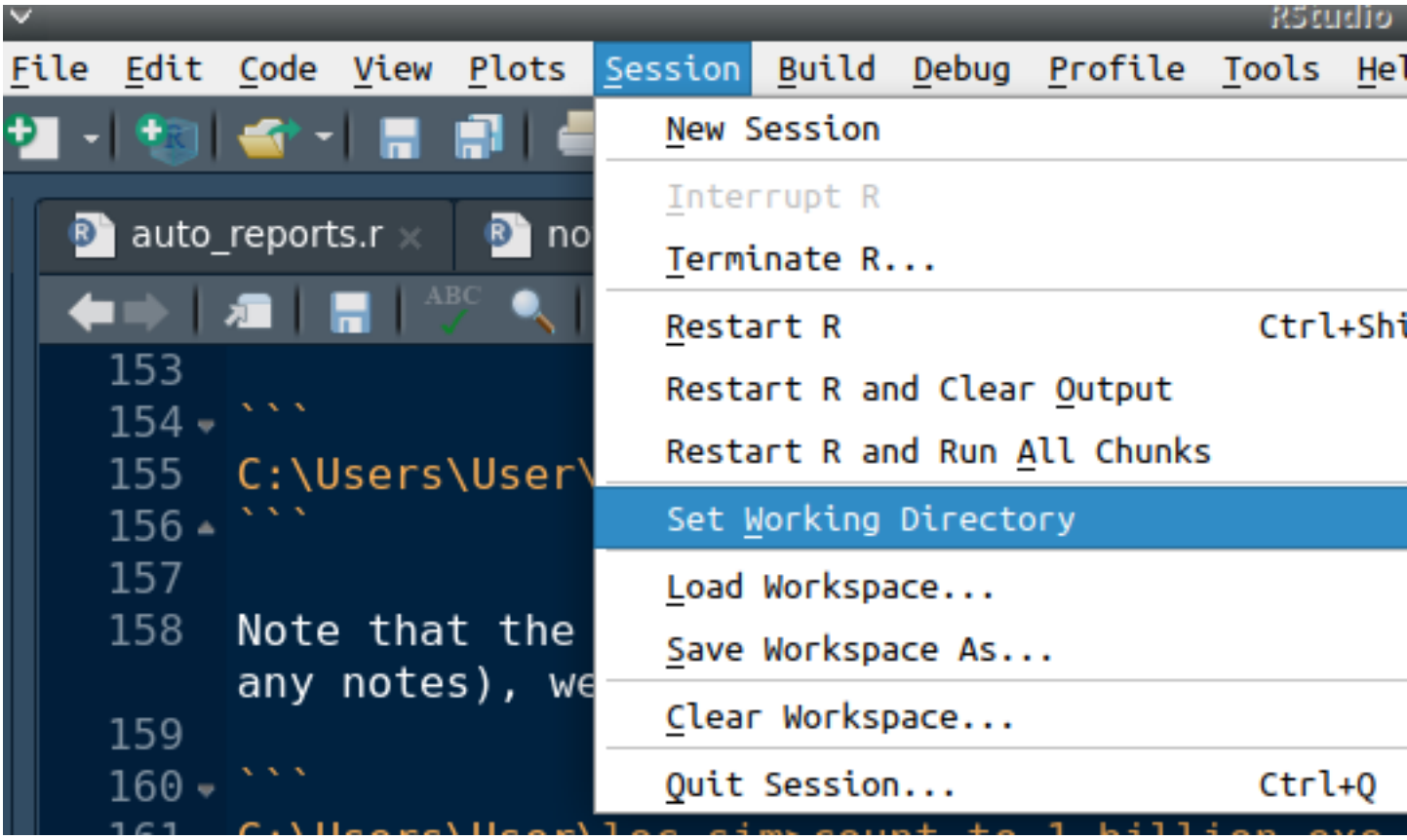


There are other ways to compile and run C code in Windows, but as far as I am aware, this is the most straightforward. Note the contrast in the process of running the interpreted (R) versus compiled (C) code, and the time that it took to run each. If you have made it this far, then you should have the basic tools that you need to run compiled code in C. In the following section, we will look at some more R code.

