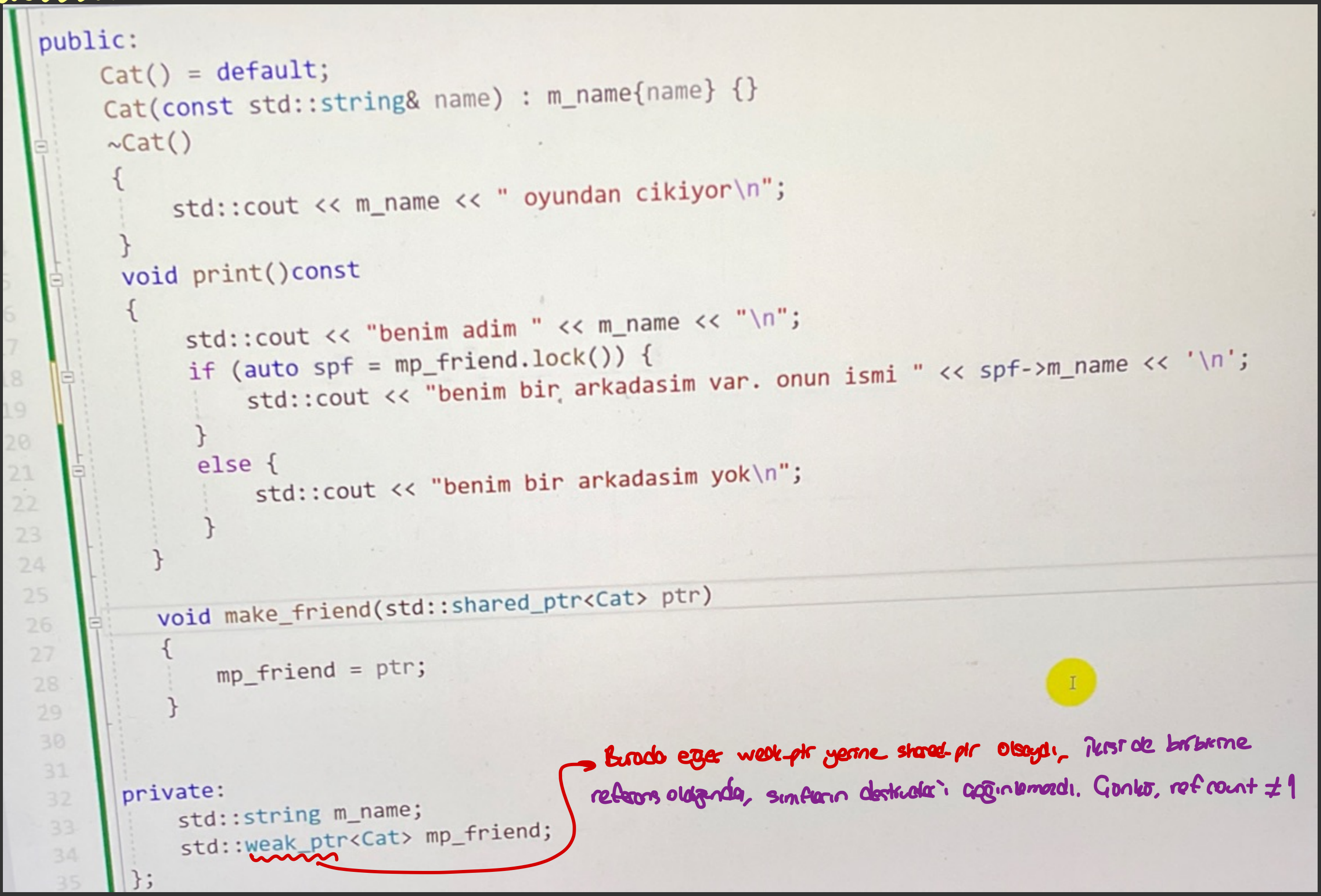
*** Cyclic Reference:**

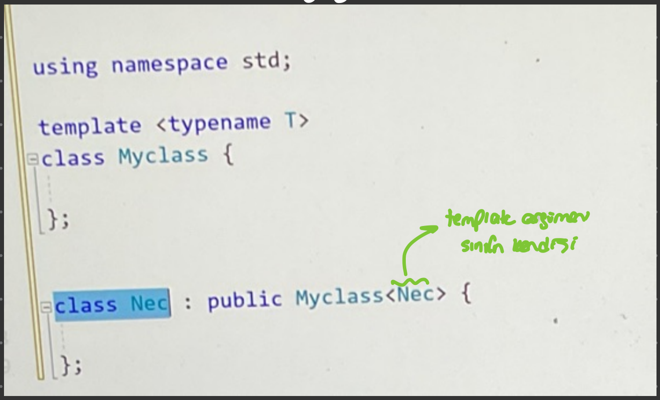


```cpp
public:
    Cat() = default;
    Cat(const std::string& name) : m_name{name} {}
    ~Cat()
    {
        std::cout << m_name << " oyundan cikiyor\n";
    }
    void print()const
    {
        std::cout << "benim adim " << m_name << "\n";
        if (auto spf = mp_friend.lock()) {
            std::cout << "benim bir arkadasim var. onun ismi " << spf->m_name << '\n';
        }
        else {
            std::cout << "benim bir arkadasim yok\n";
        }
    }

