Tekrar: · Shallow Copy ile kopyalanan pointerlar/referanslar aynı nesneye işaret eder.

```
class Myclass {

public:
    Myclass() = default;
    Myclass(int);
}
```

↳ biz default olmayan bir ctor yazdığımız anda compiler default ctor oluşturmaz. Fakat biz yine de istiyorsak

```
class Sentence {
public:
    Sentence(const char* p) : m_len{ std::strlen(p) }, m_p{ static_cast<char*>(malloc(m_len + 1)) }
    {
        if (!m_p) {
            std::cerr << "bellek yetersiz\n";
            std::exit(EXIT_FAILURE);
        }
        std::cout << this << " adresindeki nesne icin " << (void*)m_p << " adresindeki bellek alani allocate edildi\n";
        std::strcpy(m_p, p);
    }

    Sentence(const Sentence& other) : m_len(other.m_len), m_p{ static_cast<char*>(malloc(m_len + 1)) }
    {
        if (!m_p) {
            std::cerr << "bellek yetersiz\n";
            std::exit(EXIT_FAILURE);
        }

        strcpy(m_p, other.m_p);
    }

    void print()const
    {
        std::cout << (void*)m_p << " adresindeki yazi yazdiriliyor\n";
        std::cout << "[" << m_p << "]\n";
    }

    ~Sentence()
    {
        std::cout << this << " adresindeki nesne icin " << (void*)m_p << " adresindeki bellek alani geri veriliyor\n";
        std::free(m_p);
    }
private:
    char* m_p;
    std::size_t m_len;
};
```

+1 çünkü null character için de yer ayırdık!

tekrardan pointer için yer allocate edip

allocate edilen alana, kopyalama yaptığımız pointer'ın kopyalanması!

Bizim yazmamız gereken copy constructor tam olarak böyle olmalı! → Bu Deep Copy Yapıyor!

* Yeni nesneleri kopyalamak için, compiler copy ctor'u yerine deep copy yapan kendi copy ctor'umuz. Değer değil, yeni bir memory alanı oluşturulur.
  - pointer
  - ref.

* Copy Assignment: Nesneler birbirine eşitlendiğinde

• Bu bir ctor değil. Return değeri var = *this → &myclass (my class ref)

```
class Myclass {
public:
    Myclass& operator=(const Myclass&);
}

a = b;
```

a için çağırdık

b argümanıyla.

a.operator=(b)

• Bu da shallow copy yapıyor.

```
int main()
{
    Sentence sen_1{ "bugun hava cok guzel!" };

    {
        Sentence sen_2{ "umarim yarin da guzel olur" };
        sen_2 = sen_1;
    }
    (void)getchar();
    sen_1.print();

}
```

→ Bu durumda compiler'ın copy ctor'ı

• C++ dilinde atama operatörleri ile oluşturulan ifadelerin değer kategorisi L value expression

- Bir sınıfın copy ctor'ı, bir sınıf nesnesi, hayatına değerini aynı sınıf türünden olan bir nesne ile başladığında
- Bir sınıf nesnesine atama operatörüyle, aynı türden bir sınıf nesnesi atandığında, bunu gerçekleştiren Assignment.

→ mini özet:

```cpp
#define _CRT_SECURE_NO_WARNINGS

#include <iostream>
#include <cstring>
#include <cstdlib>

class Myclass {
public:
    Myclass();
    ~Myclass();
    Myclass(const Myclass&);
    Myclass& operator=(const Myclass&);
};
```

- `Myclass();` → default ctor
- `~Myclass();` → destructor → release resource yapan yazılmalı!
- `Myclass(const Myclass&);` → copy ctor → deep copy yapanını yaz!
- `Myclass& operator=(const Myclass&);` → copy assignment → rel resource and deep copy yazılmalı!
- (copy ctor ve copy assignment) → shallow copy by default

* Eğer kopyalayan atama operatör fonksiyonunu kendimiz yazıyorsak, (ve alternatif yapı yoksa copy-swap yoksa)
↳ 1. self assign'a karşı kontrol et

* Big 3 / Rule of 3:
- destructor
- copy ctor
- copy assignment

* Big 5: → yeni olarak
- \+ move assignment
- \+ move constructor.

* Move Semantiği:
- Bir nesnenin yok edilmesine doğru, kaynaklarını vermek yerine, onun değeriyle hayata gelecek nesnenin, kaynak kopyalamak yerine, o kaynağı vermesi.
- R value referanslar önce yoktu.

```cpp
void func(const int&);   // L value
void func(int&&);        // R value

int main()
{
}
```

→ aynı şekilde →

```cpp
class Myclass {

};

void func(const Myclass&);
void func(Myclass &&);

int main()
{
}
```

↳ r value ise ephemeral bir öğeyi çukurda → move semantiğine uygun

```cpp
class Myclass {
public:
    Myclass(const Myclass &);
    Myclass(Myclass &&);
}
```

- `Myclass(const Myclass &);` → const class ref = L value = Copy Constructor
- `Myclass(Myclass &&);` → R value = move constructor.

```cpp
class Myclass {
public:
    Myclass(const Myclass &);
    Myclass(Myclass &&);
    Myclass& operator=(const Myclass&);
    Myclass& operator=(Myclass&&); //move assignment
}
```

- `Myclass& operator=(const Myclass&);` → Copy assignment
- `Myclass& operator=(Myclass&&);` → move assignment

```cpp
class Myclass {

};

int main()
{
    Myclass m1;

    Myclass m2 = m1;
```

`m1` → L value olduğu için copy ctor.

```cpp
{
    Myclass m1;

    Myclass m2 = static_cast<Myclass&&>(m1);
```

Bu şekilde castlersek m1'i R value yaparak move const. çağırır.

Ama daha kolayı

```cpp
{
    Myclass m1;

    Myclass m2 = std::move(m1)
```

`std::move` → Böyle de R value olur.

Bu move yapmaz!!, R value yapar!!

Hatırlatma:

```cpp
class Myclass {

public:
    Myclass(const Myclass &other) : ax(other.ax), bx(other.bx), cx(other.cx)
    {

    }

    Myclass& operator=(const Myclas& other)
    {
        ax = other.ax;
        bx = other.bx;
        cxs = other.cx;

        return *this
    }


private:
    A ax;
    B bx;
    C cx;
}
```

- Copy constructor block → compiler's copy constructor.
- `operator=` block → compilers copy assignment

* Derleyicinin oluşturduğu Move Semantikleri:

```
//move ctor
Myclass(Myclass &other) : ax(std::move(other.ax)), bx(std:>:move(other.bx)), cx(std::move(other.cx))
{

}

Myclass& operator=(Myclass &&other)
{
    ax = std::move(other.ax);
    bx = std::move(other.bx);
    cx = std::move(other.cx);

    return *this;
```

→ Compiler's copy assignment

*

Derleyici hangi durumlarda sınıfın hangi özel üye fonksiyonlarını

bir sınıfın bir özel üye fonksiyonu aşağıdaki 3 durumdan birinde olabilir

1. yok

2. user declared (programcı tarafından bildirilmiş)

```
    Myclass();
    Myclass() = default;  → yazan kişi compiler'a yapsın der.
    Myclass() = delete;   → buna yapılan çağrı hata
```

⇒ Default = kendi oluşturması

3. implicitly declared → derleyicinin yazdığı
    defaulted
    deleted