More R code to help get started

As mentioned earlier, I want to pick up where the last session left off with some useful things to know about coding in R. This is not in any way a comprehensive list of things to know before you start to program in R. What I want to do is provide some additional tips for getting started, mainly things that I learned (or

wish I had learned) when first learning to code in R. In the last session, I introduced some coding ideas and functions for getting started with basic statistical tools (functions introduced last time were `read.csv`, `head`, `dim`, `hist`, `summary`, `plot`, `cor.test`, `lm`, `t.test`, and `aov`). Here I will focus more on data wrangling (i.e., cleaning and transforming data). We can start by loading the same Bumpus sparrow data set from last time, which you can download here (right click 'Bumpus_data.csv' and save) or here. Make sure that you are in the correct working directory before getting started (i.e., the directory where the Bumpus data file is located). If you need to change directories, you can go to the Rstudio toolbar and select 'Session > Set Working Directory > Choose Directory' and select the directory.



We can also select the working directory directly from the command line.

```
setwd("~/Dropbox/projects/StirlingCodingClub/coding_types");
```

Note that your own path will look different than mine, especially if you are running Windows. From here, we can read in the Bumpus CSV file and assign it to the variable named `dat`.

```
dat <- read.csv(file = "Bumpus_data.csv", header = TRUE);
```

We now have our data set read into R. We can confirm that everything looks good by inspecting the first six rows of the dataset using the `head` function (note, the equivalent function `tail` gives us the last six rows).

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 1 male alive    154     241 24.5 31.2 0.687 0.668 1.022 0.587 0.830
## 2 male alive    160     252 26.9 30.8 0.736 0.709 1.180 0.602 0.841
## 3 male alive    155     243 26.9 30.6 0.733 0.704 1.151 0.602 0.846
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
## 5 male alive    156     247 24.1 31.5 0.715 0.706 1.129 0.575 0.821
```

```
## 6 male alive    161     253 26.5 31.8 0.780 0.743 1.144 0.607 0.893
```

What we are looking at now is a data frame. Visually, this is a two-dimensional table of data that includes columns of various types (in this case, words and numbers). It is completely fine to think about the data frame this way, but the way that R sees the data frame is as an ordered list of vectors, with each vector having the same number of elements. I will create a smaller one to demonstrate what I mean.

```
eg_list <- list(A = c("X", "Y", "Z"), B = c(1, 2, 3), C = c(0.3, 0.5, -0.2));
eg_list;
```

```
## $A
## [1] "X" "Y" "Z"
##
## $B
## [1] 1 2 3
##
## $C
## [1]  0.3  0.5 -0.2
```

Notice that the list that I created above includes three elements, which I have named 'A', 'B', and 'C'. Each of these elements is a vector of length three (e.g., the first element 'A' is a vector that includes "X", "Y", and "Z"). The whole list is called `eg_list`, but if I wanted to just pick out the first vector, I could do so with the `$` sign as below.

```
eg_list$A;
```

```
## [1] "X" "Y" "Z"
```

Notice how only the first list element (vector with "X", "Y", and "Z") is printed. Somewhat confusingly there are at least two other ways to get at this first list element. The notation for identifying list elements is to enclose them in two square brackets, so if I just wanted to pull the first element of `eg_list`, I could also have typed the below to get an identical result.

```
eg_list[[1]];
```

```
## [1] "X" "Y" "Z"
```

Since the first element of the list is named 'A', both `eg_list$A` and `eg_list[[1]]` give the same output. There is a third option, `eg_list[["A"]]`, which is a bit more to type, but is also more stable than the `$` because `$` allows for partial matching (e.g., if a list element was named 'December', then 'Dec' would work, but be careful if you have another list element that starts with 'Dec'!).

```
eg_list[["A"]];
```

```
## [1] "X" "Y" "Z"
```

If we want to get individual vector elements of our list elements, we use a single square bracket. That is, say we wanted to just pick out the second element "Y" in the list element "A". We could do so with the following code.

```
eg_list[["A"]][2];
```

```
## [1] "Y"
```

Note that `eg_list$A[2]` or `eg_list[[1]][2]` would also work just fine. Lists are very flexible in R, and you can even have lists within lists, which could make the notation quite messy (e.g., `eg_list[[1]][[1]][2]` for the second element of the first list in the first list – I don't generally recommend structuring data in this way unless you absolutely need to for some reason). In any case, it is helpful to know sometimes that when we are reading in and working with data frames, we are really just looking at lists of equal vector lengths. We can even turn our `eg_list` into a two-dimensional data frame that looks like the `dat` Bumpus data that we read in earlier.

