C Notes

General notes for programming in the C language. This will be laid out less like a manual and more like general notes. I will be drawing parallels from Python in these notes, as it was my first coding language and this is my second. Once you've learned your first language, it's easier to learn more as the differences are primarily syntax. That said, if you are reading this, a basic understanding of another coding language is highly recommended before attempting to understand these notes.

Hello World

Any good coding manual or notes begins with how to create a "Hello World" program. Let's look at that here:

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Let's take this line by line:

- `#include <stdio.h>` : This is similar to importing a library in Python. Since C is a low-level language, we must import the ability to print using the Standard Input/Output Library, or `stdio` . Note that unlike Python, hashes `#` in C do not indicate comments.
- `int main()` : This is the primary function to be run. Everything inside this function will execute at runtime. Anything outside of it will not. You can define functions and other data outside of `main()` , but they must be called within `main()` . This is similar to `if __name__ == "__main__":` in Python, except it is still crucial in single-file scripts and programs. This will always be the starting point for any C script.
- `{` and `}` : While Python relies on indentation to define nested code and blocks, C uses curly braces. Indentation isn't as strict in C (and in fact is completely ignored by the compiler), but it is still best practice for readability to indent nested code.
- `printf("Hello World!\n");` : Our main line of code to be executed. `printf()` does functionally the same thing as Python's `print()` , including allowing for escape characters such as the newline (`\n`) we see here. It is best practice to add newlines after `printf` statements, as this is not done automatically and can make your output difficult to read if omitted. `printf` statements *only* allow for double quotes, and not interchangeable

single/double quotes as in python. Also note the semicolon. These are used at the ends of most lines of C code.
- `return 0;` : If the `main()` function returns 0 at the end of runtime. This tells the OS that the program ran successfully. Any other integer usually indicates an error.

Compiling and Running Code

C is different from Python in that it is compiled into machine language before running, while Python is interpreted. All C code must be compiled (or 'built') before it can be run. Doing this is slightly different across Operating Systems. I will be using Linux.

- Ensure that your script is saved with the `.c` extension. It our case, it would be `helloworld.c`
- Open a terminal and navigate to your script's location.
- run `cc helloworld.c` to compile your code. If you get any errors, check your code to ensure it is correct.
- run `ls` to check the files in your current directory. You should see a new filed called `a.out`. This is our compiled and executable script. This can be renamed when compiling the code by running `cc helloworld.c -o helloworld`
- Run your code with `./helloworld`. If you didn't rename it (not recommended) run `./a.out`

If you did everything correctly, you should see `Hello World!` printed in your terminal.

Variables

As in nearly every programming language, Variables are a method of storing data. However, C variables are different from Python in that they are statically typed. This means that we must define the variables' type when they are declared. While in Python we might be able to define a variable like `mynum = 75` and later redefine it as `mynum = 'Seventy-Five'`, the variable conventions are more strict in C. To declare a variable in C, you start by defining its type, followed by its name, then its definition. For example:

```c
int myNum = 75;
```

Now the `myNum` variable must always be an integer, and cannot be redefined to any other data type.
An important thing to note about string variables is that they are defined with `char` for 'character'. If you create a variable such as `char mychar = ...`, it will only be able to hold a single character. If you want to define a variable to contain an entire string, it must be defined as a character array by using square brackets.

```c
char myName[] = "Author";
```

`char` variables use different quotation marks depending on whether it is a single character or an array. Single characters can only use single quotes, while arrays (strings) can only use double quotes.

If we wanted to incorporate our `myName` variable into a `printf` statement, we would do so similarly to Python's print formatting.

```c
#include <stdio.h>

char myName[] = "Author";
int main()
{
    printf("My name is %s\n", myName);
    return 0
}
```

This is what the same script would look like in Python:

```python
myName = 'Author'
print(f'My name is {myName})
```

We use `%s` to indicate that there will be a string variable type in the position of this indicator. Then, outside of the quotes but still within the parentheses, we add a comma and define the string-type variable that will go at the indicated position.
What if we have multiple variables of differing types to go in the same `printf` statement? Let's take a look:

```c
#include <stdio.h>

char myName[] = "Author";
int myAge = 145;
int main()
{
    printf("My name is %s and I am %d years old\n", myName, myAge);
    return 0
}
```

Note that each data type has its own format specifier. `%s` for character arrays and strings, and `%d` for integers. I will have a breakdown of common format specifiers in the next section, Data Types.

