## C Programming Text I/O in C, Managing and Using Arrays

This assignment involves managing the allocation, use and deallocation of arrays in support of a simple algorithm for encrypting text. The algorithm depends on a simple, but powerful, mathematical concept.

A *permutation* is a one-to-one function from a set of nonnegative integers {0, 1, 2, …, n} to itself. We will describe permutations, not by using an algebraic formula, but by means of a table. Here's a permutation on the set {0, 1, …, 18}:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
14 11  6  7  8 12 13 15  0  4 16 17 18 10  1  2  3  5  9
```

If we call the permutation $P()$, then $P(0) = 14$, $P(1) = 11$, and so forth. We say that $P$ *maps* 0 to 14, 1 to 11, and so forth.

Now, since permutations are one-to-one, every permutation has in *inverse*. The inverse, $P^{-1}$, reverses the mappings that $P$ defines. So, $P^{-1}$ maps 14 to 0, 11 to 1, and so forth; here's the full specification for $P^{-1}$:

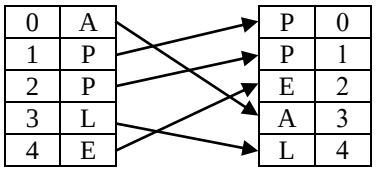
```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
 8 14 15 16  9 17  2  3  4 18 13  1  5  6  9  7 10 11 12
```

### Text Encryption

The translation of text from clear form to an encrypted form and back is a common problem in computing. A simple approach is to simply rearrange the characters of the original text, moving each character from its original position to the position defined by a permutation. This does impose a restriction, in that we must process the text in "chunks" of characters, whose length is determined by the domain of the permutation we use. Consider the permutation:

```
0  1  2  3  4
3  0  1  4  2
```

The string “APPLE” would be rearranged into the string “PLEAP”:




What about decrypting the text above? For the permutation given above, the inverse permutation is:

```
0  1  2  3  4
1  2  4  0  3
```

Apply the inverse permutation to the scrambled string “PPEAL”; you should get back the original string “APPLE”.

For this project, you will write a program that is capable of both encrypting text and decrypting text, by applying a given permutation to "chunks" of characters taken from the original text.

**Input file description and sample:**

Your program **must** read its input from a file formatted exactly as described below.

The first line of input specifies how many elements are in the domain of the permutation you will be using; in this case there are 10 elements, so the domain is the set {0, 1, …, 9}. The number of elements will vary.

The next two lines of the input file specify the permutation that was used to create the given text sample. The first of those lines just lists the elements of the domain in ascending order (you can ignore this data); read the values on the second of these lines into an appropriate array. Because the size of the domain varies, and we are not specifying a maximum value for this, you must dynamically allocate an array of exactly the right size to hold the permutation.

The next line of input specifies the number of characters in the text you will be processing. This number will also vary, and you must allocate an array of just the right size to hold the text sample.

The text to be processed is on the next line of the input file, and extends to the end of the file. In the samples below, the file contents are wrapped to fit the page width, but in the actual input files the text to be processed will all be on a single line.

You may assume that all the input values will be logically correct. For instance:

```
20
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
  0  2 19 13 15 16 14  1  8 17  4 18  3 10  5 12  6  7  9 11
302
Our loyalty is due entirely to the United States. It is due to the
President only and exactly to the degree in which he efficiently serves
the United States. It is our duty to support him when he serves the United
States well. It is our duty to oppose him when he serves it badly. --
Theodore Roosevelt
```

The given text may be clear, as shown above, or encrypted:

```
14
  0  1  2  3  4  5  6  7  8  9 10 11 12 13
  0  8  7 10  5  6 12  9 11 13  1  2  3  4
186
Hw uatdol e thm swhi onlake iesuy ecr,bert aud ger vmustenems  eyfn
hirerspposinom o; fe ivh i forohtss di ulatetseae ltbiy, hsrpca deeehes nw
lat ilrt theh mtoeislach foha T msP. --anei
```

That does not make any difference to your logic, since you just need to apply the given permutation to the given text.

There are no specified limits on the size of the domain of the permutation, or on the length of the text to be processed. You are expected to allocate suitable arrays dynamically, and to deallocate those arrays when you are done with them. Part of the grading of the assignment will take into account how efficiently you do this.

Your program must be written so that it will detect the end of each text sample (which is not the same as the end of the input file) correctly.

Note: we advise that you do not rely on logic to detect the end of the file. First, because the quotation is followed by a newline character, which is not included in the quotation. Second, because the feof() test does not work the way you may expect (Google for discussions, if you want details). Third, because you are given the exact number of characters in the quotation, and your array should be sized to match that (plus a terminator, of course), and there is no logical reason to continue reading after that many characters have been read.

**Output description and sample:**

Your program must write output data to a file formatted exactly as described below. The first two lines of output echo the permutation given in the input file. That's followed by a single blank line; blank lines contain only a newline character.

The next line shows the original input text, unmodified. That is followed by a single blank line.

The next three lines are simply column indices; a function that will write these is supplied with the base code for the assignment.

The final line of the output file shows the processed text produced by applying the given permutation to the given text.

The sample output files shown below correspond to the input data files given above:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
  0  2 19 13 15 16 14  1  8 17  4 18  3 10  5 12  6  7  9 11

Our loyalty is due entirely to the United States. It is due to the President only. . .

00000000000000000000000000000000000000000000000000000000000000000000000000000000. . .
00000000001111111111222222222233333333334444444445555555555666666666677777777778. . .
01234567890123456789012345678901234567890123456789012345678901234567890123456890. . .

Oauiy uel sed ylot rn th  nitteeUryelotids  Isdu.ei  teat tStPodsn orneltt heei y. . .
```

