* Hatırlatma:

- Sette olduğu gibi, map'in de key value'su const → Değiştirmek, Data structure'ı corrupt eder.
- Map = 1 anahtar, multimap = birden fazla anahtar.
- Set'in insert / emplace'i yalnızca key'i construct ederek argüman alır.

Fakat, map - multimap → pair construct edilecek. map'in value type'ı, pair! Tanımlar değil.

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

    map<string, int> mymap;

    pair<string, int> px{ "murat", 56 };
    //mymap.insert(px);
    mymap.insert(std::move(px));
    mymap.insert({ "alican" ,67 });
    mymap.insert(make_pair("esen", 9));
    mymap.emplace("deniz", 987);

}
```

→ insert - emplace farkı: emplace'e direkt argüman geçilebilir.

→ insert'e pair constructor'ına uygun şekilde argüman geçilmeli.

* Aynısı C++ 17'de try_emplace() eklendi.

- Map traverse edilirken, pair'ler traverse ediliyor

```
for (auto iter = mymap.begin(); iter != mymap.end(); ++iter) {
    iter->first
}
```

first — (field) public : const std::string std::pair<const std::string, int>::first
the first stored value
File: utility

→ Bu iterator, pair iterator

bu yüzden key = first
value = second

- Map template'inde 3. argüman olabilir ve default olarak std::less → map küçükten büyüğe sıralanır.

std::greater<> → explicit olarak belirtirsek, map büyükten küçüğe.

* Bir Map'in Anahtarlarından Birinin Değerini Değiştirme:

1. Eski Usul / Extract Kullanmadan:

```
string name_entry;
cout << "ismi girin : ";
cin >> name_entry;

if (auto iter = mymap.find(name_entry); iter != mymap.end()) {
    auto bdate = iter->second;
    mymap.erase(iter);
    mymap.emplace("polathan", bdate);
}
```

→ eğer varsa, return değeri o iterator tanımı

→ en başta, aradığımız key map'te var mı onu bakıyor.

→ sonra, o iterator konumdur, pair second / value'yu aldık.

→ pair'i sildik,

→ Ellere yeşil.

=> Fakat burada gereksiz construction/destruction işlemi olacak. Bu yüzden

2. Extraction Kullanarak: => Extract hem iterator, hem de key ile yapılabilir.

```
if (auto iter = mymap.find(name_entry); iter != mymap.end()) {
    auto handle = mymap.extract(iter);
    handle.key() = "polathan";
    //handle.mapped() = ???
    mymap.insert(move(handle));
}

}
```

* [] Operator Fonksiyonu: * Mapte var, multimapte yok. X

```
{
    using namespace std;

    map<string, Date> mymap;


    for (int i{}; i < 100; ++i) {
        mymap.emplace(rname(), Date::random());
    }

    cout << "size = " << mymap.size() << "\n";

    for (const auto& [name, bdate] : mymap) {
        cout << name << " " << bdate << '\n';
    }

    string name_entry{ "mehmet" };

    mymap[name_entry] = Date{ 5, 9 ,2022 };


}
```

→ Compiler şu şekilde kod üretir:

→ Önce istenilen anahtarı arar. Eğer anahtar bulunursa, geri dönüş değeri o anahtara karşılık gelen value'ya referans yani bu atama sonucunda, value değeri.

→ Eğer anahtar yoksa, name entry key'i ile entry yapar ama default initialize ederek kapsa gelir yani → pair <name_entry, >

Daha sonra pair'in second'ına erişip, Date'in değerini atar.

→ Yani bizim yerimize insert edilecek ve secondına erişip atama yapılacak.

→ Ancak, map'in type'ı default constructable değilse syntax hatası.

→ Kullanım Örneği: Bir kelimeden kaç adet aynı öğeden olduğunu saydırma

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

    vector<string> svec;
    rfill(svec, 10000, rname);
    //
    map<string, int> cmap;

    for (const auto& name : svec) {
        ++cmap[name];
    }
```

int olmasından faydalandık.

```
    std::ofstream ofs{ "out.txt" };
    if (!ofs) {
        std::cerr << "out.txt dosyasi olusturulamadi\n";
        exit(EXIT_FAILURE);
    }

    for (const auto& [name, count] : cmap) {
        ofs << name << " " << count << "\n";
    }


}
```

→ Insertten Farkı: - insert'in geri dönüş değeri pair <iterator, bool>, Yani insert edilip edilmediğini bu değere bakarak anlayabiliriz.

- [] operatör ise, var olan değeri değiştirir. Yoksa önce (key, default init) sonra değer atanır (insert)

* at() fonksiyonu: - Eğer key varsa → öğeye erişiriz.
- Eğer key yoksa → exception throw

```cpp
using namespace std;

map<string, int> mymap{
    {"deniz", 12},
    {"emre", 34},
    {"fatih", 45},
    {"ayse", 92},
};

try {
    auto value = mymap.at("salih");
}
catch (const std::exception& ex) {
    std::cout << "exception caught: " << ex.what() << '\n';
}
```

* Try Emplace:

→ Emplace'in Dezavantajları:

Diyelim ki bizim pair'imizda move-only bir nesne ile pair oluşturuyoruz, Eğer key değeri varsa biz hem boş yere kaynak kaldık, hem de map ekleyemedik.

```
try_emplace() does not move from rvalue arguments if the insertion does not happen.
This is useful when manipulating maps whose values are move-only types,
such as std::unique_ptr.

Unlike insert or emplace, these functions do not move from rvalue arguments if the insertion does not happen,
which makes it easy to manipulate maps whose values are move-only types,
such as std::map<std::string, std::unique_ptr>.
In addition, try_emplace treats the key and the arguments to the mapped_type separately,
unlike emplace, which requires the arguments to construct a value_type (that is, a std::pair)

As a bonus point, you do not have to use the ugly
std::piecewise_construct tag to pass both the key and the value constructor arguments
( unless you want to inplace construct the key too).
```

* Insert or Assign: → Cpp 17'de geldi
- [] operatörüne alternatif.
- return değeri bool, eklenip eklenmediğini kontrol edebiliriz.

