Command Line Arguments

Programs take input, either as command line arguments, files, or standard input. C programs do the exact same. Let's write a quick file to read through some command line arguments:

`arguments.c`

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

int main(int argc, char** argv){
  printf("Hello you supplied the arguments:\n");

  int i;
  for(i = 0; i < argc; i++){
    printf("Argument %d is %s\n", i, argv[i]);
  }

  printf("Thank you for viewing your arguments!");
  return EXIT_SUCCESS;
}
```

What's happening above is that we have an integer `i` declared at line 7, which we then use as an index for our `for` loop. Inside, our loop, we're setting our stopping condition as `argc`, or the count of the arguments. Afterward, we're then printing the associated index of the argument along with the string of the argument itself:

```
gcc arguments.c -o args
./args
```

> Hello you supplied the arguments:
>
> Argument 0 is ./args
>
> Thank you for viewing your arguments!

We passed in no arguments, however, we still saw `Argument 0 is ./args`. Does this remind you of anything? Possibly something similar to *positional parameters*? Let's try it with a few more arguments (all whitespace delimited):

```
./args they call them fingers, but how do they fing?
```

> Hello you supplied the arguments:

> Argument 0 is ./args
>
> Argument 1 is they
>
> Argument 2 is call
>
> Argument 3 is them
>
> Argument 4 is fingers,
>
> Argument 5 is but
>
> Argument 6 is how
>
> Argument 7 is do
>
> Argument 8 is they
>
> Argument 9 is fing?
>
> Thank you for viewing your arguments!

Regardless of how many arguments we put in, so long as we're using `argc` as a stopping condition, we can always be sure that we'll be able to grab them!

Standard Input / Output

Because `c` can work with many other programs via the command line, it also needs to be able to take in standard input and write to the standard out (as well as the standard error). To do so, we'll use the commands `getchar` and `putchar`. To make things more interesting, let's create a basic cipher! :

`standardInNOut.c`

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

int main(int argc, char** argv){
  int isDecode = argc == 2 ? strcmp(argv[1], "decode") == 0 : 0;
  int encoder = 2;

  if (isDecode){
    encoder = -2;
  }

  int characterReadFromStdIn = getchar();


  do {
```

```
16      putchar(characterReadFromStdIn+encoder);
17    } while (( characterReadFromStdIn = getchar()) != EOF);
18
19    return EXIT_SUCCESS;
20  }
```

What have we done above? Let's walk through our code step by step.

First, at line 5, we're using a ternary. This is doing in one line what an entire if/else would do in 3. First, we're determining whether or not `argc` has 2 arguments (i.e. one for the program itself, and one for a user specified argument). If there are two arguments, then we compare `argv[1]` and `"decode"` with the function `strcmp`, which returns 0 if they're equal, otherwise it returns a numeral favoring one of the two arguments (negative if the ascii values of the first argument's argument unmatched character is less than the second, and positive if it's greater). Unfortunately for us, we cannot compare two strings in `c` with `==`. We then compare the result of `strcmp` with `0`, to see if they're equal. By doing so, we'll return a boolean value to isDecode. If, however, argc is not equal to two, we just return `0`, or `false`.

At line 6, we pull the first character from the standard input with `getchar()` and store it in `characterReadFromStdIn`. Notice that it's an integer? Recall from previous courses, all characters have integer values!

Lines 7 through 11 determine wither or not we're going to be adding 2 or subtracting 2 from our standard input! This way, we can add or subtract a number from that value to either encode or decode our text based on the value of the `isDecode` boolean!

Finally, inside of our `do..while()` loop, we use `putchar` to output `characterReadFromStdIn + encoder`, and depending on what our input was (as well as our encoder's value), we'll either see some nice, decoded text, or possibly an encoded jumble of letters! We then read the next character from standard in with `getchar()`. The loop will only stop once `characterReadFromStdIn == EOF`, where `EOF` is the end of the file. Keeping the reading of getchar inside the while loop is a somewhat typical convention, however, if it looks a little too confusing, you can just as easily rip it out:

```
...
  do {
    putchar(characterReadFromStdIn+encoder);
    characterReadFromStdIn = getchar();
  } while ( characterReadFromStdIn != EOF);
...
```

However you construct your program is up to you. I would always push erring on the side of being as verbose as you can. Being clever can save you some lines in your code, however, when you go back to debug your code, often times, it's that very cleverness that comes back to bite you.

