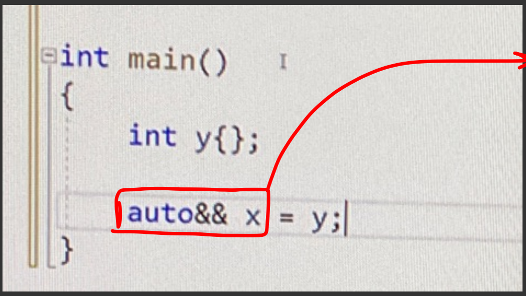
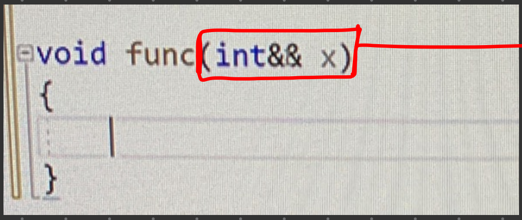
*** Hatırlatma:**

- 3 çeşit referans var, R value
  L value
  Forwarding (Universal) Referans → Hem R hem L value'e bağlanabilir



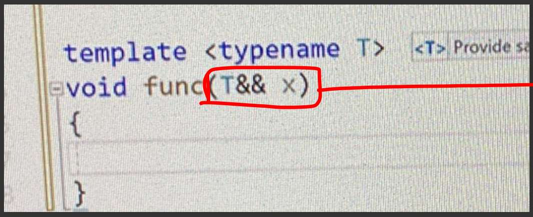
```cpp
int main()
{
    int y{};

    auto&& x = y;
}
```

→ Bu forwarding referans → çünkü auto type deduction var.



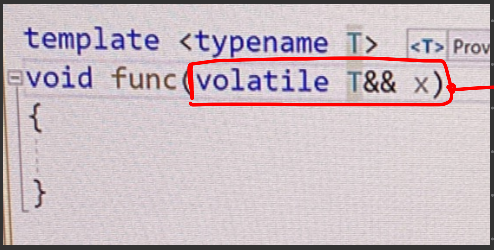
```cpp
void func(int&& x)
{

}
```

→ R value referans



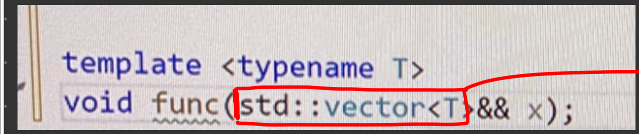
```cpp
template <typename T>
void func(T&& x)
{

}
```

→ Forwarding referans



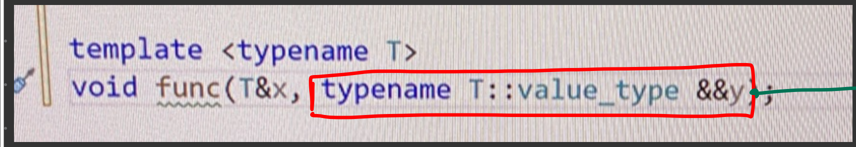
```cpp
template <typename T>
void func(volatile T&& x)
{

}
```

→ volatile/const qualifier varsa R value ref.



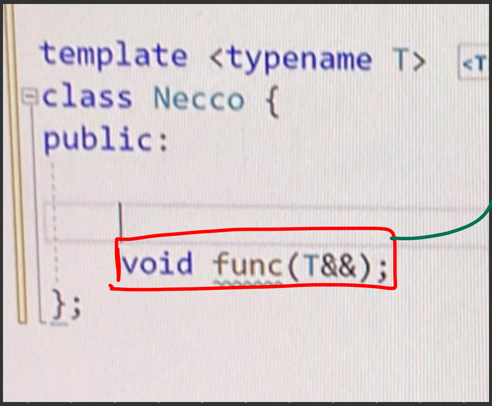
```cpp
template <typename T>
void func(std::vector<T>&& x);
```

→ R value referans



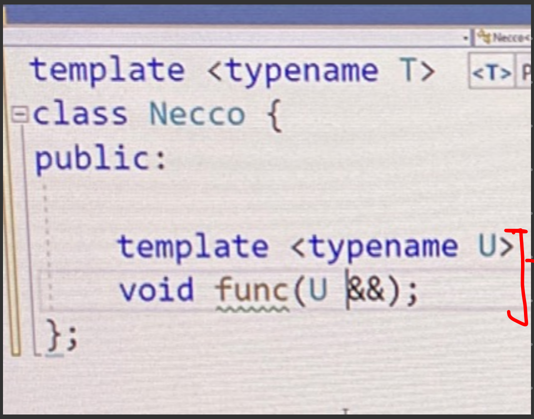
```cpp
template <typename T>
void func(T&x, typename T::value_type &&y);
```