    void make_friend(std::shared_ptr<Cat> ptr)
    {
        mp_friend = ptr;
    }

private:
    std::string m_name;
    std::weak_ptr<Cat> mp_friend;
};
```

Burada eğer weak_ptr yerine shared_ptr olsaydı, ikisi de birbirine referans olduğunda, sınıfların destructor'ı çağrılmazdı. Çünkü, ref count ≠ 1

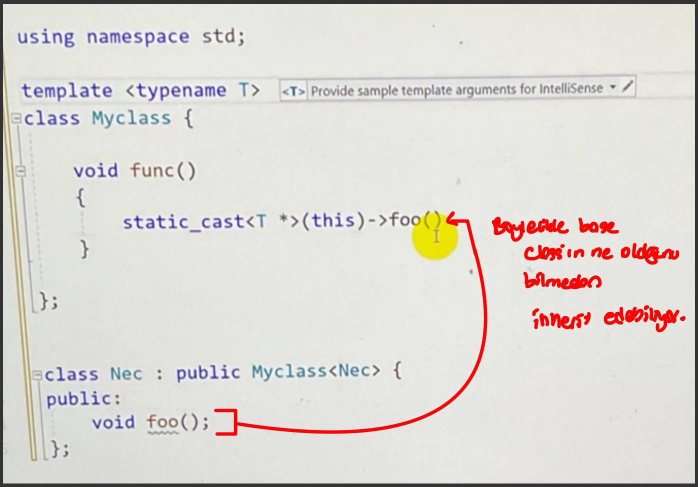
***CRTP: Curiously Recurring Template Pattern**

- Sınıf şablonundan kalıtım yoluyla bir sınıf elde edilmesi



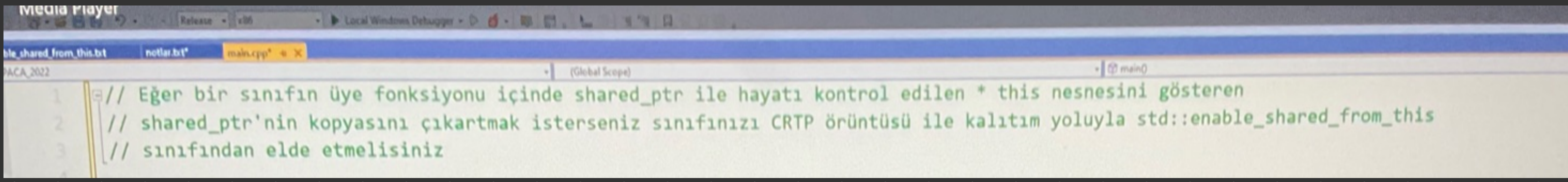
```cpp
using namespace std;

template <typename T>
class Myclass {

};

class Nec : public Myclass<Nec> {

};
```