Let's see what this code does now:

```
gcc standardInNOut.c -o inOutCipher
echo Shh! It\'s a secret! | ./inOutCipher
```

> Ujj#"Kv)u"c"ugetgv#

We just retrieved some input from standard in, read it character by character, and encoded it to our very own cipher (granted, it's not that hard to crack - each letter is only 2 more than the letter we want). Let's decode our output (be sure to escape any special characters that the shell will want to interpret), but be sure to include the `decode` argument, otherwise, we'll just further encode our message!!

```
echo Ujj\#\"Kv\)u\"c\"ugetgv\# | ./inOutCipher decode
```

> Shh! It's a secret!

Naturally, you won't always be redirecting standard in to a `c` program with a pipe, it's incredibly likely that you'll want to redirect entire files to act as input! Let's use a historical example for our cipher:

`maryQOSLetters.txt`

```
Myself with ten gentlemen and a hundred of our followers will undertake the delivery of your royal person from the hands of your enemies. For the dispatch of the usurper, from the obedience of whom we are by the excommunication of her made free, there be six noble gentlemen, all my private friends, who for the zeal they bear to the Catholic cause and your Majesty's service will undertake that tragical execution.
```

The above is some text from a letter to Mary Queen of Scots offering the assassination of Queen Elizabeth. The correspondence of the Babington Plot was encoded with a cipher that replaced letters and simple words with special characters. However, even in the 16th century, frequency analysis was a thing, and the cipher was broken (using this cipher seriously is definitely not recommended). Regardless, let's encode the above text into an unreadable jumble, and redirect the standard output to a new file:

```
./inOutCipher < maryQOSLetters.txt >> encodedLetter.txt
cat encodedLetter.txt
```

> O{ugnh"ykvj"vgp"igpvngogp"cpf"c"jwpftgf"qh"qwt"hqnnqygtu"yknn"wpfgtvcmg"vjg"fgnkxgt{"qh"{qwt"tq{cn"rgtuqp"htqo"vjg"jcpfu"qh"{qwt"gpgokgu0"Hqt"vjg"fkurcvej"qh"vjg"wuwtrgt."htqo"vjg"qdgfkgpeg"qh"yjqo"yg"ctg"d{"vjg"geqoowpkecvkqp"qh"jgt"ocfg"htgg."vjgtg"dg"ukz"pqdng"igpvngogp."cnn"o{"rtkxcvg"htkgpfu."yjq"hqt"vjg"|gcn"vjg{"dgct"vq"vjg"Ecvjqnke"ecwug"cpf"

> {qwt"Oclguv{}u"ugtxkeg"yknn"wpfgtvcmg"vjcv"vtcikecn"gzgewvkqp0

Now, let's unencode it!

```
./inOutCipher decode < encodedLetter.txt
```

> Myself with ten gentlemen and a hundred of our followers will undertake the delivery of your royal person from the hands of your enemies. For the dispatch of the usurper, from the obedience of whom we are by the excommunication of her made free, there be six noble gentlemen, all my private friends, who for the zeal they bear to the Catholic cause and your Majesty's service will undertake that tragical execution.

Writing To Files

Redirecting data from standard out is great, but piping input and output to and from files can get messy. There isn't much of a paper trail if you're moving things via stdin and std out. It's just data streams. If you wish to be more consistent and write your data to a file, thenyou'll want to use the `fopen` command. Files can be opened with 6 possible options:

Mode	Description
`r`	Opens a file as read only. File must exist, or an error is thrown.
`w`	Creates a file for writing. If the file already exists, the file is clobbered, and a brand new file is created in its place.
`a`	Opens a file for appending. If the file does not already exist, it is created.
`r+`	Opens a file for both reading and writing. File must exist, or an error is thrown.
`w+`	Creates an empty file for both reading and writing.
`a+`	Opens a file for reading and appending.

