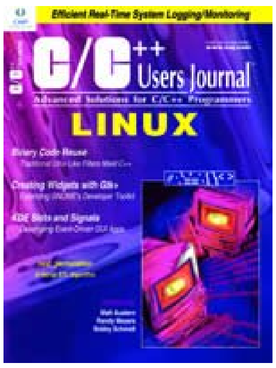
# C++ Algorithms: next_permutation()

Mar 1, 2002



**C/C++ Users Journal**
March, 2002

*Note: Thanks to Shawn McGee for pointing out an error in Figure 1. The print edition of this article in C/C++ Users Journal had an unfortunate extra line!*

My daughter's math teacher at Hockaday School in Dallas wants his sixth-grade students to enjoy their class. He's fond of sending home interesting problems that are meant to be both entertaining and enriching. As most parents probably know, this can only mean trouble!

Last week Mr. Bourek sent home a worksheet containing a set of variations on the traditional magic square. Students were given various shapes, such as triangles, stars, and so on, and asked to fill in a set of consecutive numbers at the vertices. The goal was to come up with an arrangement of numbers such that various rows, columns, and diagonals all added up to a given sum.

Kaitlin worked her way through most of the problems in fairly quick order. But the shape shown in Figure 1 managed to stump her.

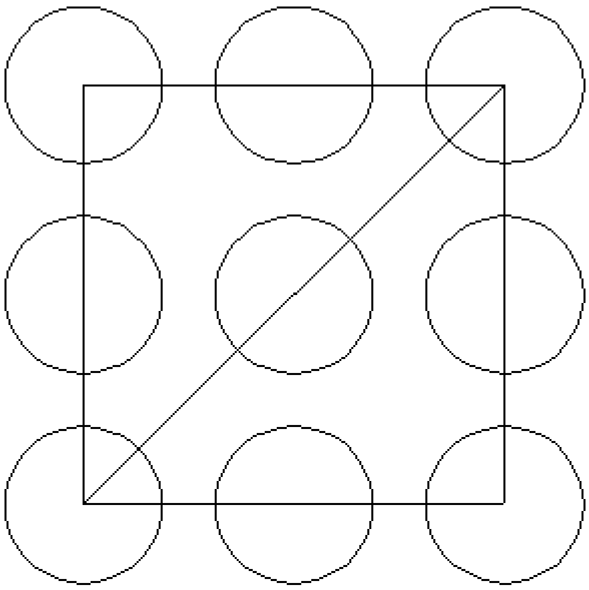


Figure 1 - Sum = 17

The problem was simple enough. All she had to do was place the numbers 1 through 9 in the nine positions of the figure so that the sum of all the straight lines was 17. Although Kate was able to knock the other problems out quickly, this one was still unsolved after fifteen minutes or so; well past the normal sixth-grade attention span.

Even worse, after another 10 minutes of my help we were no closer to a solution. That's when I decided it was time for a brute force approach. I remembered that the standard C++ library had a handy function, `next_permutation()`, that would let me iterate through all the possible arrangements of the figure with just a couple of lines of code. All I had to do was check the five different sums for each permutation and I'd have the answer in no time.

The resulting program is shown in Listing 1, and its output is given below:

```
100706 : 3 5 9 8 2 1 6 4 7
114154 : 3 8 6 5 2 4 9 1 7
246489 : 7 1 9 4 2 5 6 8 3
259937 : 7 4 6 1 2 8 9 5 3
362880 permutations were tested
```

A little quick sketching will show you that the four solutions are simply rotations and mirror images of the one true solution. You can also see that randomly putting down numbers makes the odds almost 100,000:1 against finding a solution. Not quite as bad as the lottery, but it clearly shows that random guessing isn't going to work. (My daughter asked me to give her the center position only, upon which she solved the rest of it in roughly 30 seconds.)

```
#include <iostream>
#include <algorithm>
using namespace std;

main()
```

```
{
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int count = 0;
    do
    {
        if ( ( a[0] + a[1] + a[2] ) == 17 &&
             ( a[0] + a[3] + a[6] ) == 17 &&
             ( a[2] + a[5] + a[8] ) == 17 &&
             ( a[2] + a[4] + a[6] ) == 17 &&
             ( a[6] + a[7] + a[8] ) == 17 )
        {
            cout << count <<  " : ";
            for ( int i = 0 ; i < 9 ; i++ )
                cout << a[ i ] << " ";
            cout << "\n";
        }
        count++;
    } while ( next_permutation( a, a + 9 ) );
    cout << count << " permutations were tested\n";
    return 0;
}
```

**Listing 1 - Magic.cpp**

## Magic Permutations

From this program you can see that `next_permutation()` is a handy function to have in the C++ library. In my case it meant the difference between writing an impulse program versus fiddling around with pencil and paper for another hour. What really makes `next_permutation()` interesting to me is the fact that it can generate permutations without keeping any additional information beyond the sequence being juggled. I can generate a permutation, go off and do whatever I like with it, even write the numbers out to a file and save them for later. Regardless of what I do, `next_permuation()` will always be happy to generate the next set in the series given only the previous one as input.

