

sınıfın özel üye fonksiyonları (special member functions)

default etme: özel yapan derleyici tarafından default edebiliyor (derleyici kodlarını bizim yerimize yazabilir)

user declared : durumdan vazife çıkarılan derleyici veya ben diyorum derleyici sen benim için tanımla

Eğer derleyici kurallara göre default ederken bir sentax hatası oluşursa const. nesne init ederse vs derleyici üye fonksiyonu delete eder.

default ctor:

Default constructor is the constructor which doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

*

copy constructor : sınıfın copy ctor. illa olacak (fakat delete edilebilir)

```
class Myclass{  
public:  
    Myclass(const Myclass& other) : ax(other.ax), bx(other.bx), cx(other.cx) // shallow copy  
    {  
    }  
  
private:  
    A ax;  
    B bx;  
    C cx;  
};
```

// Copy constructor
Geeks Obj1 (Obj);
or
Geeks Obj1 = Obj;

Derleyici belirli koşullar oluştuğunda special member functions yazar.

örneğin default const.'ı her zaman yazmaz

```
class Myclass{  
  
public:  
    // Myclass() = default;  
    Myclass(int); // sadece bu olursa default ctor. yazmaz  
}
```

copy constructor'ı derleyiciye bırakmayıp kendim yazacağım.
Böylece shallow copy yerine deep copy yapacağım.

copy assignment: (kopyalayan atama operatör fonksiyonu)

C++'da bir sınıf nesnesine atama operatöründen aynı türden bir nesne atandığında bu işi copy assignment fonksiyonu yapar. Bu bir constructor değildir.
Non-static, public, inline üye fonksiyonu. Geri dönüş değeri türü Myclass&

x = y;

```
class Myclass{
public:
    Myclass& operator = (const Myclass& other) // copy assignment operator overloading
    ile yapılıyor.
    {
        ax = other.ax;
        bx = other.bx;
        cx = other.cx;
        return ??? → return *this; ???
    }
private:
    A ax;
    B bx;
    C cx;
};
```

* a = b; // a.operator = (b); ile aynı *

Copy Constructor ile copy assignment farkı nedir?

to copy the data of object at the time of initialization

to copy the data of object after initialization

Shallow copy: An object is created by simply copying the data of all variables of the original object. If some variables are dynamically allocated memory from heap sector, then the copied object variable will also reference the same memory location. This will create ambiguity, run time errors and dangling pointers.

class box {

private:

```
int length;  
int breadth;  
int height;
```

public:

```
void set-dimensions(int length1, int breadth1, int height1)
```

{

```
length = length1;  
breadth = breadth1;  
height = height1;
```

}

```
void show-data()
```

{

```
cout << "length: " << length  
<< "breadth: " << breadth  
<< "height: " << height << "\n";
```

}

}

```
int main()
```

{

```
box B1, B3;
```

```
B1.set-dimensions(14,12,16);
```

```
B1.show-data();
```

```
box B2 = B1;
```

```
B2.show-data();
```

```
B3 = B1;
```

```
return 0;
```

Neden kopyalayan atama fonksiyonu kendisine atama yapılmış nesneyi döndürüyor.

a = b;

C++ dilinde atama operatörleri ile oluşturulan ifadelerin değer kategorisi L value expr. dir.

x = y = z = t; // t'nin değeri hepsine atanır. Bunu sağlayan sağdan sola öncelikli
x = (y = (z = t));

copy constructor: copying the data of object at the time of initialization

copy assignment: copying the data of object after initialization

Deep Copy: Is performed by implementing our own copy constructor.

class box{

private:

```
int length;  
int* breadth;  
int height;
```

public:

box()

```
{ breadth = new int;
```

}

void set_dimension (int len, int brea, int heig)

{

```
length = len;  
*breadth = brea;  
height = heig;
```

}

void show_data()

```
{ cout << "length: " << length  
    << "breadth: " << breadth  
    << "height: " << height << "\n";
```

}

box (box & sample) // parameterized constructors for deep copy

```
length = sample.length;
```

breadth = new int;

*breadth = *(sample, breadth);

height = sample.height;

}

```
~box()
{
    delete breadth;
}
```

```
int main()
{
    box first;
    first.set_dimension(12, 14, 16);
    first.show_data();
    box second = first;
    second.show_data();
    return 0;
}
```

int* ptr = nullptr; // hiç bir yerin göstermesi
ptr = new int; // bir yer ayırdık, oraya gösterir.
*ptr = 10; // gösterilen yere değer atadık.

