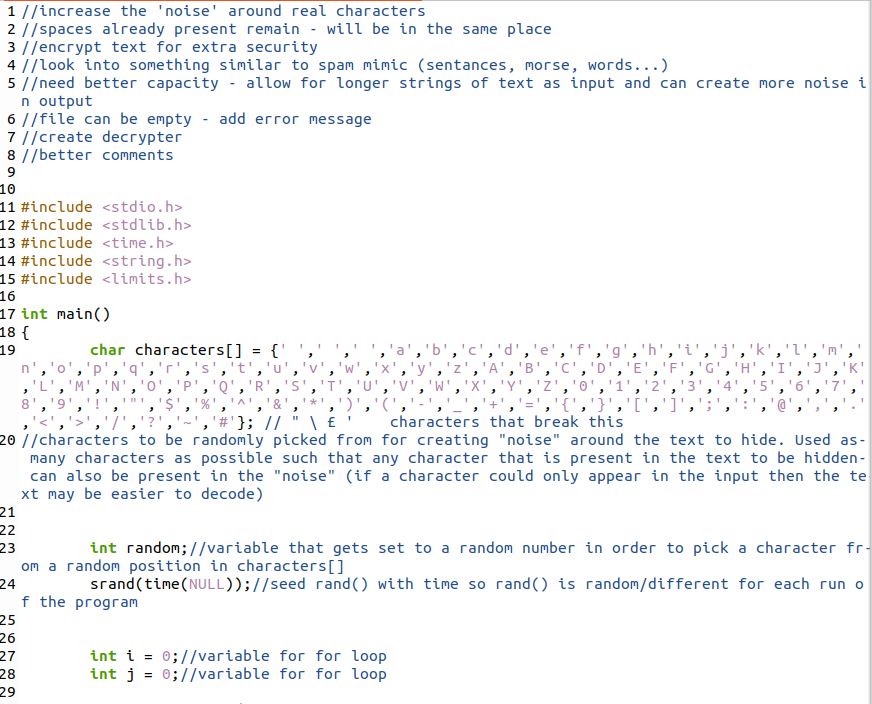
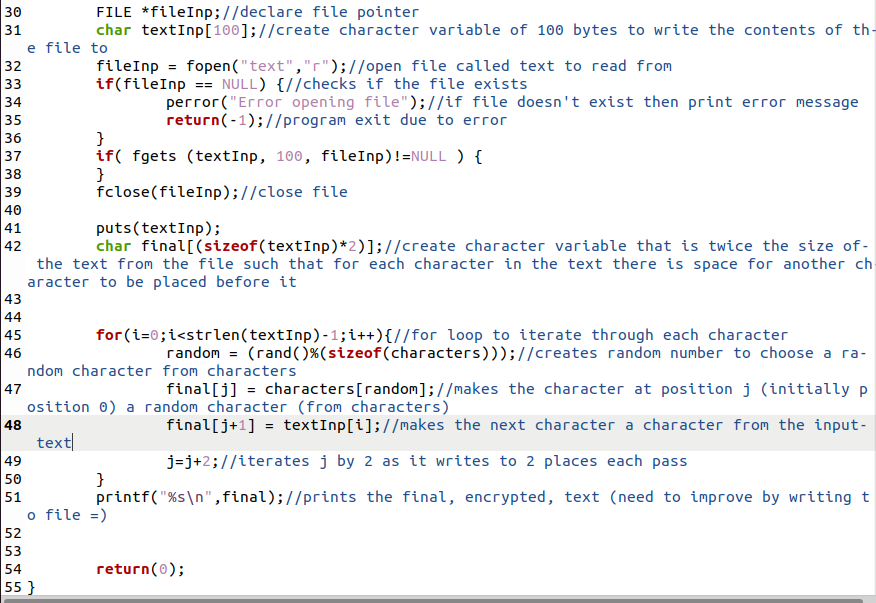
**Text Steganography**

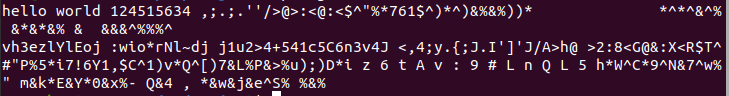
Initial Ideas

Text steganography is hiding text within other text, this can take many forms; using deliberate errors and mistakes, using zero space Unicode characters, using whitespace, highlighting characters in text (e.g. capitalisation), using a code to give character positions, spammimic.com creates an email-like text that contains the hidden text.

