# EECS 4980 / 5980 – Inside Cryptography S'17
# Programming Assignment #1
## Due Thursday, February 16, 2017 at 11:59:59 PM.

> ***NOTE***: Under no circumstances are you allowed to use *any* code from *any* source other than what you individually create from scratch, or are provided by the instructor. You may, of course, use any information in the text and the lecture slides, including the "DES Algorithm Illustrated" article by J. Orlin Grabbe, a copy of which is posted on Blackboard. Source code for DES is readily available from the web; *don't use any of it, even as a reference. If I suspect you didn't write your code, I'll ask you to explain it, potentially line-by-line, to prove to me that it's yours and yours alone.*

Your assignment is to write a (single) program to do DES encryption and decryption. Use Visual Studio 2015 to code your **Win32 Console Application** in C/C++. Call your program "DES". Do NOT write a .NET application; you will be writing *unmanaged* code. Include in your project the compilation switches to provide the directives necessary to generate native 64-bit instructions; your code will run much faster if it can work directly on 64-bit quantities.

The command-line syntax for your program will be:

```
DES <–action> <key> <mode> <infile> <outfile>
```

Note: The angle brackets are shown here ONLY to delimit the parameters, they will not actually be part of the command line (See the examples below)

All five command-line parameters are required. The "action" will be either "`-d`" or "`-e`" (case ***in***sensitive). If the action is "`-d`", you are to decrypt the ciphertext file specified by `<infile>` and place the resulting plaintext into `<outfile>`. If the action is "`-e`", you are to encrypt the file specified by `<infile>` and place the ciphertext into `<outfile>`. The `<key>` is specified as either 16 hex digits (also case ***in***sensitive), or a string of 8 characters enclosed in single quotes (note: if your 8-character key contains embedded spaces (which IS allowed), the whole thing will have to be enclosed in **double** quotes – see the examples below).

The `<mode>` parameter will be either "ECB" or "CBC" (case-insensitive, no quotes). **We will discuss what this means in class, and you will be implementing both later on, but for now just leave this as a placeholder, and only accept "ECB" as the mode.**

You must validate all of your command-line parameters (key length, contents, file existence, etc.). Errors on the command line should result in the printing of a message indicating the proper usage, and then a graceful exit. If the user supplies valid command-line parameters, the only console output your program should produce is the elapsed time it took to encrypt or decrypt.

Examples of valid command-line parameters:

```
DES –E 0123456789ABCDEF ECB plaint.txt cipher.txt
DES –D 0123456780ABCDEF ECB cipher.txt plain-again.txt
DES –e 'Pa$$word' cbc in.txt out.txt
DES –d 'Pa$$word' cbc out.txt in2.txt
DES –E "'KeY WoRd'"  EcB in.txt out.enc
```

DES works with 64-bit (8-byte) blocks at a time. If the length of the file you wish to encode does not happen to be an even multiple of 8, then you will have to pad the final block to make 64 bits. You will handle padding as follows:

Your program should be written to handle 31-bit file lengths (0 – 2 billion bytes). After validating the command-line parameters, the first thing your program should do is establish the length of `<infile>`, and use that file size as the right half (bits 32-64) of the first block to encrypt. The left half should be randomly generated garbage (we'll discard it later). Encrypt the first block (containing the garbage and the file length) and write that block to `<outfile>`.

The second block you produce will be the actual first block from the data in `<infile>`. If, in the final block of the data file, you have to pad the file to get an even multiple of 8 bytes, fill the padding bytes with random characters (0-255). Do not pad with static data (like all zeroes or all 0xFF).

When you decrypt, read the first block and decrypt it. Mask off the left half, leaving a 32-bit file size. Then process the remaining blocks. When you get to the last block, if the file size is not a multiple of 8 bytes, output only the necessary number of bytes (and discard the remaining garbage, if there is any).

The result of this padding scheme is that your ciphertext file will be 8 to 15 bytes longer than your plaintext file. If the original file was a multiple of 8 bytes in length, the ciphertext file will be exactly 8 bytes longer to accommodate the 64-bit file size block. If you have to pad with 1 to 7 bytes to get an even multiple of 8 bytes, then your ciphertext file will be 9 to 15 bytes longer than the plaintext file.

