Basic Data Manipulation and Analysis with C/C++

Session 2

Robert Palmere

2021

Topics:

- C/C++ Syntax
- Library Functions for I/O using C/C++
- Emulating NumPy functionality
- Implementation

In the corresponding Python session, we reviewed basic data manipulation and analysis. In this session, we review C/C++ syntax by writing code to read in textual data and emulate NumPy manipulations. Some of the features which make C++ a supserset of C are highlighted. Instead of using a Jupyter notebook, the code is provided as C/C++ source files to be compiled.

File Reading

There are many methods for reading textual data from local files using C/C++. The first code snippet is written in C.

```
#include <stdio.h>
#include <stdlib.h>
#define LENGTH 100

int main(){
   const char fn[] = "Data.txt";
   FILE* fp = fopen(fn, "r");
   if (fp == NULL){
```

```
        return -1;
    }
    char line[LENGTH];
    int line_counter = 0;
    while ( fgets(line, LENGTH, fp) != NULL ){
        line_counter++;
        if (line_counter > 5){
            break;
        }
        printf(''%s'', line);
    }
    fclose(fp);
    return 0;
}
```

If we did not know where to start writing this code, we would lookup the documentation regarding input and output (I/O) methods and what standard header files we can include in our source file to access these functions.

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

In this case, we are including the standard (I/O) and standard library header files which include *fgets()* and *printf()*, respectively. The < > around the header file names specify that the header files are likely to be found in the "normal" directories for these header files designated by the compiler rather than header files within our current working directory.

We then use a preprocessor macro to define "LENGTH" to be the integer 100 for the lifetime of our program. Defining a macro allows the compiler to then substitute the token "LENGTH" for the integer. This comes in handy when we wish to set many array sizes the same and will need the size for associated functions.

In the body of our program we define "fn" as an array of constant chars with the size being dictated by the string length of the file name it is defined as. We then call *fopen()* provided by stdio.h which returns a file pointer as indicated by

```
FILE* fp = ...
```

After checking to see if *fopen()* returned the file pointer "fp", a while loop is implemented to get each line of the input file as long as *fgets()* does not return

NULL.
The program can then be compiled using gcc on the terminal.

```
gcc 1-fgets.c -o $<$program-name$>$
```

Another method would be to use *feof()* in our while loop.

```
#include <stdio.h>
#include <stdbool.h>
...
do {
   ll = fgetc(fp);
   counter++;
   if (feof(fp)){ break; }
   if (counter <= 5){ printf(``%c'', ll); }
} while (true);
fclose(fp);
```

We acquire characters from the input text file using *fgetc()* until the end of file (EOF) is reached. Since we have not specified a char array to place each character from *fgetc()*, we do not form strings for each line. Another loop defining cells of a char array would be required.

A similar implementation using standard functions available in C++:

```
#include <iostream>
#include <fstream>

int main(){
   std::ifstream in(``Data.txt``);
   if (!in){ return -1;}

   std::string l;
   int counter = 0;
   while (std::getline(in, l)){
      counter++;
      if (counter > 5){ break; }
      std::cout << l << std::endl;
   }
}
```

Aside from the differently named header files, function prototypes, and use of namespaces, the solution is much the same as that seen in our C programs above. Instead of a file pointer *FILE**, C++ uses the class *ifstream* located

within the standard (std) namespace. The scope resolution operator, ::, specifies the namespace which the class is being used from. As we will discuss further in the session regarding classes, the we pass the name of the file to the constructor of "in" which is an instance of ifstream. In other words, "in" is defined as a file stream to "Data.txt". The file stream is then provided as a condition for our while loop using *getline()* and a local variable stores the string before printing.

