# Lab 3: Introduction to C

For this lab, you will need to be on a linux system. If you have a linux system or a Mac, you are already ok. If you are Windows, you will need to connect to a linux system — https://its.ucsc.edu/unix-timeshare/tutorials/how-to-connect.html . If you are on your own system, then install BitVise and connect to unix.ucsc.edu . If you are on a lab computer open the "Software" folder on the Desktop, then navigate to "Internet Tools" -> "Unix - SSH", and double click on the "unix.ucsc.edu" shortcut.

If you are doing this, you have two options, use a command line text editor on linux or work locally on your computer and copy over the files to compile and run. While it will have a steeper initial learning curve, it's suggested to use a text editor on linux, as it's very easy to forget to copy over a file which can lead to some very frustrating bugs. If you have a text editor you like, then go with that, but if you don't, emacs with CUA-bindings is probably the closest for people used to working on Windows. To use it, you will need to do the following:

1) Open .emacs by typing
   `$ nano ~/.emacs`
2) Type
   (cua-mode 1)

3) Save the file by pressing Ctrl+o and close nano by pressing Ctrl+x
4) You can now use emacs, for instance, opening up the included code by typing
   `$ emacs tracery_recursion.c`
5) The commands for copy (Ctrl+c), cut (Ctrl+x), and paste (Ctrl+v) will be as you expect. But a few won't be like Find (Ctrl+s), Find Backwards (Ctrl+r), Save (Ctrl+x Ctrl+s), or Exit (Ctrl+x Ctrl+c)

This lab will introduce you to C. C is something akin to the grandfather of Java (Java came from C++ and C++ came from Java). It is much closer to the bare metal of how a computer operates, and as such does not have a lot of the niceties that we have gotten used to in Java. This lab once again returns to Tracery, although in this case we'll be doing something that we took for granted before — splitting a line. To do this, we will become acquainted with C: how to compile a program, how to allocate memory, and how to free memory.

You will be doing the following:

- Compiling a C program
- Including the correct standard libraries to read from and print to a command line
- Iterating over the input character by character, constructing strings as you encounter the correct delimiters
- Allocating memory for the strings, filling the strings with the correct characters, and freeing memory when it is no longer needed

Open the file `tracery_recursion.c` that came with the lab. This is where you'll make all your changes.

Inside the file you will see what is basically the simplest C program possible

```
int main(int argc, char** argv){
return 0;
}
```

Just as with Java, the entry point to the program is a function called `main`, but unlike Java, this main function is floating free and is not contained in a class. In fact, C doesn't have classes: functions and data (`struct` in C) are completely separate. This main is very similar to the one you've seen in Java, and is essentially how one would write the equivalent to

```
public static void main(String[] args) {
}
```

in C.

The differences being:

1. There is no `String` class in C, and the C equivalent is a pointer to a location in memory containing characters, `char*`
2. While C has arrays, arrays are nothing more than a pointer to a memory location storing multiple of a certain type. So the C equivalent to a `String[]` can be written as a pointer to a location in memory containing pointers to locations in memory containing characters, `char**`.
3. These pointers don't have any of the nice helper functions we've gotten used to in Java, so there isn't a way to do anything like `argv.length()`. Instead, the integer `argc` contains the number of strings found in `argv`. C is somewhat nice

and gives you this for free, but any other time you have an array of strings, you'll have to keep track of the count yourself

As a start, we'll print out the arguments passed to program. To do this, we'll need to include the correct libraries. In Java, you always have access to the System library (where we've been using `System.out.println`), but in C there are very few things available to you without including the correct libraries. In C, to include a library you have to use a preprocessor directive (`#`) that tells the compiler to include the header file for the file you want to include. This is done with the code:

```
#include <stdio.h>
```

NOTE: It is common practice in C to use < > for standard libraries and " " for your personal created libraries but the following code snippet

```
#include "stdio.h"
#include <mylibrary.h>
```

is equivalent to

```
#include <stdio.h>
#include "mylibrary.h"
```

The `stdio` (Standard I/O) library contains just what you would expect, a collection of input and output functions. The first that we will acquaint ourselves with is the C equivalent to `System.out.println` — `printf` (formatted print).

