# CIS7 Unit 4 Lab: Counting and Permutations

std::next\_permutation in Algorithm STL

Lexicographically **next\_permutation** in C++

Implement next permutation, which rearranges numbers into the lexicographically next **greater permutation of numbers**. If such arrangement is not possible, it must rearrange it as the **lowest possible order** (ie, sorted in ascending order).The replacement must be in-place and use only constant extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

1,2,3 → 1,3,2

3,2,1 → 1,2,3

1,1,5 → 1,5,1

A **lexicographical comparison** is the kind of comparison generally ***used to sort words alphabetically in dictionary. It involves comparing sequentially the elements that have the same position in both ranges against each other until one element is not equivalent to the other***. The result of comparing these first non-matching elements is the result of the lexicographical comparison.

Given a word, find lexicographically greater permutation of it. For example, lexicographically next permutation of “gfg” is “ggf” and next permutation of “acb” is “bac”.

C++ Algorithm next\_permutation() function is used to reorder the elements in the range [first, last) into the next lexicographically greater permutation. A permutation is specified as each of several possible ways in which a set or number of things can be ordered or arranged. It is denoted as N! where N = number of elements in the range.

The **function is *next\_permutation(a.begin(), a.end()).*** ***It returns ‘true’ if the function could rearrange the object as a lexicographically greater permutation. Otherwise, the function returns ‘false’***. Transform range to next permutation.

***Rearranges the elements in the range [first,last***) into the next lexicographically greater permutation.

The ***first such-sorted possible permutation (the one that would compare lexicographically smaller to all other permutations) is the one which has all its elements sorted in ascending order***, and the largest has all its elements sorted in descending order.

A permutation is each one of the **N! possible** **arrangements** the elements can take (**where N is the number of elements in the range**). Different permutations can be ordered according to how they compare lexicographically to each other.

***If the function can determine the next higher permutation, it rearranges the elements as such and returns true***. If that was ***not possible*** (because it is already at the largest possible permutation), it ***rearranges the elements according to the first permutation*** (sorted in ascending order) ***and returns false.***

**Example 1:** Permutation example using array. Program contains sort() and next\_permutation() functions.

nExample 1: next_permutation in C++. Program contains array { 1, 2, 3, 4)
Program uses sort to organize values
Program uses next_permutation to rearrange the values based on comparison.

1. Using the above example 1 of next\_permutation function, create a program in C++ that includes an array of **4 elements** and uses the next\_permutation function.
2. Calculate the variation of arrangement in Example 1 program, using N!.

4! = 4 \* 3 \* 2 \* 1 = 24 arrangements

1. Input the program in IDE, then run the program. **Describe the arrangement of the elements. Provide screen capture.**

Each time a digit is being tested the following digits are moved until they are in descending order. So, if we are testing the values 1234 for permutations and are looking at the digits after 2 which in this case is 34 the next permutation would be 1243. Now that we are in descending order we know that the permutations for 2 are done and we can move onto the next digit from the possible digits left in the index that 2 is at. This ultimately leaves us with the final permutation being in descending order so in this case 4321 and this is how we know we have exhausted all possible permutations.

A screenshot of a computer

Description automatically generated

1. Replace the first element and the second element with HIGHER values than the rest of the elements. For example, **int myints** contains {11, 12, 13, 14}, the new array would be {**21, 22**, 13, 14}. Run the program and **describe the outcome permutations. What are the main differences? Provide screen capture.**

Since we are sorting before doing any of the permutations we still go from being in ascending order in the first permutation to descending order in the last permutation. The only difference between this and part b are the values in the permutations since the process remains the same. The way the permutations are calculated remains the same as described in part b and there are still 4! or 24 permutations with this set of numbers. If we hadn’t sorted we would have had an incorrect number of permutations because we could have reached a permutation where all digits are descending prematurely leading to the program thinking it had reached the last possible permutation while there are still permutations left.

A screenshot of a computer screen

Description automatically generated

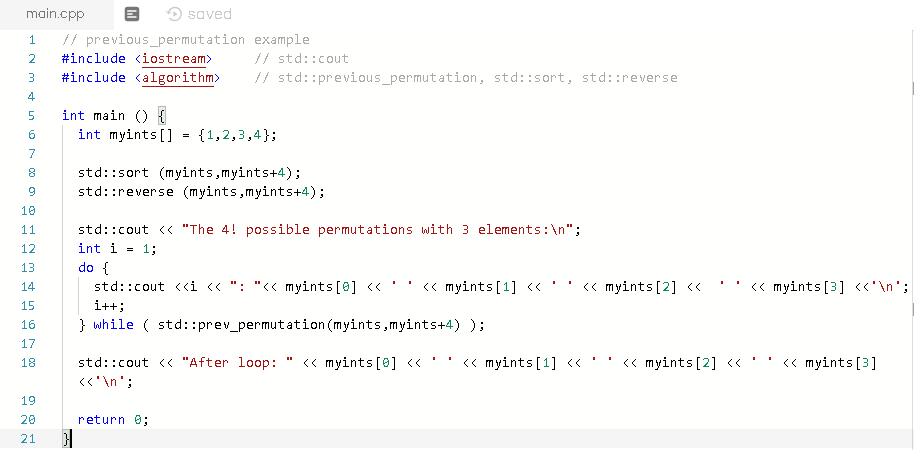
## std::prev\_permutation in Algorithm STL

