# G51PRG Exercise Two: Octal and Hidden Characters

*Steven R. Bagley*

### Introduction

Over the next couple of courseworks, we are going to create a program that will read in a file and print it to the screen so that all characters are visible. Some ASCII characters aren't normally printable (those with character codes below `32`), and so your program should convert these into a printable form. We'll use the same representation that C uses — in most cases, the octal version of the character code preceded by a backslash (e.g. `\007` for the bell character). However, for some characters (e.g. the TAB character, ASCII code `9`) C uses a single character instead (e.g. `\t`) — see the appendix for a full list of characters. Note that characters used to denote the end of line also fall in this low-ASCII region and so you'll need to make sure that a new line is started by emitting the line feed character itself (rather than printing `\012`).

As well as making the characters visible, your program should also work out whether the file uses UNIX (`LF`), Windows (`CRLF`), or Mac (`CR`) line endings. The specification below contains more details on how to do identify and report these statistics.

This coursework gets you to implement the skeleton of the program, the next coursework will get you to refine your implementation and add features — *so please don't delete your code*. This week, you'll need to submit two files: one containing your octal converter, and a second which contains your initial version of the printer. The next exercise will get you to integrate your octal converter into the main program.

There is no single correct solution for this exercise; any program which meets the specification will be given a pass mark. To achieve higher grades, your program should make appropriate use of variables and functions in your solution. In particular, to obtain the top grades it is necessary to design your own functions and make correct use of them. Full marking criteria can be found at the end of this document.

### 1. Octal conversion

Since your program is going to print out the octal representation of an integer, it needs to be able to convert an integer into an octal representation so that it can print it out. `printf` does provide the `%o` format specifier for that could do this, but for this coursework you will implement your own. In this section, you'll implement the basic functionality of an octal converter but it will print the results out backwards. To print it the correct way round rquires some parts of the C language that we have not covered yet — so you'll add that functionality later. For now, your program should just print out the steps of the calculation it needs to perform. Therefore, for the moment, this should be implemented *as a separate program*.

This program should read in a number from the keyboard (using `scanf()`) and then print out each step of converting that number into octal. Conversion to octal uses exactly the same steps as a human would use to convert a number to binary (as we saw in the G51CSA lectures), however we divide by eight rather than by two. In other words, we will repeatedly divide the number by `8`, until it reaches `0`, printing out the remainder (modulus) at each stage. We will then print this neatly out to the screen, so for the input `123` we would output:

```
123 / 8 =    15 remainder 3
 15 / 8 =     1 remainder 7
  1 / 8 =     0 remainder 1
```

We can then read the octal number up from the bottom to the top (in this case, `173`). To print the octal number out on it's own, we would need to print the numbers out in reverse order, there are several ways you could do this but for now the program should print it out as shown above. (In the next coursework, we'll use an array to print it out in the correct order.) Note we can get the nicely formatted output seen above very easily by using `printf`. If you recall, we can use the placeholder `%d` to represent an integer. However, `printf` lets us format the output by saying how many columns we want the output to take up by putting the number of columns between the `%` and the `d`. So if we use, `%5d` then printf will format each number so that it takes up five columns in the output as shown — this should be enough for the numbers we are likely to enter (anything up to 99999).

The algorithm we use here (both as a human and as a computer) is simple. We repeatedly divide the input value by eight while the result is not zero and print the result and remainder to screen using `printf`.

**2. Print File**

As described in the Introduction, the main program you will develop is to print out the contents of a file to screen. Initially, you'll use the same method we did in lectures to write the *word count* program — that is repeatedly using `getchar()` to read characters from the standard input until the end-of-file is reached. A later coursework will get you to modify your program to directly read the input from the file. Until then, you can pipe the contents of a file into your program using the UNIX redirect opeartor '`<`' like this (assuming your compiled program is called `print`).

```
./print < silly.example
```

You can use `printf` to write a character back to the output, but it is probably more efficient to use `putchar` to print them out in this case.

As well as printing out normal visible ASCII characters, you program should also make the low-ASCII characters visible, either in octal or as a C escape character (e.g. `\t`). Since your octal converter isn't finished yet, for now your program should just print a question mark character for low ASCII characters that do not have a C escape sequence. See the appendix for a list of characters that will need special handling and what should be output when they are encountered.

Finally, after it has printed out the file your program should print out a single line stating how the lines in the file are terminated. Different computer systems use different character sequences to determine where a line ends. On UNIX, this is ASCII code 10 (the line feed character), but on a Mac (at least before MacOS X) it has traditionally been ASCII code 13 (the carriage return). DOS and Windows on the other hand have made use of two characters, a carriage return immediately followed by a line feed to determine the end of a line. While reading in the characters from the file, you should store whether you see UNIX, Mac or DOS line endings, and at the end print outone of the following:

```
This file had Mac line endings.
This file had UNIX line endings.
This file had DOS line endings.
```

depending on what was found in the file. On the other hand, you may find mixed line endings in which case you should print:

```
This file has mixed line endings.
```

Note that while it would be easy to find just Mac and UNIX line endings, detecting DOS line endings will require you to keep state since you can't know when you see a CR (ASCII code 13) whether it is a Mac line ending, or the beginning of DOS line ending. Therefore, you'll need to keep track of what the last character you saw was. *In all cases, you should output a single UNIX line-ending (`\n`) regardless of what was encountered in the file.*

A selection of variously ended files can be found at:

```
http://g51prg.cs.nott.ac.uk/code/various_texts.zip
```

to let you test your program. Download these to your UNIX file space (mapped to your Windows `H:\` drive) and extract them by typing:

```
unzip various_texts.zip
```

at the UNIX command prompt.

### 3. Marking Criteria

Each sub-problem of the coursework will be assessed in three parts. Firstly, grades will be given for the use of Functions, Variables, Conditionals, and Loops where appropriate. Secondly, grades will be given for each of the specific features asked for in the coursework description document. Finally, a mark will be given for the 'Overall impression' given by the implementation.

In all cases, we are looking for solutions that make the best and most appropriate use of functions, variables, conditionals and loops, and execute in an efficient manner. All programs that correctly implement the problems described will obtain a pass mark.

All grades will be given on the scale A — E.

## APPENDIX

| ASCII Code | Printout | Meaning |
|---|---|---|
| `< 32` | Octal version, unless listed below: | |
| `8` | `\b` | Backspace |
| `9` | `\t` | ASCII Tab |
| `10` | A new line | Linefeed |
| `13`† | | Carriage Return |
| `92` | `\\` | Backslash itself should be escaped to enable us to see original backslashes in the file, otherwise `\t` would be ambiguous. |

† Note you should also output a single newline for DOS line endings too.