template argümanı sınıfın kendisi



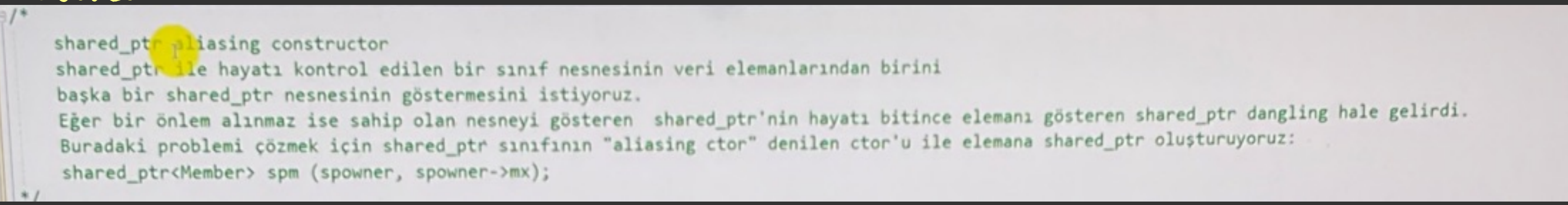
```cpp
using namespace std;

template <typename T>
class Myclass {

    void func()
    {
        static_cast<T *>(this)->foo();
    }

};

class Nec : public Myclass<Nec> {
public:
    void foo();
};
```

Böylelikle base class'ın ne olduğunu bilmeden inherit edebiliyor.

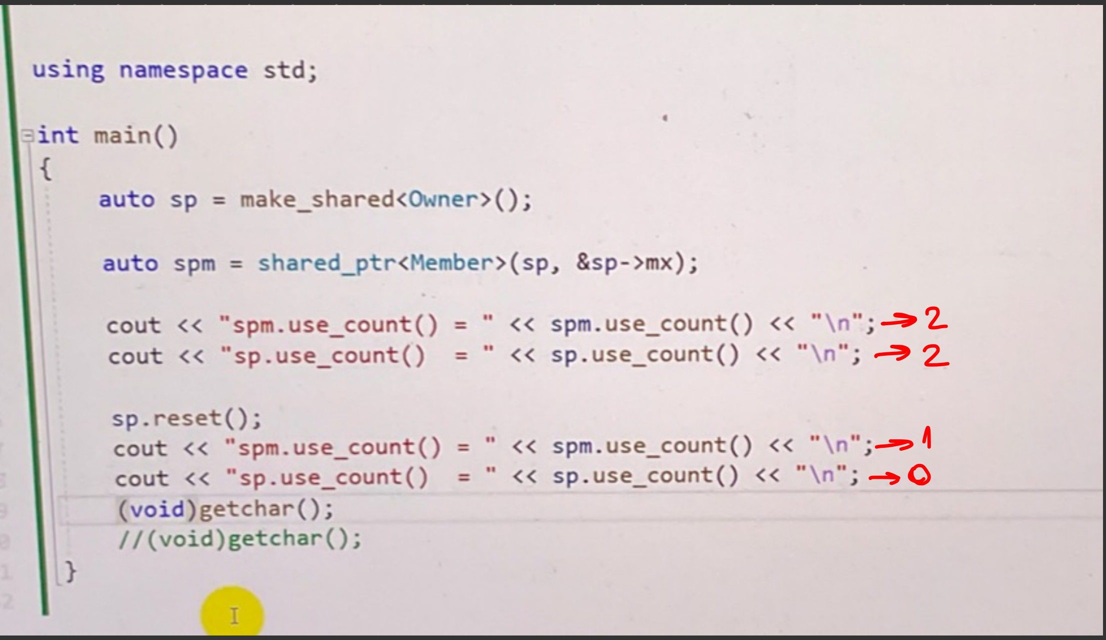


```cpp
// Eğer bir sınıfın üye fonksiyonu içinde shared_ptr ile hayatı kontrol edilen * this nesnesini gösteren
// shared_ptr'nin kopyasını çıkartmak isterseniz sınıfınızı CRTP örüntüsü ile kalıtım yoluyla std::enable_shared_from_this
// sınıfından elde etmelisiniz
```

* Shared_ptr Aliasing Constructor:



```
/*
  shared_ptr aliasing constructor
  shared_ptr ile hayatı kontrol edilen bir sınıf nesnesinin veri elemanlarından birini
  başka bir shared_ptr nesnesinin göstermesini istiyoruz.
  Eğer bir önlem alınmaz ise sahip olan nesneyi gösteren  shared_ptr'nin hayatı bitince elemanı gösteren shared_ptr dangling hale gelirdi.
  Buradaki problemi çözmek için shared_ptr sınıfının "aliasing ctor" denilen ctor'u ile elemana shared_ptr oluşturuyoruz:
  shared_ptr<Member> spm (spowner, spowner->mx);
*/
```

