

User-defined Functions and Arrays

Permutation-Based Cryptography

The translation of text from clear form to an encrypted form and back is a common problem in computing. One simple technique for encrypting text is based on the mathematical notion of a permutation. A permutation of the integers 0 through $N-1$ is simply a one-to-one function whose domain and range are both the set of integers $\{0, 1, 2, \dots, N-1\}$. In other words, a permutation transforms each integer in the set $\{0, 1, 2, \dots, N-1\}$ into another integer in the same set, with no two integers being transformed into the same result. For example, here is a permutation of $\{0, 1, 2, 3, 4\}$:

0	→	3
1	→	4
2	→	0
3	→	1
4	→	2

This permutation can be used to encrypt a sequence of five characters by moving each character from its original position to the position defined by the permutation. For example, the sequence "APPLE" would be translated into the string "PLEAP":

0	A	→	P	0
1	P	→	L	1
2	P	→	E	2
3	L	→	A	3
4	E	→	P	4

Note that we number positions starting at zero just as in C++ arrays — that's a hint of things to come. What about decrypting the text above? Well, each permutation has an inverse, another permutation that does the exact opposite of the original permutation. For the permutation given above, the inverse permutation is:

0	→	2
1	→	3
2	→	4
3	→	0
4	→	1

Apply the inverse permutation to the scrambled sequence "PLEAP"; you should get back the original sequence "APPLE". That's the key point about encryption/decryption using a permutation:

Apply the permutation, and then apply its inverse, and you get back where you started.

That means we can use a permutation to encrypt a sequence and then use the inverse of that permutation to decrypt the encrypted sequence and get back the original.

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

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 above 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:

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them.
```

There are 199 characters there, including the newlines. While we could create a permutation of $\{0, 1, \dots, 198\}$ and use it to encrypt this text, there's a simpler approach. We can create a shorter permutation, say of $\{0, 1, \dots, 19\}$ and use it to encrypt "chunks" of 20 characters until we're done with the entire text (aside from a slight problem 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
6	13	1	15	8	3	11	7	9	12	10	16	14	17	18	4	19	0	2	5

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

T	o		b	e	,		o	r		n	o	t		t	o		b	e	:
b		e	,	o	:	T	o	e	r	n			o	t	b	o		t	

then read the next 20 characters and repeat the process:

	t	h	a	t		i	s		t	h	e		q	u	e	s	t	i	o
t	h	i		e	o		s	t		h	i	t	t		a	e	q	u	s

and again (note that the newline character at the end of the first line is treated just like any other character):

n	:	\n	W	h	e	t	h	e	r		'	t	i	s		n	o	b	l
o	\n	b	e		l	n	h	h	e		t	r	:	t	W	'	i	s	n

So the text considered so far would encrypt as follows:

```
b e,o:Toern  otbo t thi eo st hitt aequso
be lnhhe tr:tW'isn
```

Continuing this way we would obtain the complete encryption:

```
b e,o:Toern  otbo t thi eo st hitt aequso
be lnhhe tr:tW'isn  s tuehnemt rniid onedT  fe
lhsfnrigsaorusgsaowfo rtouraeaoku e et,On
f rrtotera  gsanaiatms asb yod obrlsuef
t,An .p
iemogs nnep odth
```

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 199 characters in that sample and the permutation takes 20 of them at a time. That means that after we've read and processed 9 chunks of text there will be 19 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 $\{0, \dots, 19\}$, say the one from above:

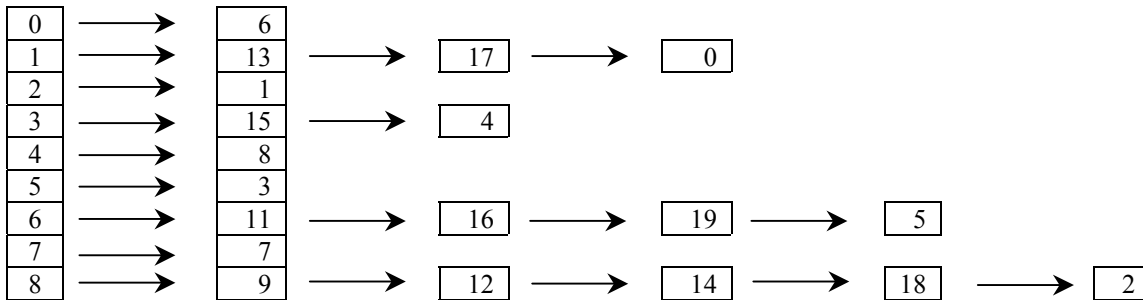
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
6	13	1	15	8	3	11	7	9	12	10	16	14	17	18	4	19	0	2	5

Let's say we need a permutation of $\{0, \dots, 8\}$ to handle 9 characters. We may construct such a permutation from the one above in the following way. Let k be in $\{0, \dots, 8\}$. Let k' be the value the original permutation maps k to. If k' is in $\{0, \dots, 8\}$ just keep that value. That takes care 0, 2, 4, 5, and 7 above. If k' is bigger than 8, we look at the value the original permutation maps k' to; call that value k'' . If k'' is in the range $\{0, \dots, 8\}$, then let the new permutation map k to k'' . If k'' is bigger than 8, continue the process by looking at the value the original permutation maps k'' to. Eventually you have to get a value that's in the range $\{0, \dots, 8\}$; when you do, let the new permutation map k to that.

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

0	1	2	3	4	5	6	7	8
6	0	1	4	8	3	5	7	2

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



Now, once we've constructed the right size permutation we can apply it to encrypt the last chunk of the original text sample. Recall the text sample from the previous page. The last text chunk is 19 characters long (including the newline at the end), so we need a permutation of $\{0, \dots, 19\}$. Using the technique just shown above, we get the permutation:

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

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

o	p	p	o	s	i	n	g		e	n	d		t	h	e	m	.	\n
.	p	\n	i	e	m	o	g	s		n	n	e	p		o	d	t	h

Decryption: The decryption problem: given a sample of encrypted text, T_E , we wish to obtain the corresponding clear text T_C . The process depends upon one simple fact:

Given a clear text sample T_C and a permutation P , there is only one encrypted text T_E that can be obtained, if the process described above is used.

Thus, decrypting text is, mathematically speaking, the inverse of the encryption operation. Fortunately, we may solve the decryption problem in a very simple manner. Every permutation P has an inverse permutation, usually denoted by P^{-1} . If we know P^{-1} , we may decrypt an encrypted text sample T_E by applying the inverse permutation to T_E in exactly the way the original permutation P was applied to create T_E .

If we know P , we may find P^{-1} easily: if P maps i to j , then P^{-1} will map j to i . (Take another look at the discussion on the first page of this spec if that wasn't clear.)

Let's take another look at the encrypted version of that quotation from *Hamlet*:

```
b e,o:Toern otbo t thi eo st hitt aequso
be lnhhe tr:tW'isn s tuehnemt rniid onedT fe
lhsfnrigsaorusgsaowfo rtouraeaoku e et,On
f rrtotera gsanaiatms asb yod obrlsuef
t,An .p
iemogs nnep odth
```

We obtained this using the permutation:

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

The inverse permutation would be:

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

Apply the inverse permutation to the first chunk of the encrypted text:

b		e	,	o	:	T	o	e	r	n			o	t	b	o		t	
T	o		b	e	,		o	r		n	o	t		t	o		b	e	:

Continue in this way, applying the inverse permutation to successive chunks until all the encrypted text has been processed and you'll obtain the original text.

Aside: What makes this technique effective is that to decrypt text you have to know the permutation P that was used to encrypt the original text. Given the encrypted text above, you don't even know how long the permutation P was. Even knowing that P was of length 20, there are $20! = 20 \cdot 19 \cdot 18 \cdot 17 \cdot \dots \cdot 3 \cdot 2 \cdot 1$ possible permutations of that length. The amount of time required to try each one of those (the brute force approach) is prohibitive if you need the answer quickly. Without knowing the length of P, the amount of time for a brute force solution becomes astronomical. It can be made even worse if the permutation itself is varied from chunk to chunk of input text.

From the back of an envelope: $20! \approx 2.43 \times 10^{18}$ — if it took one one-thousandth of a second to apply a permutation and then determine whether the resulting text made sense, processing this many permutations would take about 77,146,816 years. Of course, faster hardware would help, but it would still take over 77,000 years if one million permutations could be processed per second. Not to mention it's possible that applying the wrong permutation might produce sensible, but incorrect, text.

Input:

Your program **must** read its input from a file named `CryptoData.txt` — use of another input file name will result in a score of zero. The first two lines of the input file specify the permutation to be used. You should treat the first line as a comment (ignore it) and read the values on the second line into an appropriate array — the permutation will always be of length 20.

The next section of the input file is a data header, containing five lines of labels and data describing the text sample to be processed. The first two lines of the data header are a comment and a line containing a single character specifying whether the text sample that follows is to be encrypted ('E') or decrypted ('D') using the given permutation. The next two lines are another comment and a line containing the number of characters included in the text sample, counting spaces and newlines. The last line of the data header is just a comment.

The text sample to be encrypted or decrypted follows the data header. You may assume that the text sample will contain exactly the specified number of characters. There will be one or two blank lines after the last character of the text sample (not counted as a part of the text sample).

After that, there will be a second data header section and another text sample to be processed. Each input file will contain exactly two text samples for processing, one to decrypt and one to encrypt (in random order).

The values on each line will be separated by whitespace. You may assume that all the input values will be logically correct. For instance:

```

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

Option: D
Number of characters: 220
ouhT rWodhga iBnmobcoee ian museotDnn,ndt
h sd,A po uoopebngoefoa bi wmon onr
Ytn wty ,el t Iilrhlate.Bemy sfr eo by
oto ardI ylrhwmwi sike.yoneh a,dl L cdfMfAaneauddd
;n m hmb art ei ichttfsrs Hio,uh!e"eo"dl ng

Option: E
Number of characters: 141
If thou didst ever hold me in thy heart
Absent thee from felicity awhile,
And in this harsh world draw thy breath in pain,
To tell my story.

```

Note that you must **not make any assumptions** about the number of lines of data in the input file; the input file will comply with the specification given here. Your program must be written so that it will detect the end of each text sample correctly.

What to Calculate:

There are almost no calculations in the numerical sense. You will write a program to read the input file and use the given permutation to encrypt or decrypt (as specified) each of the two text samples. The processed text will be written to an output file, formatted as described in the next section.

Output:

The first line of your output should include your name only; the second line should include the project title as shown in the sample file. The next line should be a blank line, followed by a delimiter line to separate the file header from the output generated when the first text sample is processed. The length and characters used in the delimiter line do not matter, but it must not be blank.

Next your output file will specify how many characters were processed and whether they were encrypted or decrypted, and then echo the permutation used, as shown. Following the header is the processed text. Do not change the capitalization of the input text, and do not add or remove any newlines within the text you are processing: if the input sample text is processed properly, you should automatically generate appropriate line breaks (newlines) when you print it. **You may generate a blank line at the end of the processed text, or not, as you like.**

Following that, there will be another, identical, section of output generated when the second input sample text is processed. A delimiter line and a line containing the exit message shown below should follow the end of that section.

Your program must write output data to a file named `CryptoOut.txt` — use of any other output file name will result in a score of zero. The sample output file shown below corresponds to the input data given above:

```

Programmer: Bill McQuain
Permutation Cryptography

+++++
Decrypting 220 characters
Using:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
         3  9  0  1 10  2  4 15 14  5 16 11 17 12  6 18  7  8 13 19
Though Birnam Wood be come to Dunsinane,
And thou opposed, being of no woman born,
Yet I will try the last.  Before my body
I throw my warlike shield.  Lay on, Macduff;
And damned be him that first cries "Hold, enough!"

+++++
Encrypting 141 characters
Using:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
         3  9  0  1 10  2  4 15 14  5 16 11 17 12  6 18  7  8 13 19
toIuieerfhs d dtvhd eo harlmh tnitye
setA eo fbn rehtefmlit iaiAd cye
ihwl,nnhi hsrds ts odrahwlr tya ei pwht arbahni
T ntlSor,om yle yt.