Just writing a function to generate permutations isn't particularly hard. One easy way to tackle the problem is with a recursive approach. If the string you want to permute is *n* characters long, you execute a loop that makes one pass per character in the string. Each time through the loop you remove character *i* from the string, and keep it as a prefix. You then print out all the permutations of the remaining substring concatenated with the prefix. How do you get the list of permutations of the substring? By recursively calling the permutation function. The only additional piece of logic you need to include is the test to see if a substring is only one character long. If it is, you don't need to call the permutation function, because you already have the only permutation of the string.

For example, to print the permutations of “abc”, you will first strip off the “a” character, and then get the permutations of “bc”. To get those permutations, you will first strip off the “b” character, and get a resulting permutation list of “c”. You then strip off the “c” character, and get a resulting permutation of “b”. The results when combined with the prefix character of “a” give strings “abc” and “acb”.

You then repeat the process for prefix “b” and substring “ac”, then for prefix “c” and substring “ab”.

Using the `string` class in the C++ standard library makes it fairly easy to implement this logic. Listing 2 shows `permute.cpp` which implements this algorithm relatively faithfully.

```cpp
#include <iostream>
#include <string>
using namespace std;

void permute( string prefix, string s )
{
    if ( s.size() <= 1  )
        cout << prefix << s << "\n";
    else
        for ( char *p = s.begin(); p < s.end(); p++ )
        {
            char c = *p;
            s.erase( p );
            permute( prefix + c, s );
            s.insert( p, c );
        }
}

main()
{
    permute( "", "12345" );
    return 0;
}
```

**Listing 2 - Permute.cpp**

This approach to generating permutations is okay, but its recursive nature makes it unattractive for use in a library. To use this in a library we would have to employ a function pointer that would be invoked from deep inside the chain of function calls. That would work, but it’s definitely not the nicest way to do it.

`next_permutation()` manages to avoid this trouble by using a simple algorithm that can sequentially generate all the permutations of a sequence (in the same order as the algorithm I described above) without maintaining any internal state information. The first time I saw this code was in the original STL published by Alexander Stepanov and Ming Lee at Hewlett-Packard. The original code is shown in Listing 3.

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last) {
    if (first == last) return false;
    BidirectionalIterator i = first;
    ++i;
    if (i == last) return false;
    i = last;
    --i;

    for(;;) {
        BidirectionalIterator ii = i--;
        if (*i < *ii) {
            BidirectionalIterator j = last;
            while (!(*i < *--j));
            iter_swap(i, j);
            reverse(ii, last);
            return true;
        }
        if (i == first) {
            reverse(first, last);
            return false;
        }
    }
}
```

Listing 3 - Function *next_permutation()* from the STL

Using this function is simple. You call it repetitively, asking it to permute a given sequence. If you start with a sequence in ascending order, `next_permutation()` will work its way through all possible permutations of the sequence, eventually returning a value of false when there are no more permutations left.

## Internals of next_permutation()

You don't need to be an STL expert to understand this code, but if you've never been exposed to this new part of the C++ standard library, there are a few things you need to know.

First, iterators (and the `BidirectionalIterator` type used here) are an STL abstraction of pointers. When looking at this code you can mentally think of the iterators as pointers. The permutation sequence is defined by iterators `first` and `last`. `first` points to the first element in the sequence, while `last` points one past the last element.

The code shown in Listing 3 also uses two other STL functions. `iter_swap()` swaps the values ponted to by its two arguments. And `reverse()` simply reverses the sequence defined by its two arguments. By convention of course, the first argument points to the start of the sequence to be reversed, and the last argument points one past the end of the sequence.

To help illustrate the workings of this algorithm, I've included a listing of a permutation sequence in Figure 2. It contains all 120 permutations of a five digit sequence. With that output example, plus Listing 3, it is fairly easy to see how this code works.

The function first does a cursory check for sequences of length 0 or 1, and returns `false` if it finds either. Naturally, sequences of those lengths only have one permutation, so they must always return false.

After passing those tests, the algorithm goes into a search loop. It starts at the end of the sequence and works its way towards the front, looking for two consecutive members of the sequence where member n is less than member n+1. These members are pointed to by iterators `i` and `ii` respectively. If it doesn't find two values that pass this test, it means all permutations have been generated. You can see this is the case in Figure 2 for the very last value, '54321'.