I have chosen to do text steganography using a code that gives character positions. This involves sending a message, that can take any form, along with a string of numbers which correlate to the positions of a character within that text (e.g. ‘There was a person’ 123 would be decoded as ‘The’). I identified two aspects of this that are potential weaknesses that can be improved; firstly that the space efficiency is poor – long strings of text to be encoded may require even longer container text along with a long string of digits, and that sending the string of digits may arouse suspicion on what is supposed to be an innocuous message to any third party.

This idea was an improvement on an original idea to hide the text by simply surrounding it with “noise” text and removing the “noise” to reveal the message (steg1.c, GitHub): 



Output (‘text’ contains: hello world 124515634 ,;.;.''/>@>:<@:<$^"%\*761$^)\*^)&%&%))\* \*^\*^&^% &\*&\*&% & &&&^%%%^\*(oafojofajoaAHHHHHHHHHHHHHAIIBAIBCIBIWIBDAIND): 

There is an issue here with the size of textInp being 100 and therefore not containing every character of ‘text’ as it has more than 100 characters.

Future improvements to this would involve increasing the additional text around the message characters however I decided that there would be a better method of text steganography than this as even at this stage the size of the encoded text would be twice that of the original text and this would only increase if more “noise” was added, but without additional “noise” the text is not greatly hidden, especially when compared to image or audio steganography.

The first part of the improved version used some of what I’d done for this but ultimately would have different functionality.

Plan

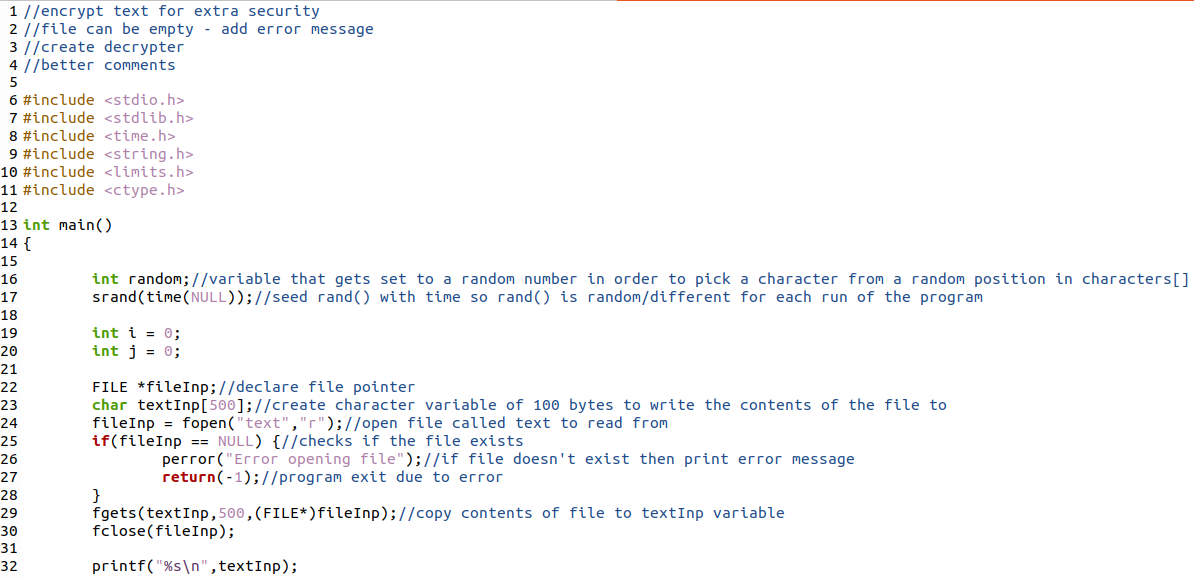
The (improved) plan for creating the text steganography follows;