→ R value referansı



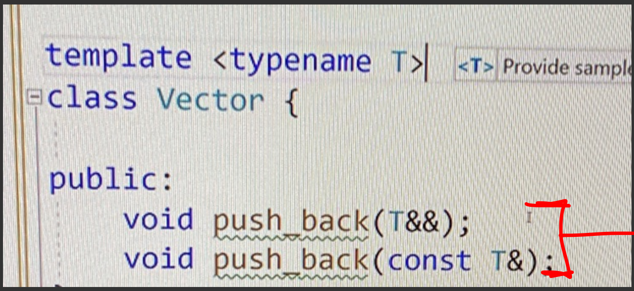
```cpp
template <typename T>
class Necco {
public:

    void func(T&&);
};
```

→ R value referans
↓
T int olsa, int&& olur



```cpp
template <typename T>
class Necco {
public:

    template <typename U>
    void func(U &&);
};
```

→ Forwarding referans

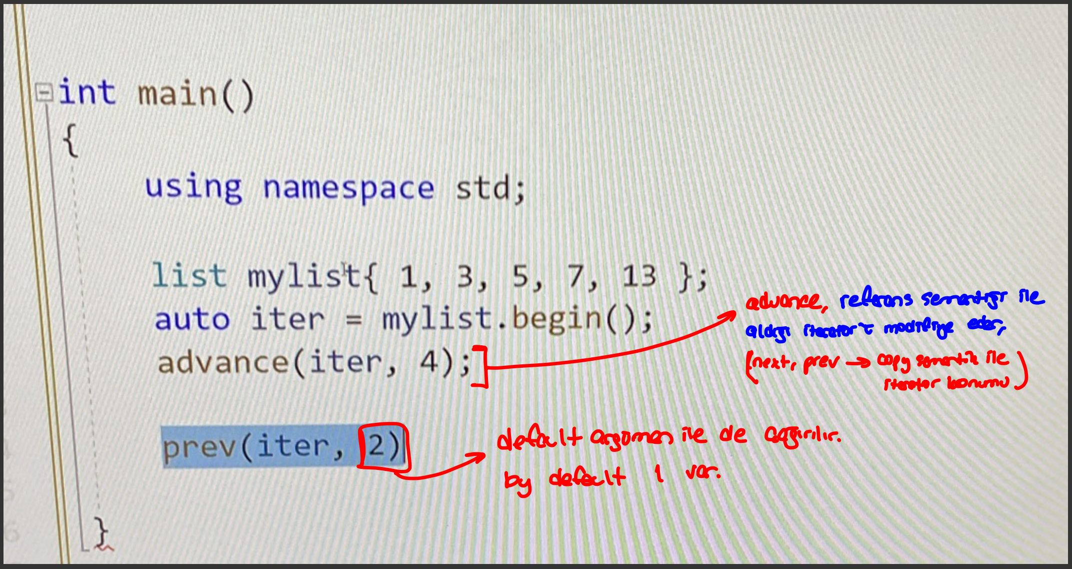


```cpp
template <typename T>
class Vector {

public:
    void push_back(T&&);
    void push_back(const T&);
```

→ push_back R value, L value olan overload'u var.!!

- Next/ Prev

• (Ayrıca prev) kopyalama semantiği ile, iteratörü alır, npos sonraki konumu döndürür.



```
int main()
{
    using namespace std;

    list mylist{ 1, 3, 5, 7, 13 };
    auto iter = mylist.begin();
    advance(iter, 4);

    prev(iter, 2)
}
```

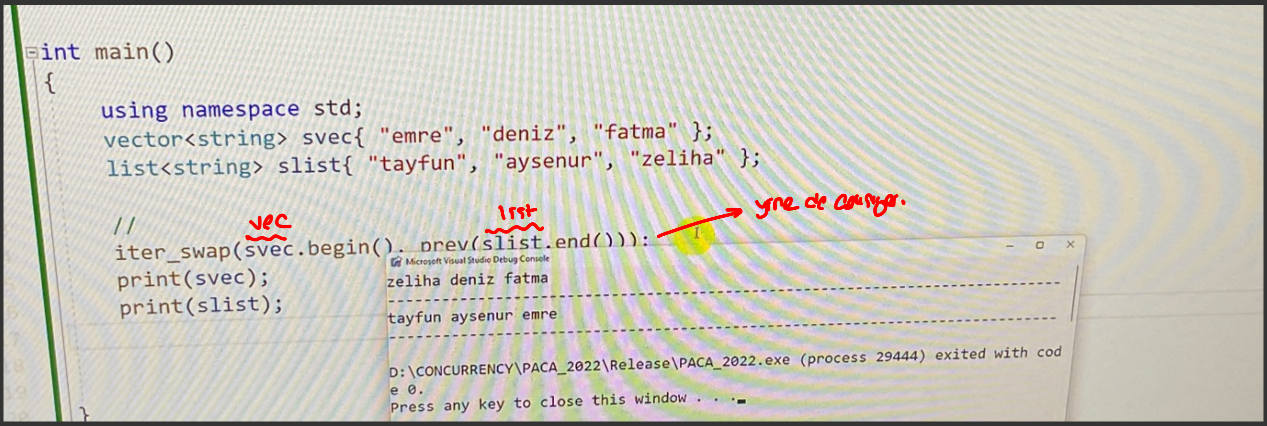
advance, referans semantiği ile aldığı iteratör'ü modifiye eder,
(next, prev → copy semantik ile iterator konumu)