```
12345 12354 12435 12453 12534 12543 13245 13254 13425 13452
13524 13542 14235 14253 14325 14352 14523 14532 15234 15243
15324 15342 15423 15432 21345 21354 21435 21453 21534 21543
23145 23154 23415 23451 23514 23541 24135 24153 24315 24351
24513 24531 25134 25143 25314 25341 25413 25431 31245 31254
31425 31452 31524 31542 32145 32154 32415 32451 32514 32541
34125 34152 34215 34251 34512 34521 35124 35142 35214 35241
35412 35421 41235 41253 41325 41352 41523 41532 42135 42153
42315 42351 42513 42531 43125 43152 43215 43251 43512 43521
45123 45132 45213 45231 45312 45321 51234 51243 51324 51342
51423 51432 52134 52143 52314 52341 52413 52431 53124 53142
53214 53241 53412 53421 54123 54132 54213 54231 54312 54321
```

Figure 2 - A sequence generated by next_permutation(a)

Once iterators `i` and `ii` have been properly located, there are still a few more steps left. The next step is to again start searching from the end of the sequence for the first member that is greater than or equal to the member pointed to by `i`. Because of the previous search for `i` and `ii`, we know that at worst the search will end at `ii`, but it might end earlier. Once this member is located, it is pointed to by iterator `j`.

Once these three iterators are located, there are only two more simple steps. First, a call is made to `iter_swap( i, j )`. This simply swaps the members pointed to by `i` and `j`.

Finally, a call is made to `reverse( ii, last )`. This has the effect of reversing the sequence that starts at `ii` and ends at the end of the sequence.

The end result is a routine that is short, simple, and runs in linear time. You really can't ask for much more than that.

## Walk through an example

For a quick look at the algorithm in action, consider what happens when you call `next_permutation("23541")`. After passing through the initial size tests, the algorithm will search for suitable values for iterators *i* and *ii*. (Remember that you are searching from the end of the sequence for the first adjacent pair where the value pointed to by *i* is less than the value pointed to by `ii`, and `i` is one less than `ii`.) The first pair of values that meet the test are seen when `i` points to 3 and `ii` points to 5. After that, a second search starts from the end for the first value of `j` where `j` points to a greater value than that pointed to by `i`. This is seen when `j` points to 4.

Once the three iterators are set, there are only two tasks left to perform. The first is to call `iter_swap(i,j)`, which swaps the values pointed to by the iterators `i` and `j`. After you do this, you are left with the modified sequence "24531". The last step is to call `reverse( ii, last )`, which reverses the sequence starting at `ii` and finishing at the end of the sequence. This yields "24135". Examining Figure 2 shows that the result demonstrated here does agree with the output of the program.

## An additional charming attribute

The algorithm shown here has one additional feature that is quite useful. It properly generates permutations when some of the members of the input sequence have identical values. For example, when I generate all the permutations of "ABCDE", I will get 120 unique character sequences. But when I generate all the permutations of "AAABB", I only get 10. This is because there are 6 different identical permutations of "AAA", and 2 identical permutations of "BB".

When I run this input set through a set of calls to `next_permutation()`, I see the correct output:

```
AAABB AABAB AABBA ABAAB ABABA ABBAA BAAAB BAABA BABAA BBAAA
```

This might have you scratching your head a bit. How does the algorithm know that there are 6 identical permutations of "AAA"? The recursive implementation of a permutation generator I showed in Listing 2 treats the permutations of "AAABB" just as it does "ABCDE", obligingly printing out 120 different sequences. It doesn't know or care that there are a huge number of identical permutations in the output sequence.

It’s easy to see why the brute force code in Listing 2 doesn’t notice the duplicates. It never pays any attention to the contents of the string that it is permuting. It couldn’t possibly notice that there were duplicates. It just merrily swaps characters without paying any attention to their value.

The STL algorithm, on the other hand, actually performs comparisons of the elements that it is interchanging, and uses their relative values to determine what interchanging will be done. In the example from the last section, you saw that an input of “24531” will generate a next permutation of “24135”. What if the string had a pair of duplicates, as in “24431”? If the algorithm were ignorant of character values, the next permutation would undoubtedly be “24134”.

In the early case, iterators `i` and `ii` were initially set to offsets of 1 and 2 within the string. But in this case, since the value pointed to by `i` must be less than the value pointed to by `ii`, the two iterators have to be decremented to positions 0 and 1. `j` would again point to position 3.

The subsequent swap operation yields “34421”, and the reverse function produces a final result of “31244”. Remember that the algorithm works by progressively bubbling the larger values of the string into position 0, you can see that this permutation has already jumped well ahead of the permutation of “24531” on its way to completion. Thus, the algorithm “knows” how to deal with duplicate values.

## Conclusion

The addition of the STL to the C++ Standard Library gave us a nice grab bag of functions that automate many routine tasks. `next_permuation()` turned out to be just what I needed to solve a sixth grade math problem. It might be time for you to look through the declarations in the `algorithm` header file to see what else standards committee laid on our doorstep.