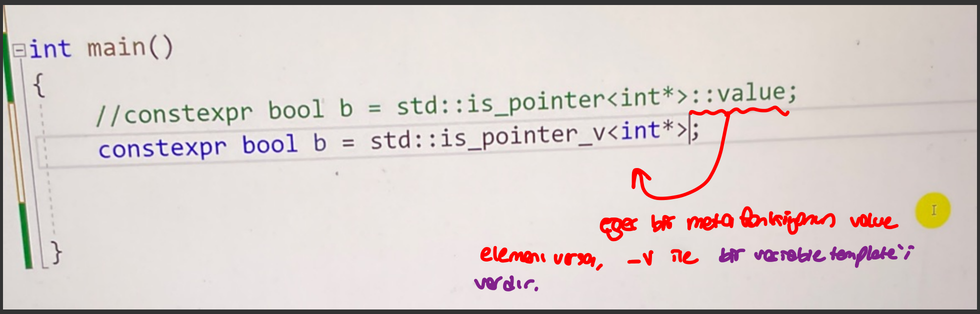


```
using namespace std;

int main()
{
    auto sp = make_shared<Owner>();

    auto spm = shared_ptr<Member>(sp, &sp->mx);

    cout << "spm.use_count() = " << spm.use_count() << "\n"; → 2
    cout << "sp.use_count()  = " << sp.use_count() << "\n"; → 2

    sp.reset();
    cout << "spm.use_count() = " << spm.use_count() << "\n"; → 1
    cout << "sp.use_count()  = " << sp.use_count() << "\n"; → 0
    (void)getchar();
    //(void)getchar();
}
```

—— 0 —— 0 ——

Type Traits Library:

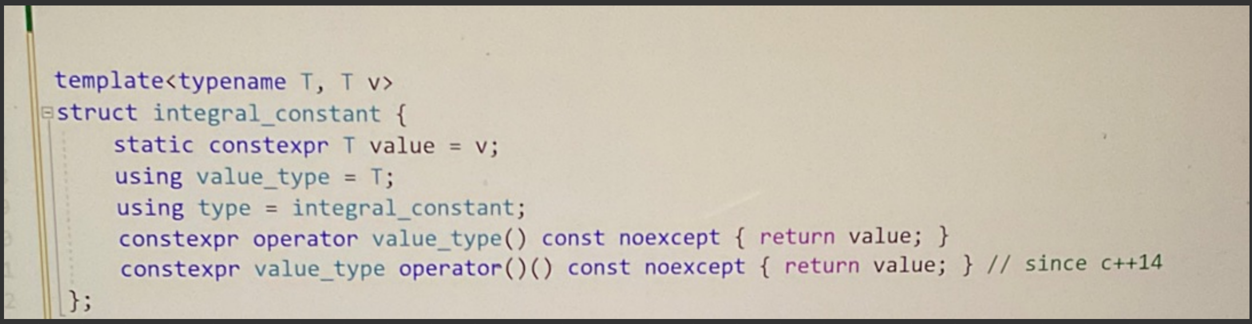
→ metafunction library
- → generic bir componentin amacı compile timeda bir sabit elde etmek
- → 〃 〃 〃 bir tür elde etmek



```
int main()
{
    //constexpr bool b = std::is_pointer<int*>::value;
    constexpr bool b = std::is_pointer_v<int*>;
}
```

eğer bir meta fonksiyonun value elemanı varsa, _v ile bir variable template'i verdir.

* Integral Constant Class Template.



```
template<typename T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant;
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; } // since c++14
};
```

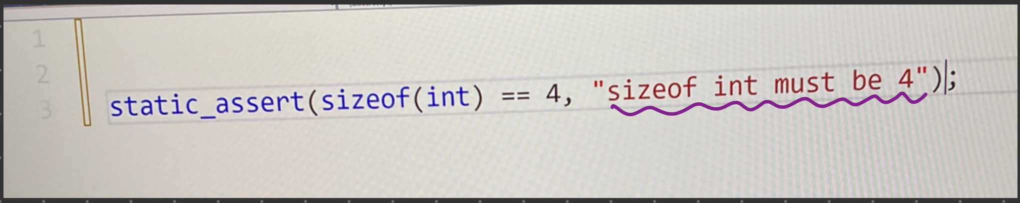
→ Varlık nedeni diğer metafonksiyonlar kalıtım yoluyla bundan elde edilmesi.