```
as.data.frame(x = eg_list);
```

```
##   A B    C
## 1 X 1  0.3
## 2 Y 2  0.5
## 3 Z 3 -0.2
```

See in the above how each element name became the column name above. Let's take another look at the `dat` data frame again.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 1 male alive    154     241 24.5 31.2 0.687 0.668 1.022 0.587 0.830
## 2 male alive    160     252 26.9 30.8 0.736 0.709 1.180 0.602 0.841
## 3 male alive    155     243 26.9 30.6 0.733 0.704 1.151 0.602 0.846
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
## 5 male alive    156     247 24.1 31.5 0.715 0.706 1.129 0.575 0.821
## 6 male alive    161     253 26.5 31.8 0.780 0.743 1.144 0.607 0.893
```

Note now that we can refer to each column in the same way that we referred to list elements (note, we could also put `dat` in list form with `as.list(dat)`). So if we just wanted to look at `wgt`, then we could type `dat$wgt`, `dat[["wgt"]]`, or `dat[[5]]`. Because R recognises the data frame as having two dimensions, we could also type the below to get all of the `wgt` values in the fifth column.

```
dat[,5];
```

```
##   [1] 24.5 26.9 26.9 24.3 24.1 26.5 24.6 24.2 23.6 26.2 26.2 24.8 25.4 23.7 25.7
##  [16] 25.7 26.5 26.7 23.9 24.7 28.0 27.9 25.9 25.7 26.6 23.2 25.7 26.3 24.3 26.7
##  [31] 24.9 23.8 25.6 27.0 24.7 26.5 26.1 25.6 25.9 25.5 27.6 25.8 24.9 26.0 26.5
##  [46] 26.0 27.1 25.1 26.0 25.6 25.0 24.6 25.0 26.0 28.3 24.6 27.5 31.0 28.3 24.6
##  [61] 25.5 24.8 26.3 24.4 23.3 26.7 26.4 26.9 24.3 27.0 26.8 24.9 26.1 26.6 23.3
##  [76] 24.2 26.8 23.5 26.9 28.6 24.7 27.3 25.7 29.0 25.0 27.5 26.0 25.3 22.6 25.1
##  [91] 23.2 24.4 25.1 24.6 24.0 24.2 24.9 24.1 24.0 26.0 24.9 25.5 23.4 25.9 24.2
## [106] 24.2 27.4 24.0 26.3 25.8 26.0 23.2 26.5 24.2 26.9 27.7 23.9 26.1 24.6 23.6
## [121] 26.0 25.0 24.8 22.8 24.8 24.6 30.5 24.8 23.9 24.7 26.9 22.6 26.1 24.8 26.2
## [136] 26.1
```

Note that the square brackets above identify the row and column to select in `dat`. Empty values are interpreted as 'select all', meaning that `dat[,]` is the same as writing `dat` – both select the entire data set. To select, e.g., only the first five columns of data, we could use `dat[,1:5]` (recall that `1:5` produces the sequence of integers, `1, 2, 3, 4, 5`). If instead we wanted to select the first three rows, then we could use `dat[1:3, ]`. And if we wanted only the first three rows and first five columns, we could use `dat[1:3, 1:5]`. The point is that the numbers listed within the square brackets refer to the rows and columns of the data frame, and we can use this to manipulate the data.

Here is a simple example. In this data set, the last five columns are measured in inches (all of the other length measurements are in mm). Assume that we wanted to put them into mm to match the 'totlen', 'wingext', and 'head' measurements. We just need to then multiply all of the values in the last five columns (columns 7-11) by 25.4 (1 inch equals 25.4 mm). We could do this column by column. For example, to multiply all of the values in the seventh column `humer`, we could use the following code.

```
dat[["humer"]] <- dat[["humer"]] * 25.4;
```

Verbally, what I have done above is to assign `dat[["humer"]]` to a new value of `dat[["humer"]] * 25.4` – that is, I have multiplied the column `dat[["humer"]]` by 25.4 and inserted the result back into the `dat` array. A short-cut for doing the rest of them all at once (columns 8-11) is below.

```
dat[,8:11] <- dat[,8:11] * 25.4;
```

I am mixing and matching the notation a bit just to get you used to seeing a couple different versions (as a side note, R allows us to assign in both directions, so we could have also typed `dat[,8:11] * 25.4 -> dat[,8:11];`, though it is very rare to do it this way). Now we should have a data set with the last five columns in mm rather than inches.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 1 male alive    154     241 24.5 31.2 17.4498 16.9672 25.9588 14.9098 21.0820
## 2 male alive    160     252 26.9 30.8 18.6944 18.0086 29.9720 15.2908 21.3614
## 3 male alive    155     243 26.9 30.6 18.6182 17.8816 29.2354 15.2908 21.4884
## 4 male alive    154     245 24.3 31.7 18.8214 17.4752 29.1084 14.8336 21.3106
## 5 male alive    156     247 24.1 31.5 18.1610 17.9324 28.6766 14.6050 20.8534
## 6 male alive    161     253 26.5 31.8 19.8120 18.8722 29.0576 15.4178 22.6822
```

Maybe we want to save this data set as a CSV file. To do so, we can use the `write.csv` function as below.

