

## \* Teker:

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

→ Her argumen temsil etmektedir.

→ Ambiguity yaratabilecek durumlar yaratır.

```
template <typename T>
T foo();
```

→ Return type ile temsil etmektedir.

```
int main()
{
    int x = foo();
```

→ foo<int>() şekilde explicit tanımlanır.

```
template <typename R, typename T, typename U>
R sum(T x, U y)
{
    /**
     * return type explicit olarak belirtildi
     * return degerinin double olmasini
     * icin explicit olarak türdeğişiklik yapılıp
     * eklenmiştir.
     */
    return x + y;
}
```

```
int main()
{
    int ival = 3;
    double dval = 3.4;

    auto result = sum<double>(12, 4.5);
    std::cout << "result = " << result << "\n";
}
```

return type explicit olarak belirtildi  
return degerinin double olmasini icin explicit olarak türdeğişiklik yapılıp eklenmiştir.

## \* Trailing Return Type:

```
#include <iostream>

int func(int x, int y)
{
    return x + y;
}

auto func(int x, int y)->int
{
    return x + y;
}
```

syntax bo gelmedi  
return type: auto  
declaration sonuna  
→ type

\* C'den Matirikler: → Return değer function pointer den bir fonksiyon

```
int foo(int, int);

// int()(int, int);
typedef int(*Fptr)(int, int);

Fptr func()
{
    return &foo;
}
```

```
int foo(int, int);

// int()(int, int);
//typedef int(*Fptr)(int, int);

int(* func())(int, int)
{
    return &foo;
}
```

template <typename T, typename U> <T>
auto sum(T t, U u)-> decltype(t + u) → scope problemi cozuldu. / Return degerin decltype() kullanildi.

\* Auto Return Type:

- C++ 11 ile eklendi.
- Fonksiyonun direct tanimlanmasi gerek. / trailing return type'la deklarasyon yeterli

```
auto func(int x, double y)
{
    return x + y;
```

```
template <typename T, typename U>
auto func(T x, U y)
{
    return x + y;
```

→ Abbreviated Template Syntax: → C++ 10. sie geldi

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



```
void func(auto x)
```

Herketlenmesi - Decltype operandun value category'ine bağlı.

P P → T

L → T & → decltype(L)

X → T && → decltype(X)

ilk pointer versya L value

### \* Decltype (auto);

→ İlk değer varsa refde, decltype'in operandı olmaz gibi ele alınır. → Çıktılarda buna tanırı.

```
int main()
{
    int x = 10;
    decltype(auto) y = (x);
}
```

y'nm tür int& ols.

int &

```
int main()
{
    int x = 10;
    int* ptr = &x;

    decltype(auto) y = *ptr;
}
```

y'ne int

```
template <typename T> decltype(auto) func(T& x)
{
    return x; ]> ival ile ilk değer无缝接
}                                [ X → int &]

template <typename T>
auto foo(T& x)
{
    return x; ]> ival ile ilk değer无缝接
}                                [ X → int

int main()
{
    int ival = 56;
    auto x = func(ival);
}
```

## \* Class Templates:

→ STL deki bir çok sınıf ve data structure, class template'tır.

\* inline fonksiyonlarında, "T" belirtilecezinde değil

```
template <typename T>
class Nec { template name ✓
public:
    void foo()
    {
        Nec x; bu da
        int
        olsun! olar
    }
};

int main()
{
    Nec<int> x;
    x.foo();
}
```

```
template <typename T>
class Nec {
public:
    void foo(T x);
    void func(T& x);
};

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

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

→ inline olmamışlarında  
⟨T⟩ *zannılsın!!*

→ Class template bir sınıf değildir *soblenin olusun specialization classtır!!* Ve her specialization olsa bir classtır.

```
template <typename T>
class A {
    // ...
};

int main()
{
    std::list<std::vector<A<int>>> soblen argman ...
    soblen argman ...
}
```

→ class template'ler, template olmamın olasılık, başka class template kullanabilir.

## \* Default Class Template Argument:

```
template <typename T = int>
class MyClass { wi }

int main()
{
    MyClass<double> x;
    MyClass<> y; ←
    //MyClass<int> y;
}
```

Fonksiyonların default argumentlarının olması ne olsun

```
template <typename T = double, typename U = int>
class MyClass {
};

int main()
{
    MyClass<int, long> x;
    MyClass<int> y; // MyClass<int, int> y;
    MyClass<> z; // MyClass<double, int> z;
}
```

Bu defaultsa

bu da def argument  
völmek zorunda

```
template <typename T>
class Nec {
};

template <typename T, typename U = Nec<T>>
class MyClass {
};

int main()
{
    MyClass<int> m1; // myclass'in int, Nec<int> olmamı
    std::cout << typeid(m1).name() << "\n";
}
```

eger belirtimizse  
default argumanı T  
ne ise bu da D

#Function Template Overload'u:

template <typename T> [Template]
void func(T x);

⇒ Aynı semantik. Bir template, biri genel func  
olسا bile bu function overloaddir.

Void func(int x); [beraber fonksiyon]

```
template <typename T>
void func(T)
{
    std::cout << "function template type T is : " << typeid(T).name() << "\n";
}
```

```
void func(int)
{
    std::cout << "non template function func(int)\n";
}
```

```
int main()
{
    func(78); // genel func int cogeler.
    func(12u); // template olusturulur.
    func('A'); // promotion / conversion olmaz X
    func("ali");
}
```

```
int main()
{
    func<int>(12);
}
```

Olsun, template int cogeler.  
Yani da func < -> (12)

Bu belirtimiz esaslı.

template olusturulur.

Promotion / conversion olmaz X

→ Bir Fonksiyonu Yalnızca Tek Bir Parametre ile Gözürmek:

```
template <typename T>
void func(T) = delete;
```

```
void func(unsigned int);
```

Yazılım abbreviates  
Template Syntax ile (C++20)

```
void func(auto) = delete;
```

```
void func(unsigned int);
```

\* Yalnızca unsigned int çağırılır. Geriye Delete edildi!!!

→ Instantiation syntax holesi oluşur.

### \* Partial Ordering Rules

\* Fonksiyon sabitleri, başka fonksiyon sabitleri ile overload edilebilir. Bu türde norma her ikisinden de olumsuz olumka verse, partial ordering rules deneye girer.

→ Eğer bu olmuyorsa / done specific yolsa = Ambiguity !!

### \* Explicit Specialization (Full Specialization):

- Belli bir template argümanları için, örneğin "Anon template olarak yorumlu yarın" bize yorumunu veriyor.

```
using namespace std;

template <typename T> Max(T x, T y)
{
    return x > y ? x : y;
}

int main()
{
    std::cout << Max("yesim", "belgin") << "\n";
    std::cout << Max("belgin", "yesim") << "\n";
}
```

Burada lexicographical compare oluyor X → Alfabetic  
sıraya göre  
değil

→ Bu template stringler hava sorunuz colisir. (String'in sonuna s eklediğimizde)

```
template <typename T>
T Max(T x, T y)
{
    return x > y ? x : y;
}

template <>
const char* Max(const char* p1, const char* p2)
{
    std::cout << "explicit spec.\n";

    return std::strcmp(p1, p2) > 0 ? p1 : p2;
}

int main()
{
    std::cout << Max("yesim", "belgin") << "\n";
    std::cout << Max("belgin", "yesim") << "\n";
}
```