

User-defined Functions and Arrays

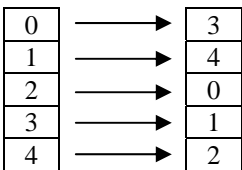
This programming assignment uses many of the ideas presented in topics 3 through 18 of the course notes, so you are advised to read those carefully. Read and follow the following program specification carefully.

You will receive two scores on this project: the runtime testing score from the Curator and the software engineering score you receive for following the instructions in the Programming Standards section below.

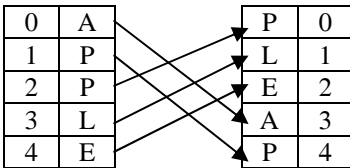
The Program Specification:

Positional Permutation Cryptography

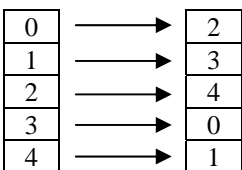
Introduction: The translation of text from clear form to an encrypted form and back is a common problem in computing. One simple technique for encrypting text is based on the mathematical notion of a permutation. A permutation of the integers 0 through N-1 is simply a one-to one function whose domain 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 $\{0, 1, 2, 3, 4\}$:



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



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:



Apply the inverse permutation to the scrambled string “PLEAP”; you should get back the original string “APPLE”. That’s the key point about the inverse of 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 string and then use the inverse of that permutation to decrypt the encrypted string 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 character strings 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 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:

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 that's more efficient. We can create a shorter permutation, say of $\{0, 1, \dots, 19\}$ and use it to encrypt groups of 20 characters until we're done with the entire text (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 $\{0, \dots, 20\}$, say the one from above:

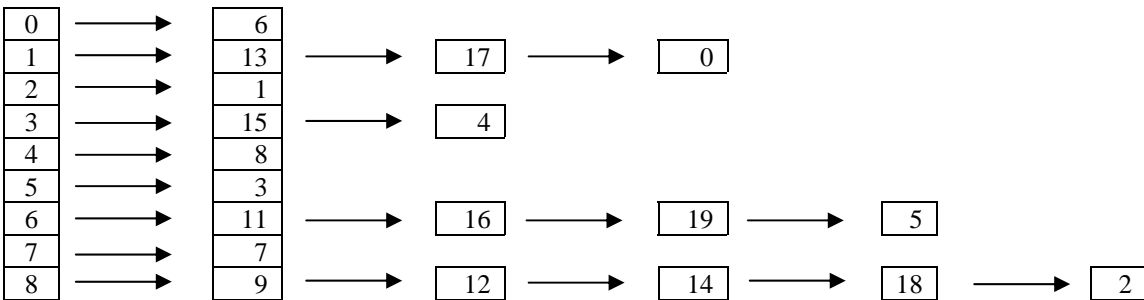
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
6	13	1	15	8	3	11	7	9	12	10	16	14	17	18	4	19	0	2	5

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, 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.

* 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 file description and sample:

Your program **must** read its input from a file named `Scrambled.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 that was used to encrypt the text. 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 three 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 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 newline after the last character of the text sample (not counted as a part of the text sample).

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
                7 15 8 1 3 16 2 17 11 19 18 4 14 0 10 5 12 6 9 13
// Number of characters:
202
// Text:
bluiekfiagn'aL s tw, saphlpyadr oowao asrTn
dud h eratttssp esotnut urohrih ee
esn dsithhna gAtn oha eIo tr. mdr elbesay ona iTtla
d fiisdolnuof lt ,uouir nfaySnidnd
 ifygnag ocnB tetMg h-i.
h
```

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 decrypt the given text sample. The processed text will be written to an output file, formatted as described in the next section.

Output description and sample:

The first line marks the beginning of the output generated when the first text sample is processed. It doesn't matter what symbol(s) you use for this line, but it must not be blank.

Next your output file will contain a three-line data header describing the processed text block, formatted as shown below. The last line of that header should be blank, as shown. Note that the permutation displayed here is the inverse of the one that was given in the input file. 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. There should be a newline, a line of delimiters, and another newline immediately after the end of the processed text whether the processed text ends with a newline or not.

Your program must write output data to a file named `Unscrambled.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:

```

+-----+
Decrypting 202 characters
Using:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
        13  3  6  4 11 15 17  0  2 18 14  8 16 19 12  1  5  7 10  9

Life's but a walking shadow, a poor player
That struts and frets his hour upon the stage
And then is heard no more.  It is a tale
Told by an idiot, full of sound and fury
Signifying nothing.  - MacBeth

```

```

+-----+

```

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.
- 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.

Programming Standards:

You'll be expected to observe good programming/documentation standards. All the discussions in class about formatting, structure, and commenting your code will be enforced. A copy of *Elements of Programming Style* is included with the course notes — if you don't have a copy I strongly suggest you read the on-line edition (available from the course website). Some specific requirements follow.

Documentation:

- You must include header comments specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- Each user-defined function must have a valid C++ prototype and the definition of each must be preceded by a block comment, explaining in one sentence what the function does, and listing each parameter to the function and explaining its purpose.
- You must use indentation and blank lines to make control structures like loops and if-else statements more readable. The second payroll code example from the course notes appendix is a good guide.

Implementation:

- 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 30 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 twelve: three for input, four for output, three for text translation, one for control, 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.
- At least one function must return an `int` value.
- Use named constants instead of variables where appropriate — in particular for array dimensions.
- Choose your control structures appropriately.

Your submission that receives the highest score will be graded for adherence to these requirements, whether it is your last submission or not. If two or more of your submissions are tied for highest, the **earliest** of those will be graded. Therefore: implement and comment your C++ source code with these requirements in mind from the beginning rather than planning to clean up and add comments later.

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 the given text sample. 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 decryption 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.

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 Programming Standards 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 in the encrypted text should really be at position k in the decoded text.

Testing:

At minimum, you should be certain that your program produces the output given above when you use the given input file. However, verifying that your program produces correct results on a single test case does not constitute a satisfactory testing regimen.

Several additional input/output file examples will be posted on the website. You may use those for additional testing.

You could make up and try additional input files as well; of course, you'd have to determine by hand what the correct output would be.

Honor Code:

In addition to the limits set in the posted course contract, students are expressly forbidden from providing the following information to the CS 1044 listserv, newsgroup, or by private e-mail to other students in this course:

- any description (C++, English or any other language) of C++ code to apply the permutation to decrypt text

You **are** allowed to describe **logically** how these things work, just as in this specification, and using either the examples here or your own examples. As usual, it **is** permissible to post an input file and the corresponding output file to the listserv. Be advised that such postings may be incorrect.

Pledge:

Beginning with Project 2, each of your submissions to the Curator 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 Automated Grader.
```

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