→ Tekrar:

```
    void func();
    static void sfunc();
};


int main()
{
    Nec mynec;

    mynec.func();
    mynec.sfunc();
}
```

→ Func, mynec nesnesi için çağırılır. mynec'in adresi bu fonksiyonun gizli parametresi this* 'a atanır.

→ Burada gizli this* yok

* Named Constructor Idiom'dan Devam:

```
class Myclass {
    Myclass();
public:
    static Myclass create_object();
};

int main()
{
    auto m = Myclass::create_object();
}
```

→ constructor private

→ bunun ile constructor çağırabiliriz.

→ Bunu neden yaparız?

- Farklı constructlar yazıp, overload etmek istiyoruz fakat signatureları aynı → bu overload'a engel bir durum.

```
class Complex {
private:
    Complex(double distance, double angle); //polar
    Complex(double r, double i, int);  //cartesian
public:
    static Complex create_polar(double d, double a)
    {
        return Complex(d, a);
    }
    static Complex create_cartesian(double r, double i)
    {
        return Complex(r, i, 0);
    }
};
```

Aynı signature

→ dummy int eklendi normalde client çağırmıyor

- Benzer şekilde otomatik yada statik ömürlü nesne oluşturmak yerine dinamik ömürlü bir nesne oluşturmak isteriz. → heap'te tutulmasını

```
class DynamicOnly {
private:
    DynamicOnly();
public:
    static DynamicOnly* create_object()
    {
        return new DynamicOnly{};
    }

};
```

* Friend Declerations:

- Sınıfın private ve protected kısımları, client kullanımına açık değil
- Friend bildirimi bunu legal kılmaktadır.
- Kimlere bu izin verilir
  - a) Free Functions (global gibi) → en sık
  - b) Başka bir sınıfın tüm fonksiyonuna → en nadir.
  - c) Başka bir sınıfa

FreeFunction

```
class Nec {
public:
    friend void gf();
private:
    int mx, my;
    void foo();
};


void gf()
{
    Nec nec;

    auto i = nec.mx;
    nec.foo();
}
```

→ Friend verilmeseydi

→ Bu fonksiyon içinde private elemanlara erişim syntax hatasıydı.

- Başka bir sınıfın tüm fonksiyonuna: → Complete type olmalı.

```
class A;
class Nec {
public:
    friend void A::foo(int); //gecersiz
private:
    int mx, my;
};
```

→ Incomplete type.

Legal Hali

```
class A {
public:
    void foo(int);
};

class Nec {
public:
    friend void A::foo(int); //gecersiz
private:
    int mx, my;
};


void A::foo(int)
{
    Nec mynec;

    mynec.mx = 10;
}
```

• Sınıfa Friendlik Vermek:

```
class Myclass{
public:
    friend class Nec;
private:
    int mx, my;
};


class Nec {
public:
    void foo()
    {
        Myclass m;

        m.mx = 4;
    }
private:
    void func()
    {
        Myclass m;

        m.mx = 4;
    }
};
```

→ Friendlik interchangable değil

A → friend → B ≠ B → friend → A

* Operator Overloading:

- Bir sınıf nesnesi, bir operatörün operandı olduğunda, bu ifadeyi bir fonksiyon çağrısına dönüştürür.
- Sözdizimi ve anlamı kolaylaştırır → çağrışıma dayalı
  - ↳ add() yerine '+' operatör gibi

→ Genel Kuralları: 1. Olmayan operatör overload edilemez. Kısıtlı var olan operatörler.