default argüman ile de çağırılır.
by default 1 var.

↪ Amacımız iteratör'ün korunmuysa, next /prev kullan

- Iter Swap:

• iki iteratör alır, o iki iteratör konumundaki nesneleri takas eder!!



```
int main()
{
    using namespace std;
    vector<string> svec{ "emre", "deniz", "fatma" };
    list<string> slist{ "tayfun", "aysenur", "zeliha" };

    //
    iter_swap(svec.begin(), prev(slist.end()));
    print(svec);
    print(slist);
}
```

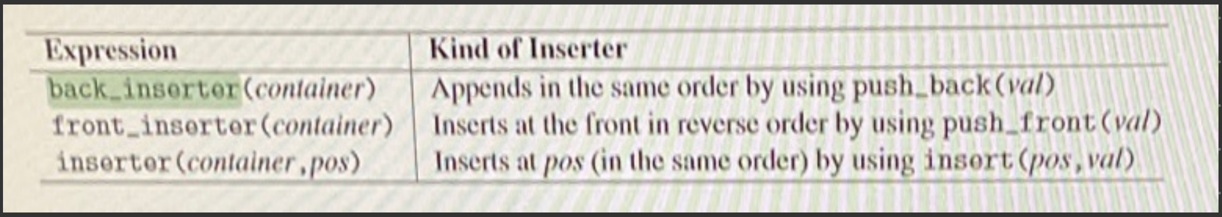
vec

list

yine de çalışıyor.

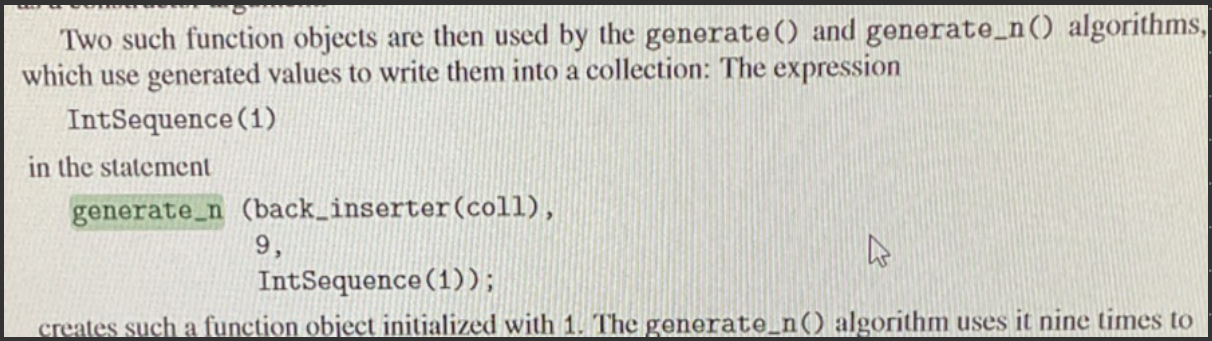
```
zeliha deniz fatma
-----------------------------------------------------------------------------
tayfun aysenur emre
-----------------------------------------------------------------------------

D:\CONCURRENCY\PACA_2022\Release\PACA_2022.exe (process 29444) exited with cod
e 0.
Press any key to close this window . . .
```

Inserterler



Expression	Kind of Inserter
back_inserter(*container*)	Appends in the same order by using push_back(*val*)
front_inserter(*container*)	Inserts at the front in reverse order by using push_front(*val*)
inserter(*container*,*pos*)	Inserts at *pos* (in the same order) by using insert(*pos*,*val*)

- Generate_n:



Two such function objects are then used by the generate() and generate_n() algorithms, which use generated values to write them into a collection: The expression

```
IntSequence(1)
```

in the statement

```
generate_n (back_inserter(coll),
            9,
            IntSequence(1));
```

creates such a function object initialized with 1. The generate_n() algorithm uses it nine times to

* Conteiners:

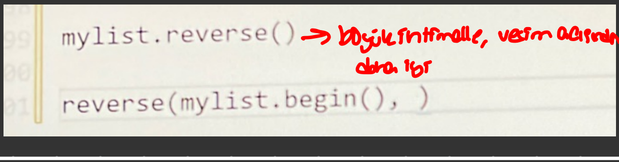
- Sequence Conteiners → vector, deque, list, forward list, array, string, vector.
- Associative Conteiners → set, multiset, map, multimap
- Unordered Associative conteiners → unordered set, unordered multiset, unordered map, unordered multimap

- Sequence containerler ekleme ve silme işlemlerini, belirli bir konum üzerinden yapar. Fakat, associative containerlar değer ile insertion yapar. (onların arkasında bir binary tree var)

- Öyle fonksiyonlar var ki, tüm containerlarda var. → clear
  → empty (returns bool true if empty)

- Bazıları da spesifik konteyner türüne özeldir.

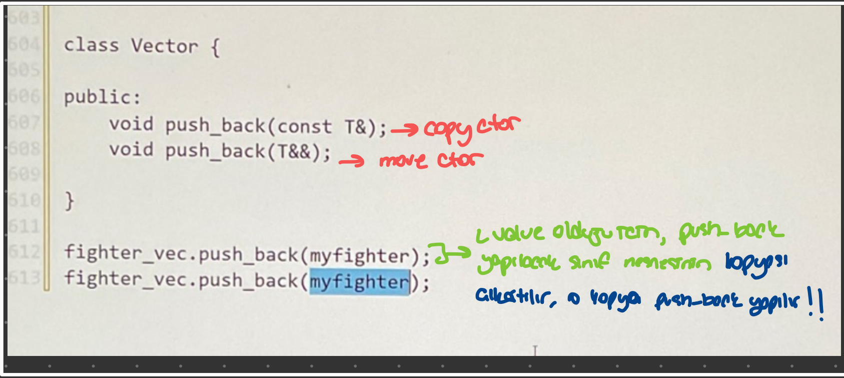
* Algoritmalar, iteratorları argüman alır, Conteiner'in kendisini almaz. Bu yüzden, container üye fonksiyonunu çağırmak daha doğru.



```
mylist.reverse()
reverse(mylist.begin(), )
```

→ büyük ihtimalle, verim açısından daha iyi

→ pointer'ın gösterdiği elemanı değiştirmektense, pointer'ı değiştirmek.



```
class Vector {

public:
    void push_back(const T&);
    void push_back(T&&);

}

fighter_vec.push_back(myfighter);
fighter_vec.push_back(myfighter);
```

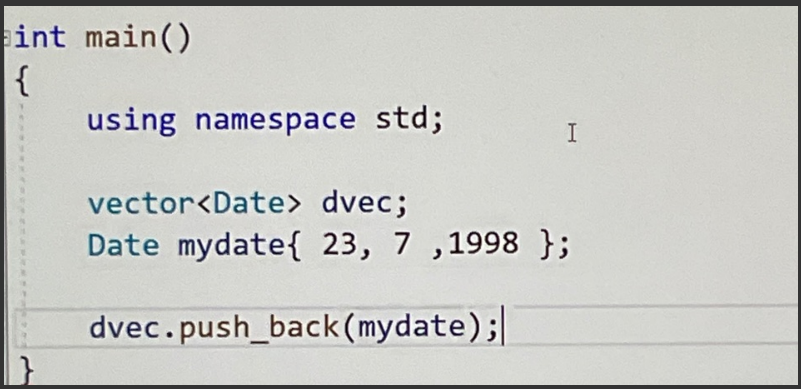
→ copy ctor

→ move ctor

→ L value olduğu için, push_back yapılacak sınıf nesnesinin kopyası çıkarılır, o kopya push_back yapılır!!

