Introduction to C
Session 1

Robert Palmere

2021

Topics:

- Data types
- Arrays
- Flow Control (loops)
- Functions

In the previous session, we reviewed introductory level Python programming. In this session, we cover similar topics using the C programming language. The C language is a "mid-level" programming language, and therefore, there are additional requirements placed on the developer to write and deploy software. Unlike Python, which is an interpeted language, C source code is required to be compiled to a binary executable before we can run the program. This session provides exposure to concepts of mid-level programming as well as and overview of C syntax / data structures and how they relate to what we have covered in the corresponding Python session.
This document serves as a guide for its associated Jupyter Notebook.

C Program Structure

Unlike C, Python requires the Python interpreter to be installed and specified either at the start of the file or at the command-line. For example,

```
$ python program.py
```

Since C is a compiled language we only need to install a C linker / compiler (e.g. gcc) to compile our programs into a binary executable. C source files follow a general format to be compatible:

```
< header files >
< function declarations or definitions >

int main(){

return 0;
}

< function definitions if function prototypes placed above >
```

In addition to C being a compiled language, it is also a strongly typed language. In Python we did not have to specify the type of the variable we are defining. The type was automatically dictated by the Python interpreter. However, in C the type of variable we declare must be specified preceeding the declaration of a variable. The type defines the storage allocated for the variable in bytes.

```
printf(``%lu\n'', sizeof(char));          //  1 byte
printf(``%lu\n'', sizeof(bool));          //  1 byte
printf(``%lu\n'', sizeof(int));           //  4 bytes
printf(``%lu\n'', sizeof(float));         //  4 bytes
printf(``%lu\n'', sizeof(double));        //  8 bytes
printf(``%lu\n'', sizeof(size_t));        //  8 bytes
printf(``%lu\n'', sizeof(long double));   // 16 bytes
```

In the above code snippet, we use the standard C library functions *printf()* and *sizeof()* functions to acquire and print the storage capacity of each of several types. Although type specification might seem unnecessary after our introduction to Python, it provides direct control over the amount of memory allocated. By having control over the amount of memory allocated and the lifetime of that memory, we can limit the resources required and improve computational efficiency of the program.

Casting

In our Python session we looked at casting variable from one type to another

using standard Python functions. Although there are functions to do this in C as well, we take a look at an example of casting without this convenience first. In the following example we will cast a character variable (char) to an integer (int):

```
char x = '0';
printf(''%c'', x);

x = x - '0';
printf(''%d'', x);
```

To cast a *char* to an *int* in the above code, we substract ASCII values. Characters are interpreted by the computer as numerical representations. By subtracting the same character from the original value we arrive at 0, but as an integer ('0' is 48 in the ASCII table so 48-48 = 0 as an integer).
We can also use functions such as *atoi()* function to convert a *char* to *int* from the C standard library for convenience.

```
char x = '0';
x = atoi(&x);
printf(''%d'', x);
```

Arrays

In stead of containers that we reviewed in our Python session, C uses arrays with each cell of our array corresponding to a memory address.

```
int array[5];
printf(''%d'', array);
```

The above line produces a warning from our compiler:
This brings us to the topic of **pointers** in C. Although the values of the array we passed to *printf* are of type int, when we passed *array* we passed the memory address being pointed to in the first cell of our array.

```
int a[5];
&a[0]
```

The above code declares an array with space for 5 integers. We use the reference operator, &, which captures the memory address being referenced by a[0].

```
&a
```

Entering &a into the jupyter-notebook displays the same memory address as a[0]. This is because an array points to the first element by default. We can then use the dereference operator to see the value held at this memory address.

```
*&a[0]
```

This is equivalent to:

```
*a
```

Functions

We can pass arguments to a function by value or "by reference" in C. The reason for the quotations around "by reference" will become clear. The following code will show how to declare and define a function in C:

```
void f(int x){
printf(''%d\n'', x);
}
```

In the above function we pass variable x by value. This means that calling *f(1)* will first make a copy of the value within the function scope (indicated by brackets) and use this copy for *printf()*. In the next function, we will redefine our function f to permit passing the argument by reference.

```
void f(int& x){
printf(''%d\n'', x);
}
```

Instead of creating a local copy of x in the function scope, the address of where the value of x is held is passed. The x within the function scope passed to *printf()* is therefore the dereferenced address

```
*&x
```

which is the integer value held at the memory address. However, **this syntax is not available in C** for passing by reference. In fact, passing by reference does not exist in C, but we can emmulate this behavior using pointers as we will see in the next example:

```
void add_another(int* x){
printf(''%p'', x);
(*x)++;
}
```

```
add_another(&v);
printf(''\n%d'', v);
```

Although we cannot pass by reference using *int&*, we can enable the function to accept a pointer and then pass the memory address to be pointed at by the function as shown above. By passing the address of v using the reference operator, the variable x within the function scope now points to the address of v. We can iterate the integer value of v by one by dereferencing the memory address.

Control Flow

We can implement control flow into our program using for / while loops.

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

Unlike Python, we have to explicitly declare the variable we are iterating for each loop. This does not have been within the same statement as shown in the other examples of a for loop.
Additional control is provided when using while loops in C. In the first example, the condition is checked before continue the while loop and in the second the code within the *do* scope is ran first before checking the condition.

```
int x = true;
int counter = 0;

while (x){
     printf("%d", counter);
     counter++;
     if (counter == 10){
         x = false;
     }
}
```

```
counter = 0;
x = true;
do{
     printf("%d", counter);
     counter++;
     if (counter == 10){
         x = false;
```

```
    }
} while (x);
```

With what we have learned about C thus far, we can generate methods with similar functionality of familiar Python containers.

```
int* range(const int v){
int values[v];
for (int i = 0; i < v; i++){
values[i] = i;
}
return values;
}
```

We will notice upon compiling the code of this function that a warning is displayed regarding returning address of stack memory associated with a local variable. This is because the memory allocated for *values* is freed once the function returns. Therefore, the pointer points to an invalid memory location. We would like to return the array after calling *range()*. To do this, we need the lifetime of the array to remain after the function returns. We can do this by declaring the array as static, we can pass the array to the function for the function to modifying without returning new memory, or we can allocated memory on heap to be manually removed using *free()*.

In order to emulate the behavior of the *range()* function of Python, we will dynamically allocate memory for our array within the function scope to be returned as new memory by the function. Passing an existing array for values to modified is also shown in the associated Jupyter notebook.

```
int* range(const int v){
int* values = (int*)malloc(v*sizeof(int));
    for (int i = 0; i < v; i++){
values[i] = i;
}
    return values;
}
```

When we use this function we have to provide the return type as *int** and free the memory being pointed to before we end the program. Otherwise, the memory is never deallocated (memory leak).

Lastly, although classes are not part of C, we can group various data types into a *struct* and access them via the dot operator.

```
struct Pie{
```

```
const char* name = ''Pizza Place'';
int slices = 8;
}

Pie p; // Generate instance of struct Pie called 'p'
printf(''Name: %s\n'', p.name);
printf(''Slices: %d\n'', p.slices);
```

There is a lot more to be learned about C which is best done through practice.