It returns ‘true’ if the function could rearrange the object as a lexicographically **smaller permutation**. If the function can determine the previous permutation, it rearranges the elements as such and returns true. If that was not possible (because it is already at the lowest possible permutation), it rearranges the elements according to the last permutation (sorted in descending order) and returns false.

A permutation is each one of the N! possible arrangements the elements can take (where N is the number of elements in the range). Different permutations can be ordered according to how they compare lexicographicaly to each other;

The first such-sorted possible permutation (the one that would compare lexicographically smaller to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.

**Example 2**: C++ permutation program using previous\_permutation() function.



1. Refer to the above example 2 and create a C++ program that contains prev\_permutation function and an array of 5 elements.
2. Calculate the possible arrangement of numbers that the Example 2 program will display.

4! = 4 \* 3 \* 2 \* 1 = 24 arrangements

1. Input the example 2 program into IDE, and run the program. **Describe the arrangement of the elements. How is it different from a program using next\_permutation function, example 1? Provide screen capture**.

Each time a digit is being tested the following digits are moved until they are in ascending order. So, if we are testing the values 4321 for permutations and are looking at the digits after 3 which in this case is 21 the next permutation would be 4312. Now that we are in ascending order we know that the permutations for 3 are done and we can move onto the next digit from the possible digits left in the index that 3 is at. This ultimately leaves us with the final permutation being in ascending order so in this case 1234 and this is how we know we have exhausted all possible permutations. The only difference between prev\_permutation and next\_permutation is that prev\_permutation is looking for ascending order to know when all possible sub permutations for a digit have been exhausted while next\_permutation is looking for descending order to know when all possible sub permutations for a digit have been exhausted. Both still achieve the same permutations and same number of permutations just in a different order.

A screenshot of a computer

Description automatically generated

1. Replace the first, second and the third element with HIGHER values than the rest of the elements. For example, **int myints** contains {**10, 11, 12**, 13} the new array would be {**20, 21,** 12, 13}. Run the program and **describe the outcome permutations**. **What are the main differences compared to outcome in #2b? Provide screen capture**.

Since we are sorting and reversing before doing any of the permutations we still go from being in descending order in the first permutation to ascending order in the last permutation. The only difference between this and #2b are the values in the permutations since the process remains the same. The way the permutations are calculated remains the same as described #2b and there are still 4! or 24 permutations with this set of numbers. If we hadn’t sorted we would have had an incorrect number of permutations because we could have reached a permutation where all digits are ascending prematurely leading to the program thinking it had reached the last possible permutation while permutations are still left for it to find.

A screenshot of a computer

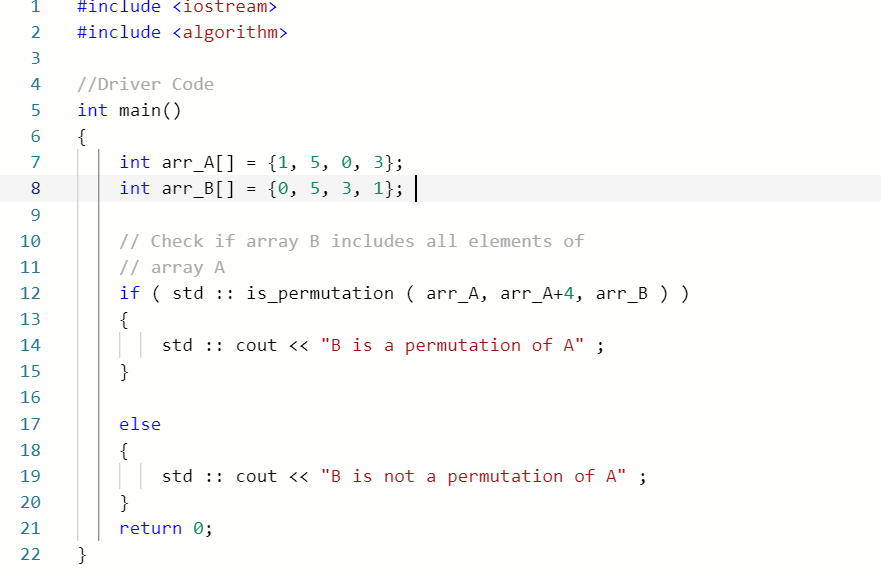
Description automatically generated

## std::is\_permutation

**The C++ function std::algorithm::is\_permutation() tests whether a sequence is permutation of other or not. It uses operator == for comparison.**

Returns true if first range is permutation of another otherwise it returns false.