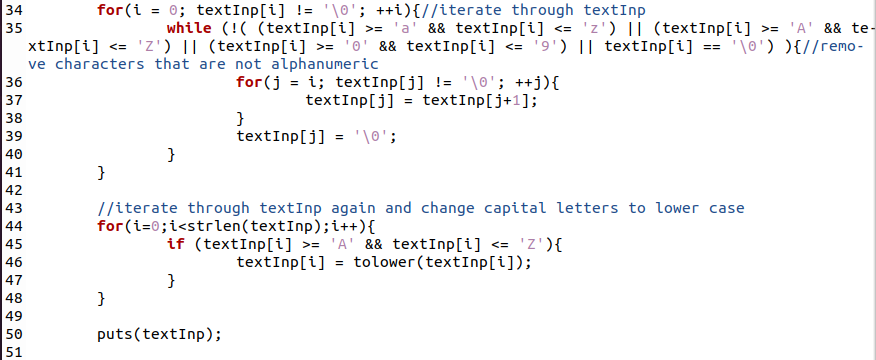
* Take text to be encoded from file
* As I’m dealing only with letters and numbers, remove all non-alphanumeric characters from the text to hide
* Use a list of words that contain all 26 letters as well as numbers 0-9
* Print these words and have a string of digits that correspond to the letters from the text that need to be hidden
* Find way of hiding the string of digits (e.g. the number of characters in the hidden string is the digit string (this would be too inefficient as the encoded text would have to be very large and there could be complications of determining if a character is at position 1 followed by the same character or is at position 11)).
* Output the encoded text to a file
* When decoding loop round the words (this would allow a smaller output compared to keeping adding words if the input text is longer)

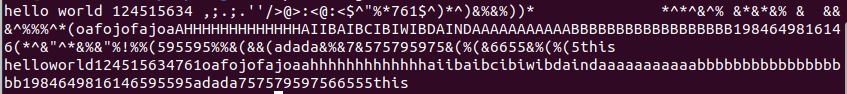
Example word list: angry, blitz, dock, hex, quid, jumps, vow, fouls, zebra, toxify, king. Letters repeat so that if a letter repeats in the input text then the program may not have to loop back through the whole of the encoded text which could allow the program to run quicker.

Coding

I used the code from my original idea to get the contents of the ‘text’ file, then began by removing non-alphanumeric characters for the variable I’d be working with.

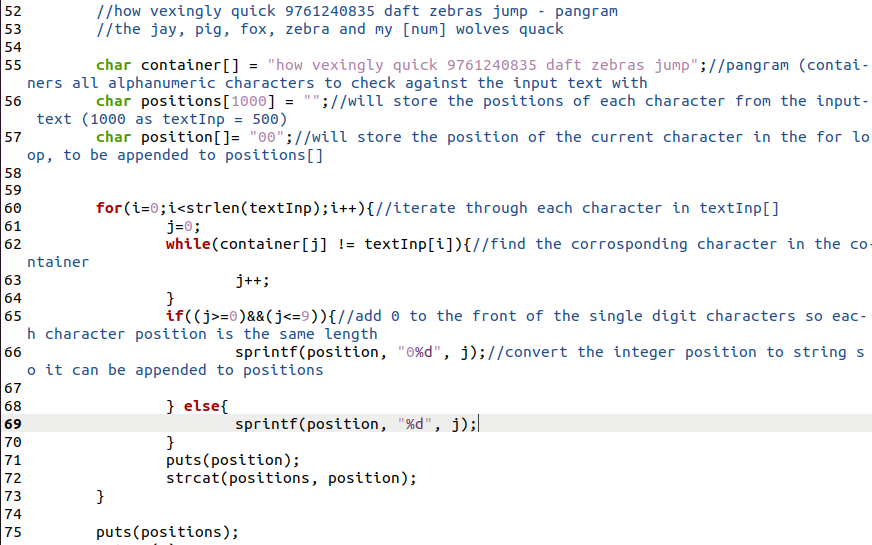




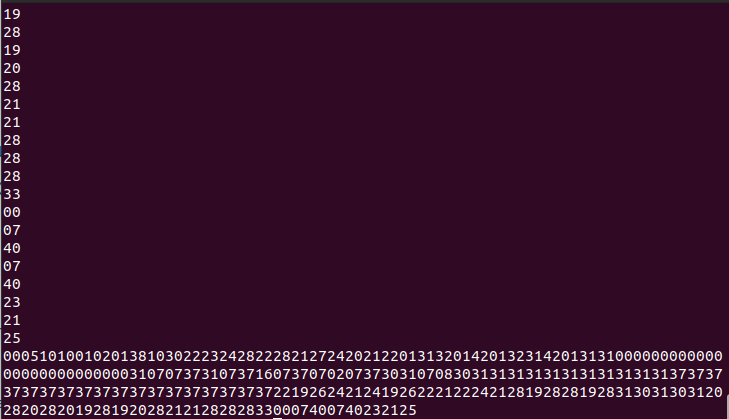
Output: 

There is once again a problem of not getting all the contents of the file past 255 characters.