```
write.csv(x = dat, file = "Bumpus_data_mm.csv", row.names = FALSE);
```

There will now be a new file 'Bumpus_data_mm.csv' in the working directory with the changes that we made to it (converting inches to mm). Note my use of the argument `row.names = FALSE`. This is just because the `write.csv` function, somewhat annoyingly, will otherwise assume that we want to insert a first column of row names, which will show up as integer values (i.e., a new first column with the sequence, 1, 2, 3, . . . , 136).

Now that you have a handle on how to refer to rows, columns, and individual values of a data set, I will introduce some functions in R that might be useful for managing data. If at any time we want to look at what objects we have available in Rstudio, then we can use the `ls` function below.

```
ls();
```

```
## [1] "dat"     "eg_list"
```

The above output should make sense because we have assigned `dat` and `eg_list`. Now say we want to get rid of the `eg_list` that I assigned. We can remove it using the `rm` function.

```
rm(eg_list);
```

The `eg_list` should now be removed from R and not appear anymore if I type `ls()`.

```
ls();
```

```
## [1] "dat"
```

Now let's look more at `dat`. Let's say that I want to find out about the attributes of this object. I can use the `attributes` function to learn more.

```
attributes(dat);
```

```
## $names
##  [1] "sex"     "surv"    "totlen"  "wingext" "wgt"     "head"    "humer"
##  [8] "femur"   "tibio"   "skull"   "stern"
##
## $row.names
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
```

```
##  [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
## [109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
## [127] 127 128 129 130 131 132 133 134 135 136
##
## $class
## [1] "data.frame"
```

You can now see the names (column names, which recall are also vector element names), row names, and the class of the object. Note that not all objects will have the same (or indeed any) attributes. But the attributes that we can see gives us a bit of information, and we can actually refer to the attributes themselves with the equivalently named `names`, `row.names`, and `class` functions. For example, if we wanted to get all of the names of dat, we could use the code below.

```
names(dat);
```

```
##  [1] "sex"     "surv"    "totlen"  "wingext" "wgt"     "head"    "humer"
##  [8] "femur"   "tibio"   "skull"   "stern"
```

The same would work for `row.names` and `class`. We can also pull out information for individual columns of data. If, for example, we wanted a quick count of the alive versus dead individual sparrows in the data set, we could use the `table` function.

```
table(dat$surv);
```

```
##
## alive  dead
##    72    64
```

If we wanted the set of unique `totlen` values, then we could use the `unique` function.

```
unique(dat$totlen);
```

```
##  [1] 154 160 155 156 161 157 159 158 162 166 163 165 153 164 167 152
```

Let's say that we wanted to know if a sparrow has total length greater than 160. We could use the code below to get a TRUE/FALSE vector for which 'TRUE' indicates a length over 160 and a 'FALSE' indicates a length of 160 or less.

```
dat$totlen > 160;
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [13]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
##  [37] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
##  [49]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
##  [61] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
##  [85]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
##  [97] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
## [121] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
## [133]  TRUE FALSE  TRUE  TRUE
```

The above looks a bit messy, but it can be quite useful if we also know that R interprets `TRUE` as having a value of 1, and `FALSE` as having a value of `0`. We can actually get R to confirm this itself, e.g., by checking if `FALSE == 1`.

```
FALSE == 1;
```

```
## [1] FALSE
```

It does not. But now we can check if `FALSE == 0`.

```
FALSE == 0;
```

```
## [1] TRUE
```

We could do the same for `TRUE` and confirm that `TRUE == 1`. Since `TRUE` is equivalent to 1 and `FALSE` is equivalent to 0, a numeric version of the `TRUE/FALSE` vector above would look like the below.

```
as.numeric(dat$totlen > 160);
```

```
##   [1] 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0
##  [38] 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0
##  [75] 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
## [112] 0 0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 1 1
```

Compare the above to the equivalent `TRUE/FALSE` vector to confirm that they are the same. We can use this to our advantage to count all of the sparrows whose length exceeds 160.

