

Combinations in C++

By **Wong Shao Voon** | 14 Sep 2009

An article on finding combinations.

- [Download demo project - 14.3 KB](#)
- [Download source - 5.95 KB](#)

Introduction

Combination is the way of picking a different unique smaller set from a bigger set, without regard to the ordering (positions) of the elements (in the smaller set). This article teaches you how to find combinations. First, I will show you the technique to find combinations. Next, I will go on to explain how to use my source code. The source includes a recursive template version and a non-recursive template version. At the end of the article, I will show you how to find permutations of a smaller set from a bigger set, using both `next_combination()` and `next_permutation()`.

Before all these, let me first introduce to you the technique of finding combinations.

The Technique

The notations used in this article

- n : n is the larger sequence from which the r sequence is picked.
- r : r is the smaller sequence picked from the n sequence.
- c : c is the formula for the total number of possible combinations of r , picked from n distinct objects: $n! / (r! (n-r)!)$.
- The $!$ postfix means factorial.

Explanation

Let me explain using a very simple example: finding all combinations of 2 from a set of 6 letters {A, B, C, D, E, F}. The first combination is AB and the last is EF.

The total number of possible combinations is: $n! / (r! (n-r)!)$ = $6! / (2! (6-2)!)$ = 15 combinations.

Let me show you all the combinations first:

```
AB
AC
AD
AE
AF
BC
BD
BE
BF
CD
CE
CF
DE
DF
```

EF

If you can't spot the pattern, here it is:

```
AB | AB
A  | AC
A  | AD
A  | AE
A  | AF
---|----
BC | BC
B  | BD
B  | BE
B  | BF
---|----
CD | CD
C  | CE
C  | CF
---|----
DE | DE
D  | DF
---|----
EF | EF
```

The same thing goes to combinations of any number of letters. Let me give you a few more examples and then you can figure them out yourself.

Combinations of 3 letters from {A, B, C, D, E} (a set of 5 letters).

The total number of possible combinations is: 10

```
A B C
A B D
A B E
A C D
A C E
A D E
B C D
B C E
B D E
C D E
```

Combinations of 4 letters from {A, B, C, D, E, F} (a set of 6 letters).

The total number of possible combinations is: 15.

```
A B C D
A B C E
A B C F
A B D E
A B D F
A B E F
A C D E
A C D F
A C E F
A D E F
B C D E
B C D F
B C E F
B D E F
C D E F
```

I'm thinking if you would have noticed by now, the number of times a letter appears. The formula for the number of times a letter appears in all possible combinations is $n!/(r!(n-r)!) * r / n == c * r / n$. Using the above example, it would be $15 * 4 / 6 = 10$ times. All the letters {A, B, C, D, E, F} appear 10 times as shown. You can count them yourself to prove it.

Source Code Section

Please note that all the combination functions are now enclosed in the `stdcomb` namespace.

The Recursive Way

I have made a recursive function, `char_combination()` which, as its name implies, takes in character arrays and processes them. The source code and examples of using `char_combination()` are in `char_comb_ex.cpp`. I'll stop to mention that function. For now, our focus is on `recursive_combination()`, a template function, which I wrote using `char_combination()` as a guideline.

The function is defined in `combination.h` as below:

```
// Recursive template function
template <class RanIt, class Func>
void recursive_combination(RanIt nbegin, RanIt nend, int n_column,
    RanIt rbegin, RanIt rend, int r_column, int loop, Func func)
{
    int r_size=rend-rbegin;

    int localloop=loop;
    int local_n_column=n_column;

    //A different combination is out
    if(r_column>(r_size-1))
    {
        func(rbegin, rend);
        return;
    }
    //=====

    for(int i=0; i<=loop; ++i)
    {
        RanIt it1=rbegin;
        for(int cnt=0; cnt<r_column; ++cnt)
        {
            ++it1;
        }
        RanIt it2=nbegin;
        for(int cnt2=0; cnt2<n_column+i; ++cnt2)
        {
            ++it2;
        }

        *it1=*it2;

        ++local_n_column;

        recursive_combination(nbegin, nend, local_n_column,
            rbegin, rend, r_column+1, localloop, func);

        --localloop;
    }
}
```

The parameters prefixed with 'n' are associated with the n sequence, while the r-prefixed one are r sequence related. As an end user, you need not bother about those parameters. What you need to know is `func`. `func` is a function defined by you. If the combination function finds combinations recursively, there must exist a way the user can process each combination. The solution is a function pointer which takes in two parameters of type `RanIt` (stands for Random Iterator). You are the one who defines this function. In this way, encapsulation is achieved. You need not know how `recursive_combination()` internally works, you just need to know that it calls `func` whenever there is a different combination, and you just need to define the `func()` function to process the combination. It must be noted that `func()` should not write to the two iterators passed to it.

