

* Meta Functions:

- Sınıflarla oluştururlar.

- Compile time'da bir tarzı bulgular / değer hesaplarlar, sınıfları kullanır.

```
template <int n>
struct Factorial {
    const static int value = n * Factorial<n - 1>::value;
};

template<>
struct Factorial<1> {
    const static int value = 1;
};

int main()
{
    int a[Factorial<5>::value];
}
```

compile time'da
recurrene bir seferde
cagrilir.

explicit specialization`ı base case
olarak kullandi

compile time'da
belir oldugu tara, array size`ı olustur.

```
template <int n>
struct Nec : public Nec<n - 1>{
    Nec()
    {
        std::cout << n << " ";
    }
};

template <>
struct Nec<0> {
};

int main()
{
    Nec<100> x;
}
```

0 icin specialization yolu

→ i den 100 ye kadar yazdir.

* Explicit Specialization dan Meyvan Gibi Mülakat Driller:

```

template<typename T>
void func(T)
{
    std::cout << "1";
}

template<>
void func(int*)
{
    std::cout << "2";
}

template<typename T>
void func(T*)
{
    std::cout << "3";
}

int main()
{
    int* p = nullptr;
    func(p);
}

```

→ Cevap: 3

- 2 numara / explicit specialization, template overload resolution'a katılmaz !!
- ★ Eğer 1 numaradan specialization oluşturmak, `int*(2)` durumunda explicit kullanır.

⇒ Explicit'in yazılışı şıra göre yanlış

```

template<typename T>
void func(T)
{
    std::cout << "1";
}

template<>
void func(int*)
{
    std::cout << "2";
}

template<typename T>
void func(T*)
{
    std::cout << "3";
}

template<>
void func(int*)
{
    std::cout << "4";
}

int main()
{
    int* p = nullptr;
    func(p);
}

```

↳ Buna cat

↳ Buna cat

↳ Cevap
U

* Static Sınıf Elemanları ve Member Fonksiyonları için Explicit Specialization.

```

template <typename T>
class A {
public:
    static int x;
};

template <typename T>
int A<T>::x = 99;

template <>
int A<int>::x = 20;

int main()
{
    std::cout << A<double>::x << "\n";
    std::cout << A<int>::x << "\n";
}

```

Her specialization
rem aynı bir
static int'ı var.

```

template <typename T>
struct Nec {
    void func()
    {
        std::cout << "primary template\n";
    }
};

template <>
void Nec<double>::func()
{
    std::cout << "explicit spec. for Nec<double>\n";
}

int main()
{
    Nec<int>{}.func();
    Nec<double>{}.func();
}

```

```

template <typename T> <T> Provide sample template arguments for IntelliSense
class A {
public:
    inline static int x{};
};

int main()
{
    std::cout << A<double>::x++ << "\n";
    std::cout << A<int>::x++ << "\n";
    std::cout << A<long>::x++ << "\n";
    std::cout << A<double>::x++ << "\n";
}

```

→ Bu eğer bir sınıf nesnesi oluydı, tür sınıfları taraf 1 adet X olurdu.

→ Şimdi specialization başına 1 adet X var

→ Cevap 0
0
0
1

* Partial Specialization: → Varyans sınıf templatelerinde

```

template <typename T>
class MyClass {
public:
    MyClass()
    {
        std::cout << "primary template\n";
    }
};

template <typename T>
class MyClass<T*> {
public:
    MyClass()
    {
        std::cout << "partial spec. T *\n";
    }
};

int main()
{
    MyClass<int> m1;
    MyClass<int*> m2;
    MyClass<int**> m3;
    MyClass<double*> m4;
    MyClass<double> m5;
}

```

→ her türini
eden T'ye de yonitir.

```

Microsoft Visual Studio Debug Console
primary template
partial spec. T *
partial spec. T **
partial spec. T ***
primary template
D:\CONCURRENCY\PACA_2022\Debug\PACA_20
Press any key to close this window...

```

```

template <typename T, typename U>
class MyClass {
public:
    MyClass()
    {
        std::cout << "primary template\n";
    }
};

template<typename T> <T> Provide sample template arguments for IntelliSense - /
class MyClass<T, T> {
public:
    MyClass()
    {
        std::cout << "partial \n";
    }
};

```

→ aynı türde
genel
partial

* Member Template:

```
template <typename T>
class MyClass {
public:
    template <typename U> void func(MyClass<U> x)
    {
        std::cout << "typeid(*this) : " << typeid(*this).name() << "\n";
        std::cout << "typeid(x) : " << typeid(x).name() << "\n";
    }
};
```

```
int main()
```

```
{  
    MyClass<int> m1;  
    MyClass<double> m3;  
    m3.func(m1);
```

Sıradanın eylemleri, aynı sınıf gizlenen member template

Arka tır specialization ile elde edildi.