The arguments to printf are:

```
printf(char* formatting_string, ...)
```

Where … is a variable number of arguments based on the formatting string. An example of this is

```
printf("%d, %d, %0.1f, %c, %s\n", 1,2,3.4,'a',"bcd");
```

which would print the line

```
1, 2, 3.4, a, bcd
```

Where `%d` means print an integer, `%f` means print a floating point number (and the 0.1 means only show the first decimal place), `%c` means print a character, and `%s` means print a string. Note that unlike `println`, `printf` doesn't automatically print a new line character, so we have to add that ourselves, `'\n'`.

Given this, you now have the tools to print out all of the arguments passed in to the program.

## Todo 1

- In TODO 1, you will add code that loops through the arguments and prints each argument on its own line

We'll now compile our program.  The command to do that is

```
$ gcc tracery_recursion.c
```

You might get an error, since most of you will probably have written code where you declare your iterator inside your for loop. C doesn't like that, and originally, you had to place all declarations at the beginning of every function before you did anything else. However, times have changed, and so has C, so now you can add the option

```
$ gcc tracery_recursion.c -std=c99
```

to tell the compiler to use the C standard that allows this , C 1999.

By default, gcc will compile this to a program named a.out which you can run with the command `$ ./a.out` .  However, we would like a more descriptive name, so run the compiler with the additional option

```
$ gcc tracery_recursion.c -std=c99 -o tracery_recursion
```

which will now generate a program called tracery_recursion which we can run with the command `$ ./tracery_recursion`.
When you run tracery_recursion with no arguments, you should get the output

```
tracery_recursion
```

If you were to run with the arguments

```
$ ./tracery_recursion abc def
```

You should get the output

```
tracery_recursion
abc
def
```

You'll notice that unlike Java, the first argument in C is always the name of the program you ran, so you'll need to keep account of that in the future when dealing with the command line.

Now we come to memory management in C.  We can declare an array in C in a very similar way to in Java:

```
int int_array[3];
```

Which declares an integer array with 3 elements. Declaring an array like this is referred to as "allocating on the stack." Stack memory is what you get when you declare a local variable of some type in a function definition. Stack memory is allocated when the function is called and is de-allocated when it returns. The memory area associated with a given function call is called a stack frame or just a frame. A frame includes memory for all local variables, formal parameters, and a pointer to the instruction in the calling function to which control will be transferred after the function returns. The function call stack is literally a stack data structure whose elements are (pointers to) these so-called frames. The frame at the top of the stack corresponds to the function currently executing. Each function call pushes a new frame onto the stack, and each return pops a frame off the stack.

The following code will compile but won't do what we would want:

```
int* make_array(){ // This function returns a pointer to an int
      int int_array[3]; // Declare an int array on the stack
     // Set the three int array locations equal to values
      int_array[0] = 1;
      int_array[1] = 10;
      int_array[2] = 100;
     // The value of the name of an array (without any []) is the
     // memory location (a pointer) of the beginning of the array.
     // So the line below returns an int*, which is a pointer to the
     // beginning of int_array.
      return int_array;
}
int main(int argc, char** argv){
  // Store the int* returned from make_array() into made_array
  int* made_array = make_array();

  // This won't work, because the pointer returned from make_array()
  // is no longer valid when make_array() returns! This is because
  // int_array was allocated on the call stack, not the heap.
  printf("%d\n",made_array[1]);
}
```

The thought would be that this should print `10`, but instead this will give us `Segmentation fault (core dumped)`

Which is due to us trying to access memory that is no longer allocated — the int_array we return in make_array essentially ceases to exist once we leave the function.  If we want to be able to return something like an array, we have to dynamically allocate the memory, referred to as "allocating on the heap."  Heap memory is not associated with the function call stack and must be explicitly allocated and de-allocated by program instructions. Heap memory is often said to be dynamically allocated, which means that the amount of memory to be used can be determined at run time. Storage in the heap is organized into blocks of contiguous bytes, and each block is designated as either allocated or free. Allocated blocks are reserved for whatever data the programmer wishes to store in them. Free blocks are simply those blocks which are not currently

allocated. It is important to remember that the code you write should ***never*** access the contents of free blocks. Most bytes in free blocks contain meaningless garbage, but some bytes contain critical information about the locations and sizes of the free and allocated blocks. If a program corrupts that information, it may crash in a way which is mysterious and difficult to diagnose.