2. Her operatör overload edilemez → sizeof
   → . (member selection)
   → ternary operator
   → :: (scope res.)
   → typeid operator
   → .* operatörü (user elemanlarının adresleri, member pointer)

3. Operatör fonksiyonları özel bir şekilde isimlendirilir. operator+
   operator! ... gibi

4. Operatör fonksiyonlar → global operator function olabilir. Static member function olamaz!!
   → member operator function

5. Primative türler için operator fonksiyonu bildirilemez. En az bir operand sınıf türü ya da enum olmalı.

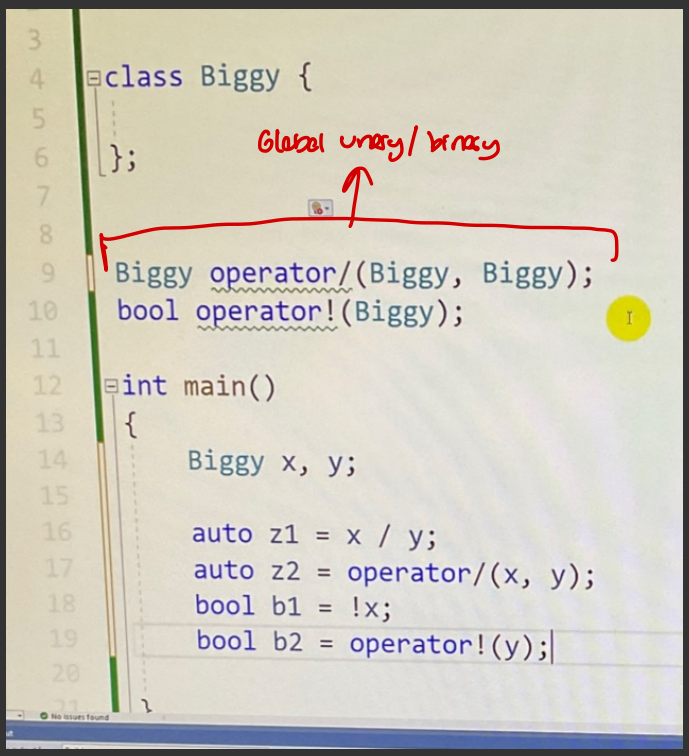
6. Tüm operatör fonksiyonları (operator notasyonu kullanmadan) isimleriyle çağrılabilir.

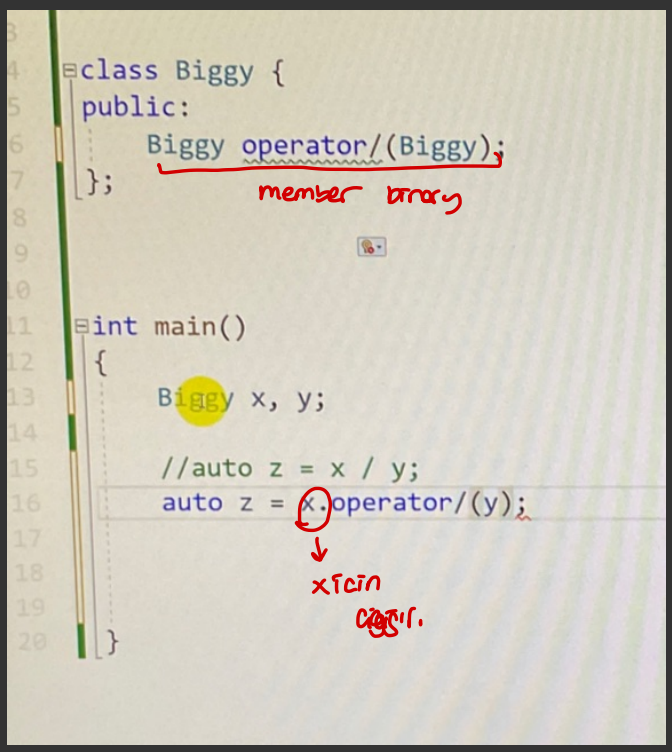
```
a + b     a.operator+(b)
x > y     operator>(x,y)
```