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13
  0  8  7 10  5  6 12  9 11 13  1  2  3  4

Hw uatdol e thm swhi onlake iesuy ecr,bert aud ger vmustenems  eyfn hirerspposino. . .

00000000000000000000000000000000000000000000000000000000000000000000000000000000. . .
00000000001111111111222222222233333333334444444445555555555666666666677777777778. . .
01234567890123456789012345678901234567890123456789012345678901234567890123456890. . .

He that would make his own liberty secure, must guard even his enemy from oppress. . .
```

In the output samples, the lines are not wrapped to fit the page width, in order to make the purpose of the column indices clearer.

**Supplied Code:**

Download the supplied tar file and unpack it in a CentOS 7 directory. You will find the following files:

```
driver.c          test driver
PermuCryptor.h*   header file for primary function
PermuCryptor.c    C shell file for primary function and private helpers
IOFunctions.h*    header file for I/O functions
IOFunctions.c     C shell file for I/O functions and private helpers
Generator.h*      header file for test case generator
Generator.o*      64-bit Linux binary for test case generator
Grader.h*         header file for grading function
Grader.o*         64-bit Linux binary for grading function
runValgrind.sh    a bash script to simplify your use of Valgrind
```

Do not modify the files marked with an asterisk (*). You may modify the driver file during your testing, but we will use the original version when grading.

Compile the code with the command:

```
gcc -o driver -std=c99 -Wall -ggdb3 driver.c PermuCryptor.c IOFunctions.c Generator.o Grader.o
```

Invoke the driver as:

```
driver OPTION [-repeat]
    OPTION is either -encrypt or -decrypt, and must be present
```

If invoked without -repeat, the program will choose a random quotation from a pool of quotations, and use that for testing. Invoked in this manner, the program will create the following files:

| | |
|---|---|
| `textIn.txt` | input data, formatted as described in the spec |
| `textOut.txt` | corresponding output, formatted as described in the spec |
| `seed.txt` | random seed value generated from the system clock |
| `scoreInfo.txt` | score information |

You will implement four functions, in two different C source files. In the file `PermuCryptor.c`, implement the primary function for the assignment:

```
/**  Apply given permutation to given quotation, storing results into
 *   processedText[].
 *
 *   Pre:   perm points to an array holding the given permutation in perm[0:permLength-1].
 *          rawText points to a C-string holding the quotation to be processed
 *          processedText points to an array large enough to hold the processed quotation
 *   Post:  the array pointed to by processedText is a C-string holding the rearranged
 *          quotation
 */
void processText(const int* const perm, int permLength, const char* const rawText,
                 char* processedText);
```

In the file `IOFunctions.c`, implement the required I/O functions:

```
/**  Reads the quotation from the input file.
 *
 *   Pre:   In is open on an input file compliant with the specification.
 *          In is positioned at the beginning of the fourth line of that file.
 *          ppText points to a char*.
 *   Post:  *ppText points to a C-string holding the given quotation.
 *   Returns:  the number of characters in the quotation.
 *   Note:
 *     This function must allocate the array to hold the quotation.
 */
int  readText(FILE* In, char** ppText);


/**  Reads the permutation from the input file.
 *
 *   Pre:   In is open on an input file compliant with the specification.
 *          In is positioned at the beginning of that file.
 *          ppPerm points to an int*
 *   Post:  *ppPerm points to an int array holding the given permutation
 *   Returns:  the number of elements in the domain of the quotation
 *   Note:
 *     This function must allocate the array to hold the permutation.
 */
int  readPermutation(FILE* In, int** ppPerm);
```