Neden kopyalayan atama fonksiyonu kendisine atama yapılmış nesneyi döndürüyor ?

a = b;

C++ dilinde atama operatörleri ile oluşturulan ifadelerin değer kategorisi L value expr. dır.

x = y = z = t; // t'nin değeri hepsine atanır. Bunu sağlayan sağdan sola öncelikli
x = (y = (z = t));

Bir nesneye kendisinin atanması (self assignment):

x = x; // hata değil

*p1 = *p2;

r1 = r2; // referans

*ptr = r; // referans

Move Semantics: (Taşıma Semantiği)

```
int main()
{
    std::string str(20000, 'A');

    std::string s{ str }; // yüksek maliyetli, bundan kurtulmak yerine pointer kopyalama ile
    // halledebilirim.
}
```

Myclass x = baska_nesne;
// baska_nesne'nin hayatı bitecekse öyle bir kod devreye girsın ki
// hayata gelen x nesnesi o kaynağı çalsın, destructor baska_nesnenin kaynağını geri
vermesin.

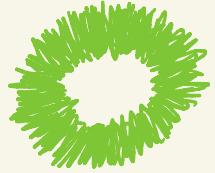
basika_nesne'nin hayatının bitip bitmemesine göre ya kaynağı çalacak veya kaynağı
kopyalayacak

```
void func(const int&); // lvalue reference
void func(int &&); // rvalue reference

int main()
{
    int x = 5;
    func(x); // üstteki
    func(5); // ikisi viable, alttaki çağrıılır
}
```



void func(const Myclass&); // lvalue reference, sınıf nesnesinin hayatı devam edecektir
void func(Myclass &&); // rvalue reference, sınıf nesnesinin başka kod tarafından kullanılma
ıhtimalı yok // kaynağı çalışınabilir !!!



```
int main()
{
    Myclass m1;

    Myclass m2 = m1; // copy constructor'dır, move constructor değil
```

// öyle bir araç olmalı ki, tür dönüştürme değil value category'i değiştirecek
Myclass m2 = static_cast<Myclass&&>(m1); // rvalue expr. // fakat zahmetli

Myclass m2 = std::move(m1); // sağ taraf rvalue expr. olur (okunuşu: stud move)

```
}
```

move doesn't move // taşıma ile alakası yok !!!

std::move(m1); // kaynak çalmak olmuyor, sadece rvalue yapmış oluyoruz.

Myclass m2 = std::move(m1); // kaynağı çalan m2 nesnesi

```
void foo(Myclass && r);

int main()
{
    Myclass x;

    foo(std::move(x)); // bu fonksiyonun çalınmasıyla ilgisi yok // atama ya da kopyalama
    // yapılmadı

    foo( r value expression bir myclass nesnesi);
}
```

// kaynağı nasıl çalarım ?

```
void foo(Myclass && r)
{
    Myclass m = r; // yine kopyalar, çalmaz
    Myclass m = std::move(r); // şimdi kaynak çalınmış olur
}
```



// öyle fonksiyon yazalım ki gönderilen ifade lvalue ise kaynağı kopyalayalım rvalue ise çalalım
// cevabı iki fonksiyon yazmak

```
class Myclass{

}

void func(const Myclass& r) // lvalue için çağrılır
{
    Myclass m = r; // kaynağı kopyaladım
}

void func(Myclass&& r) // rvalue için çağırılır
{
    Myclass m = std::move(r); // kaynağı çaldım
}
```

derleyicinin yazdığı move constructor

```
Myclass(Myclass &&other) : ax(std::move(other.ax)), bx(std::move(other.bx)),  
cx(std::move(other.cx))  
{  
  
}  
  
Myclass& operator = (Myclass &&other) // move assignment  
{  
    ax = std::move(other.ax);  
    bx = std::move(other.bx);  
    cx = std::move(other.cx);  
}
```



derleyici hangi durumlarda sınıfın hangi özel üye fonksiyonu yazıyor ?

bir sınıfın bir özel üye fonksiyonu aşağıdaki 3 durumdan birinde olabilir

1. not exist
2. user declared (programcı tarafından bildirilmiş)
3. implicitly declared (programcı bir bildirim yapmadı, derleyici yazıyor)

*-----

1. not exist

```
class Myclass{  
public:  
    Myclass(int); // not exist  
}
```

*-----

2. user declared (alttakiler gibi 3 tipte)

```
Myclass();  
Myclass() = default;  
Myclass() = delete;
```

*-----

3. implicitly declared

```
default  
deleted
```