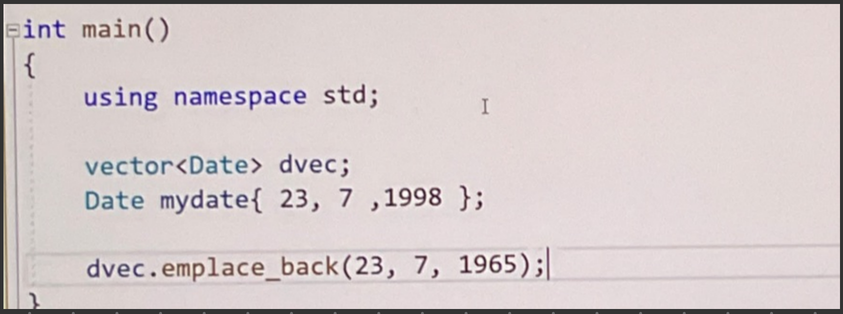
• Bu tip insert fonksiyonları, overload edilerek yazılır.
↓
push_back
push_front
⋮

- Push vs Emplace fonksiyonları:



```
int main()
{
    using namespace std;

    vector<Date> dvec;
    Date mydate{ 23, 7 ,1998 };

    dvec.push_back(mydate);
}
```

→ push_back(const T&) parametreli fonksiyonu çağrıldı. Bu overload, vector'ün sağladığı bellek alanında, ilgili adresi *this pointer'ı olarak kullanıp, sınıfın copy constructor'ına çağrı yapıyor.

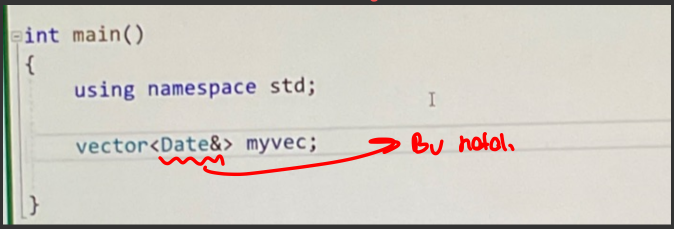


```
int main()
{
    using namespace std;

    vector<Date> dvec;
    Date mydate{ 23, 7 ,1998 };

    dvec.emplace_back(23, 7, 1965);
}
```

→ emplace back ise, aldığı argümanları, perfect forwarding mekanizması ile, sınıfın constructor'ına forward eder. (No copy, no move)

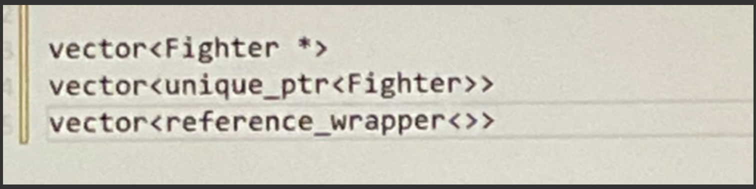
→ Copy / move constructable olmayan sınıflar da artık insert edilir.

* Containerlar, referans semantiği ile çalışmaz X. Fakat, bunu yapmak, dolaylı yollarla mümkün.



```
int main()
{
    using namespace std;

    vector<Date&> myvec;
}
```

→ Bu hatalı.



```
vector<Fighter *>
vector<unique_ptr<Fighter>>
vector<reference_wrapper<>>
```

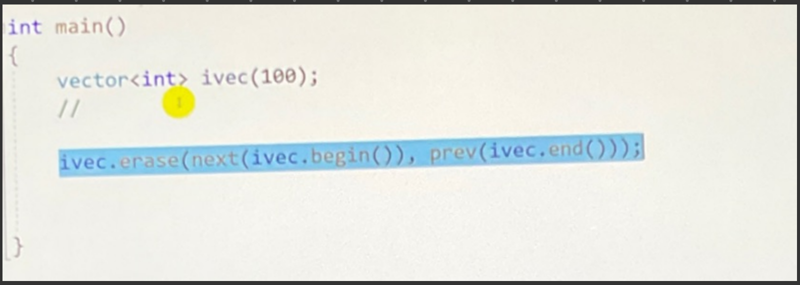
↳ Containerda öğenin kendisini tutmak yerine, container dışında hayatı devam eden nesneler containerda tutmak ve onlara erişip / değiştirmek için 3 yöntem bu.

* Vector:

→ Dynamic Array implementation.

- Dynamic array contiguous memory allocated dır. indexi bilinen öğeye O(1) 'de erişir.
- Sondan ekleme işlemi, amortized constant time O(1). Diğer işlemler linear complexity O(n)
- Array hariç, diğer tüm containerlarda olduğu gibi, vector'ün de 1 template parametresi → Allocator
  ↳ extreme durumlar hariç, default argument olarak, std::Allocator<T>

* at fonksiyonu, [ ] ile aynı. tek fark [ ] de generic index, exception throw etmez, at eder.



```
int main()
{
    vector<int> ivec(100);
    //

    ivec.erase(next(ivec.begin()), prev(ivec.end()));
}
```

→ İlk ve sonuncu öğe hariç, hepsini siliyor.

→ range olmadan da çağrılır. O zaman hepsini siler.

↪ reserve: kapasiteyi reserve eder. → realocation önüne geçer.