The typical way of filling out the parameters is `n_column` and `r_column` is always 0, `loop` is the number of elements in the `r` sequence minus that of the `n` sequence, and `func` is the function pointer to your function (`nbeg` and `nend`, and `rbeg` and `rend` are self-explanatory; they are the first iterators and the one past the last iterators of the respective sequences).

Just for your information, the maximum depth of the recursion done is `r+1`. In the last recursion (`r+1` recursion), each new combination is formed.

An example of using `recursive_combination()` with raw character arrays is shown below:

```
#include <iostream>
#include <vector>
#include <string>
#include "combination.h"

using namespace std;
using namespace stdcomb;

void display(char* begin, char* end)
{
    cout<<begin<<endl;
}

int main()
{
    char ca[]="123456";
    char cb[]="1234";

    recursive_combination(ca, ca+6, 0,
                          cb, cb+4, 0, 6-4, display);
    cout<<"Complete!"<<endl;
    return 0;
}
```

An example of using `recursive_combination()` with a vector of integers is shown below:

```
#include <iostream>
#include <vector>
#include <string>
#include "combination.h"

typedef vector::iterator vii;

void display(vii begin, vii end)
{
    for (vii it=begin; it!=end; ++it)
        cout<<*it;
    cout<<endl;
}

int main()
{
    vector<int> ca;
    ca.push_back (1);
    ca.push_back (2);
    ca.push_back (3);
    ca.push_back (4);
    ca.push_back (5);
    ca.push_back (6);
    vector<int> cb;
    cb.push_back (1);
    cb.push_back (2);
    cb.push_back (3);
    cb.push_back (4);

    recursive_combination(ca.begin (), ca.end(), 0,
                          cb.begin(), cb.end(), 0, 6-4, display);
    cout<<"Complete!"<<endl;
    return 0;
}
```

The Non-Recursive Way

If you have misgivings about using the recursive method, there is a non-recursive template function for you to choose. (Actually there are two.)

The parameters are even simpler than the recursive version. Here's the function definition in *combination.h*:

```
template <class BidIt>

bool next_combination(BidIt n_begin, BidIt n_end,
                     BidIt r_begin, BidIt r_end);

template <class BidIt>

bool next_combination(BidIt n_begin, BidIt n_end,
                     BidIt r_begin, BidIt r_end, Predicate Equal );
```

And its reverse counterpart version:

```
template <class BidIt>

bool prev_combination(BidIt n_begin, BidIt n_end,
                     BidIt r_begin, BidIt r_end);

template <class BidIt>

bool prev_combination(BidIt n_begin, BidIt n_end,
                     BidIt r_begin, BidIt r_end, , Predicate Equal );
```

The parameters *n_begin* and *n_end* are the first and the last iterators for the n sequence. And, *r_begin* and *r_end* are iterators for the r sequence. *Equal* is the predicate for comparing equality.

You can peruse the source code for these two functions in *combination.h* and its examples in *next_comb_ex.cpp* and *prev_comb_ex.cpp*, if you want.

A typical way of using *next_combination* with raw character arrays is as below:

```
#include <iostream>
#include <vector>
#include <string>
#include "combination.h"

using namespace std;
using namespace stdcomb;

int main()
{
    char ca[]="123456";
    char cb[]="1234";

    do
    {
        cout<<cb<<endl;
    }
    while(next_combination(ca, ca+6, cb, cb+4));
    cout<<"Complete!"<<endl;

    return 0;
}
```

A typical way of using *next_combination* with a vector of integers is as below:

```
#include <iostream>
#include <vector>
#include <string>
#include "combination.h"
```

```

template<class BidIt>
void display(BidIt begin,BidIt end)
{
    for (BidIt it=begin;it!=end;++it)
        cout<<*it<<" ";
    cout<<endl;
}

int main()
{
    vector<int> ca;
    ca.push_back (1);
    ca.push_back (2);
    ca.push_back (3);
    ca.push_back (4);
    ca.push_back (5);
    ca.push_back (6);
    vector<int> cb;
    cb.push_back (1);
    cb.push_back (2);
    cb.push_back (3);
    cb.push_back (4);

    do
    {
        display(cb.begin(),cb.end());
    }
    while(next_combination(ca.begin (),ca.end (),cb.begin (),cb.end()) );

    cout<<"Complete!"<<endl;

    return 0;
}

