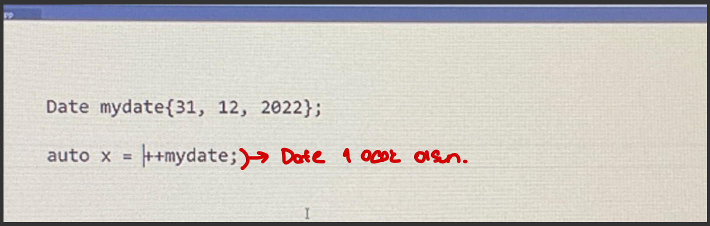
* Tekrar:

- Sınıf nesneleri, operatörün operandı olduğunda (belirli operatörler için) bu operatör işlemini fonksiyon çağrısına dönüştürür.
- Global ya da non-static member function olabilirler
- Amaç kolaylık sağlamak.



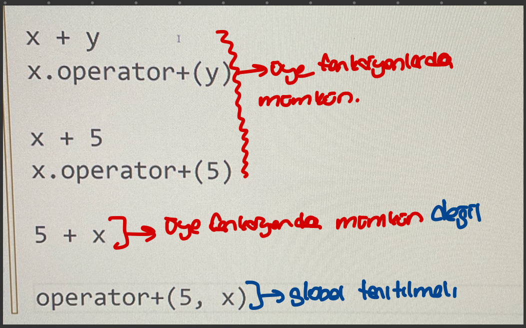
```
Date mydate{31, 12, 2022};

auto x = ++mydate;
```
→ Date 1 ocak olsun.

→ **Genel kurallar:**
- Olmayan operatör overload'lanamaz.
- Bazı operatörlerin global overload'u yasak
- İsimlendirme operator ifadesi yanına token
- Static member func olamaz.
- Öncelik sırası değişmez.

* Neden Global Operator Functionlar Var?:

sınıf türünden

1. Üye operator fonksiyonları, binary operatorler için (x+y) / x.operator+(y) olur.

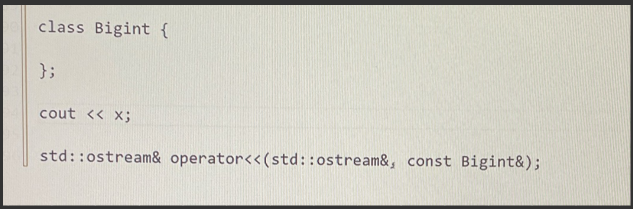


```
x + y
x.operator+(y)

x + 5
x.operator+(5)

5 + x

operator+(5, x)
```
→ Üye fonksiyonlarda mümkün.

→ Üye fonksiyonda mümkün değil

→ global tanıtılmalı

2. İki custom type class, birbirleriyle işleme sokulmak istendiğinde

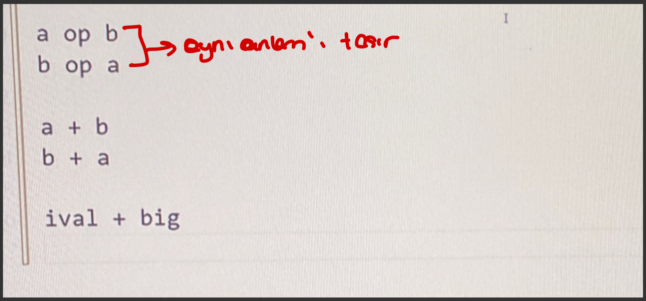


```
class Bigint {

};

cout << x;

std::ostream& operator<<(std::ostream&, const Bigint&);
```

→ Ne zaman global, ne zaman sınıf için tanıtalım, Genel Tavsiye:

• Hem global, hem class içi aynı operatörler için tanıtma → bu ambigity

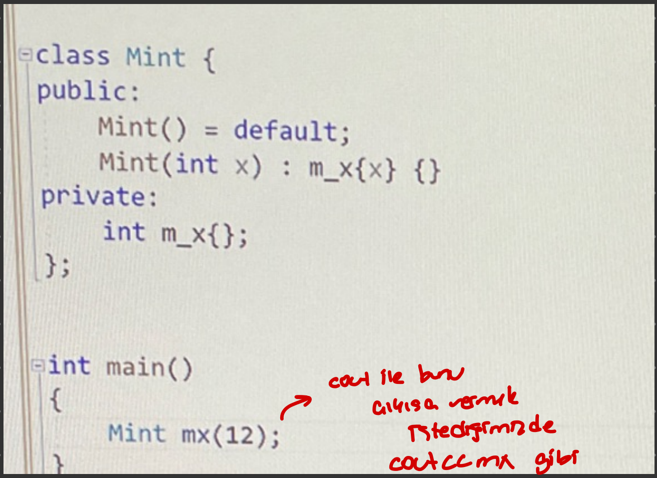
1. Simetrik operatörleri global tanıt



```
a op b
b op a

a + b
b + a

ival + big
```

} → aynı anlamı taşır

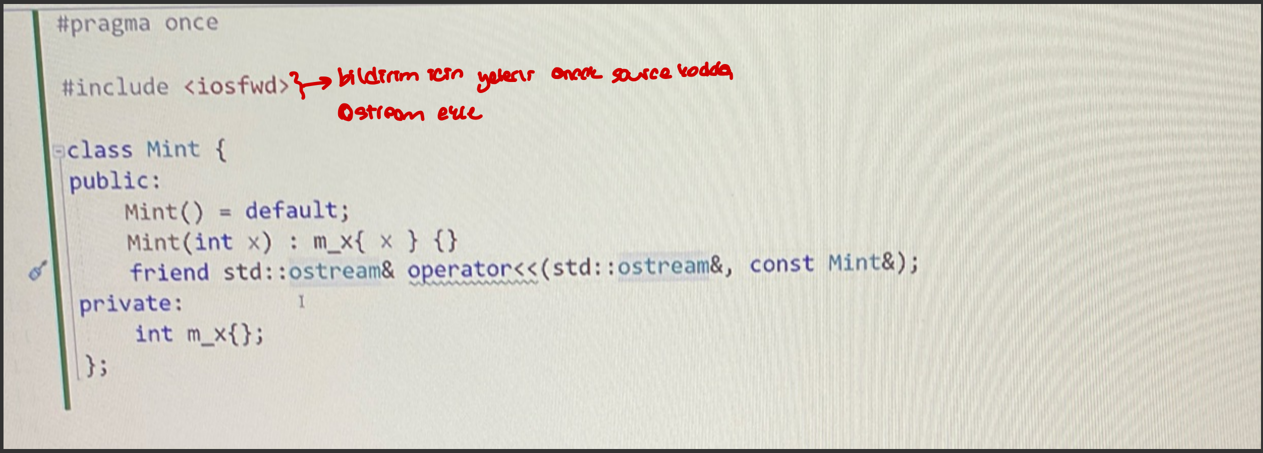
2. Custom classları, primative tip gibi kullanabileceğimiz senaryolar için



```
class Mint {
public:
    Mint() = default;
    Mint(int x) : m_x{x} {}
private:
    int m_x{};
};


int main()
{
    Mint mx(12);
}
```

↗ cout ile bunu çıkışa vermek istediğimizde cout<<mx gibi

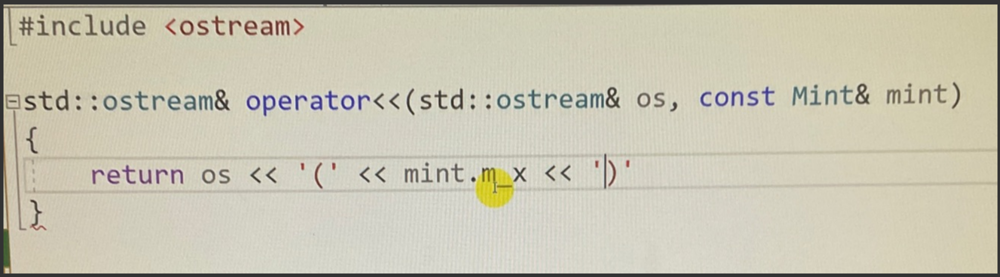
→ In header File:



```
#pragma once

#include <iosfwd>

class Mint {
public:
    Mint() = default;
    Mint(int x) : m_x{ x } {}
    friend std::ostream& operator<<(std::ostream&, const Mint&);
private:
    int m_x{};
};
```