Note too that the variable names at the end of the `printf` statement are listed in the order in which they are referenced. This is true for all variables, regardless of data type.

Data Types & `printf`

The following is a breakdown of some of the most commonly used data types in C, as well as their format indicators. Note that this is not all of them, but the rest are more advanced and situational.

Type	Description	Indicator
`int`	Integer - a whole number without decimals.	`%d`
`float`	Floating Point - A 32-bit decimal number.	`%f`
`double`	Double - a 64-bit decimal number.	`%lf`
`char`	Character - a single ASCII character. char definitions only use single quotes.	`%c`
`char[]`	Character Array - A full string of ASCII characters. Character array definitions only use double quotes.	`%s`

I remember theses as d = digit, f = float, lf = long float, c = character, s = string.
Note that you don't necessarily need to define these as variables before inserting them into a `printf` statement. The following is a valid statement:
`printf("My favorite number is %d\n", 50);`

You can also do some interesting things with numbers within a `printf` statement, such as defining a math operator instead of a single number.
`printf("%d", 5+4);`
This will print `9`.

If you expect to receive a decimal number as the returned value of your operation, you must define it as %f. Even if all numbers in the operation are digits, if the returned value will be a decimal number, it's best to define it as such. If our operation were `5/4` and we defined it as %d in the `printf` statement, the printed value would be `1`, as it rounds to the nearest whole number. If any of your operands or expected return value is a floating point, it's best to define the expected returned format indicator as `%f`. Note also that within `printf()` statements, `%f` can be used to indicate both floats AND doubles. However, this is not true for other statements such as `scanf()`, which will be covered in a future section.

There are a few other useful number functions which can also be used outside of `printf` statements. Note that these can only be used by importing `#include <math.h>`:

- `pow(x,y)` - Returns an exponent as x^y.

- `sqrt(n)` - Returns the square root of n.
- `ceil(f)` - rounds the passed-in float *up* to the nearest integer.
- `floor(f)` rounds the passed-in float *down* to the nearest integer.

Comments

As mentioned before, hashes `#` do not indicate comments in C as they do in Python. To comment a single line in C, you would indicate it with a double front slash.

```
//This is a comment and will not be run as code.
```

Docstrings in C are different as well. Rather than three double quotes before and after the docstring, you indicate them with one front slash and one asterisk.

```
/*
This whole section is one big comment.
I can put as much text as I want between these tags.
None of it will be run as code.
*/
```

Note that the starting tag is `/*` and the ending tag is `*/`. The difference is subtle but crucial.

Constants

A constant is a variable type that cannot be changed or redefined later in the code. By convention, these are named using all caps.

```
const int DAYS_PER_WEEK = 7;
```

Note that constants can be any variable type, but they are most commonly integers and floats.

Getting User Input

In Python, getting input from a user is as easy as using the `input()` function. However, in C, it is slightly more complex. We start by creating an empty variable, printing instructions for the user via `printf()`, then saving that input to the empty variable's *pointer*. (We will cover pointers later.)

```
#include <stdio.h>

int age;
int main()
```

```
{
    printf("Enter your age: ");
    scanf("%d", &age);
    printf("You are %d years old.", age);

    return 0;
}
```

Things to note:

- We do not put our instructions for input into the same function as the actual gathered input. We ask for input via `printf()` by printing the instructions on the screen, and then `scanf()` only collects the input itself. We rely on two different functions to communicate what we want from the user, and then actually obtain that input.
- We must specify the data type of the input we are requesting from the user. This is done via a format specifier. Note that even though we are not outputting a string, we still encase the specifier in double quotes.
- Within the same `scanf()` statement, we define the variable *pointer* that we write our user input to, as indicated by the ampersand `&`. `scanf()` works essentially in the opposite direction as `printf()` where the value we define is what will be written to instead of called.
- Quick oversimplified note about pointers: A pointer is essentially the value stored at a given place in memory. In this case, the `age` variable is looking at a specific spot in memory to pull its info. When we write to the age pointer with `&age`, we are changing the value that `age` sees, and therefore changing the value that it calls. Again, we will discuss pointers in detail later on.
- Finally, we print our user input in the form of a `printf()` statement that references the variable we just wrote to.

You can also get user input as doubles (`%lf`) and a single character (`%c`) as long as the variable and `scanf` statement are set up to accept these data types. *NOTE*: When getting user input for a single character, we must add a space before the format specifier. `" %c"` instead of just `"%c"`.
Getting a string from a user is slightly different. To start, when we create our empty string variable, we need to declare the maximum length of the expected string (in characters) within the square brackets.

`char name[20];`

Then, in the `scanf` function, we do *not* modify the pointer, rather the variable itself.

`scanf("%s", name);`

This is because our `name` variable is an array, and we are writing to the array rather than the pointer. More on arrays later.

Now this will allow us to accept a *one word* string from the user. If we insert any spaces, `scanf` will stop reading at the space, and only output the characters before it. For example, if we input "John Smith", the `scanf` function will only read "John".
The best way to get a multi-word string from a user is to replace `scanf` with another function called `fgets`. This function reads a line of characters, and stores it in a specified string. Syntax for `fgets` is as follows:

`fgets(buff, n, stream);`

Where `buff` is the string buffer where the input will be stored. In our case, the `name` variable. `n` is the maximum number of characters to be stored, including the `null` terminator character which is invisible but crucial. and `stream` is the input source. For our purposes, we will use `stdin` for 'standard input'. So in the case of our program above, we replace `scanf()` with:

`fgets(name, 20, stdin);`

One thing to note with `fgets()` is that it terminates with a newline character, so anything printed after your `fgets()` user input will be printed on a new line.

Full example of our full name gathering program looks like this:

```c
#include <stdio.h>

char name[20];
int main()
{
    printf("Enter your name: ");
    fgets(name, 20, stdin);
    printf("Your name is %s", name);

    return 0;
}
```

Arrays

An array can be compared most closely to a list in Python, but it's not quite the same. While in Python you can keep any number of data types mixed together in a single list, Arrays in C only allow for a single data type. Syntax for creating an array looks like this:

`int luckyNumbers[] = {4, 8, 15, 16, 23, 42};`

Just like defining a string (which is an array of characters) we specify the data type that will be going into our array, the variable name, open and closed square brackets, and the array itself. In the case of integers and floats/doubles, we encase the array in curly braces, with each element separated by a comma.
Indexing an array is very similar to Python, where index positions start at 0 and calling a specific index involves calling the variable name, followed by the index position in square brackets. If we wanted to print the number 15 from the above, we would do something like this:

`printf("The third number is %d\n", luckyNumbers[2]);`

You can also reassign values via indexing.

`luckyNumbers[1] = 200`

Much like string arrays, we can create an empty array variable by defining the type, name, and maximum amount of values to be stored in that array inside the square brackets.

`int unluckyNumbers[20];`

Now we have an empty array that can hold up to 20 whole numbers, and can be added to later. Arrays do not have an `append()` function as they do in Python. If you have an array, you can only add to them by assigning specific index positions as outlined above. There is also no way to easily delete values from arrays, so be sure you know what you're doing before appending an array.

Functions

Functions in C are fairly similar to Python in that you can define a block of code with a name (either inside or outside of your `main()` function) and then call the function name and arguments later in your code. The `main()` function where we put all of our runtime code is, to no one's surprise, a function. It gets automatically called when our script is compiled and run. Functions in C differ slightly in syntax from that of Python. Let's take a look:

```
#include <stdio.h>

void sayHi()
{
    printf("Hello User!");
}

int main()
{
    sayHi();
    return 0;
}
```

This will print `Hello User!` to the console.

