# CP Lab 8: Recursion and some File work

## Instructions

The purpose of this Lab is to help you get more programming experience with C, and introduce you to *recursive programming* and *file structures*. We will be building on what you have done in the previous lab and in your classes.

All Labs for CP will take place on the DICE machines in AT5.05 (Thursday and Tuesday) or AT6.06 (Monday). For working alone, you may use any of the DICE labs in AT, as long as they are not booked out by a class. You have 24-hour access to AT levels 3,4,5: if your card/pin does not allow you in, use the ITO contact form on our course webpage to contact the ITO.

## Aims

- Get you extra experience with the DICE system, and its use for C programming.
- Write some 'recursive programs'. As well as the concept of *recursion*, you will also use the basic concepts of assignment-to-variables, input, output and the `if`-statement.
- Give you experience of reading data from *files* and writing data into *files*.

The techniques studied in this lab are widely used in programming. However, they will not be assessed in the exam, so if you have not completed earlier labs, you will be better off to continue working on those, and perhaps come to this lab during the revision period.

## Prerequisites

You should complete most of the tasks in the previous labs before attempting this one.

### Getting started

- Log into one of the DICE machines.
- Bring up a *terminal window*, move to your home directory, and create a new subdirectory called `lab8`.



## Stage A Towers of Hanoi

This puzzle was invented in 1883 by the French mathematician Édouard Lucas, complete with a mythic background story (of which there is no earlier trace). So his story goes: in ancient Vietnam, a group of monks were told by a prophecy to move gold discs from one stack onto another. The initial stack had 64 discs threaded onto one stack, arranged in order of decreasing size, with the largest disc on the *bottom* and the smallest disc at the *top*. The monks were challenged to move all discs from the 'initial' stack (stack 1) onto a 'final' stack (stack 3), using a 'middle' stack (stack 2) to help, and following these constraints:

(i) Only one disc can be moved in one step;

(ii) A larger disc can never be placed above a smaller disc.

When they finish their task, the world will end.

How can they move the discs?

> The *Towers of Hanoi* is the problem of determining a series of steps which obey rules (i) and (ii), and which when carried out from the initial configuration (with the discs arranged from largest up to smallest on stack 1, and the other stacks empty), will result in an end configuration where the discs are arranged on stack 3 from largest up to smallest.

It is well known that a nice way to solve the *Towers of Hanoi* problem is via *recursion*.

The key is to relate the Towers of Hanoi problem for `n` discs to the version where there are only `n-1` discs. This relationship can be described as follows:

(a) Move `n-1` discs from the initial stack to the original middle stack, using the original final stack as the temporary stack (obeying rules (i) and (ii)).

(b) Next, move the last disc (the biggest) from the (original) initial stack to the (original) final stack.

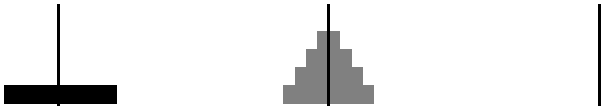
(c) Finally, move the `n-1` discs now on the original middle stack to the original final stack, this time using the original initial stack as the temporary stack. (obeying rules (i) and (ii)).

Note that when we just have *one disc* to move (`n` = 1), that we do not need to use recursion – we can just move it directly from initial stack to final stack. This is the 'base case' for Towers of Hanoi.

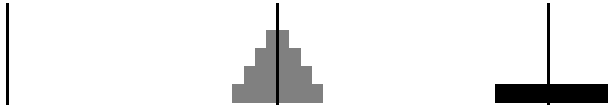
For example, suppose we have five discs:

We reduce it to a 4-disc problem by fixing the largest disc:

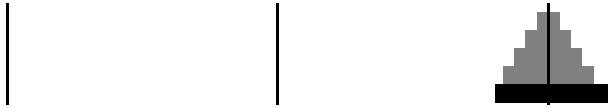
Now we solve the 4-disc problem, using stack 3 (the original final stack) as the middle stack, to move the discs on to stack 2 (the original middle stack):



Now in a single step we move the biggest disc to the end:



and now we again fix the biggest disc, and solve the 4-disc problem again to move the smaller discs to the end, using the original initial stack as the temporary middle stack:



Our goal is to write out a series of moves of the form 'From *i* to *j*', where *i* and *j* are either 1, 2, or 3, which will describe the solution to the Towers of Hanoi problem for the given number of discs `n`.

Your task is to write a program called *hanoi.c* to write out this series of moves.

Your program should include a recursive function (*recursion* can only be achieved by using functions) called `towers` taking four parameters: `discs` (the number of discs), `start` (the index of the 'initial stack'), `end` (the index of the 'final stack') and `temp` (the index of the temporary stack). The `towers` function should be based on (a), (b), (c) above. The function prototype for `towers` should be as follows:

```
void towers(int discs, int start, int end, int temp)
```

We can implement a solution by taking the series of steps for (a) first; then after that adding the step 'From `start` to `end`' (for step (b)); and then finally adding all the steps for (c). Note that (a) and (c) are *recursive calls* to (slightly) smaller instances of the Towers of Hanoi problem (with some rearrangement of stack roles). *Also note* that in applying (a), (b), (c), in our `towers` implementation, that we must work in terms of `start`, `end` and `temp`, rather than 1, 2, 3, because we will need to have the solution even with rearrangements of stack roles.

Your main routine should then read `n` rom the user, and call
`towers(n,1,3,2)`
to show the sequence of moves required to move `n` discs from stack 1 to stack 3.

After you have written the program, please compile it using

```
gcc -Wall hanoi.c↵
```



### Your program `hanoi.c`

☐ Once you have written your program, need to test *correctness*.