→ iki parçalık videonun ilk partının sonuna kadar hep örnek vardı.

***Static Assert:**

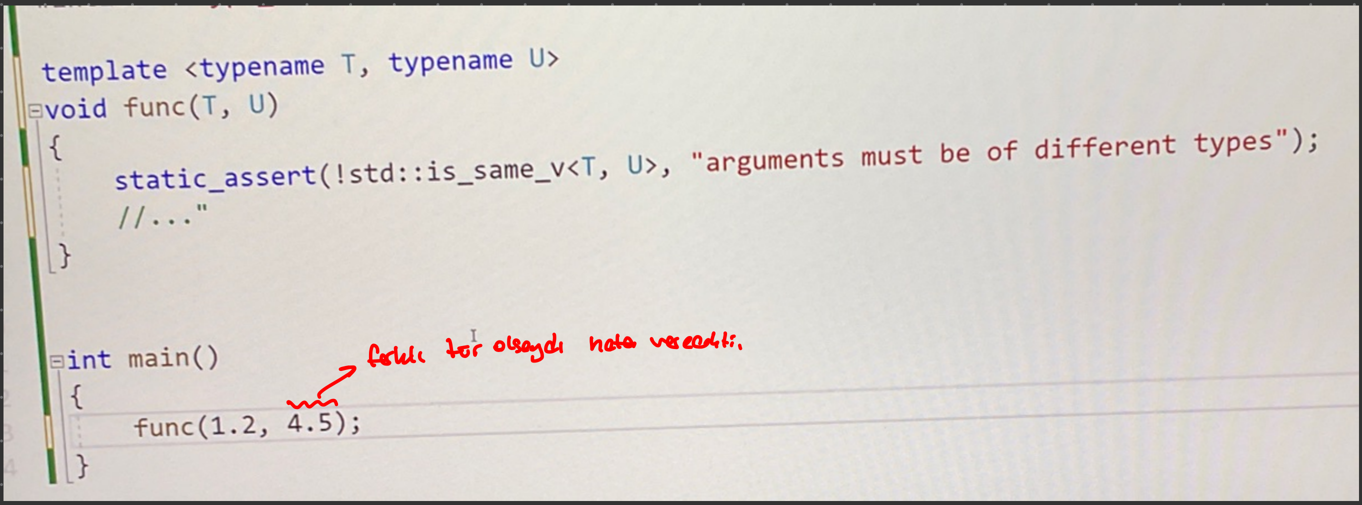
→ Static Assert bir keyword. Kütüphane eklemeye gerek yok

→ Cpp 17 öncesinde bir string literali ile kullanılması gerekiyordu.



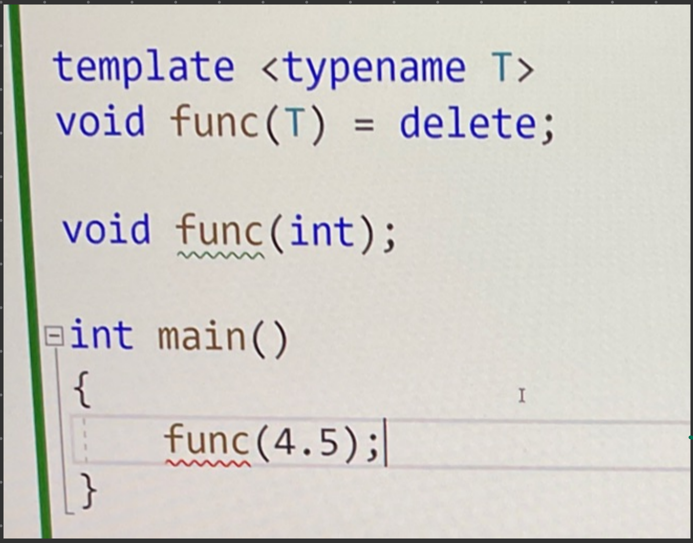
```cpp
static_assert(sizeof(int) == 4, "sizeof int must be 4");
```

→ Compile time'da eğer bu koşul sağlanmazsa static assert failed compile error verecek ve benim yazdığımız string kullanılacak.