```
/**  Writes the permutation to file, formatted as specified.
 *
 *   Pre:   Out is open at the beginning of an output file.
 *          *pPerm holds the permutation
 *          The permutation's domain is 0..permLength-1.
 */
void writePermutation(FILE* Out, const int* pPerm, int permLength);
```

Each of these functions is called from the function `main()` in the test driver file, `driver.c`.

You must not change the function interfaces, and you must not modify any of the supplied header files. You will only be submitting your completed versions of `PermuCryptor.c` and `IOFunctions.c`, so changes to any other files will not be available when your solution is graded.

You may write as many helper functions, in either file, as you see fit. Any such functions must be declared as `static`, in the relevant `.c` files (not the `.h` files).

### Incremental development and hints:

First of all, you must allocate the arrays for the permutation and the text dynamically, because you will not know how long they need to be until your code is reading the input file. You should use C-strings for the text; that is, you should use an array that allows room for one extra character (a string terminator). We suggest this is a good time to use the function calloc(), since that will zero out all the elements in the array when you allocate it; this way, your string always has a terminator after last character.

You'll find that it's easier and faster to produce a working program by practicing incremental development. In other words, don't try to solve the entire problem at once. First, develop your design. When the time comes to implement your design, do it piece by piece. Here's a suggested implementation strategy for this project. As usual, verify and correct as necessary after each addition to your implementation.

- First, focus on reading and storing the permutation; this is easy to test by reading the permutation from the input file and then writing your stored permutation to the output file (which is eventually necessary anyway). You should run the driver once to generate a test case, then run it with `-repeat` in order to debug your code.
- Second, focus on reading a text sample. Use the test case generated by the driver, read the text and store it into an array, and then echo the array contents to an output file. The hardest part of this may be to stop reading at the correct time; you may find the `fgetc()` function to be useful since it does return whitespace characters, and it returns `EOF` when you've reached the end of the file.
- Third, implement the code to translate a single permutation-length chunk of the text (you're now storing correctly) using the given permutation. This shouldn't require much modification of the earlier code since you can use the same output code here as well. Do not try to put the transformed code into the same array as the original text. Test this thoroughly. The test generator can create fifty different text samples for your use.
- Fourth, extend your logic to handle successive permutation-length chunks of text, and stop when there are fewer than that many characters left.
- Finally, determine how to handle the last chunk of text, which is likely to contain fewer characters.

Now you have a substantially complete program. At this point, you should clean up your code, eliminating any unnecessary instructions and fine-tuning the documentation you already wrote. Always test again after making any modifications, even if all you've done is edit for comments!

You can store the given permutation *P* in an array, say `A[]`. Then `A[k] == j` if and only if the permutation *P* maps k to j. Another way of saying this is that `A[k]` is the location to which the character at position k should be moved. So, you only have to store the "second line" of *P*.

The basic model of execution is indicated by the code in `driver.c`:

- get the permutation and its length from the input file
- get the quotation from the input file (as a C-string, so you actually don't need to store its length)
- process the quotation, in permutation-length chunks, storing each rearranged chunk into a second array
- log results to the output file

The basic model of the processing here is something like:
Read the discussion of encrypting and decrypting at the end of this specification for more suggestions.

### What to Submit

For this assignment, you must organize your source code in two of the files we have supplied: `PermuCryptor.c` and `IOFunctions.c`. You will submit a tar file containing those two C source files and nothing else.

You will be allowed multiple submissions; the final one will be graded.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

### Grading

This assignment will be graded automatically, using the same grading code we have supplied, but using the same test cases for everyone. We will run multiple tests on your submission, using quotations and permutations of different lengths. Therefore, you should be sure to do the same, testing with clear text and with encrypted text.

We will also use Valgrind to check:

- whether you have, in fact, used dynamic allocation
- whether you have deallocated all the arrays properly*
- whether your solution performs any invalid reads or invalid writes, indicating that you have array bounds issues
- whether you have allocated excessively large arrays in order to avoid invalid reads and writes
- whether you have any uses of uninitialized values; this could indicate array bounds issues, or failure to properly terminate your C-strings

A penalty of up to 20% will be applied to your score if your solution exhibits any such bad behavior.

* During your testing, this may require that you modify the "clean up" section in `driver.c`. During grading, we will use a version of `driver.c` that has been correctly modified.

### Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the Curator System.
//
//    <Student Name>
//    <Student's VT email PID>
```

**We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.**

### Change Log

| Version | Posted | Pg | Change |
|---|---|---|---|
| 6.00 | TBA | | Base document. |

## Some general comments

Completing the assignment does not require understanding the following discussion; but it may broaden your understanding of the motivation for the assignment.