Now that the input text is in a more usable form I can begin to encode the text, to begin with I found text that could be used as a container, a sentence which contains all the letters and numbers. By using a real sentence as opposed to random text it can be argued that the message is better hidden as random text may arouse suspicion.



Output:



At the moment the program is also outputting the value of ‘position’ on each iteration through the loop, this is just for testing purposes to make sure that the desired output is obtained – such as correct values for character positions and each value being equal in length (2 digits).

I think it is important that each position is the same length as this will make decoding later on easier – if single digits were also used then an output such as: 1234 may indicate a value at 1,2,3,4 or 12,3,4 or 1,23,4… with no distinction between them as opposed to 012304 with each two digits being one position.

At the moment only the positions string is outputted, this needs to be outputted with the container text. Outputting the container with the positions string can be done in various ways, once all the functionality is achieved I can then improve on the current method by choosing a method that is less likely to arouse suspicion – by having the overall output of container and positions look like normal text.



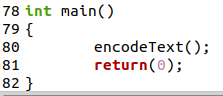
Output (previous printfs/puts… now removed:



This is initially a basic way of outputting everything – I have used another sentence that containers all the letters and which has the positions string within it so it takes the same form as the first sentence with the number in it, this can be seen as a form of obfuscation; the second sentence may appear to be more important for decoding than it actually is. By doing this it effectively splits each component of the text steganography into two parts, one in each sentence. In the future an improvement could be to further split these parts to further hide the text and make it more difficult to see that there is hidden text and more difficult to decode it. I could also hide the positions string or incorporate it in the text in a more natural way; a successful way of doing this could be writing text that has a total number of characters equal to the positions string but this would result in inefficiency due to the number of characters required. I could also further obfuscate the output to further hide the decoding process (e.g. add extra digits to each position which appear in the output but can be ignored, random capitalisation, more sentences/text so the actual text needed is a smaller part of the output… An issue with further obfuscation is that it will often lead to a larger (more inefficient) output or an output that looks less like ‘real’ text, thus possibly arousing more suspicion.

The encoding now works (although can be improved upon in the future) so I decided to start on decoding functionality. Before this I put the code already completed into a new function so encoding and decoding would have their own function.

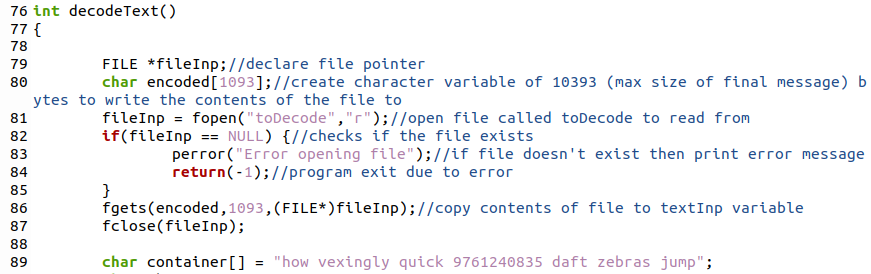




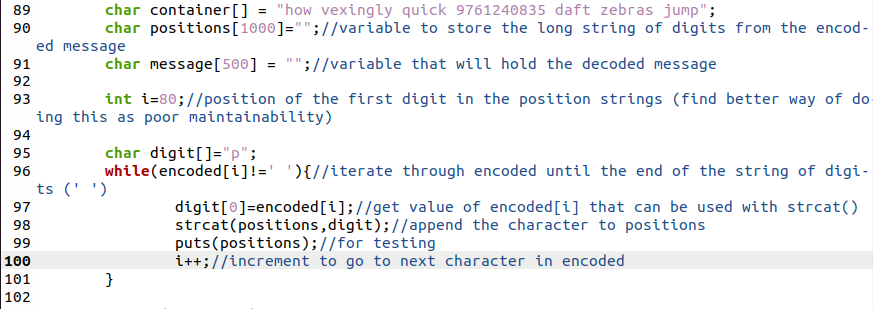
At this stage I also realized that future improvements should include:

* A way for the user to specify if they want to encode or decode
* Have the encoded text be outputted to a file
* Be able to have the user specify the file they want to encode or decode

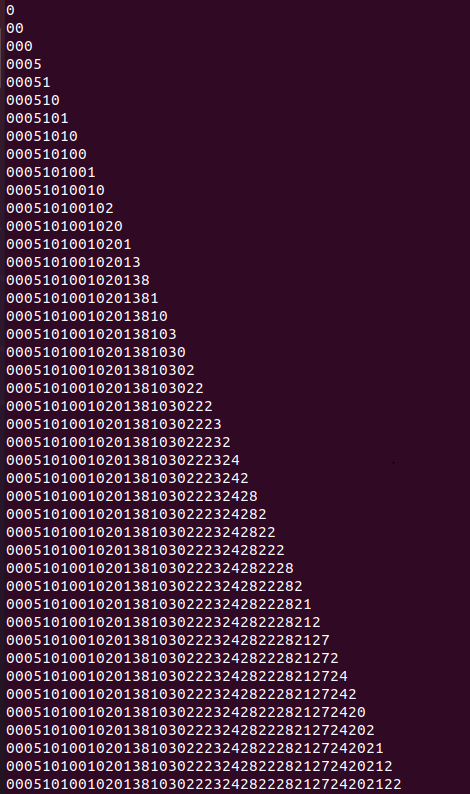
To start the decoding function I once again get the text from a file and store it in a variable. In the future I will ask the user for the file name.



Next I began by creating a while loop to extract the ‘positions’ string.

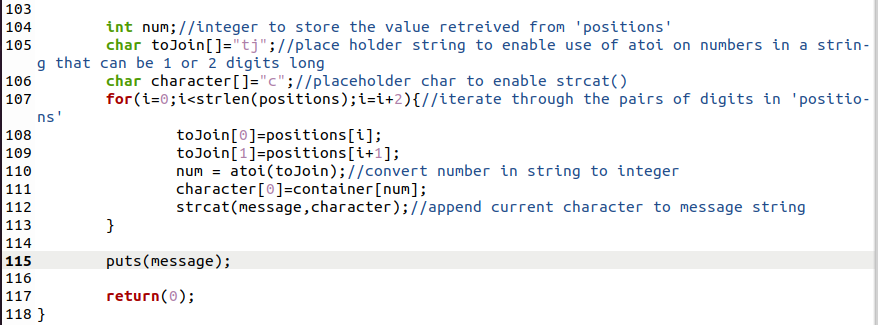


Output:



This matches the string of digits in the input file.

I would now need to use each pair of digits to compare to the container string:



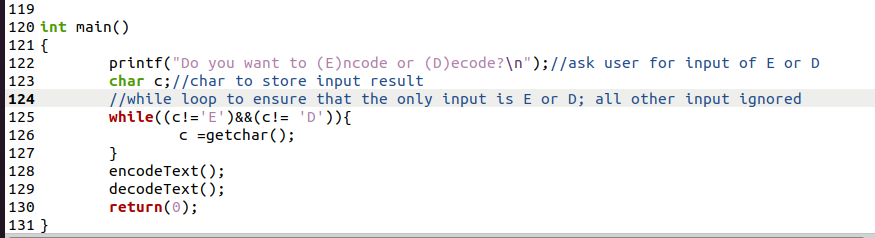
Output:



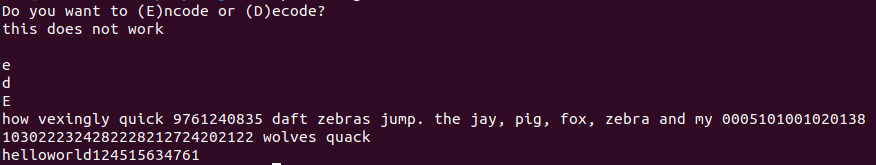
This is the expected output.

The program now encodes and decodes text, next I would need to improve the functionality by incorporating a way for the user to specify a file to encode and move the output to a file with a specified name, as well as give the user the option of encoding or decoding.