- `void sayHi()` - Every function definition in C begins with a data type - similar to a variable. This tells C what data type will be returned from the function. This function returns `void`, or no returned data. `main()` returns an integer (0 if successful, or another integer if there are errors) so it is defined with `int`. More on return statements later. We then define the function's name, and arguments (if any) within the parentheses. Note how there is no semicolon after function signatures.
- `{` and `}` - As before, all code to be executed by the function goes between these curly braces. Indentation is technically not required (and in fact is ignored by the compiler) but is still required per style convention for readability.
- `sayHi();` - Skipping ahead a few lines because we know what the code in between does. Within `main()` we call our function name, as well as arguments (if any) within the parentheses. Note how we *do* use a semicolon when calling a function.

When we do define parameters for our function, we do so as if we were defining a variable - by defining the expected data type, as well as the variable name.

```
void sayHi(char name[])
{
    printf("Hello %s!\n", name);
}
```

Now, within our `main()` function, we can call `sayHi("Author");` to output `Hello Author!`

Once you understand syntax and parameter definition, function use is essentially the same as Python.

Return Statements & Function Prototyping

Let's establish a sample script to use as an example for this section:

```
#include <stdio.h>

double cube(double num)
{
    return num*num*num;
}
int main()
{
    printf("Answer: %f", cube(7.0));
}
```

In this script, we've created a function to output the cube of a given double, then in the `main()` function, we've printed out the cube of 7.0.

As we know with Python functions, we can call `return` at the end of the function to return the specified data and break the function. The same concept is true in C, only we must specify exactly what data type we expect to return at the initial definition of the function. Rather than using the `def` keyword, we use data types like `int`, `float`, or `void`.

Important note regarding function order:
Function order matters! If you attempt to call a function before the place in code that it is defined, the compiler will throw an error. This means that, in the case above, we MUST define `cube()` first, because it is getting called later inside of `main()`. Attempting to call `main()` BEFORE `cube()` will give an error, because that function hasn't been defined yet chronologically.

One way around this is by prototyping functions - where the function is defined below `main()`, but its function signature is copy/pasted above `main()`. The signature is essentially the definition line, where we create the function, give it a name, and outline its parameters. Note that we we use function prototyping, a semicolon is required.

Here is an example of function prototyping using the same function as above:

```c
#include <stdio.h>

double cube(double num);

int main()
{
    printf("Answer: %f", cube(7.0));
}
double cube(double num)
{
    return num*num*num;
}
```

Note that we've defined the full `cube()` function below `main()`, but we've prototyped it by putting its signature before `main()`.

If Statements

C has if/elif/else statements much like Python. The primary difference is Syntax. it should also be mentioned that `elif` in C can only be expressed as `else if`. Syntax for these statements looks like this:

```
if(condition1){
    //code to run under condition 1
} else if(condition2){
    //code to run under condition 2
} else {
    //final code to run if none of the above conditions are met
}
```

Note how the conditions (such as 2 > 1) are in parentheses after each respective if statement, and the blocks of code to be run for each condition are contained within their own set of curly braces after the condition is defined.
If statements tend to use logical operators, so here is a guide:

- `&&` - AND operator. Both sides of AND must be true.
- `||` - OR operator. One or both sides of the OR operator must be true.
- `!` NOT operator. Commonly used in conjunction with mathematical operators, such as `!=` for "Is not equal to". NOT can also be used to negate a statement. Writing `if(!(3<2))` in C is the same as writing `if not 3<2` in python.

Switch Statements

Switch Statements are similar to `if/else` statements in that they provide an output based on several specified cases. However, Switch statements are best utilized when the cases are simple and specific, while `if/else` statements are suited to more broad and complex conditions.