```
sum(dat$totlen > 160);
```

```
## [1] 55
```

Note that what we have done is produce that same `TRUE/FALSE` vector as above, but then summed up all the values. Because `TRUE` values are 1 and `FALSE` values are 0, the number of `TRUE` values get counted up. We can do multiple comparisons though too. Say that we wanted to find out the number of individuals that have a total length either greater than 160 mm or less than or equal to 155 mm. We could make this work using the 'or' operator, represented in R by a vertical line `|`.

```
sum(dat$totlen > 160 | dat$totlen <= 155);
```

```
## [1] 76
```

Notice how the `|` separates the two comparisons, and the `<=` sign is used to indicate 'less than or equals to' (a `>=` would indicate 'greater than or equal to, and `==` is simply 'equal to', as we saw above).

We could also try to pull out a subset of individuals who have a total length greater than 160 mm and are female. We could make this work using the 'and' operator, represented by an ampersand in R, `&`.

```
sum(dat$totlen > 160 & dat$sex == "female");
```

```
## [1] 14
```

Notice how the `&` separates the two comparisons and "female" is placed in quotes (else R will look for an object called 'female' and come up empty with an error message – try this out!).

Now say we wanted to identify the row numbers of the individuals with a total length above 165 mm. We could first find these individuals and assign their rows to a new variable using the `which` function.

```
inds_gt_165 <- which(dat$totlen > 165);
inds_gt_165;
```

```
## [1] 18 57 59 79 83 84 86
```

Now that we have these rows of individuals with a total length above 165 mm, we could use these values in `inds_gt165` to view just these rows of the `dat` data frame.

```
dat[inds_gt_165,];
```

```
##     sex  surv totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 18 male alive    166     253 26.7 32.5 19.4818 19.4310 31.2420 15.2400 22.3012
## 57 male  dead    166     251 27.5 31.5 18.2880 17.5514 28.3972 15.5448 21.5138
## 59 male  dead    166     250 28.3 32.4 19.1516 18.2372 29.9466 15.4178 23.2664
```

```
## 79 male  dead    166     245 26.9 31.7 18.1610 17.6530 28.1178 15.2654 21.5138
## 83 male  dead    166     256 25.7 31.7 19.1008 19.0754 30.1498 15.1130 21.7932
## 84 male  dead    167     255 29.0 32.2 19.4310 18.9230 30.4038 16.2052 21.7170
## 86 male  dead    166     254 27.5 31.4 19.3040 18.8468 28.5496 15.3416 23.2156
```

Notice how the values of `totlen` are all above 165, and the row numbers to the left match the values in `inds_gt_165`. One more quick trick – say that we wanted to check if a living individual was in this subset (obviously we can see that the first one is alive, but pretend for a moment that the data frame was much larger). We could use the `%in%` operator to check.

```
"alive" %in% dat$surv;
```

```
## [1] TRUE
```

Again, note how 'alive' is placed in quotes. We could also check to see if numeric values are in the data set. For example, we could ask if the value '250' appears anywhere in `dat$wingext`.

```
250 %in% dat$wingext;
```

```
## [1] TRUE
```

Next, I want to demonstrate three useful functions in R, `tapply`, `apply`, and `lapply`. All of these functions essentially apply some other function across an array, table, or list. Say that we want to find the means of females and males in the data set. We could do this with the `tapply` function.

```
tapply(X = dat$totlen, INDEX = dat$sex, FUN = mean);
```

```
##   female     male
## 157.9796 160.4253
```

Note that in the above, the argument `X` is what we want to do the calculations across (total length), `INDEX` does the calculation for each unique element in the vector (i.e., 'female' and 'male' in `dat$sex`), and `FUN` indicates the function ('mean' in this case, but we could do 'sd', 'length', 'sum', or any other calculation that we want). We could use all of this to calculate, e.g., the standard error of females and males.

```
N_dat  <- tapply(X = dat$totlen, INDEX = dat$sex, FUN = length);
SD_dat <- tapply(X = dat$totlen, INDEX = dat$sex, FUN = sd);
SE_dat <- SD_dat / sqrt(N_dat);
SE_dat;
```

```
##    female      male
## 0.5220396 0.3435868
```

The function `lapply` works similarly, but over lists. We can remake that example list from earlier.