Reading From Files

Often times you'll need to write programs that read data from a file, be it a log file, csv, or any other file, assuming you have permissions. To read from a file, we'll need to do something quite similar to what we did with reading from the standard input, but instead of using `getChar()`, we'll be using file pointers to move from character to character, and we'll also need to open and close the files with `fopen` and `fclose`:

`readFile.c`

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

int main(int argc, char** argv){
  if (argc < 2) {
    printf("You need to provide a file, or else this whole operation will fail!\n");
    return EXIT_FAILURE;
  }

  FILE * inputFilePointer;
  int inputFileCharacter;

  inputFilePointer = fopen(argv[1], "r");

  if (inputFilePointer == NULL) {
    printf("Bad file input! %s cannot be opened", argv[1]);

    return EXIT_FAILURE;
  }

  inputFileCharacter = fgetc(inputFilePointer);
  while(inputFileCharacter != EOF) {
    printf("%c", inputFileCharacter);

    inputFileCharacter = fgetc(inputFilePointer);
  }

  fclose(inputFilePointer);

  return EXIT_SUCCESS
}
```

In the above code, what we're doing is first and foremost, checking to see if a user even passed in a file. If they did, then great, but if not, then we need to exit the program immediately. You can't read a file you haven't been provided. We then declare an file pointer. What this is going to do is let us look at each individual byte of a file! And just like reading from the standard input, we need an integer to take in the integer value of each character!

We then want to open the file with our `fopen` command. `argv[1]` is the file name that we passed in, with `r` as the mode of file access. We then want to check to see if the pointer is null. If so, that means there was some error in reading the file, but in all likelihood, the filename is just in correct!

Finally, just like with the reading in of standard input, we want to create a while loop where we can read over each character, and then print it to the standard output with `printf`! When we finally reach the `EOF`, the conditional clause in the while loop will fail, and we then kick out to finally close our file!

Writing To Files

Curiously, writing to a file isn't too much different than reading a file, however, it does look different enough in that we have to use a new command, `fprintf`. The `fprintf` is just like the `printf` except we're directing the program where to print the data, whereas `printf` just prints the data entirely to standard out! Let's write a quick program to read in user input and print it to a file! To read in user input, we're going to use a familiar command, `getchar`:

`printToFile.c`

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

int main(int argc, char** argv){
  if (argc < 2) {
    printf("You didn't provide a file name! All output will be printed to \"output.txt\"!\n");
  }
  char* outputFile = argc == 2 ? argv[1] : "output.txt";

  FILE * outputFilePointer;
  int outputFileCharacter;

  outputFilePointer = fopen(outputFile, "a");

  if (outputFilePointer == NULL) {
    printf("Bad file! %s cannot be opened", outputFile);

    return EXIT_FAILURE;
  }

  printf("Please enter something to append it to the file %s. When you're finished, type ~", outputFile);

  outputFileCharacter = getchar();
  while(outputFileCharacter != '~') {

    fprintf(outputFilePointer, "%c", outputFileCharacter);

```

```
28      outputFileCharacter = getchar();
29    }
30
31    fclose(outputFilePointer);
32
33  }
```

In the above file, we're again ensuring that if the user doesn't provide a file, we're doing something. This time, since the program isn't 100% reliant on a user providing a file to us, we can just create a default file for them. At line 8 is where we use a ternary to determine what the output file's name ought to be.

Again, we create an output file pointer, and an output file character. We then open the new file with our `fopen` command, and we're using the `a` for append. This way, if we want to open the program again, we don't wind up overwriting our actual file, we just append the new data to the end! We also want to check if we actually opened the file, though, just in case. `c` is very powerful, however, it's also very delicate.

Finally, we read in individual characters from standard in with `getchar`, and then immediately write those characters to a file, just so long as that character is not a tilde. One thing to note, we're writing each character to the file with `fprintf(outputFilePointer, "%c", outputFileCharacter);`. We use the file pointer to tell `fprintf` where to print the data, and then we tell what to print. Since we're only printing a single character before we get a new one, we're only using the `%c`, but this could literally be anything you so choose! Try changing it and seeing what happens!

Reading From And Writing To Files

We've learned how to quickly read from a file, and we've seen how to write to a file. Let's combine both of these to adapt the cipher from before to read data from a file as a parameter, and then write to a file as well!