Syntax for switch statements is as follows. Assume this is in the context of a larger script with necessary elements such as the `main()` function.

```
char grade = 'A';

switch(grade){
case 'A':
    printf("You did great!");
    break;
case 'B':
    printf("You did pretty well!");
    break;
case 'C':
    printf("You barely passed!");
    break;
case 'D':
    printf("You did very poorly.");
```

```
        break;
    case 'F':
        printf("You failed!");
        break;
    default:
        printf("Invalid grade!");
        break;
}
```

Let's break this down:

- We begin by defining a variable - `grade`. With Switch statements, it looks at the specified variable and adjusts its output according to the defined cases.
- We define a switch statement and pass our watched variable into the parentheses. After this, *all* cases are defined within a single set of curly braces.
- This is where it begins to resemble an if statement. `case 'A':` is similar to writing `if(grade == 'A')` From here we define the code to run if this case is true.
- One big difference between if & switch statements is the presence of `break`. if statements do not require this. They will output based on the appropriate condition, and the rest of the code will keep running. Meanwhile, switch statements without `break` will continue executing subsequent cases in an event called "Fall through."
- `default` at the end of a switch statement is similar to `else` in an if statement. If none of the other cases are true, the switch statement will run the default code.

Structs

A struct in C is essentially a collection of information stored in a single place. Once you've set up a struct, you can create multiple instances of the same struct that holds different values of the pre-defined struct variables. Let's take a look at what this looks like in practice:

```
struct Student{
    char name[50];
    char major[50];
    int age;
    double gpa
}
```

We start with the `struct` keyword, followed by the name of the struct, which is capitalized by convention. All variables for this struct are stored within a single set of curly braces. Note that the closing curly brace is followed by a semicolon. Inside of the struct, we define several variables that are relevant to whatever we happen to be keeping track of. In this case, we are storing basic info on students for a school admin program. Note that the variables are all empty.

We will assign their data later.
Now that we have defined the struct and the data that goes into it, we can now create an instance of this struct and assign its variables data.

```c
struct Student student1;
student1.age = 22;
student1.gpa = 3.2;
```

Now we have an instance of our Student struct called `student1`, with their age and GPA stored. You may have noticed that we have not yet assigned a name or major to `student1`. This is because those would be strings. As we know, in C, strings are simply arrays of characters. As such, we cannot simply assign a brand new value to this array. Instead we will use the String Copy function to copy the value of a string and assign it to the array.

```c
strcpy(student1.name, "Jim");
strcpy(student1.major, "Business");
```

The syntax of the String Copy function is:
`strcpy(destination, source);`

where `destination` is the pointer to the destination character array where the string will be placed, and `source` is the pointer to the character array to be copied.

We can create multiple instances of this struct for `student2`, `student3`, etc. and assign their values accordingly. We could then call these individual values with `student1.name` or `student3.gpa` etc.

Let's take a look at what this would look like with two students, and print out their names and ages.

```c
#include <stdio.h>

int main(){
    struct Student{
        char name[50];
        char major[50];
        int age;
        double gpa
    };

    struct Student student1;
        strcpy(student1.name, "Jim");
        student1.age = 22;
        strcpy(student1.major, "Business");
        student1.gpa = 3.2;

    struct Student student2;
        strcpy(student2.name, "Pam");
        student2.age = 23;
        strcpy(student2.major, "Art");
        student2.gpa = 4.0;

    printf("Student 1's name is %s and their age is %d", student1.name,
student1.age);
    printf("Student 2's name is %s and their age is %d", student2.name,
student2.age);

    return 0;
}
```

While Loops

While loops are basically the same conceptually across most languages. As we know from Python, a while lop will repeatedly loop through its specified code until the predefined condition is no longer true. The same is true in C. The biggest differences between the two are of course syntax, and the fact that C doesn't have a direct contemporary to `while True:`. However, we can still create an infinite while loop by creating unbreakable conditions.
While loop syntax in C looks like this:

```
while(condition){
    //code to run;
}
```

We can create a simple program that counts to 5 using a while loop and incrementation:

```
#include <stdio.h>

int main(){
    int index = 1;
    while(index < 5){
        printf("%d\n", index);
        index++;
    }
    return 0;
}
```

Of course since we have a foundational knowledge of while loops, this script shouldn't be too much of a challenge to read. My reason for including it is to showcase a real example of a while loop, and to display what incrementation looks like. In Python we might have written `index += 1`, but adding 1 to `index` in C looks like `index++`. The same is true of decrementation, where we use `--` instead of `++` Adding or subtracting any other amount from `index` would be the same as Python, with `index += n` or `index -= n` respectively. There is also prefix incrementation/decrementation, but that's a topic out of the scope of these notes.

