* Necö'nun sorusu:

```
class Myclass {

};

void func(Myclass);

int main()
{
    // func({});
}
```

→ Bu syntax hatası değil! Myclass Default Ctor'u çağrılarak fonksiyona argüman gönderir.

```
template<typename T, typename C = less<T>....>
class set {


private:
    //
    C m_comp;
}


mymap.insert({3, 8})
```

* Container'ımsı Yapılar:

```
using namespace std;


// std::array
//
// std::tuple;
// std::pair;
//
// std::stack
// std::queue
// std::priority_queue
```

// std::array → wrapper for C array

// std::tuple; // std::pair; → Bir fonksiyon, 2+ değer döndürmesini bunlarla

// std::stack // std::queue // std::priority_queue → container adaptors

* Std:: Array:

→ Sınıf olduğu için, bir interface ve type tanımlayıcıları var, normal arrayde yok

→ " " " std ile uyumlu

→ Exception throwing

→ Fonksiyon parametresi olarak olamaz / Return tipi olarak olamaz ama wrapper'a alabilir

→ Diziler, array decay olur

std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```

(since C++11)

T: argüman tipi

N: Array size

Default olarak struct

std::array is a container that encapsulates fixed size arrays.

This container is an aggregate type with the same semantics as a struct holding a C-style array `T[N]` as its only non-static data member. Unlike a C-style array, it doesn't decay to `T*` automatically. As an aggregate type, it can be initialized with aggregate-initialization given at most N initializers that are convertible to T: `std::array<int, 3> a = {1,2,3};`.

The struct combines the performance and accessibility of a C-style array with the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc.

std::array satisfies the requirements of *Container* and *ReversibleContainer* except that default-constructed array is not empty and that the complexity of swapping is linear, satisfies the requirements of *ContiguousContainer*, (since C++17) and partially satisfies the requirements of *SequenceContainer*.

There is a special case for a zero-length array (`N == 0`). In that case, `array.begin() == array.end()`, which is some unique value. The effect of calling `front()` or `back()` on a zero-sized array is undefined.

An array can also be used as a tuple of N elements of the same type.

*

```
struct Data {
    int a, b, c;
};

int main()
{
    Data mydata = { 2, 5, 9 };
}
```

Aggregate Type

Aggregate initializer

→ Bizim unordered map'ler aggregate init :)

↓

Eğer struct yerine, class olsaydı syntax hatası!

```
template <typename T, std::size_t N>
struct Array {
    T ma[N];
};

int main()
{
    Array<int, 5> ax{ 1, 2, 3, 4, 5 };
}
```

Bu şekilde aggrate init yapılmazsa, Default initialize edilir.
= size kadar 0 değer!

```cpp
template<typename T, std::size_t N>
void print(const std::array<T, N>& ar)
{
    for (const auto& x : ar)
        std::cout << x << ' ';

    std::cout << '\n';
}



int main()
{
    using namespace std;

    array<int, 5> a{ 1, 3, 6, 9 ,2 };


    print(a);


}
```

* int * p = a syntax hatası. Array Decay yok ✓

* 2 Boyutlu Array:

```cpp
using artype = std::array<int, 3>;

int main()
{
    using namespace std;

    array<artype, 4> ax{ {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {2, 2, 2}} };

    print(ax);
```

Eğer bu en dıştaki 2 braket olmasa syntax hatası

* So-Called Containers:

- Data memberları bir container
- Begin, end yok, range based for loopla gezilemez.

* Stack:

```
template<typename T>  <T> Provide sample ter
class Stack {
public:
    void push(const T&);
    void pop();
    bool empty()const;
    std::size_t size()const();
    T& top();
    const T& top()const;
protected:
    std::vector<T> mc;
};
```

→ STL, bunu deque ile implement etmiş. 2. template parametresi < deque<T> >