I started working on getting the user to choose whether they want to encode or decode, beginning with asking and getting a valid input:

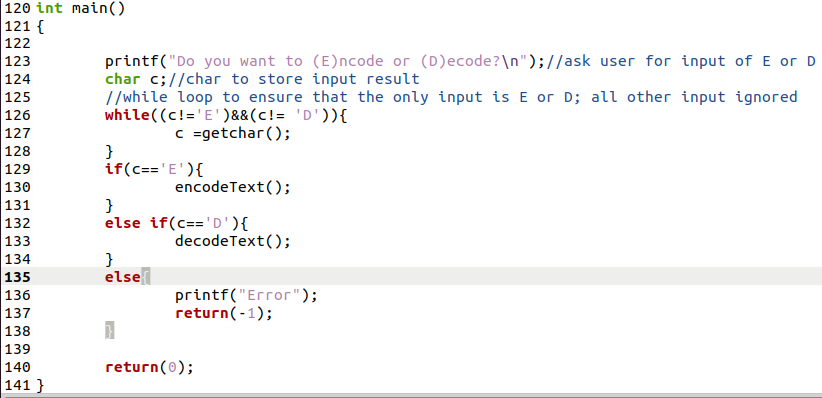


Output:

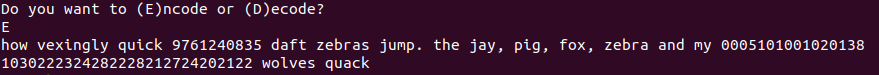


One issue with this currently is that typing text which includes an E or D will still be accepted, additionally putting text containing both an E and a D will be accepted – taking the one that appears first as the value for c. As this still allows the program to function and does not break it this is accepted for now but can be improved in the future.

By only accepting E or D I am now able to confidently direct the program to either encodeText() or decodeText() – if other inputs were accepted then a subsequent if statement may break if the input differs from the expected value(s) of c. (i.e if(c==’E’)… if(c=’D’)…. would break if c was equal to neither ‘E’ or ‘D’).



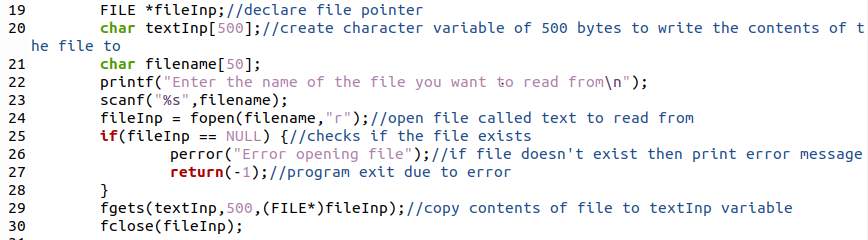
Output(s):



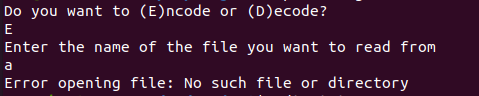


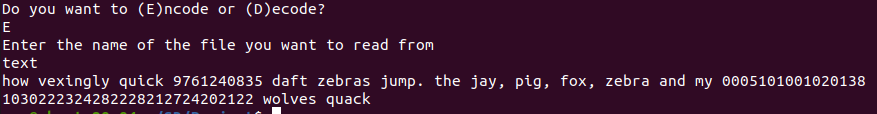
Now it was time to work on adding functionality through files.

I started on the encodeText() function by asking the user to enter the name of the file to read the text from.



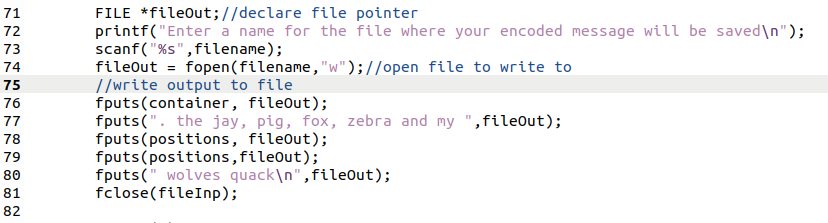
Output(s):



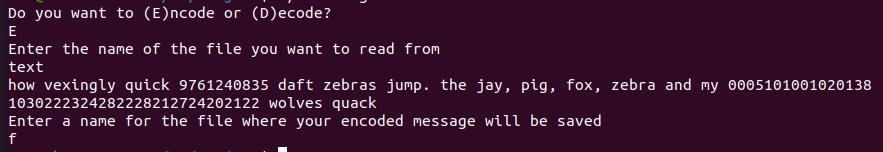


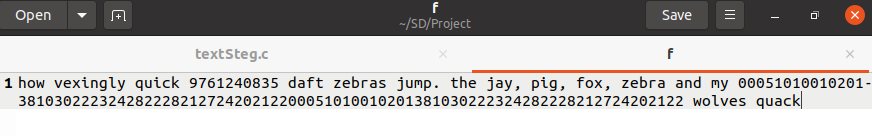
At this stage a possible point of error arises as a file may have a valid name but could be an image file or an audio file etc… There may be a solution here when the rest of the project is incorporated as other parts will be able to handle different file types.