C also has a while loop variant called a do while loop, where the block of code contained within the while loop is run first before checking if the while condition has been met. Syntax looks like this:

```c
#include <stdio.h>

int main(){
    int index = 6;
    do{
        printf("%d\n", index);
        index++;
    }while(index < 5);
    return 0;
}
```

In this do while loop, the `printf` and `index++` lines will still run once even though `index` is greater than 5. This is because the instructions are defined before the condition.

For Loops

For Loops in C differ a bit from For Loops in Python. In Python, For loops are iterator-based, meaning it loops through an iterable and stops when it is out of iterators. Examples being `for i in range(10):` or `for letter in "string":`. In these cases, the for loop will stop when it has looped for each integer in the range 0-9, and when it has looped 6 times (the number of letters in "string") respectively. Python is capable of automatically counting these loops, and stopping when required.
In C, we do not have access to these same iterators, We can't give it any old range or string and say "Loop this many times." Instead we must manually specify the amount of times a loop will happen. In the cases of the examples above, we would need to tell C "Loop 10 times" or "Loop 6 times". For loops also have the ability to increment more than once per loop, but we'll discuss that later.
Below is the syntax of a For loop in C:

```
for (expression1; expression2; expression3){
    //Code block to be executed;
    }
```

- Expression 1 is executed one time *before* the execution of the code block.
- Expression 2 defines the condition for executing the code block
- Expression 3 is executed every time *after* the code block has been executed.

A more practical example looks like this:

```
for (int i = 0; i < 5; i++){
    printf("%d\n", i);
}
```

This will print numbers 0-4.

- Expression 1 - `int i = 0;` : This is the integer variable that is checked before this for loop is run. You could also specify another integer outside of the loop: `int i;` - and then later within this loop rewrite expression 1 to simply assign the variable a value. `for (i = 0...;` etc.
- Expression 2 - `i < 5;` This is the condition that is checked just before the block of code is run. As long as `i` is less than 5, the code will continue to run.
- Expression 3 - `i++`. As discussed previously, `++` means "increment by 1." This will add 1 to the value of `i` each time the loop is executed.

So in essence, we define an incrementor variable, check if the incrementor is under 5, then execute our code and increment our incrementor by 1. This repeats until the incrementor is no longer less than 5.

This kind of incrementation is very common in C, especially in beginner programs. Of course you can decrement `i` with `i--` for counting down, or you can only print even numbers with `i += 2`. You can also define multiple variables in expression 1, and control multiple variables in expression 3 - but this is slightly more advanced than the scope of these notes.

If we wanted to loop through an array similar to how we might in Python, we need to know how many elements are in our array, and then specify that number as the amount of times to run the loop in expression 2.

```
int luckyNumbers[] = {4, 8, 15, 16, 23, 42};

for(int i = 0; i < 6; i++){
    printf("%d\n", luckyNumbers[i]);
}
```

So in this example, we specify `i < 6` because there are 6 elements in `luckyNumbers`. Since indexing starts at 0, `i` will have incremented 6 times by the time this expression becomes true and the loop stops. Then, we index our `luckyNumbers` array by u sing `i` as the index position. As it increments, so will the index position.

