- Tekrar:
- Derleyiciye kod yazdıran kod => template / meta code
- C++ 20 standartlarında - function template
  - class template
  - variable template
  - alias template
  - concept
- Template < typename / class T > şeklinde tanıtılır. T = template argument → + yerine int geldiği durumda
  ↳ Template parameter.

* Template parametresine karşılık gelecek template argümanı explicit söylenmese bile, compiler anlayabilir.

```
template <typename T, typename U, typename F>


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

2) explicit template argument  }→ std::vector<int> gibi sınıfı belirtiriz.
```

* Non-Type Parameter: → bir tür yerine bir sabitin kullanılmasıdır

```
template<int X>
class Myclass {

    X
};
```

x'in değeri 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:

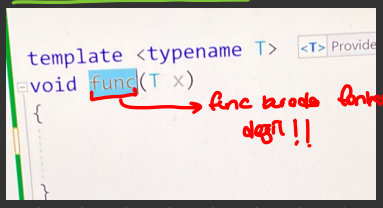
```
void swap(int& x, int& y)
{
    int temp{ x };
    x = y;
    y = temp;
}

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

• Template olmadan böyle overload ettirerek, swap yapıyordu. Template ile sonsuz sayıda veri yapısını elde edebildik. (Class swap gibi...)

- Function template ile ilgili kavramlar: - Template Instantiation:
  - Derleyicinin template'ten artık üretilen kodu üretmesi

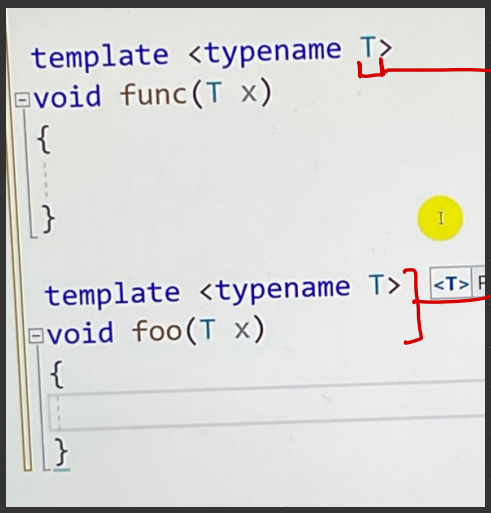
- Template Name:



func burada fonksiyon adı değil!! / Func, template function adı.

- Template Specialization:
  - Template instantiation'dan sonra elde edilen ürünün kendisi (int mi, double mı anlaşılması)
  - Neden "açılımı" olarak ifade ettiğimiz şey specialization'dır.
  - Template ID olarak da geçer. Tek template'e yazılacak specialization yapılabilir.

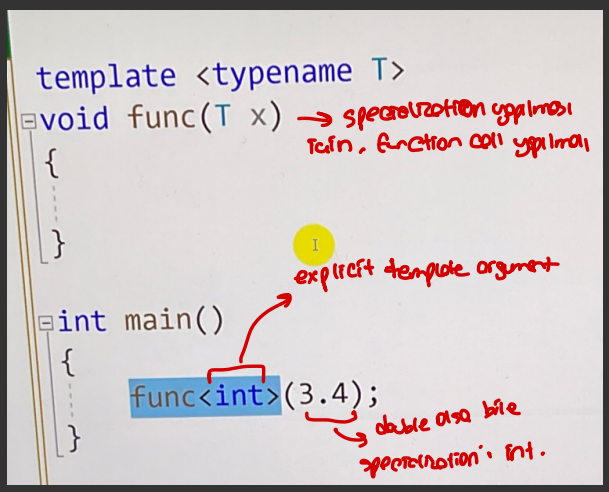
*Dikkat:



Buradaki T'nin bir scope'u var. Aşağıdaki foo için de T yazmamız.

Bu yüzden ayrı olarak tanımladı

- Ayrıca templateler de overload edilebilir.



specialization yapılması için, function call yapılmalı

explicit template argument

double olsa bile specialization: int.

*Template Argument Deduction:

- auto type deduction'a çok benzer, yanlızca 1 fark var.

```
template <typename T>
class TypeTeller;


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

int main()
{
    const int ival = 10;

    func(ival);
}
```

→ Incomplete type bir sınıfa doğru yaptı

→ Burada type deduction'ın ne olduğunu bulmamıza yardımcı bir. Çözüldü ve hepsi

→ const'luk ve referanslık, auto'da olduğu gibi düşecektir !!

→ Compiler nota olarak bize türü söyledi.

```
arted: 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
```

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

}


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

    func(a);
}
```

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

```
template <typename T>                <T> Provide sample template arguments for IntelliSense
void func(T &x)  // void func(int (&x)(int))
{
    TypeTeller<T> t;
}

int foo(int);

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

*Dikkat!!

```
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);

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

T → const char *

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

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

T → const char [4]

"veli" olsa kod bozulurdu.

```
template <typename T>
class TypeTeller;

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

int foo(double);

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

T = int

U = double olur.

Normal olarak C++ dilinde reference to reference yoktur.

→ reference to reference olursa, reference collapsing yapılır.

reference collapsing

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

+ 2.05 ders
güzel

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

}

int main()
{
    int x = 10;

    func(x);
}
```

T&& → Bu R value değil, Forwarding reference.

Argüman, value kategorisi neyse, T türünün anlamı da aynı.
Reference collapsing'e göre X' de int & olacak!!

T'nin türü int& const X L value

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

}

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

T = int
X = int &&

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

}

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

Bunun adı bu yüzden forwarding reference / universal reference