**Example 3**: C++ permutation example using is\_permutation() function.



1. Following the above example 3 of is\_permutation function, create a C++ program that contains 2 arrays of **5 elements that have the same elements in different order**.
2. Calculate the number arrangements or permutation given arr\_A = {1, 5, 0, 3}.

4! = 4 \* 3 \* 2 \* 1 = 24 arrangements

1. Run the program and **describe the outcome. Provide screen capture**.

The program reports that B is a permutation of A. Since B is just a rearrangement of the same values contained within A, B must be a permutation of A.

A screenshot of a computer

Description automatically generated

1. **Change the element values of the second array to be different that the first array and run the program. Describe the outcome. Provide screen capture.**

The program reports that B is not a permutation of A. Because B now contains elements not contained within A, B cannot be a permutation of A.

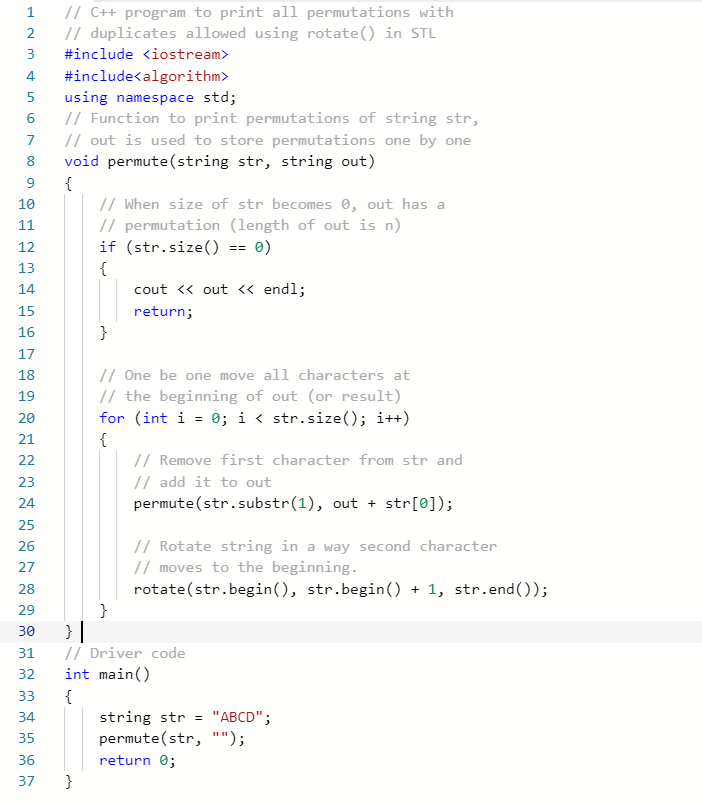
A screenshot of a computer

Description automatically generated

## Permutation String with Function

In Algorithm STL, **rotate** is a function that changes the order of the elements in the range [first,last], in such a way that the element pointed by middle becomes the new first element.

**Example 4:** C++ program that uses display permutation of an assigned string.



1. Use the above example 4 program and create the Heap’s algorithm program in C++.
2. Calculate the number of permutation that example 4 program will display.

4! = 4 \* 3 \* 2 \* 1 = 24 permutations

1. Run the program. **Provide a screen capture**. **Describe the permutation output in comparison with Example 1 program.**

The permutations generated by this program are simply generated by rotations unlike example 1 which used descending order to know when all the possible sub permutations for a digit had been exhausted. Some look close like the second where ABCD becomes ABDC but others do not follow the same descending pattern. The number of permutations is the same as example 1. Unlike example 1 the first permutation is whatever the first input was, and the last permutation is the reverse of the first permutation. The first permutation does not have to be sorted in ascending order to begin with.

A screenshot of a computer

Description automatically generated

1. Edit example 4 program to contain a string of 6 characters. **How many arrangements do you expect with 6 characters? Describe the output.**

6! = 6 \* 5 \* 3 \* 3 \* 2 \* 1 = 720 arrangements expected

Program outputs 720 permutations. The first permutation is the same as the input string and the last permutation is simply the reverse of the first permutation. Some rotations still follow the same descending pattern from example 1 while others do not follow this pattern. It appears that only 2 characters are swapped between each permutation.

Submit a document (.docx or .pdf) that contains screen answers and screen captures in Canvas.