2D Arrays & Nested Loops

A 2D array in C can be best compared to a list of lists in Python. Each element of the array is itself an array. In Python, this might look like:
`nums = [[1,2,3],[4,5,6],[7,8,9]]`

While in Python the lists could be any length and contain any data, C is a bit more strict in how a 2D array can operate. Let's look at an example:

```
int nums[3][3] = {
    {1,2,3},
    {4,5,6},
    {7,8,9}
    };
```

Note: This can be written all on the same line as in the Python example. I formatted it this way for clarity.
Note that our definition line contains two sets of square brackets, indicating a 2D array. The number of square bracket sets directly relates to the number of dimensions in the array. A 3D array (an array containing arrays that contain arrays) would have 3 sets of square brackets. For now, we will focus on 2D arrays. The numbers inside the square brackets indicate the number of sub-arrays, and the number of elements in each sub-array in that order. This means that, in this example, this 2D array *must* contain 3 sub-arrays, and each sub-array *must* contain 3 elements.

Indexing sub-arrays is similar to the way it's done in Python. Two sets of square brackets are used. In the first, we specify the index position of the sub-array to pull from. In the second, we specify the index position of the element to be indexed. For example, if we wanted to print the number 6 from our `nums` array, it would look like this:

`printf("%d", nums[1][2]);`

This is because the sub-array containing the number 6 is at index position 1, and 6 itself is at index position 2 within the sub-array.

As with normal arrays, the 2D array variable can be defined with no data, and index positions assigned later in code.

Nested Loops

These are included within the 2D array section because the two concepts go hand in hand. A nested loop functions exactly the way you'd think - it's a loop within a loop, where the second loop executes as many times as specified, and then continues doing so as long as the primary loop's condition is true. If we wanted to print every element of each sub-array in our `nums` example, it would look like this:

```
int i,j;
for(i = 0; i < 3; i++){
    for(j = 0; j < 3; j++){
        printf("%d\n", nums[i][j]);
        }
    }
```

The first thing to note here is that we used a trick to assign multiple variables on the same line of code. It's very handy for situations like this.
Next note that, once you understand the syntax, nested loops function basically the same as in Python (with the exception of general for loop differences as laid out previously.) To further drive home this point, here is the same code in Python:

```
nums = [[1,2,3],[4,5,6],[7,8,9]]

for row in nums:
    for num in row:
        print(num)
```

Memory Addresses

Whenever dada is written in C (and really most places when using a computer) that data is being stored in your computer's physical memory, or RAM. Examples of storing data in C include variables, arrays, structs, etc. For this discussion, we'll be talking about variables.

Let's say we have a variable: `int age = 30;`. With the creation of this variable, C has stored its value `30` in physical memory. When we want to access its value, we call the variable name `age`. This is not new - we've been working with variables for awhile now.
Whenever C wants to access the value of `age`, it refers to the specific memory address where the value of `age` is stored. It doesn't call `age`, it calls `00660FF0` (or whatever its assigned address happens to be)

We can print out a variable's physical address like this:
`printf("%p", &age);`

We are u sing a new format specifier here: `%p`. This specifies a pointer in C, allowing us to print the memory address of the `age` variable, specifically its pointer as denoted by the preceding ampersand `&`.

Pointers

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. It is essentially another data type like `int`, `char`, `float`, etc. Where these hold integers, characters, and decimal numbers respectively, pointers are a data type that holds memory addresses. We can also create pointer variables just as the others. However, since it can be pretty tricky for humans to memorize the hex values of physical memory addresses, in C we normally store pointer variables as pointers to other existing variables. In practice, it looks like this:

```
int age = 30;
int* pAge = &age;
```

Let's break down the syntax:

- `int age = 30;` First we define the variable whose pointer we are storing. Nothing new here, but it is important to have this variable pre-defined.
- `int*` - Memory addresses are hex values that contain letters and numbers. This means that they can't be stored in a regular `int` variable, and a character array is the wrong data type. So we sue a special variation of the `int` data type called `int*`, which is specifically for pointers.
- `pAge` - conventionally, pointer variables begin with a lowercase p (for "pointer") followed by the capitalized name of the variable whose pointer they are storing.
- `&age` - as we know, a variable name preceded by an ampersand `&` indicates that variable's memory address rather than its value.

