

- Tekrar:

- Derleyiciye kod yazdırın kod \Rightarrow template / metacode

- C++ 20 standartlarında - function template

- class template

- variable template

- alias template

- concept

- Template < typename/class T > şeklinde tanımlılır. T = template argument \rightarrow + yerine mt geldiği durumda
 \hookrightarrow Template parametre.

* Template parametreler kamilik gelecek template argümanı explicit sayılmıyorsa bile, compiler onaylar.

```
1 template <typename T, typename U, typename F>
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

1) deduction (çıkarım)
function templates only \rightarrow std::swap() \rightarrow tür belirtmeden
class template (CTAD)

2) explicit template argument \rightarrow std::vector<int> gibi infer belirtile.

* Non-Type Parameter: \rightarrow bir tsm yerine bir sabit kullanılması

```
template<int x>
class MyClass {
    // ...
};
```

x makinesi 20 ise, derleyici, template kodda 20 sabitini yazar.

* Type ve non-type parametreler bir arada kullanılabilir. (std::array)

```
int main()
{
    std::array<int, 10> ax;
```

* Function Templates:

```
void swap(int& x, int& y)
{
    int temp{ x };
    x = y;
    y = temp;
}

void swap(double& x, double& y)
{
    double temp{ x };
    x = y;
    y = temp;
}
```

* Template Olmadan böyle overload ettirerek, swap yapılır. Template ile aynı şekilde yazışları elde edilir. (class swap gibi...)

• Function template ile ilgili kavramlar:

- Template Implementation:

- Derleyicinin template'leri orijinal kodu üretmesi

- Template Name:

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

func burada fonksiyon adı / Func, template function adı.

- Template Specialization:

- Template imlementasyonunu sonra elde edilen türün kendisi (int mi, double mı anlattık)
- Neceğin "öçümü" olarak ifade ettiğimiz şey specialization'dır.
- Template ID olarak da geçer. Tek template'e yazılan specialization uygulanır.

• Dikkat:

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

template <typename T>
void foo(T x)
{
}
```

Burada T'nin bir scope'u var. Aslında bu fonksiyon da T yazmamızla.

Bu yüzden ayrı olarak tanımlı

- Ayrıca template'ler de overload edilebilir.

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

int main()
{
    func<int>(3.4);
}
```

→ specialization uygulanır
türü, function call uygulanır

explicit template argument

double olsa bile
specialization: int.

*Template Argument Deduction:

- auto type deduction'a çok benzer, yalnızca 1 tane var.

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

```
template <typename T>
void func(T x)
{
    TypeTeller<T> t;
}
```

→ Incomplete type bir sınıfı çağırıyor

```
int main()
{
    const int ival = 10;
    func(ival);
}
```

→ constluk ve referanslık, auto deduce olduğu gibi düşüncelidir !!

→ Burada type deduction'ın ne olduğunu bilmeden yazılmıyorsa, özellikle var hene

Started: Project: MART2022, Configuration: Release Win32 -----

T2022\main.cpp(11,16): error C2079: 't' uses undefined class 'TypeTeller<int>'
T2022\main.cpp(19): message : see reference to function template instantiation 'void func<int>(T)' being compiled

→ Compiler nota olarak bize törü söylüyor.

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

```
int main()
{
    int a[] { 1, 4, 5 };
    func(a);
}
```

→ Array decay ne, a'nın ilk elemanının adresine dönüyor. T = int* olur.

```
template <typename T>
void func(T &x) // void func(int (&x)(int))
{
    TypeTeller<T> t;
}

int foo(int);

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


*Divide!!

```
template<typename T>
void func(T x, T y);
```

```
int main()
{
    func(10, 1.20);
}
```

int

double

Burada Ambiguity var!

```
template<typename T>
void func(T x, T y);
```

const char *

```
int main()
{
    func("ali", "veli");
}
```

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

const char [N]

```
int main()
{
    func("ali", "ece");
}
```

"veli" olsa
kod bozulurdu.

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

```
template <typename T, typename U>
void func(T (*)(U))
{
}
```

T = int

U = double olur.

```
int foo(double);
```

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


Normal olarak C++ dilinde reference to reference yoktur. I

→ reference to reference olursa, reference collapsing uygulanır.

reference collapsing

T&	&	T&
T&	&&	T&
T&&	&	T&
T&&	&&&	T&&

4. 2.05 önce
gözet

```
template <typename T>
```

```
void func(T&& x)
```

→ Bu R value değil, Forwarding reference.

Argümanın, value kategorisi neyse, T türünün çıkarması da aynı.
Reference collapsing'e göre x de rnt b olucak!!

```
int main()
```

```
{  
    int x = 10;  
    func(x);  
}
```

→ T'nin türü int'dir çünkü x L value

```
template <typename T>
```

```
void func(T&& x)
```

```
{  
    T rnt;  
    x = rnt; // I  
}
```

```
int main()
```

```
{  
    func(x, 10);  
}
```

```
template <typename T>
```

```
void func(T&&)
```

→ Bunun adı bu yüzden forwarding reference / universal reference

```
int main()
```

```
{  
    int x = 10; I  
    const int cx = 20;  
    func(x);  
    func(cx);  
    func(10);  
}
```