**Encryption:** Given a permutation P and a sample of clear (unencrypted) text $T_C$, we can apply the permutation to the clear text to obtain encrypted text $T_E$. The process is described in this section.

The scheme described on the previous page assumes that the permutation is the same length as the string of characters you are working with. That's OK for individual words, but not so good for long text samples like:

```
Since the general civilization of mankind, I believe there are more
instances of the abridgment of the freedom of the people, by gradual
and silent encroachments of those in power, than by violent and sudden
usurpations. -- James Madison
```

There are 237 characters there. While we could create a permutation of {0, 1, . . . , 236} and use it to encrypt this text, there's a simpler approach that's more efficient. We can create a shorter permutation, say of {0, 1, . . . 19} and use it to encrypt groups of 20 characters until we're done with the entire text (slight difficulty with the last chunk of text).

For instance, take the permutation:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 9 | 5 | 1 | 12 | 17 | 14 | 6 | 16 | 7 | 19 | 18 | 2 | 3 | 15 | 0 | 10 | 11 | 4 | 13 |

Now take the first 20 characters from the text given above and apply the permutation:

| S | i | n | c | e |  | t | h | e |  | g | e | n | e | r | a | l |  | c | i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | c | n | e | c | n | h |  | S | i | l |  | e | i | t | r | e |  | e | g |

then take the next 20 characters and repeat the process:

| v | i | l | i | z | a | t | i | o | n |  | o | f |  | m | a | n | k | i | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | i | f |  | i | l | i | n | v | i | n | k | z | n | t | m | o | a | o |  |

and again:

| d | , |  | I |  | b | e | l | i | e | v | e |  | t | h | e | r | e |  | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| e | I |  | t |  |  | l | e | d | , | r | e |  | a | e | h | i | v | e | v |

So the text considered so far would be encrypted as follows:

```
acnecnh Sil eitre egaif ilinvinkzntmoaoeI t  led,re aehibev
```

Continuing in this way we would obtain the complete encryption:

```
acnecnh Sil eitre egaif ilinvinkzntmoao eI t  led,re aehibevemano  nres
ofecirts een hrd tof tbtiamghff p dmhee reetoeo ledunlygop a,dba
arhloani n smeettcenrcofn rooes we ,hpsti talenhyv t andbn  oiidpasu s
sond.ntueruo dia-e  -naJMmss
```

There's just one little problem. The permutation takes 20 characters at a time and the total number of characters isn't a multiple of 20. That means that when we get down to the end we don't have enough characters in the last "chunk" of input to match up with the permutation entries.

**The End Game:** When we get to the end of the sample text our logic must be altered slightly. Consider the sample input text given above. There are 237 characters in that sample and the permutation takes 20 of them at a time. That means that after we've read and processed 11 chunks of text there will be 17 characters left. That doesn't match the length of our permutation, so the approach described before won't quite work. What we need is a permutation that matches the length of this last chunk. Here's a simple (and for this project, required) way to create such a permutation.

Take a permutation of {0, . . . , 19}, say the one from above:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 9 | 5 | 1 | 12 | 17 | 14 | 6 | 16 | 7 | 19 | 18 | 2 | 3 | 15 | 0 | 10 | 11 | 4 | 13 |

We need a permutation of {0, . . . , 16} to handle 17 characters. We may construct such a permutation from the one above in the following way. Let $k_1$ be in {0, . . . , 19}, the domain of the original permutation. Let $k_2$ be the value the original permutation maps $k_1$ to. If $k_2$ is in {0, . . . , 16} just keep that value. That takes care of all but 5, 10, and 11 above. If $k_2$ is bigger than 16, look at the value the original permutation maps $k_2$ to; call that value $k_3$. If $k_3$ is in the range {0, . . . , 16}, then let the new permutation map $k_1$ to $k_3$. If $k_3$ is bigger than 18, continue the process by looking at the value the original permutation maps $k_3$ to. Eventually you have to get a value that's in the range {0, . . . , 16}; when you do, let the new permutation map $k_1$ to that.

Applying this idea to the permutation given above, we get:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 9 | 5 | 1 | 12 | 11 | 14 | 6 | 16 | 7 | 13 | 4 | 2 | 3 | 15 | 0 | 10 |

Why? Well, using the original permutation given above, we have:

| | | | | |
|---|---|---|---|---|
| 5 | → | 17 | → | 11 |
| 10 | → | 19 | | 13 |
| 11 | → | 18 | → | 4 |

Now, the last chunk would be the text shown below, encrypted as shown:

| | - | - | | J | a | m | e | s | | M | a | d | i | s | o | n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| o | | d | i | a | - | e | | | - | n | a | J | M | m | s | s |