Dereferencing Pointers

To dereference a pointer means to access the specific info being stored at a given variable. Let's set up an example:

```
int age = 30;
int* pAge = &age;
```

As before, `pAge` stores the pointer for `age`. If we wanted to proint out this pointer, we would do so via:
`printf("%p", pAge);`

However, to dereference the `pAge` variable means to access the actual data stored in the memory address of this pointer - in other words, the actual value of `age`. In this case, `30`.
`printf("%d", *pAge);` 0

Two differences to note:
Firstly, `30` is an integer, not a pointer. As such, we replace the `%p` format specifier with `%d`.
Secondly, we use the asterisk `*` to indicate that we are dereferencing this pointer variable.

We can also dereference a pointer directly. It doesn't necessarily need to be a pointer variable.
`printf("%d", *&age);`

This has the same effect as the previous example. It may seem a bit silly in this small example scenario - after all we could just print `printf("%d", age);` - but working with pointers in this way is the backbone of C programming.
You can reference and dereference pointers as many times as you want. The following line of code will still run in context:
`printf("%d", *&*&age);`

Writing Files

C is capable of reading, writing, and appending external files. This is done by creating a pointer to a new location and using various file-related functions to open and manipulate the file. For now, we will focus on writing new files and appending to them.

```
FILE* fpointer = fopen("employees.txt","w");
fclose(fpointer);
```

Let's go over what's happening here:

- `FILE*` - This is similar to declaring the data type in a variable, only `file` isn't a data type in its own right. This all-caps indicator is a special signal to C that we will be working with an external file. The following asterisk `*` indicates that we are working with a pointer, similar to how we used `int*` to declare a pointer variable previously.
- `fpointer` - This is the name of our variable. Often abbreviated to `fptr`.
- `fopen()` - This is the function that does the heavy lifting. Short for "File Open."
- `"employees.txt"` - The name of the file to open. If this were an existing file that we were reading from, we would need to specify a file path if it existed in a different directory from where we ran the script. Otherwise since we are creating a new file, it will get written to this directory. Note that this is in double quotes.

- `"w"` - The mode that we will use to handle this file. "w" is for "write", meaning we will be writing to this file. Since it doesn't exist yet in our filesystem, we will be creating a new file. Other common modes are "r" for reading existing files, and "a" for appending to the end of an existing file. Note that this is also in double quotes.
- `fclose()` - This saves the file and removes it from working memory, allowing the script to be terminated safely and preventing corruption of the file contents. We pass in the `fpointer` variable to specify the variable to our file to be closed.

In order to write to this file, we would use `fprintf()` between `fopen()` and `fclose()`. `fprintf()` is similar to `printf()`, but it writes to our specified file instead of the console.

`fprintf(fpointer, "Employees are people.");`

So in this case, we specify to write to the file located at our `fpointer` variable, and then write the string `"Employees are people."` Note that in the "w" mode, everything in this file gets overwritten if this file already exists or you run this code again and modify the `fprintf` statement. We can add to the end of an existing file rather than overwriting it by using the "a" mode. All of the current text in the file will remain, and the content of your `fprintf()` statement will get added to the end.

Reading Files

To read a file in C, we will combine the file handling techniques we learned above with `fgets()` to print out lines of our file.

```
char line[255];
FILE* fpointer = fopen("Employees.txt","r");

fgets(line, 255, fpointer);
frintf("%s", line);

fclose(fpointer);
```

All of this code should be familiar to you. We begin by defining an empty character array called `line` with a buffer size of 255. We then open our file with the "r" mode, and use `fgets` to copy `fpointer` to `line`. See "Getting User Input" for more info there. By default, it will only copy one line at a time. Then we print our `line` array to read the first line of the file.
As we use this function to read lines of a file, an invisible cursor will increment to the next line in the file. THis means that if we called our line begiinging with `FILE*` twice in a row, it would skip over the first line, and print the second line with `line`.
Using this information, we can print out an entire file by using a while loop:

```c
while(fgets(line,255,fpointer)){
    printf("%s", line);
}
```