CTAD:

- Class template argument deduction
- Cpp 17
- Template argumanlarını explicit olarak, belirtmek zorunda değiliz.

```
pair p(123, 6.7);
```

→ Normal bir class gibi. Template mi değil mi diye bakmadan bu şekilde yazıldı. Constructor'a gelen argümanları alıyor.

- Hem type, hem de non-type parametreler için CTAD yapılabilir.

```
template <typename T>
class Myclass {
public:
    Myclass(const T &) {
        std::cout << typeid(T).name() << '\n';
    }
};

template <typename T>
Myclass<T> make_myclass(const T&t)
{
    return Myclass<T>(t);
}

int main()
{
    using namespace std;

    auto mx = make_myclass(20);

}
```

- Cpp 17 olmadan, CTAD kullanmadan, böyle bir yöntem kullanılabilir.

```
template <typename T>
class Myclass {
public:
    explicit Myclass(const T &) {
        std::cout << typeid(T).name() << '\n';
    }
};

int main()
{
    using namespace std;

    Myclass m = 23;

}
```

Constructor explicit ise, CTAD yapılamayabilir!

```
template <typename T, std::size_t N>
class Nec {
public:
    Nec(T(&)[N])
    {
        std::cout << "type T is: " << typeid(T).name() << '\n';
        std::cout << "constant N is " << N << '\n';
    }
};

int main()
{
    double da[]{1. ,3., 5, 9.};

    Nec mynec(da);
}
```

double

4

double [4]

→ CTAD ile argüman olarak, array gönderilebilir!

```
template <typename T = double>
struct Nec {
    T val;
    Nec() : val() {}
    Nec(T val) : val(val) {}
    // ...
};

Nec nec1{ 10 }; // Nec<int>
Nec nec2;       // Nec<double>
```

default template argument!

Default olarak double olarak çıkarım!

```
template <typename T = int, typename U = double,typename W = long>
class Myclass {
public:
    Myclass(T = T{}, U = U{}, W = W{}) {

    }
};
```

→ Argüman gönderilmediği zaman, T türünden value init edilmiş değerle construct edilir.

→ Aritmetik türler = 0, class türler default init

CTAD ile kısmi çıkarım yapılamaz!

→ default argüman olmayan yerde, tüm argümanları explicit belirtmemek!

```
std::array a1{ 1, 2, 3, 5, 6 };
```

✓ Çıkarım yapılır!

```
std::array<int> a1{ 1, 2, 3, 5, 6 };
```

X Çıkarım yapılamaz! Size-t parametre eksik!

* Templatelerde Reference Wrapper:

* Function templatelere de argüman olarak referans geçilir. Yoksa kopyalama burada da var!

```
class Pred {

public:
    bool operator()(int)const;

private:
    int m_a[1024]{};

};

int main()
{
    using namespace std;

    Pred mypred;
    vector<int> ivec(100000);
    ///
    auto iter = find_if(ivec.begin(), ivec.end(), mypred);
}
```

kopyalama yapılır!

```
class Pred {

public:
    bool operator()(int)const;

private:
    int m_a[1024]{};

};

int main()
{
    using namespace std;

    Pred mypred;
    vector<int> ivec(100000);
    ///
    auto iter = find_if(ivec.begin(), ivec.end(), ref(mypred));
}
```

kopyalama olmaz!

+ Not:

- Copy gibi yazma algoritmalarının return değeri: yazma işleminin son bulduğu iteratör!
- for each return değeri: bir callable. Range deki öğeleri o callable'a yollar!

Instantiation List:

```cpp
#include <vector>

std::vector v1{ 1, 2, 3 }; // vector<int>
std::vector v2{ v1, v1 }; // vector<vector<int>>
std::vector v3{ v1 }; // vector<int>
```

* Deduction Guides:

- Scott Meyers, derleyicinin nasıl bir tür çıkarttığını anlamak için, typeteller adında bir yol önermiş,

```cpp
template <typename T>
class TypeTeller;

template <typename T>
class Myclass {
public:
    Myclass(const T&)
    {
        TypeTeller<T> x;
    }

private:
    T mx;
};

int main()
{
    Myclass m{ "ali" };
}
```

- `template <typename T> class TypeTeller;` → forward declaration
- `Myclass(const T&)` → const olduğu için T = char [4] / const olmasaydı T = const char [4]
- `TypeTeller<T> x;` → syntax hatası çünkü incomplete type tarafından nesne oluşturuldu!
- `m{ "ali" }` → T için char [4]

```
Build started...
------ Build started: Project: CTAD, Configuration: Release x64 ------
main.cpp
C:\Users\necat\source\repos\CTAD\main.cpp(11,17): error C2079: 'x' uses undefined class 'TypeTeller<T>'
        with
        [
            T=char [4]
        ]
C:\Users\necat\source\repos\CTAD\main.cpp(9): message : while compiling class template member function 'Myclass<char [4]>::Myclass(const T (&))'
        with
        [
```

- `T=char [4]` → compile hatası olarak çıkardığı türü bize söyledi!

```cpp
template <typename>
class TypeTeller;

template <typename T>
class Myclass {
public:
    Myclass(const T& r) : mx{r} {}

private:
    T mx;
};

template <typename T>
Myclass(T)->Myclass<T>;

int main()
{
    int a[40]{};

    Myclass m(a);
}
```

