

* Tekrar:

- Sınıf nesneleri, operatörün operandı olduğunda (bizzir operatörler için) bu operatör ismini **fonksiyon çağırısına** dönüştürür.
- Global ya da non-static member function olabilir.
- Amaç kolaylık sağlamaktır.

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

```
auto x = ++mydate; } → Date 1 önce alın.
```

→ Genel Kurallar:

- Olmayan operatör overloadlanamaz.
- Bazı operatörlerin global overload'u yasa.
- İşlenendirme operatör ibaresi yanına token.
- Static member func. elemanı.
- Öncelik sırası değişmez.

* Neden Global Operatör Functionlar Var ?

1. Üye operatör fonksiyonları, binary operatörler için $(x+y)$ / $x.operator+(y)$ olur. ↖ sınıf. türünden

```
x + y  
x.operator+(y) → Üye fonksiyonlarda mümkün.  
  
x + 5  
x.operator+(5)  
  
5 + x → Üye fonksiyonunda mümkün değil  
operator+(5, x) → global tanımlı
```

2. Her custom type class, birbirleriyle işlem yapılması istendiğinde

```
class Bigint {  
  
};  
  
cout << x;  
  
std::ostream& operator<<(std::ostream&, const Bigint&);
```

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

- Hem global, hem class içi aynı operatörler tanıtmak → bu ambiguity

1. Simetrik operatörleri global tanıta

```
a op b  
b op a } aynı anlamı: taşır
```



```
a + b  
b + a
```



```
ival + big
```

2. Custom classları, primitive tip gibi kullanabiliyoruz. Senaryo için

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

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

cout ile buru
çıkırsa yanlış
istediğimizde
cout << mx gibi

→ In Header File:

```
#pragma once
```

```
#include <iosfwd> } bildirim için yeterli önce source kodda  
ostream ile
```

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

→ In source(.cpp) file:

```
#include <ostream>
```

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


→ Inline olarak tanımlanmış hali ise:

```
#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{};
};
```

→ Explicit ve Inserters:

```
4
5 class Mint {
6 public:
7     Mint() = default;
8     explicit Mint(int x) : m_x{ x } {}
9     friend std::ostream& operator<<(std::ostream&, const Mint&);
10    friend std::istream& operator>>(std::istream&, Mint&);
11    void set(int val);
12 private:
13     int m_x{};
14 };
15
16
```

→ explicit olduğu için, implicit olarak ile yapıta geçilemez
Copy constructor ile yapıta geçilemez.

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

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

*C++ 70 Dersin 42. Dersin Overloadlar → Karşılaştırma operatörleri için.

```
minth.cpp | notlar.txt | main.cpp | (Global Scope) | operator<=(const Mint & lhs, const Mint & rhs)

    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);
}

No issues found
```

→ 25. de birim testimize yazacağız.

Dejenerant ile ve yz.