When you decrypt a ciphertext file, your decrypted file should be bit-identical to the original plaintext file – same length, no extra or missing characters, etc.

The first byte in the input file will be the most-significant byte in your starting block (i.e., if the file contains "ABCDEFGH", then the "A" (ASCII 65) should be in the left-most (most-significant) byte, and the "H" (ASCII 72) would be the right-most byte (least-significant). Depending on how you code your application, endian-ness may be an issue you have to address.

The Windows command-line program FC (File Compare) can be used with the "/B" option to compare two files at the binary level:

```
>FC /B file1 file2
```

Submit your code as a 7-Zip archive (zip, bzip, gzip, tar, rar, etc. are ***not*** acceptable) of the entire Visual Studio Workspace (including all source and binary subdirectories). Post your submission to Blackboard. Make sure your code is well-commented, and that the comments include your name!

There is no reason to use an object-oriented approach – this can all be procedural code. Your code should be well-designed (appropriately modularized) and well-documented. There is no reason for shoddy documentation; in fact, extra documentation will probably help you write the code. Resist the temptation to code it all with minimal / no comments and then add the documentation later. If you write your comments BEFORE you write the code, the comments will serve as a trail of breadcrumbs for you to follow as you write the code.

The comments in your code should demonstrate to me that you understand how the code works.

If you have any questions, or run into difficulties, e-mail me as soon as possible. Do not delay in starting your work; there's a lot to do here. None of it is particularly difficult, but as they say, "the devil is in the details." Debugging crypto code is particularly challenging, because, even when it works right, the output looks like garbage. When it doesn't work right, the output is _still_ garbage, and it is very difficult to look at the output and try to determine where the problem might have occurred (i.e., it's hard to tell why you have the _wrong_ garbage!)

DO NOT use bitsets, vectors, collections, or any other STL type or extension to C/C++. It should go without saying that you may not use a crypto library. Do not use a 64-element integer (or string) array to represent 64 bits. What you really want to be writing is essentially assembly code using C's syntax – you don't need any classes for this project. Think in terms of logical operations you can use on values in a CPU register.

DES works on a 64-bit quantity. Think in terms of assembly language – You should use a 64-bit (unsigned) integer as your primary data type, and your bit-manipulation operations should all be logical, rather than arithmetic. There are some nuances here for you to consider. Put some thought and design into this before you start coding. I STRONGLY suggest you start by coding some debugging routines to help you tell what's going on in your program (perhaps something to output a 64-bit quantity in binary, hex, and ASCII (to the extent that the characters are printable). Next, code the permutations and S-boxes, the key generator, and the command-line parameter validation and file code – coding this project works well when built (largely) from the bottom-up (just keep an eye on top-down).

For file I/O, you will want to use either C-Style FILE I/O, or C++ I/O streams – your choice.

The first phase of this project will be for you to implement ECB mode. Later, we will add CBC mode support to it. In order to be better poised to make the changes later, the core of your program should be a function (perhaps called DES?) that takes in a 64-bit quantity and returns a 64-bit quantity. Whether you leave the array of keys and the action (encrypt / decrypt) as global variables or pass them as parameters to DES() is up to you. There is no sense whatsoever in having a function to encrypt and another to decrypt.

Think also in terms of efficiency. Your completed code should run quickly (after all, this IS code that would be built into system utilities; we don't want to wait while we number-crunch our files just to encrypt / decrypt them!). I am providing a 5 MB test file, SHAKESPEARE.TXT, which contains the complete works of Shakespeare's plays. Your program should encrypt / decrypt this file in _well_ under 10 seconds, depending on your CPU and speed. On my 3-year-old desktop PC, my DES code process the Shakespeare file in about 3 seconds if it's on the hard drive; about double that on a flash drive. Your program must output, as it ends, the amount of time taken to encrypt or decrypt. Measure this in milliseconds, but report it in decimal seconds with three decimal places (as in "Elapsed time: 3.141 seconds"). Your program should not have a "press enter to exit" delay at the end. This is a command-line utility, so it should do its thing and exit.

