`C++11` standartları ile kapatılan eksikliklerden biri de `next` ve `prev` fonksiyonları. Adımlayıcılar `(iterator)` üzerinde çalışan bu işlevleri anlayabilmek için önce ihtiyaç noktasını görmemiz gerekiyor. Elimizde aşağıdaki gibi bir kap olsun:

```
std::vector<int> ivec(100);
```

Bu kabın tuttuğu tüm öğeler üstünde bir algoritmayı koşturmak için `begin` ve `end` işlevlerinden alacağımız adımlayıcıları kullanmamız gerekir, değil mi?

```
std::for_each(ivec.begin(), ivec.end(), [](int &r){r * = 2;});
```

Yukarıdaki deyimde standart `for_each` algoritmasıyla `ivec` isimli `vector` nesnesinde tutulan tamsayıların hepsini iki katına çıkartmış oluyoruz.

Peki ya, `vector`'ün tüm öğeleri için değil de belirli bir aralığı üstünde bir algoritmayı koşturmak istersek ne yapacağız? Bu amaçla adımlayıcı aritmetiğinden faydalanabiliriz. Örneğin ilk ve son öğe haricindeki tüm öğeleri içeren aralık üstünde işlem yapmak istediğimizi düşünelim:

```
std::for_each(ivec.begin() + 1, ivec.end() - 1, [](int &r){r * = 2;});
```

Eğer kabımız bir `vector` ise bu işlemde bir sorun yok. Çünkü `vector` kabının adımlayıcıları tamsayı ile toplama arayüzünü de sunan `"random acces iterator"` kategorisinde.

```
ivec.begin() + n
```

gibi bir ifadeyle `ivec` kabında tutulan `n` indisli öğenin konumunu tutan bir adımlayıcı elde edebiliyoruz.

Peki kabımız bir `list` nesnesi olsaydı?

```
list<int>ilist(100);
///
std::for_each(ilist.begin() + 1, ilist.end() - 1, [](int &r){r * = 2;}); //geçersiz
```

kodumuz geçerli değil. Çünkü `"bidirectional iterator"` kategorisinde olan `list` sınıfının adımlayıcılarının arayüzünde toplama işlemi yok. Şöyle bir kod yazabiliriz:

```
#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
 list<int> ilist{ 5, 56, 7, 48, 11, 23, 17, 39 };
 //
 auto iter1 = ilist.begin();
 ++iter1;
 auto iter2 = ilist.end();
 --iter2;
 for_each(iter1, iter2, [](int &x) {x *= 2; });

 for (int i : ilist)
  cout << i << " ";
}
```

`<iterator>` başlık dosyasında bulunan `next` ve `prev` işlevleri bu tür durumlarda işimizi bir hayli kolaylaştırıyor. Şimdi her iki işlevi de inceleyelim:

```
template <class ForwardIterator>
ForwardIterator next(ForwardIterator iter,
typename iterator_traits<ForwardIterator>::difference_type n = 1);
```

İşlevimiz birinci parametresi bir adımlayıcı istiyor. İşlevin değerle çağrıldığına `(call by value)` dikkat edin. İşlevin `2.` parametresine belirlenen adımlayıcı konumundan kaç sonraki konumu elde etmek istiyorsak o tamsayıyı geçiyoruz. İkinci parametrenin varsayılan argüman olarak `1` değerini aldığını görüyorsunuz. Yani ikinci parametreye bir arguman gönderilmez ise işlevden geri dönüş değeri olarak bir sonraki öğenin konumunu tutan bir adımlayıcı elde ediyoruz. iter, `"forward iterator"` arayüzüne sahip bir adımlayıcı olmak üzere

```
next(iter)
```

ifadesi ile `iter` konumundan bir sonraki öğenin konumunu elde ediyoruz.

```
next(iter, n)
```

ifadesi ile `iter` konumundan `n` sonraki öğenin konumunu elde ediyoruz.

`next` işlev şablonunun gerçekleştirimi aşağıdaki gibi olabilir:

```
template<class ForwardIt>
ForwardIt next(ForwardIt it,
typename std::iterator_traits<ForwardIt>::difference_type n = 1)
{
 std::advance(it, n);
 return it;
}
```

Standart `advance` işlevinin referans yoluyla aldığı adımlayıcıyı `n` pozisyon ilerlettiğini hatırlayalım. Eğer `n` negatif bir tamsayı ise işleve gönderilen adımlayıcı `n` pozisyon eksiltiliyor.

Gelelim `prev` işlevine. Bu işlev ile bir adımlayıcı konumundan `n` önceki konumu elde ediyoruz:

```
template<class BidIter>
BidIter prev(
  BidIter it,
  typename std::iterator_traits<BidirIt>::difference_type n = 1);
```

Adımlayıcılar üzerinde eksiltme işlemi yapabilmek için adımlayıcının en az `"bidirectional iterator"` kategorisinde olması gerekiyor. iter, `"bidirectional iterator"` arayüzüne sahip bir adımlayıcı olmak üzere

```
prev(iter)
```

ifadesi ile `iter` konumundan bir önceki öğenin konumunu elde ediyoruz.

```
prev(iter, n)
```

ifadesi ile `iter` konumundan `n` önceki öğenin konumunu elde ediyoruz.

`next` ve `prev` işlevlerini kullanarak bir kap içinde tutulan öğelerin herhangi bir aralığı üstünde işlem yapabiliriz. Daha önce yazdığımız geçersiz koda geri dönelim:

```
#include <list>
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace std;

int main()
{
 list<int> ilist{5, 56, 7, 48, 11, 23, 37, 49};
 for_each(next(ilist.begin()), prev(ilist.end()), [](int &r){r *= 2; });

 for (int i : ilist)
  cout << i << " ";
}
```

Yukarıdaki kodda

```
next(ilist.begin())
```

ifadesi ile `ilist` kabında tutulan ikinci öğenin konumunu

```
prev(ilist.end())
```

ifadesi ile kapta tutulan son öğenin konumunu elde ediyoruz. `for_each` algoritmasıyla ilist listesindeki ilk ve son öğe haricindeki tüm öğeleri `2` katına çıkartmış olduk.

`next` ve `prev` işlevlerine geçersiz bir adımlayıcı konumunun geçilmesi çalışma zamanı hatasına neden oluyor. Eksiltme ya da artırma sonucu geçersiz bir konumun elde edilmesi yine çalışma zamanı hatası. Her iki durumda da bir hata nesnesi `(exception)` gönderilmediğini hatırlatalım.