To do this, we need to include another standard library, `stdlib.h`. `stdlib` gives us a lot of functions, but the ones we care about for dynamic memory are:

```
void* malloc (size_t size);
void* calloc (size_t num, size_t size);
void free (void* ptr);
```

`void*` is similar to `Object` in Java: it is a generic pointer to anything, but we don't know anything about the type. When `malloc` or `calloc` are called, they allocate a block of memory (`size` bytes for `malloc` and `num*size` bytes for `calloc`), and return a pointer to the first location in that block of memory. So, the a dynamic version of the code above is:

```
int* make_array(){
      // Allocate enough memory to store the integers on the heap
      // instead of the stack.
     int* int_array = malloc(sizeof(int)*3);
     int_array[0] = 1;
     int_array[1] = 10;
     int_array[2] = 100;
     return int_array;
}
int main(int argc, char** argv){
  int* made_array = make_array();

  // Now this printf will work because the memory for int_array
  // stays around after make_array disappears.
  printf("%d\n",made_array[1]);
}
```

We use, `sizeof(int)` , to tell the compiler how many bytes we want to use for each element of the array. In this case we want to allocate enough memory to store 3 ints. We use the function `sizeof()` because how many bytes it takes to store an integer in C is dependent on the computer we are running it on. So `sizeof(int)` will return different values on different computers, and our code will now work on different computers.

While `malloc` is useful, `calloc` is a slightly better way to do this. `calloc` has two separate parameters, `num` and `size`, so we don't have to do the multiplication (it just looks a bit cleaner), and it zeroes out the new memory block. This might not seem like much, but it's actually very helpful for us as programmers. When `malloc` returns a block of memory, that block might still have stuff in it. A very easy mistake to make is to not properly initialize your memory, and this can be hard to catch as the stuff in the block of memory might be close enough to not look obviously wrong. So, the better way to write the above function is:

```
int* make_array(){
      int* int_array = calloc(3,sizeof(int));
      int_array[0] = 1;
      int_array[1] = 10;
      int_array[2] = 100;
      return int_array;
}
int main(int argc, char** argv){
  int* made_array = make_array();
  printf("%d\n",made_array[1]);
}
```

If you were to compile and run this code, it would seem to do exactly what you think it should. But there's one thing it is missing. Unlike Java, we are responsible for all of the memory we dynamically allocate in C. Java has garbage collection and figures out on its own what to do with dynamically allocated memory (which in Java is any object created with the `new` keyword). But in C this code will have a memory leak. This means that memory is allocated and never freed, which will eventually cause your computer to run out of memory (if the offending code was run enough). So, we need to add the line

```
free(made_array);
```

to the end of the program to release the memory back.

```
int* make_array(){
      int* int_array = calloc(3,sizeof(int));
      int_array[0] = 1;
      int_array[1] = 10;
      int_array[2] = 100;
      return int_array;
}
int main(int argc, char** argv){
  int* made_array = make_array();
  printf("%d\n",made_array[1]);
  free(made_array);
}
```

Having seen how we can allocate and free memory — it's now time to implement a function that will make a new string, and fill it with the contents of the buffer, up to a certain point.

## Todo 2

- Now implement the code for `char* make_string_from(char* from, int count)`
- This function will allocate memory for a new string of size `count+1`. The plus 1 is because C strings are referred to as null-terminated (the last element of the string is the `\0` (a zero) and it tells functions like printf that the end of the string has been reached).
- Loop through the characters in `from` up to index `count` and put them into your new string
- Finally, return the new string.

Now, we are going to read in from standard input, as we did in the last lab. Unlike the last lab, we are going to be reading in one character at a time.

You will run your code with

```
$ ./tracery_recursion abc def < grammar-story.txt
```

which is functionally equivalent to

```
$ cat  grammar-story.txt | ./tracery_recursion abc def
```

So, we are going to loop through the input one character at a time, checking to see what those characters are, storing some of them, and making and printing strings for others.

Recall that a line in a grammar looks like

```
color:red,blue,green
```

Where everything before a `':'` is the name of the rule, and everything between `','` is an expansion. Before, all of the strings we received were split on the new lines, so we also have to be aware of them. The above actually looks like:

```
"color:red,blue,green\n"
```

Because the newline character is denoted as `'\n'`. Before this next part, we are going to declare a few variables — `char_buffer`, `buffer_index`, `rule`, and `expansion`. `char_buffer` will be a statically allocated array of characters, that we will fill up with the characters that we read in one at a time, using `getchar()`. Whenever we read in a character that isn't a `':'`, `','`, `'\n'`, or `EOF` we will put that character into `char_buffer`, moving `buffer_index` up by one. When we read a `':'`, we know that everything in `char_buffer` up to `buffer_index` is a rule, and we'll set `rule` to be a new string with those characters using the helper function we made in Todo 2. We will then reset `buffer_index` to 0.