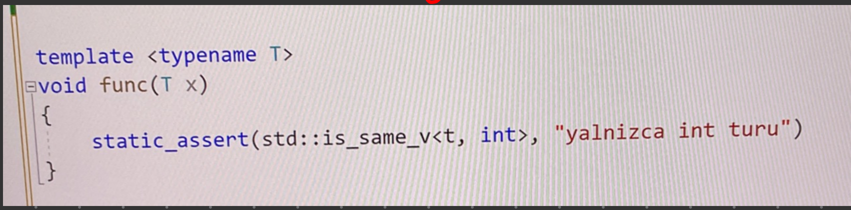
```cpp
template <typename T, typename U>
void func(T, U)
{
    static_assert(!std::is_same_v<T, U>, "arguments must be of different types");
    //..."
}


int main()
{
    func(1.2, 4.5);
}
```

farklı tür olsaydı hata verecekti.



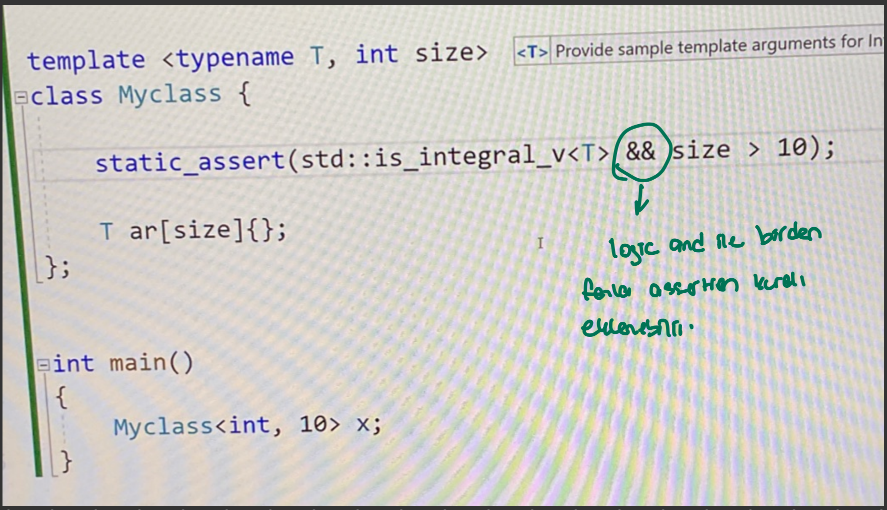
```cpp
template <typename T>
void func(T) = delete;

void func(int);

int main()
{
    func(4.5);
}
```

→ Fonksiyon sadece int argümanla çağrılabilir.



```cpp
template <typename T>
void func(T x)
{
    static_assert(std::is_same_v<t, int>, "yalnizca int turu")
}
```