```
eg_list <- list(A = c("X", "Y", "Z"), B = c(1, 2, 3), C = c(0.3, 0.5, -0.2));
eg_list;
```

```
## $A
## [1] "X" "Y" "Z"
##
## $B
## [1] 1 2 3
##
## $C
## [1]  0.3  0.5 -0.2
```

If we wanted to confirm the length of each of the three list elements, we could do so with `lapply`.

```
lapply(X = eg_list, FUN = length);
```

```
## $A
## [1] 3
##
## $B
## [1] 3
##
## $C
## [1] 3
```

We can try to use a function like 'mean' too, but note that we cannot get the mean of "X", "Y", and "Z", as this does not make any sense. If we try to do it, then R will give us an answer of `NA` for the first element with a warning, then calculate the means of the numeric elements.

```
lapply(X = eg_list, FUN = mean);
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
```

```
## $A
## [1] NA
##
## $B
## [1] 2
##
## $C
## [1] 0.2
```

Finally, the function `apply` works similarly on arrays of numbers. That is, we need to have an object with two (or more) dimensions in which every element is a number. To get an example, we can just use the last nine columns of `dat` (i.e., everything except `sex` and `surv`). We can define this below in a new object.

```
dat_array <- dat[,3:11];
head(dat_array);
```

```
##   totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 1    154     241 24.5 31.2 17.4498 16.9672 25.9588 14.9098 21.0820
## 2    160     252 26.9 30.8 18.6944 18.0086 29.9720 15.2908 21.3614
## 3    155     243 26.9 30.6 18.6182 17.8816 29.2354 15.2908 21.4884
## 4    154     245 24.3 31.7 18.8214 17.4752 29.1084 14.8336 21.3106
## 5    156     247 24.1 31.5 18.1610 17.9324 28.6766 14.6050 20.8534
## 6    161     253 26.5 31.8 19.8120 18.8722 29.0576 15.4178 22.6822
```

Now say that we wanted to get the mean value of every column. We could go through individually and find `mean(dat_array$totlen)`, `mean(dat_array$wingext)`, etc., or we could use `apply`.

```
apply(X = dat_array, MARGIN = 2, FUN = mean);
```

```
##    totlen   wingext       wgt      head     humer     femur     tibio     skull
## 159.54412 245.25735  25.52500  31.57279  18.59691  18.10852  28.79258  15.30126
##     stern
##  21.33432
```

The `MARGIN` argument is the only different one from `lapply` and `tapply`, and it is a bit confusing at first. It states the dimension over which the function will be applied, with 1 representing rows and 2 representing columns. This makes more sense when you consider that numeric arrays (unlike data frames) can have any number of dimensions. For example, consider this three dimensional array.

```
eg_array <- array(data = 1:64, dim = c(4, 4, 4));
eg_array;
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   21   25   29
## [2,]   18   22   26   30
## [3,]   19   23   27   31
## [4,]   20   24   28   32
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   33   37   41   45
## [2,]   34   38   42   46
## [3,]   35   39   43   47
## [4,]   36   40   44   48
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,]   49   53   57   61
## [2,]   50   54   58   62
## [3,]   51   55   59   63
## [4,]   52   56   60   64
```

See how the array has four rows, four columns, and four different layers. Now if we wanted to pull out, e.g., the first row and second column of the fourth layer, we could do so with square brackets as below.

```
eg_array[1, 2, 4];
```

```
## [1] 53
```

We could also use `apply` across the third dimension (which I've been calling 'layer') to get the mean of each.

```
apply(X = eg_array, MARGIN = 3, FUN = mean);
```

```
## [1]  8.5 24.5 40.5 56.5
```

We could even get the mean column value *for each layer* with the somewhat confusing notation below.

```
apply(X = eg_array, MARGIN = c(1, 3), FUN = mean);
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    7   23   39   55
## [2,]    8   24   40   56
## [3,]    9   25   41   57
## [4,]   10   26   42   58
```

You will probably never need to actually do this, but it is possible. To read the above, note that the each row represents a layer in the original `eg_array`, and each column represents a row of that layer. So, e.g., the mean of 1, 5, 9, and 13 is 7; the mean of 2, 6, 10, and 14 is 8, and so forth. Moving onto the next layer, the

mean of 17, 21, 25, and 29 is 23. Again, you will almost never need to do this, but it can be useful if you are working with multidimensional arrays.

There are a lot more tips and tricks that I could potentially introduce here, but I will leave it to you to explore a bit. In the next session, we will look at out to write custom R functions, followed by how to use loops in R. Both of these topics are extremely useful for R coding.