Similarly, whenever we read a `','` or `'\n'` we know that we've just finished reading an expansion and we will set `expansion` to be a new string with the correct characters from `char_buffer`. We will now print `"A potential expansion of rule '<RULE>' is '<EXPANSION'>\n"`

e.g., with the above line, we should expect to see:

```
A potential expansion of rule 'color' is 'red'
A potential expansion of rule 'color' is 'blue'
```

```
A potential expansion of rule 'color' is 'green'
```

If the character is `EOF`, then we have hit the end of the file, we are done.

## Todo 3

- Use `getchar()`, which takes a character from standard input one at a time, to loop through all of the input in standard input, until we hit the `EOF` character. EOF is a constant (like null) that you can use to tell that you've reached the end of the input.
- If the character is a `':'` , then make a new string called rule with the contents of the buffer up to the current index, and reset `buffer_index` to 0.
- If the character is either a `','` or `'\n'` then we know we have just read in an expansion, so make a new string called expansion with the contents of the buffer up to the current index, and reset the index to 0
- Otherwise, put the character into `char_buffer`, a statically allocated character array of size 1000, moving to the next index after you read a character
- Whenever you read in an expansion — you should print the following line `"A potential expansion of rule '<RULE>' is '<EXPANSION'>\n"` - where `<RULE>` is the last rule you read in, and `<EXPANSION>` is the expansion you just read in.
- As a final note, make sure you `free` all memory you allocate. A good practice to get in is to initialize all pointers to `NULL`. When you want to free a pointer, check to see if it is `NULL`. If it isn't, then free it and set it to `NULL`. If it is, then don't free it, since it's already pointing to nothing). To check if your program is properly freeing all memory use a tool called valgrind to check your code like so:

```
$ valgrind --leak-check=full -v ./tracery_recursion <
grammar-story.txt
```

Which will tell you if you have a memory-leak, and if so, where.

Expected Behavior

If you run your code with

```
$ ./tracery_recursion < grammar-story.txt
```

You should get the output

```
An expansion for rule 'animal' is 'cat'
An expansion for rule 'animal' is 'emu'
An expansion for rule 'animal' is 'okapi'
An expansion for rule 'emotion' is 'happy'
An expansion for rule 'emotion' is 'sad'
An expansion for rule 'emotion' is 'elated'
An expansion for rule 'emotion' is 'curious'
An expansion for rule 'emotion' is 'sleepy'
An expansion for rule 'color' is 'red'
An expansion for rule 'color' is 'green'
An expansion for rule 'color' is 'blue'
An expansion for rule 'name' is 'emily'
An expansion for rule 'name' is 'luis'
An expansion for rule 'name' is 'otavio'
An expansion for rule 'name' is 'anna'
An expansion for rule 'name' is 'charlie'
An expansion for rule 'character' is '#name# the #adjective#
#animal#'
An expansion for rule 'place' is 'school'
An expansion for rule 'place' is 'the beach'
An expansion for rule 'place' is 'the zoo'
An expansion for rule 'place' is 'Burning Man'
An expansion for rule 'adjective' is '#color#'
An expansion for rule 'adjective' is '#emotion#'
An expansion for rule 'origin' is 'once #character# and #character#
went to #place#'
```

## Grading

TODO 1: 20 points
TODO 2: 30 points
TODO 3: 35 points

Style: 15 points

As a note, it is far more important to turn in code that compiles and is missing a TODO or only partially does a TODO, than it is to attempt a TODO and make your code uncompilable.  If you code doesn't compile, you will receive 40 points off, so make sure your code compiles before you submit it (even if you are missing a TODO).

# Turning the code in

- Create a directory with the following name: `<student ID>_lab3` where you replace `<student ID>` with your actual student ID. For example, if your student ID is 1234567, then the directory name is `1234567_lab3`.
- Put a copy of your edited `tracery_recursion.c` file in the directory.
- Compress the folder using zip. Zip is a compression utility available on mac, linux and windows that can compress a directory into a single file. This should result in a file named `<student ID>_lab3.zip` (with `<student ID>` replaced with your real ID of course).
- Upload the zip file through the page for Lab 3 in canvas.