CS4023 Week04 Lab Exercise

Lab Objective: We commented at the end of last week's lab that our `copyFile` program was inflexible in that the "from" and "to" files were hard-coded into the source code. We will now remedy this. We will also use the lab to look at the software development process of `C` or C`++` .

Here's a quick summary of the tasks:

❶ Create a directory (folder) for this week's work

❷ Copy the `C` source code `copyFile.c` from last time into your directory (this week's)

❸ Write a more useful version of the `copyFile` program so that it can accept "command line arguments"

❹ Modify this program now so that it can perform a very weak type of encryption. In doing this you will learn about

- the way characters (`char`s) are represented internally in `C`
- `strcmp`, how strings (of characters) are compared in `C`
- breaking your program into multiple files which is vital for maintaining large programming projects

❺ Once the program is working according to the specifications given below you should submit it for marking using the handin mechanism

In Detail

❶ You can create this week's lab directory with

```
mkdir ~/cs4023/labs/week04
```

This assumes that you already had created the directory `~/cs4023/labs/` from last time. A little shortcut that instructs `mkdir` to make any necessary "parent" directories along the way is

```
mkdir -p ~/cs4023/labs/week04
```

Now change your working directory to this since all of our compiling, etc. will be done in this directory. See last week for how to change working directory.

❷ To start off we need our `copyFile.c` source file from last time. If you did not succeed in getting last week's lab completed you can use the `readFile.c` code instead. So you will issue one of the following two `cp` commands:

```
cp ../week04/copyFile.c .
```

or

```
cp ~cs4023/labs/week04/readFile.c copyFile.c
```

Note the difference in the last arguments of these two commands. In the second we are changing the name so we have to give that as the destination. In the former case we are keeping the same name so we only need to give the destination directory, which is "here", the current "working directory", or in shell-speak '`.`' (dot). (In that command we also used `../` which is shell-speak for "look one level of directory up from where we are now".)

❸ When a `C` program is entered as a process on the OS's ready queue the first point of execution is the function called `main()`. In last week's lab this function did not take any arguments. Now we are going to change this so that arguments specified on the command line can be passed in to our program. (This is akin to calling a function with different parameters.) To do this you will change

```
int main()
```

to

```
int main(int argc, char* argv[])
```

The first argument, `argc` indicates how many arguments our program was called with – including the program's name itself. This might seem a bit odd but there are good reasons for it. The second argument to `main`, is a block of memory that allows us to access the command-line arguments of our program. It is called `argv` where the 'v' at the end is to denote a vector (or sequential block) of data. We can access a particular argument that our program was called with by means of `argv[i]`. Remember that `C` programmers waste nothing so they start counting from 0 instead of 1. In `argv[0]` we can gain access to the program name.[1] So if we gave a command from the shell prompt:

```
copyFile here there
```

then the character string "copyFile" is associated with `argv[0]`. Further, if we gave this command from the shell prompt:

```
../week04/copyFile here there
```

then the string of characters associated with `argv[0]` would be "../week04/copyFile".

[1]For anybody who has ever asked themselves questions like "Who am I?" or "How did I get here?" (but not, like Fr. Jack Hackett, "Are those my hands?") this will strike a chord, for it provides a mechanism for a program to answer the former question.

Tip o' the mornin' (t'you)

`argc` tells us how many arguments; `argv` gives a way of accessing each of those arguments. Remember that the name of the program running counts as an argument and can be accessed in `argv[0]`.

In order to open the first file for reading you will need to modify your first `fopen()` system call so that it uses the first argument given to on the command line. (We call the command itself the zeroth argument.)

The first part of today's lab is to write this `copyFile` program. Once you think you have it working you must test it with the `diff` command. Assuming that you had already created a file `here`, if you gave the command:

```
copyFile here there
```

then you would ensure that the source and destination files are identical with:

```
diff here there
```

`diff` operates silently so if the files are identical you will see *no* output from `diff`.