You are required to ALSO write a second program (or set of programs) that will analyze the input and output files, and do some statistical analysis on each. Use Excel (or MATLAB, if you wish) as necessary to create graphs and tables.

Write up your findings in an MS Word document (.doc/docx), and include this Word file (and the Excel file, as well as the source for this second program) in your 7-Zip archive for the whole project. The encryption process is some heavy-duty code, and it's worthy of a proper analysis; don't prepare a shoddy one- or two-page report with your findings – do it justice; you have a 5 MB file of input, so you should be able to come up with some pretty good statistics. Allow plenty of time to write this analysis program and prepare the report. If you are coding DES up until 24 hours before it's due, your analysis will be too thin. This is to be a proper, formal lab report – document what you expected to find, what you observed, offer explanation as to why you observed what you did, etc. Embed your histograms in your document, clearly labeled (“See Figure 1 below”, with a caption under the graph that says “Figure 1 - <description of what’s in Figure 1>”

Your analysis is to consist of (but not necessarily be limited to):

1) Single-bit frequency: how many one and how many zero bits are in the input / output files? You may want to pick several (all?) 8-byte blocks and see what the distribution of ones and zeroes are within blocks (how many are 32/32, how many are 31/33, 30/34,…0/64).
2) Single-byte frequency (for a text file, this should look a LOT like the standard English letter-frequency histogram we saw from Chapter 3). Display this as a histogram of all 256 byte values for both the input and output files. For the input file, produce a second histogram, showing byte values 10 through 127.
3) DI-gram frequency: Create a 65536-element array of integers. The subscripts will be made up of all of the two consecutive byte sequences in the files. Remember, "Them" contains three digrams – "Th", "he", and "em". Sort this array, and plot the whole thing as a histogram, with no X-axis. Also, plot the first 30 (most-frequently-occurring) digrams and label them on the X-axis
4) TRI-gram frequency: Create a 16,777,216-element array of integers, and do the same thing as for the digrams.
5) Octet frequency: Create an integer array of 640,000 elements, along with a second array of 64-bit values (same size). Read the ciphertext file 8 bytes at a time, and build a histogram of the "octo-grams" with no label on the X axis. The easiest way to do this is just to scan the array top-to-bottom for the just-read octet, and if it's in the list, increment the counter; otherwise, add it to the list, and set the counter to 1. This will probably run 20-30 minutes, as this is an O($n^2$) approach. If you want to build a binary tree to speed this up, you can, but you'll spend longer developing a fast solution than you will brute-forcing it the easy way.
6) Analyze the frequency distributions – compare the min / max / range / mean / median / standard deviations of both the plaintext and the ciphertext files. In a perfect encryption scheme, the ciphertext will have no patterns, so the distributions should be flat (or perhaps normally distributed). To the extent that your resulting distributions are not flat (or are non-gaussian or multi-modal), identify what occurs disproportionally often, and offer an explanation as to why.

What I have outlined is the minimum that should be in your lab report – don't be afraid to "go fishing". If you observe something particular (some kind of pattern) in the output (ciphertext) file, try to measure it, and do your best to come up with a hypothesis as to where it came from.

YOU HAVE PLENTY OF TIME IF YOU GET STARTED NOW, BUT WAITING TO START IS A RECIPE FOR DISASTER. I WILL NOT EXTEND THE DEADLINE – WE HAVE TOO MANY OTHER PROJECTS TO WORK ON!

I will grade this project largely on correctness – if your code doesn't work, then you won't get much credit, no matter how many LOC you turn in. If your code doesn't even compile, that's even worse. The requirements for this program are clearly specified – it's up to you to meet ***all*** of the requirements. If your code runs but produces incorrect results (encrypts and decrypts to something other than a bit-identical copy of the original file), then the degree to which it doesn't match the original is how far it is from "correct", and your grade will reflect that. The lab report is a key component of this project – just writing correct code is not enough for a high grade; this one requires a good presentation of the results in the lab report as well.