In the next example, we will write a function to read the matrix of values presented in "Data.txt" and print these values to the standard output. The function for reading in the files and generating a matrix of values is shown below. First the dimensions of the data are determined before allocating memory and reading in the values into the return data matrix. Please refer to Session 1 on C programming if you are unsure why we need to allocate memory in this way within the scope of the function definition.

```
float** readlines(const std::string& fn){
   std::ifstream in(fn);
   int rc, cc;

   std::string l;
   std::string d = " ";

   while (std::getline(in, l)){
      rc++;
      if (rc == 1){
         int i = 0;
         for (; i < l.size(); ++i){
            if (l[i] = ' '){
               c++;
            }
         }
      }
   }
   cc = cc + 1;
   in.close();

   float** data = new float*[rc];
   for (int i =0; i < rc; i++){
      data[i] = new float[cc];
   }

   std::fstream ia(fn);
   int cl = 0;
   while (std::getline(ia, l)){
      std::string x_str = l.substr(0, l.find(d));
      std::stirng y_str = l.erase(0, l.find(d) + d.length());
```

```
      data[cl][0] = atof(x_str.c_str());
      data[cl][1] = atof(y_str.c_str());
      cl++;
   }
   return data;
}
```

The C++ solution is not too different. We include <string> and <vector> for convenience.

```
std::vector < std::vector<float> > readlines(const std::string&
    fn){
   ...
   std::vector < std::vector<float> > data;
   while (std::getline(in, l)){
      std::vector<float> row;
      ...
      row.push_back(atof(x_str.c_str()))
      row.push_back(atof(y_str.c_str()))
      data.push_back(row);
      row.clear();
   }
   return data;
}
```

<vector> is convenient because it houses its own memory management (i.e. vector handles *free()* and *malloc()* for us). After generating a vector containing the x and y for a row, it is appended to another vector, "data", which holds all rows. We are able to then index the vector as we might for a 2D NumPy array for printing.

Emulating NumPy

Now that we understand some ways to read data from a text file using C/C++, we will look at how to generate a structure which can handle NumPy-style computations. We will do this using C (6-numpy.c) and C++ (7-numpy.cpp).

```
typedef struct Numpy{
   int rows, cols;
   float** data;
} np;
```

```
struct Numpy* array(int rows, int cols){
   struct Numpy* mat = (np*)malloc(sizeof(np));
   mat->rows = rows;
   mat->cols = cols;
   float** data = (float**)malloc(sizeof(float*) * rows);
   for (int i = 0; i < rows; i++){
      data[i] = (float*)calloc(cols, sizeof(float));
   }
   mat->data = data;
   return mat;
}
...
```

Depending on your familiarity with C, this may look complicated. However, it is very close to the functions we have already written, but reorganized. Starting from the top of the code snippet, we define a struct called "Numpy" as was introduced in the first session. This struct contains data types we will need to generate a zero-filled matrix. Additionally, we used the *typedef* keyword to generate a global alias for the struct as "np". We could also separate these statements.

```
struct Numpy{ ... };
typedef struct Numpy np;
...
```

By setting the typedef "np" as a Numpy struct, we can use "np" as a type identifier. For example when passing to a function:

```
void function(np array){...}
```

The function expects objects of type "np" which is struct Numpy. Without typedef we would write:

```
void function(struct Numpy array){...}
```

After defining the struct, we move on to writing a function which returns a pointer to a Numpy struct. Within the function definition we dynamically allocate memory for a Numpy struct pointer "mat". Since "mat" points to a Numpy struct, we can use the arrow operator to access its variables and define them using integers passed by value. Zero-initialized memory is allocated for the 2d array which then defines the "data" attribute of the struct. Finally we define a function for printing of our new type:

```
void print(np* a){
   for (int i = 0; i < a->rows; i++){
      for (int j = 0; j < a->cols; j++){
         printf(``%f'', a->data[i][j]);
      }
      printf(''\n'');
   }
}
...
```

Instead of a C struct, we will use a class in C++.

By default classes in C++ are private which means their attributes would not be accesible outside of the class definition unless permissions are given. C structs cannot hold function prototypes, but in a C++ class we can include the prototype for generating an array as part of the Numpy class. The corresponding definition of this function declaration is performed outside using the scope resolution operator.

```
Numpy Numpy::array(int nrows, int ncols){ ... }
```

Rather than using *malloc()* to instantiate the class, we only have to state the class name and the name tag we assign the instance.

```
Numpy obj;
```

Also, the *new* keyword is used instead of calloc(). If we want to put the code we have written together into one program to read and analyze data, we only have to add the class and associated functions to a header file as shown in 10-analysis.c and 10-analysis.h.