For example, to move a stack of 3 discs from stack 1 to stack 3, the following should be output:

```
From 1 to 3
From 1 to 2
From 3 to 2
From 1 to 3
From 2 to 1
From 2 to 3
From 1 to 3
```

☐ Submit your program with
`submit cp lab8 hanoi.c`

## Stage B Accounts

Now you are going to create a sequential access file that might be used in an accounts system to help keep track of the amounts owed from or to a company's clients. Each client is represented in the system by a record consisting of an account number (assumed to be an `int`), the client's name (assumed to be a string of at most 30 `char`) and the client's balance (assumed to be a `double`). The information on a client will be added to a file `clients.txt`, one line per client.

### Adding clients

Your first mission is to write a short program called `addentry.c` which repeatedly asks the user for details of another client, and then saves those details as a new line in the records file `clients.txt`.

- The main resource you need is the slides for Lecture 20 on 'Files in C'.
- It is safe to assume that you should have the following three variables in your program (to hold the details of the current client being read from the user):

  ```
  int account;
  char name[30];
  double balance;
  ```

- You will probably also want to have a `char` variable to hold a `'y'/'n'` response from the user (about whether to continue to input another client).
- The first task of your program will be to define a *stream* which you can use to *append* data to `clients.txt`.

  You will use the command `fopen` to do this - see slides18.pdf for how.
- You will have one loop in your program, which will:
  - Ask the user for account number (use `printf` to write to standard output), then read this from standard input using `scanf`.
  - Ask the user for name (`printf`), then read this (`scanf`). You can assume there are no spaces in names.

  - Ask the user for the account balance of this account (`printf`), then read this using `scanf`.
  - Write these details as a new line in the `clients.txt` file (*this time* use `fprintf` with the stream name for `clients.txt` to do this).
  - Ask the user if there is another client to add (`printf`), and read in `'y'` or `'n'` with `scanf`.

    In using `scanf`, you will probably have to use `scanf(" %c", ...` (with a space before `%c`)

- At the end, remember to `fclose` your stream to `clients.txt`.

After you have written the program, please compile it using

```
gcc -Wall addentry.c ↲
```

Once your program is compiling, as an initial test of your program, enter the following inputs:

```
100 Jones 0.00
200 Doe -345.67
400 Stone -42.16
500 Rich 224.62
```

In order to check that your program has entered this information, you can type `more clients.txt ↲` at the command line.

## Listing debtors

After you have created `clients.txt` file now you can write another program called `listdebtors.c` which reads the `clients.txt` file sequentially and lists (to standard output, using `printf`) accounts which have balances of less than zero (those who owe the company money).

## Your result for `addentry.c` and `listdebtors.c`

- ☐ Both programs should be *correct*.
- ☐ You can check the correctness of your `addentry.c` program by looking at the file that you have created and written.
- ☐ Assuming you use the suggested inputs above, the result of `listdebtors.c` should be

  ```
  200, Doe, -345.76
  400, Stone, -42.16
  ```

- ☐ Submit your programs with
  `submit cp lab8 addentry.c listdebtors.c`



# Stage C String Manipulation Program 2

Now we are going to write a string manipulation program as we did in lab7. But this time the functions are going to be recursive, to illustrate the differences between iterative and recursive programming.

Remember throughout this Stage, that 'recursion' means *using other calls to the same function*. So for the purposes of this task, please use a recursive approach, *even if* there may be a simpler solution without recursion.

Your program should be named *stringman2.c* and have the following capabilities:

(i) You should implement a recursive function called `funct1` that takes a string (an array of `char`) and the length of the string as arguments, prints the string and returns nothing. The function should be implemented recursively, the 'base case' being to do nothing when it receives an array of length zero.

Your function should have the following prototype:

```
void funct1(char* str, int len)
```

(ii) You should also implement a recursive function called `funct2` which takes a character array (ie a string) as an argument, prints the string it represents *in reverse* (back-to-front), and returns no value. The function should be implemented recursively, with the 'base case' being to do nothing at all if the input starts with the terminating null character `'\0'`.

```
void funct2(char* str)
```

(iii) Your program will also implement a recursive function `funct3` that returns 1 if the substring stored in the given array `str`, between the indices `left` and `right`, is a palindrome, and 0 otherwise.

```
int funct3(char* str, int left, int right)
```

For each of the three functions, you will be passing it a string from `main`. To avoid compiler warnings, it is best to pass strings that are declared as `char` arrays.[1] For working with these strings:

- You should assume some fixed maximum length for strings and always declare this amount of space (an array with this many `char` entries) for any string you work with in `main`. Ensure that all strings end with `'\0'`.
- In functions `funct1` and `funct2`, you might like to use `putchar` instead of `printf`.
- Depending on how your `main` function is written, you may need to `#include <string.h>`

Apart from writing the three functions, you might want to use the `main` function (with some adaptations) that you used in the previous lab. This is the `main` from `stringMan.c` which scans in strings from the user (using `scanf`) and asks the user which function they want to test.

After you have written the program, please compile it using

```
gcc -Wall stringMan2.c ↲
```

[1] Alternatively, you can change the prototype of your functions to be `void funct1(const char *str, int len)` etc., and then you can pass string constants like `"radar"` directly to them.

## Your result for `stringMan2.c`

☐ Program should be *correct*. To test:

- For the choice 1 and the string `university` the output should be `university`.
- For the choice 2 and the string `university` the output should be `ytisrevinu`.
- For the choice 3, if the input is `madam` the result should be `palindrome`.
- For the choice 3, if the input is `history` the result should be `not a palindrome`.

☐ Submit your program with
`submit cp lab8 stringMan2.c`

*Julian Bradfield* after *Çiğdem Beyan and Mary Cryan, updated Nov 2016*