- `->Myclass<T>` → ok'tan sonra, çıkarımın nasıl olmasını istediğimizi yazıyoruz!
- `int main()` → Artık int[40] yerine int* çıkarımında bulunacak çünkü array decay olacak!

```
template <typename T>
class Myclass {
public:
    Myclass(T);
};

template <typename T>
Myclass(T)->Myclass<T&>;
```

Argüman olarak T geldiğinde T& çıkarımı yapılacak!

```
template <typename T>
Myclass(T)->Myclass<T&>;

int main()
{
    int x{};

    Myclass m(x);
}
```

constructer da T& parametreli oldu

```
template <typename T>
Myclass(T)->Myclass<T&>;

int main()
{
    Myclass m(10);
}
```

r value parametre syntax hatası

→ T& parametre geçilmeliydi!

```
Myclass(char)->Myclass<long>;
Myclass(short)->Myclass<long>;
Myclass(int)->Myclass<long>;
Myclass(unsigned)->Myclass<long>;

int main()
{
    Myclass m1('A');
    Myclass m2(23);
    Myclass m3(23u);
}
```

char, short, int ve uint için deduction guide verilmiş

her için long çıkarımı yapılacak!

Aggregate Typelar için ilk değer Atması:

```
template<typename T>
struct Nec
{
    T str;
};

int main()
{
    //Nec<int> x1 = 10;
    Nec<int> x2(10);
    Nec<int> x3{23};
    Nec<int> x4={23};
}
```

syntax error. Aggregateler bu şekilde init edilemez

✓ ✓ ✓

7.5.5. Aggregate Classes

An aggregate class gives users direct access to its members and has special initialization syntax. A class is an aggregate if

- All of its data members are `public`
- It does not define any constructors
- It has no in-class initializers (§ 2.6.1, p. 73)
- It has no base classes or `virtual` functions, which are class-related features that we'll cover in Chapter 15

For example, the following class is an aggregate:

+ struct, union, array gibi

→ C++17 de deduction guide aggregateler için gerekliydi! 20' de değil!

2.30' da bir örnek var. Onu bir daha izle

* Structured Binding: • Cpp 17 ile dile eklendi

- Diziler, tüm üyeleri public olan yapılar ve tuple-like sınıflarla kullanılabilir!
- Gereksiz kopyalamanın önüne geçen bir araç

```
7  struct Nec {
8      int x{};
9      double dval{};
10     std::string s{ "necati" };
11 };
12
13 std::pair<int, double> foo();
14
15 int main()
16 {
17     int ar[3]{ 1, 2, 3 };
18
19     auto [x, y] = foo();
20     auto [a1, a2, a3] = ar;
21
22     Nec mynec;
23
24     auto [i, d, name] = mynec;
25
26 }
27
```

structured binding de auto kullanmak zorunlu!

```
6  auto[x, y] = var;
7  auto[x, y](var);
8  auto[x, y]{var};
```

*

```
struct Nec {
    double dval{};
    int a[5]{};
};

int main()
{
    Nec mynec;

    auto [d, x] = mynec;
}
```

int[5] olarak int* değil!

→ Çünkü,

```
Nec mynec;

auto [d, x] = mynec;
// Nec __abc = mynec;
```

Çıkarım mynec (= sağ taraf) türün yapılır, d, x türün değil!

d ve x bu __abc'nin veri elemanları için kullanılan alias'lar!

```
// Nec __abc = mynec;
//d ===> __abc.dval
//x ===> __abc.a
```

→ Derleyicinin oluşturduğu sınıfa erişmek için,

* Arraylar için Structured Binding:

```
1  int a[2]{0, 1};
2
3  auto [x,y] = a; // x = 0, y = 1
```

can be viewed as:

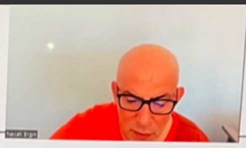
```
1  int a[2]{0, 1};
2
3  int __e[2]{a[0], a[1]};
4  #define x __e[0]
5  #define y __e[1]
```

define göstermelik

Array Elemanlarına Referans:

```
int a[2]{0, 1};

auto& [x,y] = a;
```



the change'll be like:

```
int a[2]{0, 1};

int (&__e)[2] = a; // now the anonymous object do effect on the array
#define x __e[0];
#define y __e[1];
```

```
class Myclass {

    int a{ 10 };
    int b{ 20 };
    int c{ 30};
    friend void foo();
};

void foo()
{
    auto [x, y, z] = Myclass{};
}
```

private öğelere, friendlik verilmiş bir fonksiyon structured binding uygulayabilir ✓

– friend olmasa syntax error!

Structured binding yapılırken, öğe sayısı tamamen aynı olmalı! Kısmi binding yapılamaz ✗

```
class Person {
public:
    std::string m_name{ "murat" };
    std::string m_surname{ "yilmaz" };
};

int main()
{
    using namespace std;

    Person p;

    auto [name] = p;
}
```

2 öğe

tek öğe olduğu için syntax error!