C# List in Collection

How does List work under the hood in .NET?

A List is one of the most used data types in .NET. You can dynamically add elements without taking care of how that happens. But do you know what is going on under the hood?

What is a List?

To begin with, let's ask ourselves what is an Array like int[] in .NET? An array is a contiguous block of memory that can hold a fixed number of elements. The elements are stored in a row and can be accessed by an index. The most crucial part here: **a fixed number of elements**. Once you create an array, you cannot change its size. Did you ever notice that there is no Add method? That is the big difference to a List. A List is a "dynamic array". It can grow and shrink as you need it. You can add and remove elements at any time.

How does a List work?

A list has two fundamental properties you are dealing with: Count and Capacity. Count is the number of elements in the list. Capacity is the number of elements the list can hold without resizing. The Capacity is always greater or equal to the Count. If we want to add more items than the Capacity we have to resize the list. So let's see this in action. What happens is:

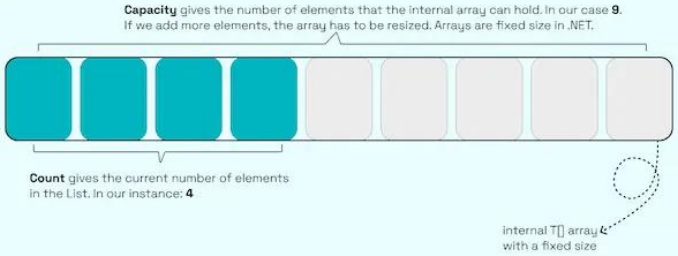
- We create a new internal array with a new size (usually double the size of the old one).
- We copy all elements from the old array to the new one.
- We replace the old array with the new one.
- We add the new element to the list.

Here is a visual representation of the process:

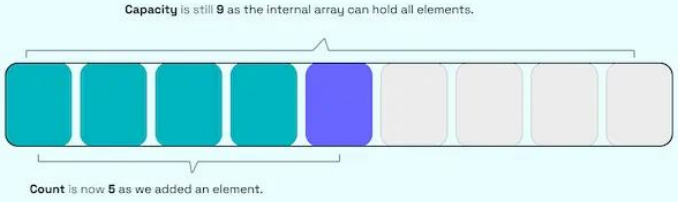
List<T> in .NET - How it works under the hood

List is one of the most used data types in .NET but how does it work internally?
So let's see how some common operations work with List.

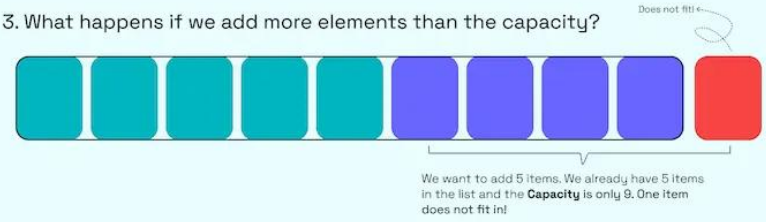
1. List is a wrapper around a fixed array.



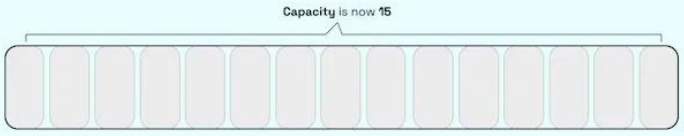
2. Adding elements to the list



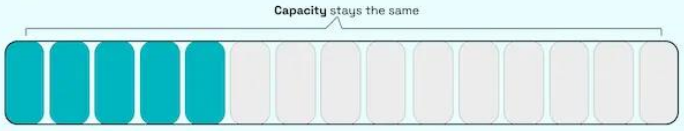
3. What happens if we add more elements than the capacity?



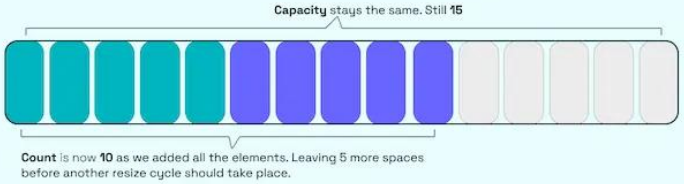
So we create a new array with **empty** larger size that can hold more items.



Now we can copy the content of the "old" array into the "new" one.



And now we add the elements, that didn't had place before!



.NET

Steven Giesel

This is a very expensive operation. It is O(n) where n is the number of elements in the list. So if you have a list with 1000 elements and you add one more, you have to copy all 1000 elements to a new array. Did you ever notice that you can pass in an initial capacity to the constructor? This is a performance optimization. If you know that you will add 1000 elements to the list, you can set the initial capacity to 1000. This way, you avoid the resizing of the list. But be careful. If you set the initial capacity too high, you will waste memory. So you have to find the right balance between performance and memory consumption.

On a side note: The StringBuilder works the same way. It has an internal array of characters. If you append a new character to the StringBuilder, it will resize the internal array if needed.

Some points

This image was originally published on LinkedIn and Twitter. It received some questions I want to answer here.

"How does it decide how many empty slots to create?"

The current .NET implementations (.NET 6 / 7 / 8) do start with 4 entries and double the size each time the space isn't enough. So we end up with 4, 8, 16, 32, 64 and so on slots.

You can define the initial capacity: new List<int>(100);. This can have impact on the performance but may waste some memory. If you know for certain you will have a rather large list, where you add elements one-by-one, it makes absolute sense to define the initial capacity!

"Why isn't it implemented as a linked list?"

A linked list doesn't suffer the resize issues as you just know the pointer to the next element.
A LinkedList is rather a special collection than a generalized one. For starters accessing an element by an indexer is **O(1)** access time (var secondElement = myList[1]). This would take **O(n)** time as we have to traverse from the start to the wished position in a LinkedList. A LinkedList is more appropriate if you want to add or remove elements in the middle of the list rather than the end.

That clashes a bit with the common use case of adding elements add the end. That is why we have two different list types (or even more). Of course, there are other implications in memory management, but that is something for later (here more details).

Conclusion

A List is a dynamic array. It can grow and shrink as you need it. If you add more elements than the current capacity, it will resize the internal array. The cool thing is that you don't have to deal with this, but anyway it is good to know what is going on under the hood!