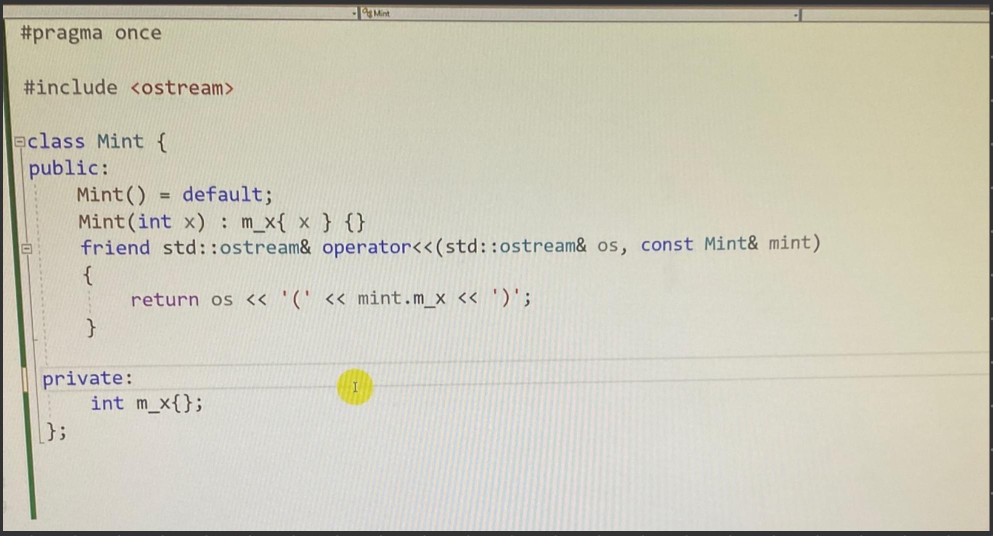
} → bildirim için yeterli ancak source kodda Ostream ekle

→ In source(cpp) file:



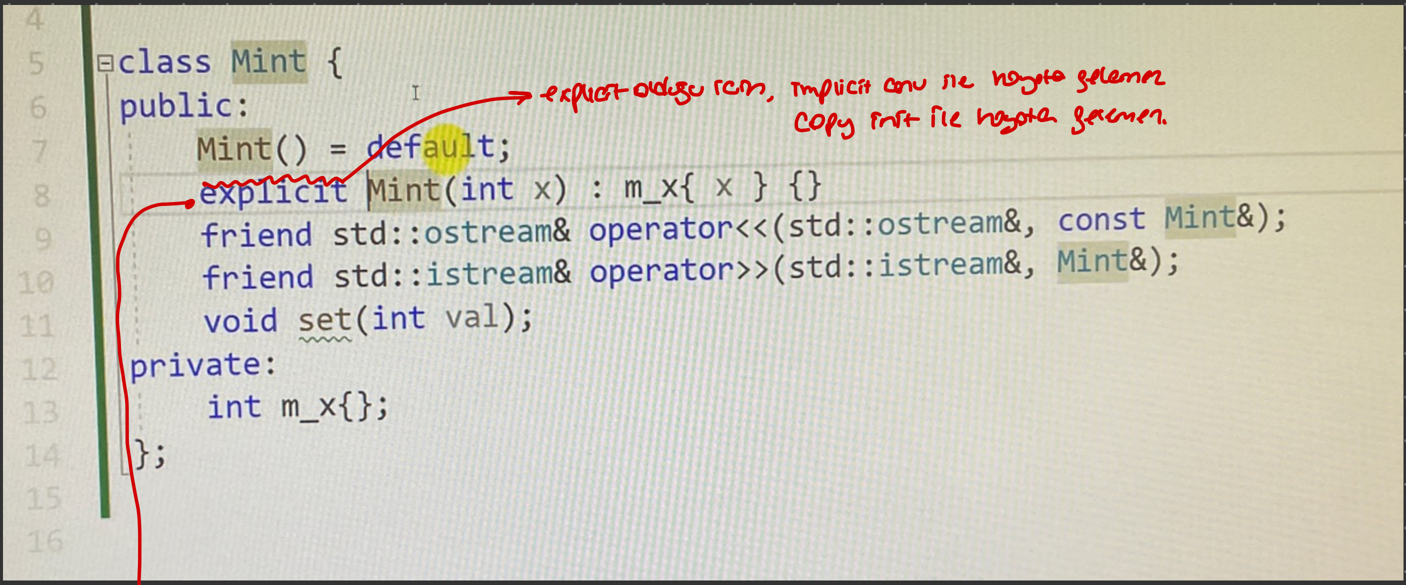
```
#include <ostream>

std::ostream& operator<<(std::ostream& os, const Mint& mint)
{
    return os << '(' << mint.m_x << ')'
}
```