+++++
All done ... thank you for using PermuCrypt

```

You are not required to use this exact horizontal spacing, but your output must satisfy the following requirements:

- You must use the specified labels and exit message, and include your name in the first line as shown.
- You must arrange your permutation output in neatly aligned columns as shown; `setw(3)` is suggested.
- You must use the same ordering of the header output as shown here.
- You must print a newline at the end of the last line.

Evaluation:

Everything that you have been told about testing in class applies here. Do not waste submissions to the Curator in testing your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that are provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 100, based upon the runtime testing performed by the Curator System. We will also be evaluating your submission of this program for documentation style and a few good coding practices. This will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

Read the *Programming Standards* page on the CS 1044 website for general guidelines. You should comment your code in the same manner as the code given for the first two programming assignments. In particular:

- You should have a header comment identifying yourself, and describing what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every major block of code should have a comment describing its purpose.
- Adopt a consistent indentation style and stick to it.

Your implementation must also meet the following requirements:

- Choose descriptive identifiers when you declare a variable or constant. Avoid choosing identifiers that are entirely lower-case.
- Use C++ streams for input and output, not C-style constructs.

- Use `char` arrays, of the appropriate dimension, to hold the text being encrypted or decrypted. Note: using `string` variables to do this is strictly prohibited. You may use `string` variables for other purposes.
- Use at least three arrays, including at least one of type `int` and at least two of type `char`.
- You may use file-scoped function prototypes, and you may use file-scoped constants.
- You may not use file-scoped variables of any kind.
- You must make good use of user-defined functions in your design and implementation. To encourage this, the body of `main()` must contain no more than 20 executable statements and the bodies of the other functions you write must each contain no more than 40 executable statements. An executable statement is any statement **other than** a constant or variable declaration, function prototype or comment. Blank lines do not count.
- You must write at least six functions, besides `main()`. For reference, my solution uses eight: two for input, two for output, two for text translation, one to find the inverse of a permutation, and one utility function.
- The definition of `main()` must be the first function definition in your source file. The remaining function definitions should be grouped logically.
- Function parameters should be passed appropriately. Use pass-by-reference only when the called function needs to modify the parameter. Pass array parameters by constant reference (using `const`) when pass-by-reference is not needed.
- Use named constants instead of variables where appropriate — in particular for array dimensions.
- Choose your control structures appropriately.

Submitting your program:

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* and submission link can be found at:

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

Incremental Development:

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 a single text sample. Use the sample input given with this specification, but delete the section containing the sample to be decrypted. Implement code to read the file header (permutation and sample text description) and make sure those are read correctly. Then implement code to read the sample text in 20-character chunks — echo it without any changes to be sure you've got that right and that you're handling the last, short chunk correctly.
- Second, implement the encryption code to translate the text (you're now reading 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. Test this thoroughly. A number of samples will be posted to the website.
- Third, add the logic to determine whether you're to encrypt or decrypt, if you didn't write that earlier. Then implement the decryption code. That should be a fairly simple modification of the decryption code you wrote earlier, at least if you're thinking about the process correctly. Again, you should be use the same output code whether you're printing encrypted or decrypted text to the output file.

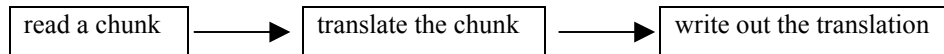
Now you have a substantially complete program. At this point, you should clean up your code, eliminating any unnecessary instructions and fine-tuning the documentation you already wrote. Check your implementation and output again to be sure that you've followed all the specifications given for this project, especially those in the Evaluation section above. At this point, you're ready to submit your solution to the Curator.

Implementation Hints:

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 .

You can also think of the inverse permutation P^{-1} in terms of the array $A[]$: if $A[k] == j$ then P^{-1} would map j to k ; said another way, the character that’s at position j should really be at position k .

About the requirement you declare and use enumerated type. Notice that you read the chunks of the text sample the same way whether you’re going to encrypt it or decrypt it, and the same goes for writing the translated chunks to the output file. If you understand the hints above, encrypting and decrypting are (different but) very similar. The basic model of the processing here is something like:



All you have to do is select the appropriate type of translation (encryption or decryption).

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 header comment for your program:

```
// 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,  
//   or any other unauthorized source, either modified or unmodified.  
//  
// - If any C++ language code or documentation used in my program  
//   was obtained from another 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>
```

Failure to include this pledge in a submission is a violation of the Honor Code.