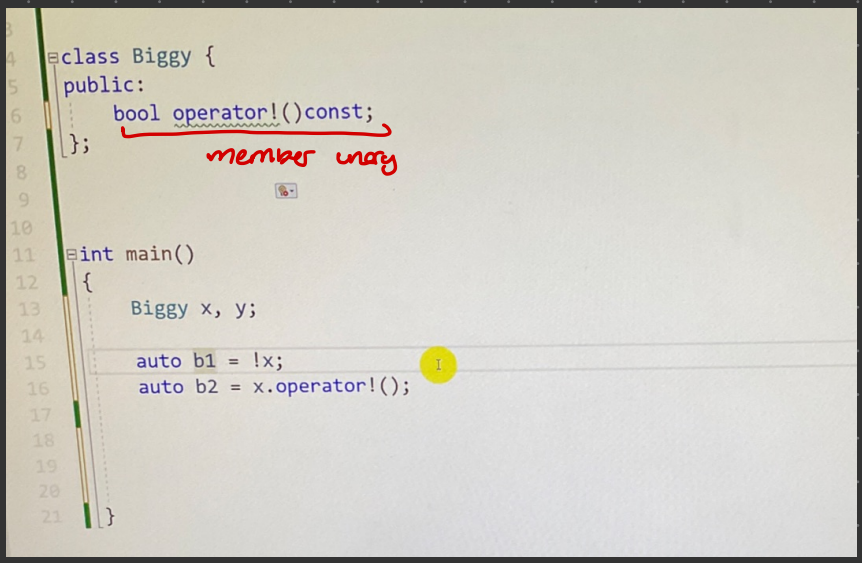
7. Bazı operatörler için yalnızca member operator function olabilir. Global operator fonksiyonu tanımlanamaz.

> * Arity of Operator: Operator kaç adet operand aldığı (unary ve binary)

8. Bu overload, operatörün arity'si ile aynı sayıda operand almalı. Yoksa syntax error







```
+    +x           sign operator
     x + y        addition operator

-----------------------------------------------

-    -x           sign operator
     x - y        subtraction operator

-----------------------------------------------

*ptr              dereferencing op.
x  * y            multiplication

-----------------------------------------------
&

     &x           address of
     a & b        bitwise and
```

9. Function operator call hariç varsayılan argüman ile bildirim yapılamaz.

```cpp
class X {
public:
    int operator()(int x = 10);
};

int main()
{
    X xa;

    xa();
    xa.operator()(10);
    xa(10);
}
```

Short Circuit Behavior:

```cpp
int main()
{
    int x = 0;
    int y = 10;

    auto b = x && y++;

    std::cout << "y = " << y << "\n";
}
```

- x → Bu operand önce işlenir.
- y++ → Kısa devreden dolayı bu işlenmez.
- && → Logic && ilk operand 0 olduğu için Short Circuit yapılacak

```
Myclass operator+(const Myclass&, const Myclass&);
Myclass operator*(const Myclass&, const Myclass&);
bool operator>(const Myclass&, const Myclass&);


int main()
{
    Myclass m1, m2, m3, m4;

    auto b = m1 + m2 * m3 > m4;
    auto b2 = operator>(operator+(m1, operator*(m2, m3)), m4);
}
```

```
class Ostream {
public:
    Ostream& operator<<(int);
    Ostream& operator<<(double);
    Ostream& operator<<(Ostream&(*)(Ostream&));
};

Ostream& operator<<(Ostream&, const char*);
Ostream& operator<<(Ostream&, char);

int main()
{
    using namespace std;

    int x = 1;
    double dval = 3.4;
    char c = 'A';

    cout << x << " " << dval << " " << c << endl;

    operator<<(operator<<(operator<<(cout.operator<<(x), " ").operator<<(dval), " "), c).operator<<(endl);
}
```

Bu member → cout.operator(x)

Bunu yapan global olan → operator<<(cout, c)

Bu aslında bir function pointer

Mutlaka

```
class Date {
public:
    bool operator<(const Date&)const;
};

// a < b
// a.operator<(b)
```

sağ operand const

sol operand const