→ STL'de çok sık kullanılan sınıf abstrakter `pair` ve `tuple` → tired form

pair tane aynı form!

tuple'da degerler farklı olabilir.

```
int main()
```

```
{  
    using namespace std;
```

```
pair<int, string> p; // pair nesnesi default initialize  
                      // constructor / factory default  
                      // init ekstra pair value  
                      // isteme edilebilir  
std::cout << p.first << "\n";  
std::cout << p.second.length() << "\n"; ↓  
                                int=0  
                                ptr = nullptr.
```

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

```
struct Pair {
```

```
    template <typename X, typename Y>
    Pair& operator=(const Pair<X, Y>&);
```

I *Bu özelde member template
yapılmış, "int = string" bir pair
esleştirmek.*

```
using namespace std;
```

```
int main()
```

```
{  
    Pair<double, double> p1;
```

```
    Pair<int, int> p2;
```

```
p1 = p2;
```

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

```
struct Pair {
```

```
    Pair& operator=(const Pair&);
```

*normal bir copy assignment
hale gelmesi olur.*

```
using namespace std;
```

```
int main()
```

```
{  
    Pair<double, double> p1;
```

```
    Pair<int, int> p2;
```

```
p1 = p2;
```

* C de bir tane yarar 2 deger return etmesi:

```
struct Retval {  
    int x;  
    double dval;  
};  
  
struct Retval foo(void)
```

→ ihr elementi olan bir struct olusturabilir.

→ C'ta da bunu yapisile saglasi!

* Pair özet:

```
int main()  
{  
    using namespace std;  
  
    pair<int, double> p1;  
    pair<long, char> p2{ 34L, 'A' };  
    //since C++17  
    pair p3{ 3.4f, 8.7 };  
  
    auto p4 = make_pair(34, 4.5);
```

(local variable) std::pair<int, double> p4

Search Online

```
template <typename T, typename U>  
class MyClass {  
public:  
    MyClass(const T&, const U&);  
};  
  
template <typename T, typename U>  
MyClass<T, U> make_myclass(const T& t, const U& u)  
{  
    return MyClass<T, U>{t, u};  
}
```

make pair kolu

* Pair'in elementleri yine pair olabilir.

* Pair'in Tnesesi konusunu yok. Function template ile o nesneyi yazdirabiliriz.

```
#include <bitset>  
  
template <typename T, typename U>  
std::ostream& operator<<(std::ostream& os, const std::pair<T, U>& p)  
{  
    return os << "[" << p.first << ", " << p.second << "]";  
}  
  
int main()  
{  
    using namespace std;  
  
    std::cout << make_pair(make_pair(12, 4.5), make_pair("ali", bitset<16>(23))) << "\n";  
}
```

Microsoft Visual Studio Debug Console
[[12, 4.5], [ali, 000000000010111]]

D:\CONCURRENCY\PACA_2022\Debug\PACA_2022.exe (process 24472) exited with code 0.
Press any key to close this window . . .

* Portakal kendi arsında kasıtlarının yelpizisidir. [first < first, second < second] ile de aynı logik false / true oluyor.

* Altış Template (T'ler esasında şablon):

```
typedef int (*FCMP)(const char*, const char*);
```

```
using FCMP = int (*)(const char*, const char*);
```

return değerini olası ve 2
const char* kasıtların farklı bir
pointer'i

→ using'in sonunda diğer olası.
Ayrıca typedef, template olusurken
kullanılmaz X.

```
template <typename T>  
using Ptr = T*;
```



```
int main()  
{  
    Ptr<int> p;
```

```
template <typename T, int size>  
using Array = T[size];
```



```
int main()  
{  
    Array<int, 20> a[5];  
    int a[5][20];
```

elde edilen array 0'dan
5 elementli array türündür

```
template <typename Key>  
using gset = std::set<Key, std::greater<Key>>
```



```
int main()  
{  
    std::set<int, std::greater<int>> myset;
```

gset<int>
yazınca aynı sayıya elde edebiliriz artı.

```
template <typename T>  
using add_pointer_t = typename std::add_pointer<T>::type;
```



```
int main()  
{  
    //std::add_pointer<int>::type  
    add_pointer_t<int>
```

type trait
bu şekilde yazılursa

STL'deki
hali
Daha
verenek sonra