→ Inline olarak tanıtılmış hali ise:



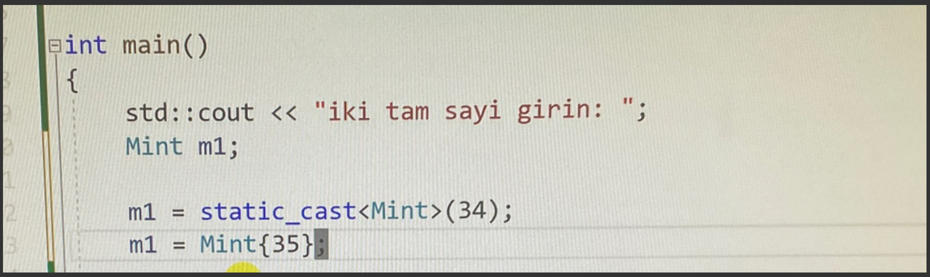
```cpp
#pragma once

#include <ostream>

class Mint {
public:
    Mint() = default;
    Mint(int x) : m_x{ x } {}
    friend std::ostream& operator<<(std::ostream& os, const Mint& mint)
    {
        return os << '(' << mint.m_x << ')';
    }

private:
    int m_x{};
};
```

→ Exerter'a ek Inserter:



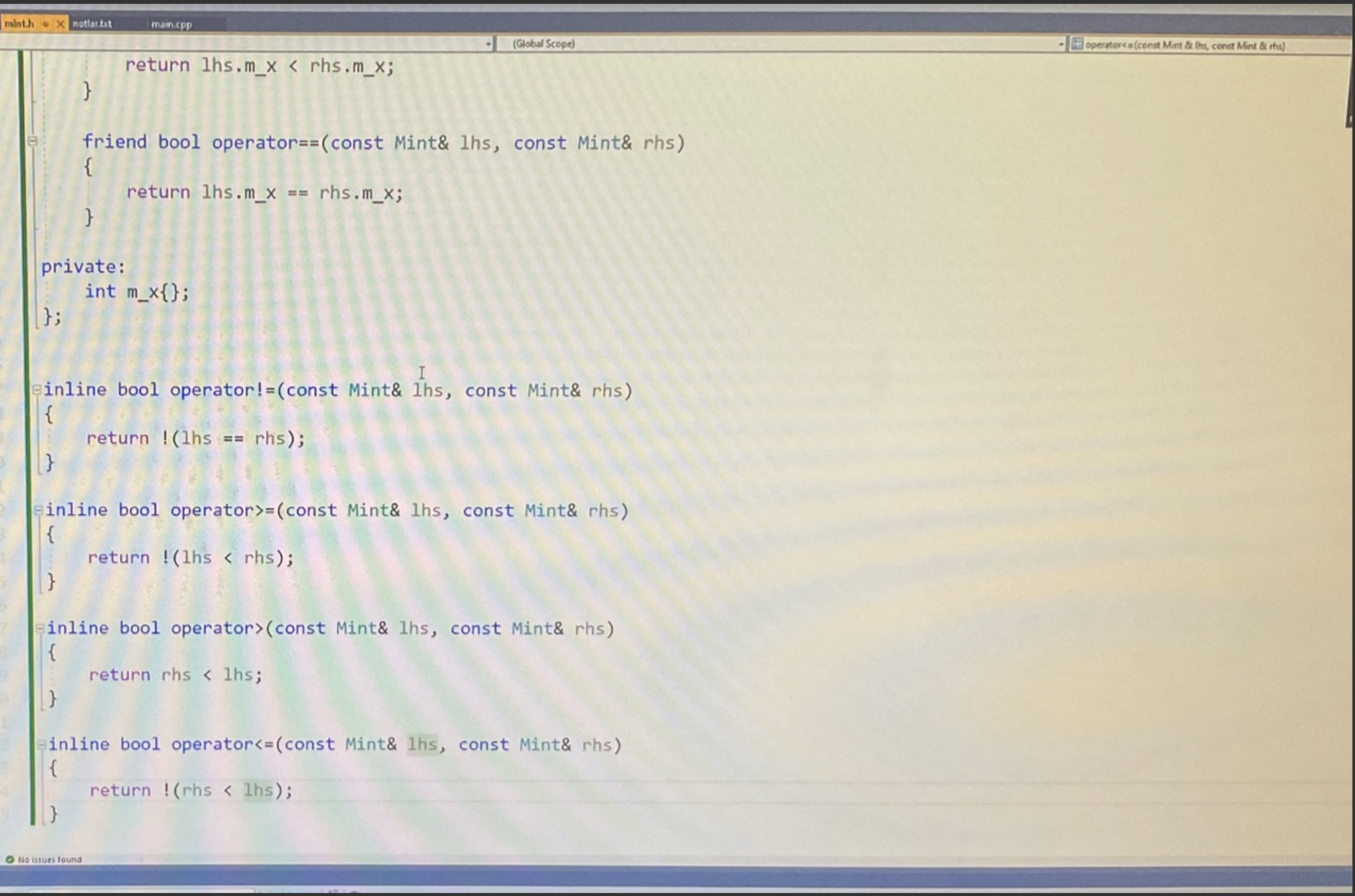
```cpp
class Mint {
public:
    Mint() = default;
    explicit Mint(int x) : m_x{ x } {}
    friend std::ostream& operator<<(std::ostream&, const Mint&);
    friend std::istream& operator>>(std::istream&, Mint&);
    void set(int val);
private:
    int m_x{};
};
```

→ explicit olduğu için, implicit conv ile hayata gelemez
copy init ile hayata gelemez.



```cpp
int main()
{
    std::cout << "iki tam sayi girin: ";
    Mint m1;

    m1 = static_cast<Mint>(34);
    m1 = Mint{35};
```

*Cpp 20 önceki diğer overloadlar → Karşılaştırma operatörleri için.



```cpp
        return lhs.m_x < rhs.m_x;
    }

    friend bool operator==(const Mint& lhs, const Mint& rhs)
    {
        return lhs.m_x == rhs.m_x;
    }

private:
    int m_x{};
};


inline bool operator!=(const Mint& lhs, const Mint& rhs)
{
    return !(lhs == rhs);
}

inline bool operator>=(const Mint& lhs, const Mint& rhs)
{
    return !(lhs < rhs);
}

inline bool operator>(const Mint& lhs, const Mint& rhs)
{
    return rhs < lhs;
}

inline bool operator<=(const Mint& lhs, const Mint& rhs)
{
    return !(rhs < lhs);
}
```

→ 20'de bizim yerimize yazabiliyor.

Devamını izle ve yaz.