❹ The second task for today is to modify this program so that, if the `-f` flag is given as the first command-line argument a very simple form of encryption called *flipping* is performed. If a file is copied in *flip* format then all numerical characters and upper and lower case letters should be replaced in the destination by their *flip*.

So what is the flip of a character? If we take the lower-case letter 'e', the fifth letter of the alphabet as an example, then when it is *flipped* it should be replaced by the fifth letter of the alphabet from the *end* of the lower-case alphabet. Likewise, for an upper-case letter or a number. For your reference I have given you a "before and after" file. They can be found in `~cs4023/labs/week04` and were generated as follows:

```
copyFile -f sample sample.flip
```

and are repeated here.

The contents of `sample`:

```
abcdefghijklmnopqrstuvwxyz;
ABCDEFGHIJKLMNOPQRSTUVWXYZ,
0123456789.
```

The contents of `sample.flip`:

```
zyxwvutsrqponmlkjihgfedcba;
ZYXWVUTSRQPONMLKJIHGFEDCBA,
9876543210.
```

Handy reference chart for *flip*

Notice how the punctuation characters at the end of each line made it through safely (without change).[2]

So how – finally! – do we implement in code the flip? What you *could* do would be to have a giant, mega, 62-way[3] "if" statement along the lines of:

```
if (c == 'a')
   putc('z', destfile);
elsif (c == 'b')
   putc('y', destfile);
 :
// ouch!
```

The two clinching reasons for not doing things this way are that it is very error-prone and not very maintainable (for making changes later). Most of all though our *professional pride* should drive us to search for something more elegant.

We can do something a lot more elegant and more compact once we realise how C deals with characters. In all computer languages everything must be represented ultimately as a series of 0s and 1s. In C, the letters 'a' to 'z' are encoded *consecutively*. This means that whatever 8-bit pattern is used to represent the letter 'd', *when viewed as an integer*, the letter 'e' can be viewed as an integer one larger. As we will see below it is *irrelevant* what the value of the integer corresponding to any of them are: **all we need to know is that they occur consecutively**. The previous was put in bold so that you would remember it.

Likewise the letters 'A' to 'Z' occur consecutively but strangely they are not consecutive to their lower-case counterparts. The digits '0' to '9' are also consecutive but, again, they are isolated form the other two "blocks" by other characters – punctuation, I think.

What this means is that we could write a function to translate any lower-case character to its corresponding upper-case as follows:

```
char toUpper(char c)
{
  if ('a' <= c && c <= 'z') // a valid lowercase
    return c-'a' + 'A';     // c-'a' is offset

  return c;     // wasn't LC so leave alone
}
```

toUpper()

You will need to write a function `flipChar()` that takes a single character as argument and does something like the `toUpper()` function above. Because there are three different cases to consider (lower-case, upper-case and digits) you will need to distinguish between the three cases and act accordingly.

[2]In order to keep things simple, in this assignment we will treat anything outside the ranges 0-9, a-z and A-Z as being non-alphabetic and let them pass through unchanged.

[3]62: 26 lowercase + 26 uppercase + 10 digits.

At a very high level the program will operate very similarly to your `copyFile` program in the first part of the lab. The difference here is that each character that is read is either flipped, or not, depending on the first command-line argument. If it is `-f` then flipping is in effect – and the origin and destination file command-line arguments are in `argv[2]` and `argv[3]`. A nice way to achieve all this is to

- test firstly at the beginning of your program if flipping is in effect and storing the result in an integer that will be either 0 (false) or 1 (true):

  ```
  int flipping = (strcmp(argv[1], "-f") == 0);
  ```

  We compare the string pointed to by `argv[1]` against the string "-f" using the `C` function `strcmp()`. If they are *equal* then `strcmp()` returns 0; in any other case what is returned is the subtraction of the characters where they first differ, so you can tell which string is larger.

