# Introduction to coding

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

Brad Duthie

18 OCT 2021

## 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;
}
```

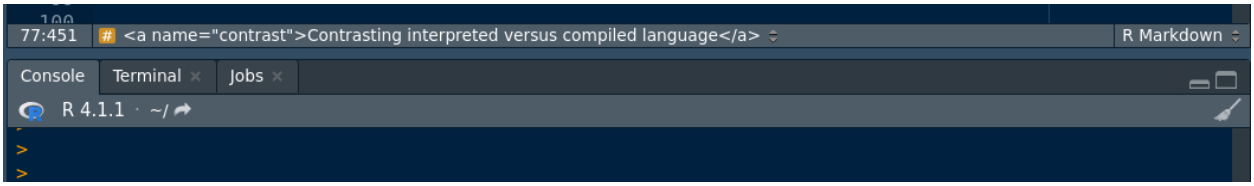
Once pasted, save the file as `count_to_1_billion.c` (remember where you save it). If you get a box that pops up asking "Are you sure you want to change the type of the file so that it is no longer an R script?" then click "Yes." Note that you could have also pasted the code into a text editor such as 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
```

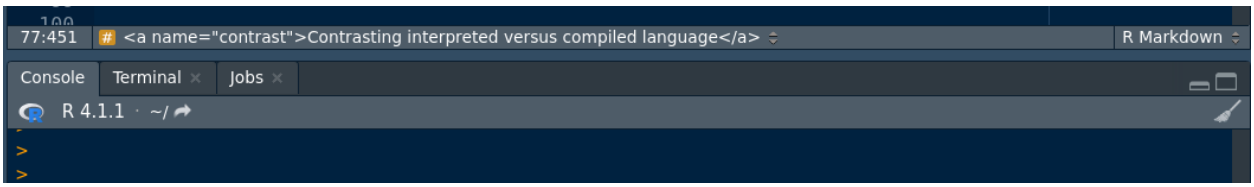
The gcc compiler should be on most Linux and Mac computers. The `-Wall` above tells the compiler to give us warnings if necessary, and the rest of the code tells us to use the `count_to_1_billion.c` file to build the program `count_to_1_billion`. If there are no warnings, then the compiler should execute, 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. You can do this by finding an external one on your computer (it's an app called 'Command Prompt' in Windows), or by using the 'Terminal' tab right within Rstudio (see the middle tab below).




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 can again use the same `cd` command (hint `cd ..` moves you up a directory), typing `cd loc_sim` followed by the Enter key.

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

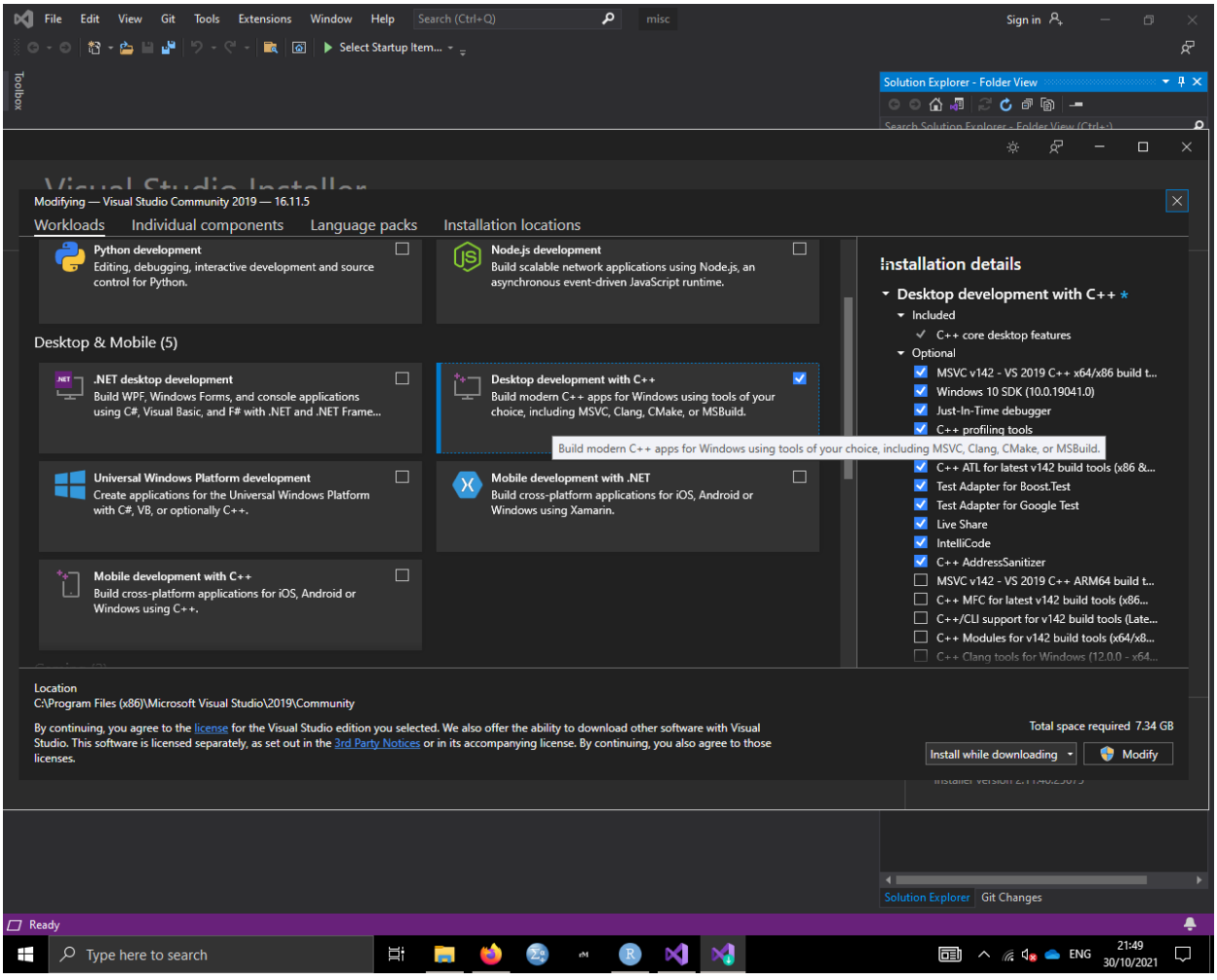
This puts me in the loc_sim directory (on my Windows machine). If I want to view the contents of this directory, I can type `DIR` (I should see my file).

```
C:\Users\User\loc_sim>DIR
 Volume in drive C has no label
 Volume Serial Number is XXX-XXX

 Directory of C:\Users\User\loc_sim

30/10/2021 21:08    <DIR>          .
30/10/2021 21:08    <DIR>          ..
09/02/2021 22:28               255 count_to_1_billion.c
               1 File(s)            255 bytes
               2 Dir(s)  112,115,658,752 bytes free
```

Now let's compile the code. Microsoft windows does not actually come with a pre-installed compiler for the 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).



I might also recommend installing the Linux tools as well (scroll down).

## More R code to help get started

```
read.csv
head
dim
dat[4, 6]; # First row, second column?
hist
summary
plot
cor.test
lm
t.test
aov
```

## References