`inOutCipherOfFiles.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char** argv){
5    if (argc == 1 || argc != 4) {
6      printf("Bad input! The program's input requires: ./program en/decode
   fileInput fileOutput");
7
8      return EXIT_FAILURE;
9    }
10
11   int isDecode = strcmp(argv[1], "decode") == 0;
12   FILE* inputFilePointer;
```

```
  FILE* outputFilePointer;

  char* fileName = argv[2];
  char* outputFileName = argv[3];

  char inputCharacter;
  char outputCharacter;

  int encoder = 2;

  if (isDecode){
    encoder = -2;
  }

  inputFilePointer = fopen(fileName, "r");
  outputFilePointer = fopen(outputFileName, "w");

  if (inputFilePointer == NULL) {
    printf("Null File Pointers!");

    return EXIT_FAILURE;
  }

  if (outputFilePointer == NULL) {
    printf("Output file error!");

    fclose(inputFilePointer);
    return EXIT_FAILURE;
  }

  do {
    inputCharacter = fgetc(inputFilePointer);
    if(inputCharacter == EOF) break;

    outputCharacter = inputCharacter + encoder;
    fprintf(outputFilePointer, "%c", outputCharacter);
  } while (inputCharacter != EOF);

  fclose(inputFilePointer);
  fclose(outputFilePointer);

  return EXIT_SUCCESS;
}

```

In the above code, we have two separate file pointers, input and output. We need both of these to open two separate file streams because one's reading from a certain file, and another's writing to a completely different file! We also have input characters and output characters! This is because we want to make our code as explicit as possible as to what's happening! Notice that we're doing exactly as we did before with reading and writing to files with the `fgetc` and the `fprintf`, but this time, we're adding our `encoder` value to it! How neat is that!

Get Options with `getopt`

While this is not in the book, this is crucial information to know. Attempting to enforce how users input data based on positional input parameters can be difficult. People will forget and input different parameters in different places and ultimately cause a mess. To get around this, we can use `getopt` to have our `c` program take flagged inputs! Let's change our final program to not use positional parameters for our files, but instead use `getopt`:

`getoptCipher.c`

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

void help() {
  printf("The options for this program are: \n");
  printf("-i inputFileName (required)\n");
  printf("-o outputFileName\n");
  printf("-e encode the file with a provided shifter\n");
  printf("-d decode the file with a provided shifter\n");
}

int main(int argc, char** argv){
  char* fileName = NULL;
  char* outputFileName = "output.txt";
  int encoder = 0;

  int option;
  while ( (option = getopt(argc, argv, "hi:o:e:d:")) != -1) {
    switch(option) {
      case 'h':
        help();
        break;
      case 'i':
        fileName = optarg;
        break;
      case 'o':
```

```
        outputFileName = optarg;
        break;
      case 'e':
        encoder = atoi(optarg);
        break;
      case 'd':
        encoder = -(atoi(optarg));
        break;
      case '?':
         if (optopt == 'c')
           fprintf (stderr, "Option -%c requires an argument.\n", optopt);
         else if (isprint (optopt))
           fprintf (stderr, "Unknown option `-%c'.\n", optopt);
         else
           fprintf (stderr,
                    "Unknown option character `\\x%x'.\n",
                    optopt);
         return 1;
      default:
        help();
        break;
    }
  }

  if (fileName == NULL) {
    printf("Input file name is required!\n");
    return EXIT_FAILURE;
  }

  if (encoder == 0) {
    printf("encoding is 0, this will just print the unciphered text to
another file %s\n", outputFileName);
  }


  printf("Input File: %s\n", fileName);
  printf("Output File: %s\n", outputFileName);
  printf("Encoding: %d\n", encoder);

  FILE* inputFilePointer;
  FILE* outputFilePointer;

  char inputCharacter;
  char outputCharacter;

  inputFilePointer = fopen(fileName, "r");
```

```
  outputFilePointer = fopen(outputFileName, "w");

  if (inputFilePointer == NULL) {
    printf("Null File Pointers!");

    return EXIT_FAILURE;
  }

  if (outputFilePointer == NULL) {
    printf("Output file error!");

    fclose(inputFilePointer);
    return EXIT_FAILURE;
  }

  do {
    inputCharacter = fgetc(inputFilePointer);
    if(inputCharacter == EOF) break;

    outputCharacter = inputCharacter + encoder;
    fprintf(outputFilePointer, "%c", outputCharacter);
  } while (inputCharacter != EOF);

  fclose(inputFilePointer);
  fclose(outputFilePointer);

  return EXIT_SUCCESS;
}


```

First and foremost we want to call `getopt` with a loop since we'll be iterating over options provided by the user.

The command `getopt(argc, argv, "hi:o:e:d:")` tells us that there are 5 potential options for our program! Option `h` is for help. Since it has no following `:` , that means that it takes no arguments! Options `i` and `o` take in input files and output file names respectively, while options `e` and `d` update the encoder and decoder, where the user supplies a number key!