I then added code to write the output to a file (and create one if needed).

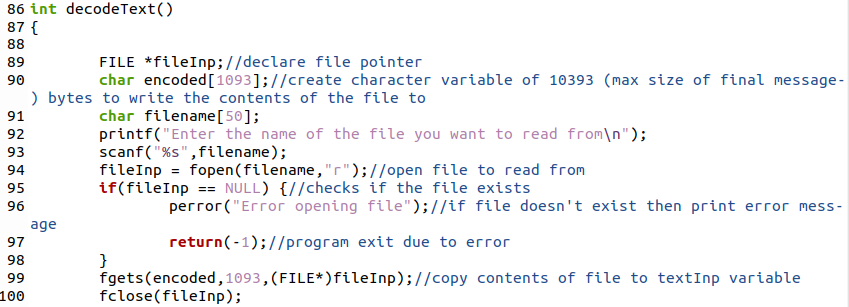


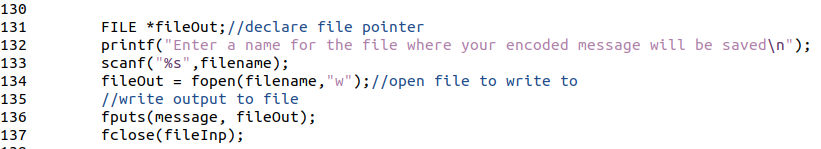
Output:



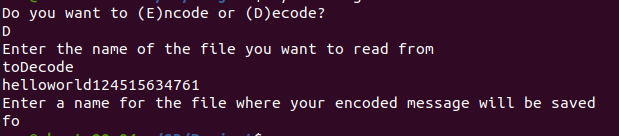


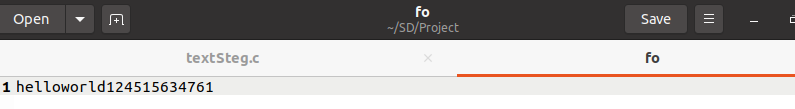
I then did the same for the decodeText() function.





Output:

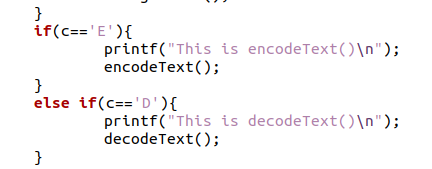




Testing

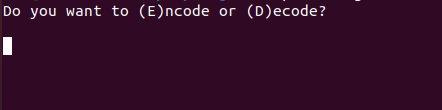
Testing for main() function:

(For testing purposes I added two printf lines to indicate whether the program goes to encodeText() or decodeText()):



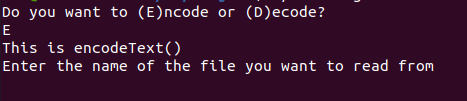
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Empty input | Press enter without typing anything | New line, program continues waiting for input | Yes |

Output:



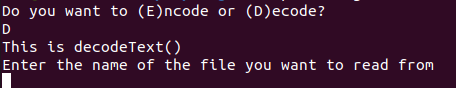
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Enter ‘E’ to go to Encoding | E | Program goes to encodeText() code (will ask for input file) | Yes |

Output:



|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Enter ‘D’ to go to Encoding | D | Program goes to decodeText() code (will ask for input file) | Yes |

Output:

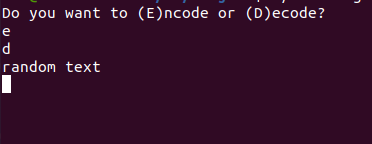


|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Enter random text | e | New line, program continues waiting for input | Yes |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
|  | d | New line, program continues waiting for input | Yes |

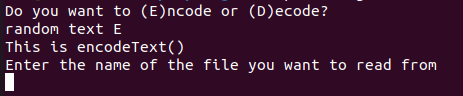
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
|  | random text | New line, program continues waiting for input | Yes |