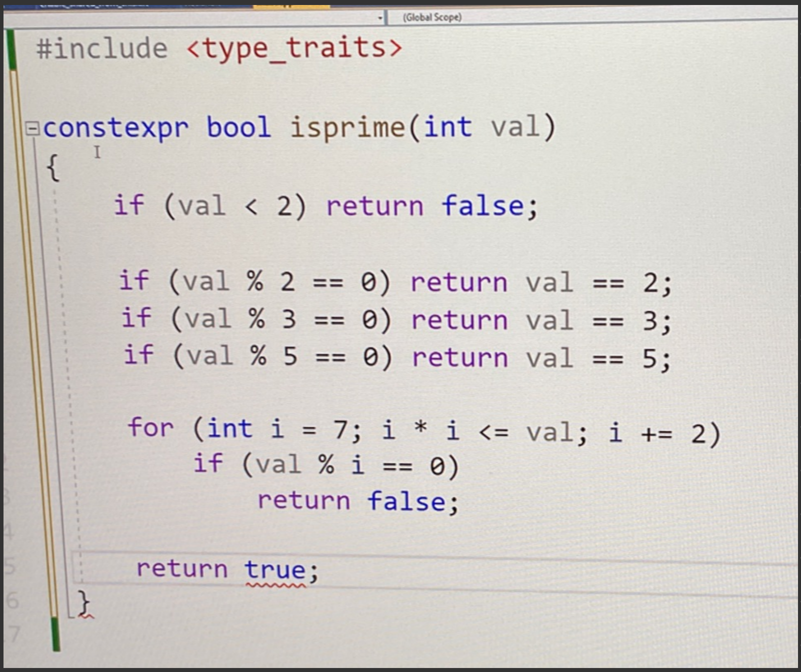
static assert alternative



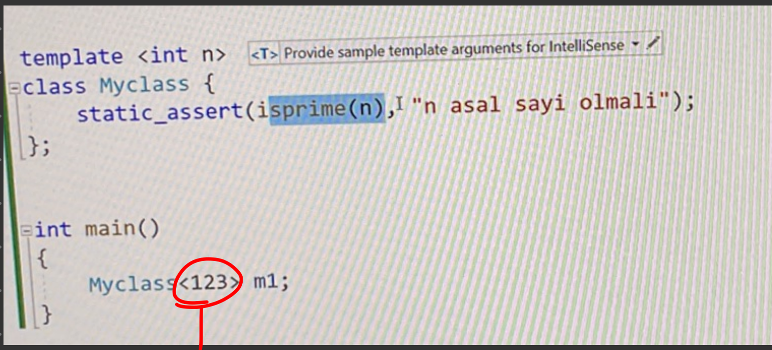
```cpp
template <typename T, int size>
class Myclass {

    static_assert(std::is_integral_v<T> && size > 10);

    T ar[size]{};
};


int main()
{
    Myclass<int, 10> x;
}
```

logic and ile birden fazla assertion koşulu eklenebilir.

+ Static assertlarında constexpr bir fonksiyona çağrı yapılabilir.



```cpp
#include <type_traits>

constexpr bool isprime(int val)
{
    if (val < 2) return false;

    if (val % 2 == 0) return val == 2;
    if (val % 3 == 0) return val == 3;
    if (val % 5 == 0) return val == 5;

    for (int i = 7; i * i <= val; i += 2)
        if (val % i == 0)
            return false;

    return true;
}
```

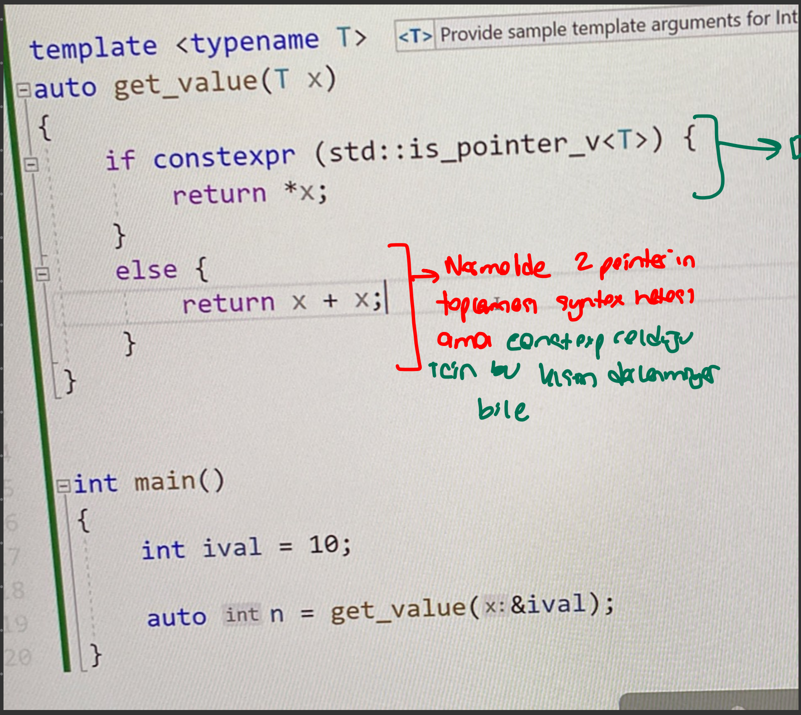


```cpp
template <int n>
class Myclass {
    static_assert(isprime(n), "n asal sayi olmali");
};


int main()
{
    Myclass<123> m1;
}
```

sadece prime num açılımı çalışır.

+ Constexpr if:

– run time if'den bağımsız. if işlemini compile timeda yapan bir araç.



```cpp
template <typename T>
auto get_value(T x)
{
    if constexpr (std::is_pointer_v<T>) {
        return *x;
    }
    else {
        return x + x;
    }
}


int main()
{
    int ival = 10;

    auto n = get_value(&ival);
}
```

Derlenen kısım.

Normalde 2 pointer'ın toplanması syntax hatası ama constexp olduğu için bu kısım derlenmez bile