```

Certain conditions must be satisfied in order for next_combination() to work

- 1 All the objects in the n sequence must be distinct.
- 2 For `next_combination()`, the r sequence must be initialized to the first r-th elements of the n sequence in the first call. For example, to find combinations of r=4 out of n=6 {1,2,3,4,5,6}, the r sequence must be initialised to {1,2,3,4} before the first call.
- 3 As for `prev_combination()`, the r sequence must be initialised to the last r-th elements of the n sequence in the first call. For example, to find combinations of r=4 out of n=6 {1,2,3,4,5,6}, the r sequence must be initialised to {3,4,5,6} before the first call.
- 4 The n sequence must not change throughout the process of finding all the combinations, else results are wrong (makes sense, right?).
- 5 `next_combination()` and `prev_combination()` operate on data types with the `==` operator defined. That is to mean if you want to use `next_combination()` on sequences of objects instead of sequences of POD (Plain Old Data), the class from which these objects are instantiated must have an overloaded `==` operator defined, or you can use the predicate versions.

When the above conditions are not satisfied, results are undetermined even if `next_combination()` and `prev_combination()` may return `true`.

Return Value

When `next_combination()` returns `false`, no more next combinations can be found, and the r sequence remains unaltered. Same for `prev_combination()`.

Some information about next_combination() and prev_combination()

- 6 The n and r sequences need not be sorted to use `next_combination()` or `prev_combination()`.
- 7 `next_combination()` and `prev_combination()` do not use any static variables, so it is alright to

find combinations of another sequence of a different data type, even when the current finding of combinations of the current sequence have not reached the last combination. In other words, no reset is needed for `next_combination()` and `prev_combination()`.

Examples of how to use these two functions are in *next_comb_ex.cpp* and *prev_comb_ex.cpp*.

So what can we do with next_combination()?

With `next_combination()` and `next_permutation()` from STL algorithms, we can find permutations!!

The formula for the total number of permutations of the r sequence, picked from the n sequence, is: $n!/(n-r)!$

We can call `next_combination()` first, then `next_permutation()` iteratively; that way, we will find all the permutations. A typical way of using them is as follows:

```
sort(n.begin(), n.end());
do
{
    sort(r.begin(), r.end());
    //do your processing on the new combination here
    do
    {
        //do your processing on the new permutation here
    }
    while(next_permutation(r2.begin(), r2.end()))
}
while(next_combination(n.begin(), n.end(), r.begin(), r.end() ));
```

However, I must mention that there exists a limitation for the above code. The n and r sequences must be sorted in ascending order in order for it to work. This is because `next_permutation()` will return `false` when it encounters a sequence in descending order. The solution to this problem for unsorted sequences is as follows:

```
do
{
    //do your processing on the new combination here

    for(cnt i=0; cnt<24; ++cnt)
    {
        next_permutation(r2.begin(), r2.end());
        //do your processing on the new permutation here
    }
}
while(next_combination(n.begin(), n.end(), r.begin(), r.end() ));
```

However, this method requires you to calculate the number of permutations beforehand.

So how do I prove they are distinct permutations?

There is a `set` container class in STL we can use. All the objects in the `set` container are always in sorted order, and there are no duplicate objects. For our purpose, we will use this `insert()` member function:

```
pair <iterator, bool> insert(const value_type& _Val);
```

The `insert()` member function returns a pair, whose `bool` component returns `true` if an insertion is made, and `false` if the `set` already contains an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element is inserted or where the element is already located.

proof.cpp is written for this purpose, using the STL `set` container to prove that the permutations generated are unique. You can play around with this, but you should first calculate the number of permutations which would be generated. Too many permutations may take ages to complete (partly due to the working of the `set` container), or worse, you may run out of memory!

If you are interested, you can proceed to read the second part of the article: [Combinations in C++, Part 2](#).

History

- 14 September 2009 - Added the example code.
- 21 February 2008 - Added the finding combinations of vectors in the source code.
- 26 November 2006 - Source code changes and bug fixes.
 - All functions are enclosed in the `stdcomb` namespace.
 - Solved a bug in `prev_combination` that `!=` operator must be defined for the custom class, unless the data type is a POD.
 - `next_combination` and `prev_combination` now run properly in Visual C++ 8.0, without disabling the checked iterator.
 - `next_combination` and `prev_combination` have a predicates version.
- 30 July 2003 - First release on CodeProject.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)


About the Author

Wong Shao Voon



I am currently working as a software developer in a company specialized in 3D building visualization. I am extremely interested in optimizing techniques, for example, CPU SIMD instructions like the Intel SSE2, multi-threading techniques on multi-core/SMP processors and GPGPU languages like OpenCL, DirectCompute and nVidia's CUDA.

Like many Singaporeans, my hobbies include reading, karaoke, watching movies and anime, play games and jogging.

Software Developer
 Singapore

I wish I have more time to write articles for CodeProject since I have a few ideas (long overdue) to write about. And I always explain the working behind the code in my articles. I hope you like my articles on CodeProject!