Notes

insert_or_assign returns more information than operator[] and does not require default-constructibility of the mapped type.

Feature-test macro
__cpp_lib_map_try_emplace

Hash Table:

- Anahtarla değere erişmek ortalama O(1)
- Hashing → Fonksiyon bizim anahtarımızı, int'e çevirsek, value'lar da bir vektörde 0 indexte tutulacak.

(çarpışma)

→ Ayrıca anahtar da constant time'da oluşturursa, (ki buna eşşek yok) . O(1)'de search / find.

- Farklı iki anahtar, aynı indexe atanırsa: Collision

→ Collision olduğunda ne yapılıyor?

- en çok kullanılan: -vektörün elemanlarını linked list yapması. Yaşanan her collisionda linked listin eleman sayısı artıyor.

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

    unordered_set<
}
```

custom typeler için explicit specialization yapmanız gerekir.

```
unordered_set<typename _Kty, typename _Hasher = hash<_Kty>, typename _Keyeq = equal_to<_Kty>, typename _Alloc = allocator<_Kty>>
```

```
using namespace std;

cout << hash<int>{}(8'726'345) << "\n";
cout << hash<int>{}(8'726'346) << "\n";
cout << hash<int>{}(8'726'347) << "\n";
```

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

    string s1{ "erdem" };
    string s2{ "eray" };
    string s3{ "emirhan" };
    string s4{ "emre" };

    cout << hash<string>{}(s1) << "\n";
    cout << hash<string>{}(s1) << "\n";
    cout << hash<string>{}(s1) << "\n";
    cout << hash<string>{}(s1) << "\n";

}
```

→ Hash'in specialization'ını verken bu şekilde özet

```
using namespace std;

string s1{ "erdem" };
string s2{ "eray" };
string s3{ "emirhan" };
string s4{ "emre" };

cout << hash<Date>{}(Date{1, 4, 1987})
}
```

Bu specialization'ı bir yapmak zorundayız,

Custom typelarda hata veriyor.

Alternatif 1:

```
template <>
struct std::hash<Date> {
    std::size_t operator()(const Date& date)const
    {
        return hash<int>{}(date.month_day()) + hash<int>{}(date.month()) + hash<int>{}(date.year());
    }

};
```

Alternatif 2:

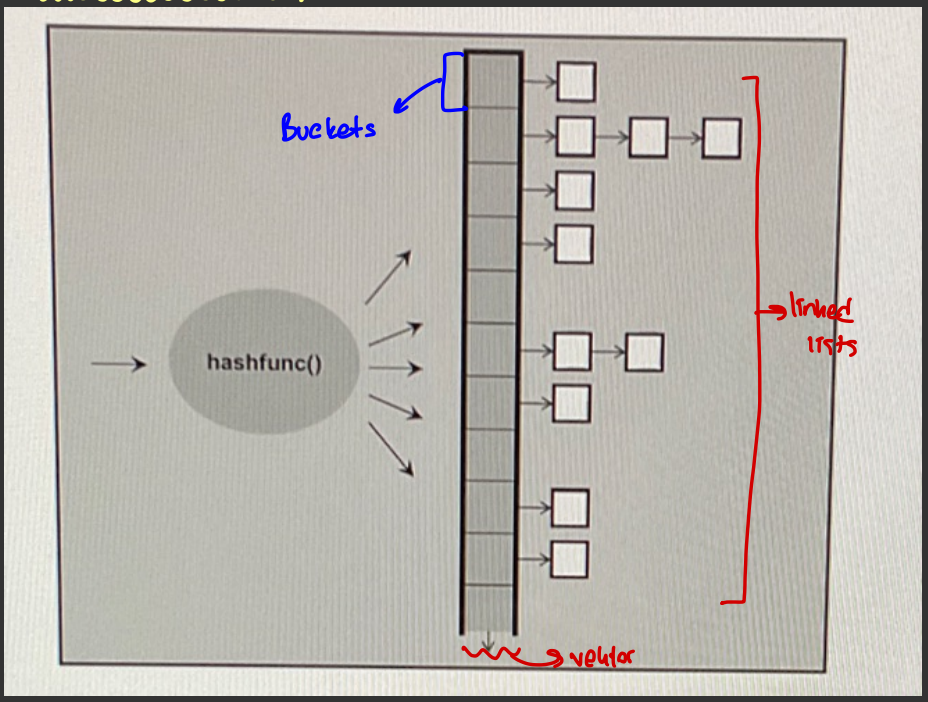
- Boost kütüphanesi

```cpp
template <>
struct std::hash<Date> {
    std::size_t operator()(const Date& date)const
    {
        return hash_val(args:date.month(), args:date.year(), args:date.month_day());
    }
};

int main()
{
    using namespace std;

    unordered_set<Date> myset;
}
```

boost kütüphanesinde hash.h

< Date, hash<Date>>

* Bucket Interface:




```cpp
unordered_set<string> myset;

for (int i = 0; i < 100; ++i) {
    myset.insert(_Val:rname());
}

ofs << "size = " << myset.size() << "\n";
ofs << "bucket count = " << myset.bucket_count() << "\n";
}
```

BU

SIZ

- Amaç BU sayıyı SIZ bucket'a collision olmadan yerleştirmek.

* iterator ile bucket interface'i traverse edebiliriz!!!