  Before you can call a function in `C` you must give the compiler fair warning. To "declare" the `strcmp()` function to the compiler all you have to do is to have it read the declarations in the `string.h` file. This is done by putting at the top of your program:

  ```
  #include <string.h>
  ```

- Now, when you want to write a character to the destination within your main while loop you ask if `flipping` is in effect and you execute the following

```
if (flipping)   // same as: if (flipping == 1)
  putc(flipChar(c), destfile);
else
  putc(c, destfile);
```

There is one final task to this week's lab. While it may appear gratuitous or contrived in this small case it is important enough to introduce this concept early in the series of lab exercises you will do. This is the idea of decomposing programs into multiple source files. All serious software is made up of code coming from multiple files and/or libraries so it is a good habit to get into now.

In our case here you should put your `flipChar()` function and any other functions that you might write for the assignment in a `C` file called `utils.c`. This file will be *compiled* separately into object code, `utils.o`, and then linked together with a separately compiled `copyFile.o` object file as shown below. Whenever you make a change to your `flipChar()` function you will need to recompile `utils.c` with:

```
gcc -c utils.c
```

and then relink the two object files with:

```
gcc copyFile.o utils.o -o copyFile
```

Similarly, changing any of the source code of `copyFile.c` necessitates a recompilation of this with:

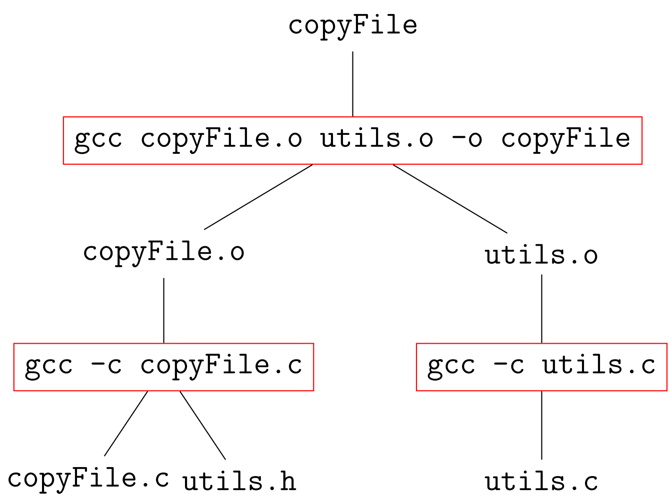
```
gcc -c copyFile.c
```

and re-linking of the two object files.

You will call the `flipChar()` function from `copyFile.c` so when this is being compiled the compiler had better be pre-warned of its existence. You do this by putting all the `utils`-related declarations in a file called `utils.h` and giving the compiler knowledge of its contents by putting

```
#include "utils.h"
```

just below all of the other `#include` directives that you have in your `copyFile.c` file. I have written this for you and it is available with all the others.



❺ This is the second lab that you will be assessed on. To be assessed you will need to have the program written and submitted by the deadline below.

The command for submitting this lab via the `handin` mechanism is:

```
handin -m cs4023 -p w04
```

Labs that are to be handed in are open for handin-gin at 09.00 on Tuesday of the week it is *assigned* and, in order to get full marks, are due by 16.00 on Tuesday of the *following* week[4]; a lateness penalty applies to submissions made until 18.00, Friday after that. This gives you at least one week to work on each lab that is to be assessed and handin without penalty. The lab sheet will be available for reading prior to the first lab so that you can read it beforehand and come to the lab with questions.

Subject to last-minute changes, the planned schedule of lab assignments due for handin-g in are:

[4] There are a couple of exceptions where you will be given 2 weeks to complete the lab.

Lab. Week	Assessed	DueDate (16.00, Tue)
Week01	✗	
Week02	✗	
Week03	✓	Week04
Week04	✓	Week06
Week05	✗	
Week06	✓	Week07
Week07	✗	
Week08	✓	Week09
Week09	✓	Week10
Week10	✓	Week11
Week11	✗	