Character Translation: Case study in design

- we want to produce a program that translates the characters in a file. That is, we have a file made up of 8 bit characters that use one character set, or encoding, and we want to convert it to another character set

- we can store the character translation in a table, where the first column is the old character and the second column is the new character, this table is stored in a file that is read when the program runs

- for simplicity we will assume that the input file will be read from standard input and the output will be written to standard output

- the specification for our program is:

Pre-Conditions:
    an input file in the old character encoding
    a translation table that converts from the old encoding to the new encoding
Post-Conditions:
    a new file that contains all the information in the input file converted to the new character encoding

- **how do we start the design of this program??**

- one way is to divide the program by functionality, there are two basic functions

- **read the translation table**
- **convert the input file**

- this suggests that the program should be divided into two procedures, one for each function

- clearly our program has the following general structure:

```
void main ( ... ) {

    read_translation ( ... );
    convert_file ( ... );

}
```

- both these procedures use the translation table, so we need to design a data structure for it. **So let's do it.**

- the translation table converts from one character encoding to another character encoding. The obvious data structure for this is a lookup table, with each entry having two fields: one for the old character code and one for the new character code

- this is the format used to store the translation table on disk, so it seems to make sense

- for each character in the input file we need to search the translation table to find its entry, we then get the new character code from that entry

- this table lookup must be performed for each character in the input file, **so it must be efficient**

- we could sort the translation table and use a binary search, **but we can do much better**

Consider the nature of the problem

- the input to the search is an 8-bit character, so we have at most 256 possible values for the input, and they are:

  0 -> 255

- why not use the input character as an index into the translation table, completely avoiding the search??

- thus the translation table data structure will be an array of char of length 256

- now we can make an interesting simplification, observe:

The translation table can be initialized to the **identity translation**, each character is translated to itself

- the translation table that is stored on disk only needs to contain the characters that are different.

This save a lot of time if the two character sets are quite similar, as is often the case.

- the translation table data structure will be:

```
char translation [256];
```

- we can now produce the main program, we will assume that the only parameter to this program is the name of the file that contains the translation table

```
void main (int argc, char* *argv) {
    char translation [256];

    if (argc != 2) {
        printf ( "usage: translate translation_file\n" );
        exit (0);
    }

    read_translation ( translation, argv[1] );
    convert_file (translation);
}
```

- now we need to design the read_translation procedure. The input to this procedure is an uninitialized translation table, and the name of the file is the translation table is to be read from. The main result of this process is a valid translation table

- for pre and post conditions we have:

Pre-Conditions:

    an uninitialized identity translation table
    the name of the file containing the translation table

Post-Conditions:

    a valid translation table

- this procedure has three main translation steps:

Initialize the table, open the table file, read the file

- initializing the translation table is a simple **for loop**
- for reading the file we can use our standard template

```
void read_translation (char translation[256], char* name)
{
    FILE* fid;
    int i, n;
    char old, new;

    for ( i = 0; i < 256; i++ )      /* make the identity table
* /
        translation[i] = i;

    fid = fopen (name, "r");
    if ( fid == NULL ) {        /* never forget this */
        printf ( "Can't open translation file: %s\n", name );
        exit (1);
    }

    while (TRUE) {
        n = fscanf ( fid, "%c %c", &old, &new );
        if (n <= 0)
            break;
        translation [old] = new;
    }
}
```

Procedure to convert the input file

- the convert_file procedure reads the characters in the input file one at a time and converts them to the new character encoding

- this procedure is basically a loop that read characters to the end of file, again we can use our standard file reading template

```
void convert_file ( char translation [256] ) {
    char c;

    while (TRUE) {
        c = getchar();
        if (c < 0)        /* EOF or getchar failure   * /
            break;
        putchar (translation [c]);
    }
}
```

- we didn't produce invariants for the loops that read the two input files. In general these invariants don't make much sense, they just state that the file is read.

The only time we really need an invariant on a file read is when there is a significant amount of processing for each item read

- for convert_file we could produce an invariant.

This invariant would state that the total characters in the output file would be equal to the total characters read from the input file. Also that the i'th character of the output file is the translation of the i'th character of the input file

- loop termination is assured, because the input files are of finite length, so the break statement will eventually be executed.