

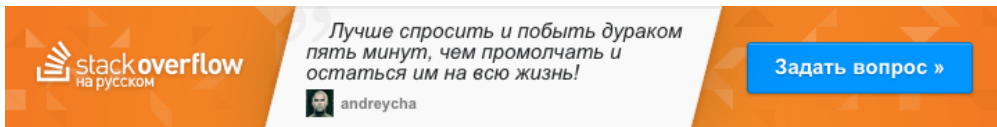
Announcing Stack Overflow Documentation

We started with Q&A. Technical documentation is next, and we need your help.

Whether you're a beginner or an experienced developer, you *can* contribute.

I want to help →

std::next_permutation Implementation Explanation



I was curious how `std::next_permutation` was implemented so I extracted the the `gnu libstdc++ 4.7` version and sanitized the identifiers and formatting to produce the following demo...

```
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

template<typename It>
bool next_permutation(It begin, It end)
{
    if (begin == end)
        return false;

    It i = begin;
    ++i;
    if (i == end)
        return false;

    i = end;
    --i;

    while (true)
    {
        It j = i;
        --i;

        if (*i < *j)
        {
            It k = end;

            while (!(*i < *--k))
                /* pass */;

            iter_swap(i, k);
            reverse(j, end);
            return true;
        }

        if (i == begin)
        {
            reverse(begin, end);
            return false;
        }
    }
}

int main()
{
    vector<int> v = { 1, 2, 3, 4 };

    do
    {
        for (int i = 0; i < 4; i++)
        {
            cout << v[i] << " ";
        }
        cout << endl;
    }
    while (::next_permutation(v.begin(), v.end()));
}
```

The output is as expected: <http://ideone.com/4nZdx>

My questions are: How does it work? What is the meaning of `i`, `j` and `k`? What value do they hold at the different parts of execution? What is a sketch of a proof of its correctness?

Clearly before entering the main loop it just checks the trivial 0 or 1 element list cases. At entry of the main loop `i` is pointing to the last element (not one past end) and the list is at least 2 elements long.

What is going on in the body of the main loop?

c++ c++11 permutation stl-algorithm lexicographic

edited Sep 21 '15 at 12:55



Shreevardhan

3,324 2 12 32

asked Jul 14 '12 at 10:37



Andrew Tomazos

25.5k 15 91 176

5 Answers

Let's look at some permutations:

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
...
```

How do we go from one permutation to the next? Firstly, let's look at things a little differently. **We can view the elements as digits and the permutations as numbers.** Viewing the problem in this way **we want to order the permutations/numbers in "ascending" order.**

When we order numbers we want to "increase them by the smallest amount". For example when counting we don't count 1, 2, 3, 10, ... because there are still 4, 5, ... in between and although 10 is larger than 3, there are missing numbers which can be gotten by increasing 3 by a smaller amount. In the example above we see that `1` stays as the first number for a long time as there are many reorderings of the last 3 "digits" which "increase" the permutation by a smaller amount.

So when do we finally "use" the `1`? When there are only no more permutations of the last 3 digits.

And when are there no more permutations of the last 3 digits? When the last 3 digits are in descending order.

Aha! This is key to understanding the algorithm. **We only change the position of a "digit" when everything to the right is in descending order** *because if it isn't in descending order then there are still more permutations to go* (ie we can "increase" the permutation by a smaller amount).

Let's now go back to the code:

```
while (true)
{
    int j = i;
    --i;

    if (*i < *j)
    { // ...
    }

    if (i == begin)
    { // ...
    }
}
```

From the first 2 lines in the loop, `j` is an element and `i` is the element before it.

Then, if the elements are in ascending order, (`if (*i < *j)`) do something.

Otherwise, if the whole thing is in descending order, (`if (i == begin)`) then this is the last permutation.

Otherwise, we continue and we see that `j` and `i` are essentially decremented.

We now understand the `if (i == begin)` part so all we need to understand is the `if (*i < *j)` part.

Also note: "Then if the elements are in ascending order ..." which supports our previous observation that we only need to do something to a digit "when everything to the right is in descending order". The ascending order `if` statement is essentially finding the leftmost place where "everything to the right is in descending order".

Let's look again at some examples:

```
...
1 4 3 2
2 1 3 4
...
2 4 3 1
3 1 2 4
...
```

We see that when everything to the right of a digit is in descending order, we **find the next largest digit and put it in front** and then **put the remaining digits in ascending order**.

Let's look at the code:

```
It k = end;

while (!(*i < *--k))
    /* pass */;

iter_swap(i, k);
reverse(j, end);
return true;
```

Well, since the things to the right are in descending order, to find the "next largest digit" we just have to iterate from the end, which we see in the first 3 lines of code.

Next, we swap the "next largest digit" to the front with the `iter_swap()` statement and then since we know that digit was the next largest, we know that the digits to the right are still in descending order, so to put it in ascending order, we just have to `reverse()` it.

edited Jan 14 at 23:58



Jamie

434 4 13

answered Jul 14 '12 at 11:32



quasiverse

4,908 2 14 29


4 Amazing explanation – Desolator Jul 14 '12 at 22:02

3 nice explanation! – XiaJun Aug 21 '13 at 6:47


1 Thank you for explaining the code. – Abhijeet Jan 12 '14 at 22:18

1 Thanks for explanation! This algorithm is called [Generation in lexicographic order](#). There are numbers of such algorithm in [Combinatorics](#), but this is the most classical one. – chain ro Mar 10 '15 at 2:59

What is the complexity of such algorithm? – user72708 Jan 5 at 21:46

 **stackoverflow**
на русском

Фантомный баг не даёт уснуть?
Ищи его на StackOverflow.

 Ella Svetlaya

Задать вопрос »

The gcc implementation generates permutations in lexicographical order. [Wikipedia](#) explains it as follows:

The following algorithm generates the next permutation lexicographically after a given permutation. It changes the given permutation in-place.

1. Find the largest index k such that $a[k] < a[k + 1]$. If no such index exists, the permutation is the last permutation.
2. Find the largest index l such that $a[k] < a[l]$. Since $k + 1$ is such an index, l is well defined and satisfies $k < l$.
3. Swap $a[k]$ with $a[l]$.
4. Reverse the sequence from $a[k + 1]$ up to and including the final element $a[n]$.

answered Jul 14 '12 at 10:56



TemplateRex

41.7k 12 94 166

AFAICT, all implementations generate the same order. – MSalters Jul 20 '15 at 8:08

Knuth goes into depth about this algorithm and its generalizations in sections 7.2.1.2 and 7.2.1.3 of *The Art of Computer Programming*. He calls it "Algorithm L" -- apparently it dates back to the 13th century.

edited Aug 25 '14 at 17:05



rvghne

4,638 2 21 46

answered Jan 22 '14 at 2:47



racarate

254 5 18

1 Can you please mention the name of the book? – [Grobber](#) Jan 28 '14 at 7:00

3 TAOCP = The Art of Computer Programming – [racarate](#) Jan 31 '14 at 1:27

Here's a complete implementation using other standard library algorithms. Study the sub-algorithms for intuition on the entire thing.

```
template <typename I, typename C>
//requires BidirectionalIterator<I> && StrictWeakOrdering<C, ValueType<I>>
bool my_next_permutation(I begin, I end, C comp)
{
    const auto rbeg = std::make_reverse_iterator(end);
    const auto rend = std::make_reverse_iterator(begin);
    bool had_greater_permutation{false};
    auto pivot = std::is_sorted_until(rbeg, rend, comp);
    if (pivot != rend) {
        std::iter_swap(pivot, std::upper_bound(rbeg, pivot, *pivot, comp));
        had_greater_permutation = true;
    }
    std::reverse(rbeg, pivot);
    return had_greater_permutation;
}

template <typename I>
//requires BidirectionalIterator<I> && TotallyOrdered<ValueType<I>>
bool my_next_permutation(I begin, I end)
{
    return my_next_permutation(
        begin, end, std::less<typename std::iterator_traits<I>::value_type>{}
    );
}
```

Demo

std::prev_permutation is left as an exercise.

edited Nov 9 '15 at 0:11

answered Sep 21 '15 at 12:36



Brian Rodriguez

1,414 2 13

There is a self explanatory possible implemetation on [cppreference](#) using `<algorithm>` .

```
template <class Iterator>
bool next_permutation(Iterator first, Iterator last) {
    if (first == last) return false;
    Iterator i = last;
    if (first == --i) return false;
    while (1) {
        Iterator i1 = i, i2;
        if (*--i < *i1) {
            i2 = last;
            while (!(*i < *--i2));
            std::iter_swap(i, i2);
            std::reverse(i1, last);
            return true;
        }
        if (i == first) {
            std::reverse(first, last);
            return false;
        }
    }
}
```

Change the content to lexicographically next permutation (in-place) and return true if exists otherwise sort and return false if it doesn't exist.

edited Sep 22 '15 at 16:34



Drew Noakes

121k 72 384 481

answered Sep 21 '15 at 12:58



Shreevardhan

3,324 2 12 32