Output:



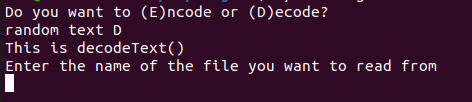
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Entering random text including an ‘E’ | random text E | New line, program continues waiting for input | No. Program reads the ‘E’ despite the random text and goes to the encodeText() code |

Output:



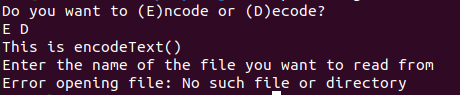
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Entering random text including an ‘D’ | random text D | New line, program continues waiting for input | No. Program reads the ‘D’ despite the random text and goes to the decodeText() code |

Output:



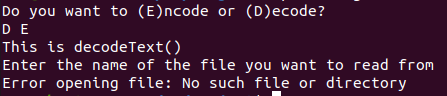
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Entering ‘E’ and ‘D’ | E D | Goes to encodeText() (based on previous tests results) | No. Goes to encodeText() and takes ‘D’ as input for the file name. |

Output:



|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Entering ‘D’ and ‘E’ | D E | Goes to decodeText() but erroneously takes ‘E’ as a input for file name (based on previous tests results) | Yes |

Output:



[testing with ‘E’ or ‘D’ followed by any text showed the program taking the extra input as input for the file name]

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Correct Output?** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Correct Output?** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Correct Output?** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Correct Output?** |
|  |  |  |  |

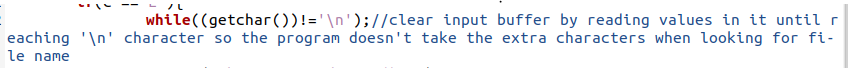
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Correct Output?** |
|  |  |  |  |

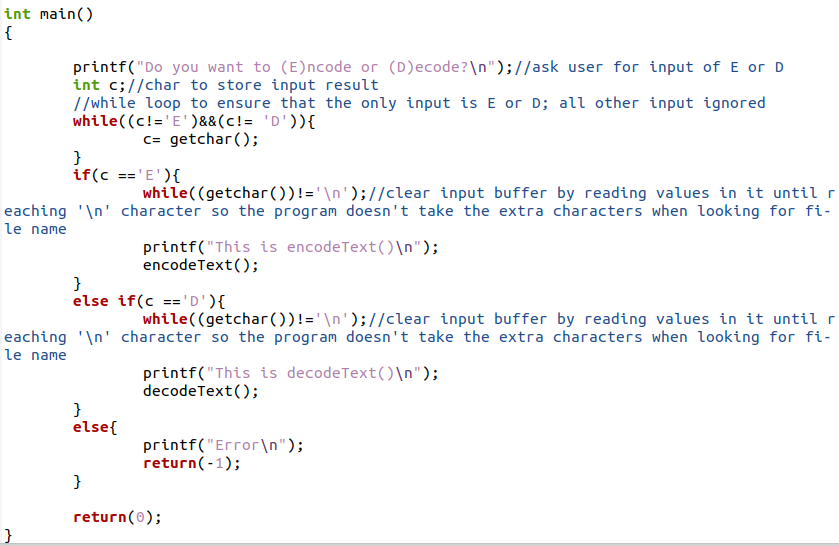
|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Correct Output?** |
|  |  |  |  |

Improvements After Testing:

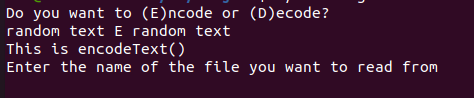
[main()]

In order to prevent the program taking inputs after writing ‘E’ or ‘D’ for file names (e.g. input of ‘E random text’ would fill the input buffer with ‘random text’ which would then be taken as an input) I added code that loops through the extra characters until the ‘\n’ character is founder, marking the end of the input:



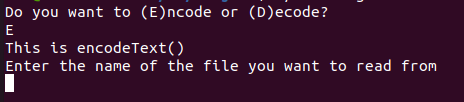


Output:



|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Entering ‘E’ followed by text | E random text | Goes to encodeText(). (Doesn’t take any input after asking for file name) | Yes |

Output:



|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output** | **Output Matches Expected?** |
| Entering ‘D’ followed by text | D random text | Goes to decodeText(). (Doesn’t take any input after asking for file name) | Yes |

Output:

