

Tekrar:

• operator new / delete global olarak ya da bir class'a özgü overload edilebilir. → Cunku replaceable yani bilmem gerekmeyen yerler.



Sadece size_t ve void* olan operator new i için
yapılmış XXX Syntax hatalı veya.

• Operatör new fonksiyonları, exception throw etmez! Basitçe örneksel olarak, global bir function pointer eger nullptr ise exception throw eder.

```
typedef void (*new_handler)(void);

new_handler get_new_handler(void);
new_handler set_new_handler(new_handler);

void* operator new(std::size_t n)
{
    for (;;) {
        void* vp = std::malloc(n);
        if (vp) {
            return vp;
        }
        auto fp = get_new_handler();
        if (!fp) {
            throw std::bad_alloc{};
        }
        fp();
    }
}
```

* Set_new_handler:

```
std::vector<void*> pvec;

void func()
{
    for (int i{}; i < 1000'000; ++i) {
        std::cout << i << "\n";
        pvec.push_back(operator new(1024 * 1024 * 10));
    }
}

void myalloc()
{
    static int cnt{};
    std::cout << "myalloc islevine yapılan " << ++cnt << ". çağrı\n";
    if (cnt == 10)
        std::set_new_handler(nullptr);
}

int main()
{
    std::set_new_handler(myalloc); → exception throw etmesini bz saglams olduk.
    try {
        func();
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
    }
}
```

Bu durumda set_new_handler ile kaydı yapılan fonksiyon neler yapabilir?

1. Bir takım işlemlerle dinamik bellek alanı elde edilmesine yönelik şartları sağlayabilir.
2. set_new_handler'a çağrı yapar
 - nullptr gönderir.
 - başka bir fonksiyonu kayda alır.
3. kayıt edilen fonksiyonun kendisi std::bad_alloc ya da bad_alloc sınıfından kalıtım yoluyla elde edilmiş bir sınıfından exception throw eder.

* Teksor: moved from state : Moved from state → nota valid dir ama degetirin genetir yolu

```
int main()
{
    using namespace std;

    ifstream ifs{ "textfile.txt" }; I
    if (!ifs) {
        cerr << "cannot open file\n";
    }

    std::string sline;
    vector<string> svec;

    // sline burada moved-from-state e sahip olabilir
    while (getline(ifs, sline)) {
        svec.push_back(std::move(sline));
    }
    ↗ Verim kazandırır. Sadece pointer
    ↗ Gerekli moved-from-state dır bir nesneye yarınca eklenmeye yarılır.
}
```

```
int main()
{
    std::string str(100'000, 'a'); I
    std::string&& r = std::move(str); ← beraber like r value
    std::string s; ← Move deant move argumens

    s = std::move(r); ← First move koyalı carına.
    ↗ Move deant move argumens
    ↗ Kaynak değişimi için move dır move argument sonası olursa.

    }
}
```

```
class Sentence {
public:
    Sentence(Sentence&& other) : mlen(other.mlen), mp{ other.mp } {
        other.mp = nullptr; ← smart kontrahentin bu şekilde move ediyor.
        other.mlen = 0; ← Kontrahentimiz tari birz evleniyor.
        /**
     }
private:
    size_t mlen;
    char* mp;
};
```

* Smart Pointers:

• Dinamik ömürli nesnelerin, nesnelerini kontrol etmek üzere kullanılabilecek pointer-like sınıflarıdır.

• Raw/Naked Pointers kullanımı, şu sorular yanıtnez → Bu yüzden dinamik ömürli nesnelerde kullanımını risklidir.

```
// pointer'in gösterdiği nesne dinamik ömürli bir nesne mi?
// pointer dinamik ömürli bir nesneyi gösteriyor(du) şu anda dangling pointer mı?
// bu pointer dinamik ömürli bir nesneyi gösteriyor olsun
    peki dizi mi?
    tekil bir nesne mi?

// delete p;
// delete [] p;
p->~Fighter();
```

```
// bu pointer dinamik ömürli bir nesneyi gösteriyor olsun
    onu göstermeye olan tek pointer bu mu?
```

sadece delete etmek yeterli mi? → Delete servisi başka işlem de yapabilir.

point edilen nesne adresinde artık o nesnenin bulunmaması

- Standart Olarak 2 adet smart pointer eklendi \rightarrow unique_ptr
 \rightarrow shared_ptr (ve ona gelmesi olması için weak_ptr)
- Unique ptr sınıfı \rightarrow exclusive ownership stratejisini implement eder.
 - kaynaktan tek sahibi olur
 - O kaynaktan yalnızca o sahibi varlığından emsası.
 - kaynaktan sahibi olan nesnenin (pointer) hedefi bitişinde kaynaktan释放 (free).

\Rightarrow Dinamik sınırlı nesne örneği
zorunlu değil / C API'ye de örneği.
- Shared ptr sınıfı \rightarrow shared ownership stratejisini implement eder.
 - bir kaynaktan birden fazla smart ptr esasına sahiptir
 - O kaynaktan çeşitli son pointer değişkenlerin hedefi bitişinde kaynaktan released (free).
 - Reference counting

* Eğer unique / want / shared ptr ve member functionları kullanılsatça `<memory>` kütüphanesi include edilmeli

* Unique_ptr:

```
template <typename T, typename D = default_delete<T>>
class unique_ptr {
public:
    ~unique_ptr()
    {
        // D{()}(p);
    }

private:
    T *p;
};

template<typename T>
struct default_delete {
    void operator()(T *p)
    {
        delete p;
    }
};
```

\Rightarrow unique pointer, asında bir raw pointer wrapper

```
using namespace std;

unique_ptr<int> uptr{ new int };

if (uptr) {
    std::cout << "dolu\n";
}
else {
    std::cout << "bos\n";
}

if (uptr == nullptr) {
    std::cout << "bos\n";
}
else {
    std::cout << "dolu\n";
}

if (uptr != nullptr) {
    std::cout << "dolu\n";
}
else {
    std::cout << "bos\n";
}
```

unique pointer bu şekilde
hedefle getirilir.

yani bu
hedef
demez

```

unique_ptr<Point> uptr(new Point{ 2, 34, 45 });
cout << *uptr << "\n";
uptr->set(4, 4, 4);
cout << *uptr << "\n";
}

```

→ scope sonunda, unique pointer destrutor çalışır.
→ Dolayısıyla Point sınıfının destrutoru çağırılır.

```

int main()
{
    using namespace std;
    I
    unique_ptr<Point> uptr(new Point{ 2, 34, 45 });

    uptr = new Point{ 12, 5, 7 };
}

```

Hayır! above code unique ptr nesnesine atama yapılış syntax error.

```

using namespace std;
I
unique_ptr<Point> uptr = new Point{ 1, 3, 6 };

```

→ Syntax hatalı çünkü copy init yapılmıyor
 - Unique ptr'inin T'ye parametresiz constructor'ı yoktur.
 - Ya da direkt initialization yapılış direct initialization

* Make Unique Function Template:

```

template <typename T, typename ...Args>
std::unique_ptr<T> MakeUnique(Args && ...args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

```

→ implementation'ı şov içinde, standard bibliyotek
kapsığında make_unique.

```

int main()
{
    using namespace std;

    I auto uptr = MakeUnique<Point>(2, 4, 6);

    cout << *uptr << "\n";
    uptr->set(1, 1, 1);
    cout << *uptr << "\n";
}

std::cout << "main devam ediyor\n";
}

```

→ make_unique ile new kullanmadan initialize
edilir.

- Unique_ptr nesnesi kopylanmaya, kopyeli olurken.

```

using namespace std;
auto upx = make_unique<Point>(3, 5, 8);

auto upy = upx;

```

- unique_ptr nesneleri moveledir

```
int main()
{
    using namespace std;

    auto upx = make_unique<Point>(3, 5, 8);

    cout << (upx ? "dolu" : "bos") << "\n"; → Dolu,
    auto upy = std::move(upx); //move ctor → Vakıflarını aktardı. Bu nesneler nullptr
    cout << (upy ? "dolu" : "bos") << "\n"; → Bos , Artık upx dolu
    (void)getchar();
```

I

```
int main()
{
    using namespace std;

    auto upx = make_unique<Point>(3, 5, 8);
    auto upy = make_unique<Point>(6, 6, 6);

    cout << *upx << "\n";
    cout << *upy << "\n";

    std::cout << "upx => " << (upx ? "dolu" : "bos") << "\n";
    std::cout << "upy => " << (upy ? "dolu" : "bos") << "\n";

    upx = std::move(upy); → önce kendine oturmuş yaptığımız nesnenin abstractası
                           sonra yeni boyutları oburlar, cogrılır.
    std::cout << "upx => " << (upx ? "dolu" : "bos") << "\n";
    std::cout << "upy => " << (upy ? "dolu" : "bos") << "\n";

    (void)getchar();
```

I

- Benzer şekilde = nullptr direkt de negatif sınırlayıp, destanızı cogrular.

```
int main()
{
    using namespace std;

    auto upx = make_unique<Point>(3, 5, 8);
    std::cout << "upx => " << (upx ? "dolu" : "bos") << "\n";
    I
    upx = nullptr;
    (void)getchar();
}
```

- Yerine dolma alternatif olarak:

```
//upx = nullptr;
//upx.reset();
upx.reset(nullptr);
```

→ reset fonksiyonu çağırılır.

```

int main()
{
    using namespace std;

    unique_ptr<Point> uptr;
    std::cout << "uptr => " << (uptr ? "dolu" : "bos") << "\n";
    uptr.reset(new Point{ 2, 5, 9 });
}

std::cout << "main devam ediyor\n"

```

Reset'e argument olarak new ile yaradı bir obje
görebiliriz. Kognak olmak onu tifte.

```

int main()
{
    using namespace std;

    auto p = new Point{ 2, 4, 6 };

    unique_ptr<Point> upx{ p };
    unique_ptr<Point> upy{ p };
}

```

→ Undefined Behavior !!!

İkisi de aynı kaynaktır. Kaynaktan sızınması durumunda
⇒ DANGLING POINTER OLUSACAK

+ Release Fonksiyon:

```

int main()
{
    using namespace std;

    auto upx = make_unique<Point>(1, 4, 7);
    std::cout << (upx ? "dolu" : "bos");
    cout << *upx << "\n";
    Point* ptr = upx.release(); // release dan sonra bos
    std::cout << (upx ? "dolu" : "bos");
    (void)getchar();
}

```

→ Mükemmel bir释放 ama kaynaktan destructor'u çağırmez!
→ release'in return değeri bir pointer. O pointer'i alıp
delete etmeli.

→ Eğer release'in return değeri tutulmasa, resource +
memory leak olur.

```

using namespace std;

auto upx = make_unique<Point>(1, 4, 7);
std::cout << (upx ? "dolu" : "bos");
cout << *upx << "\n";

unique_ptr<Point> upy{ upx.release() } // Cetvel std::move
std::cout << "upx ==> " << (upx ? "dolu" : "bos") << "\n";
std::cout << "upy ==> " << (upy ? "dolu" : "bos") << "\n";
(void)getchar();

```

Cetvel std::move
yaptık

* Get Fonksiyonu:

```
int main()
{
    using namespace std;

    auto upx = make_unique<Point>(1, 4, 7);

    std::cout << "upx ==> " << (upx ? "dolu" : "bos") << "\n";
    Point* p = upx.get();
    std::cout << "upx ==> " << (upx ? "dolu" : "bos") << "\n";
    cout << *upx << "\n";
    cout << *p << "\n";
```

→ upx hala dolu smart pointer
→ p de dolu raw pointer



```
int main()
{
    using namespace std;
    Point* p;

    {
        auto upx = make_unique<Point>(1, 4, 7);
        p = upx.get();
        cout << *upx << "\n";
    }

    std::cout << *p << "\n"; //ub
```

Undefined Behavior

→ Control scope'ın bitmesi upx silmesi, ve p alonging pointer olacak!

* Unique_ptr'in Kendi Inisier Fonksiyonu var:

```
int main()
{
    using namespace std;
    auto upx = make_unique<Point>(1, 4, 7);

    std::cout << *upx << "\n"; ➤ custom func
    std::cout << upx << "\n"; ➤ returns the wrapped raw pointer
    std::cout << upx.get() << "\n"; ➤ " " " "
    | I
}
```

Aynı

* Unique Pointer'lar Kendi Delete'ını Elekerebilir:

```
struct PointDeleter {
    void operator()(Point* p)
    {
        std::cout << p << " adresindeki Point nesnesi delete ediliyor\n";
        delete p;
    }
};

int main()
{
    using namespace std;

    {
        unique_ptr<Point, PointDeleter> uptr{ new Point{4, 7, 9} };
        cout << *uptr << "\n";
    }

    std::cout << "main devam ediyor\n";
}
```

```

void del_point(Point* p)
{
    std::cout << p << " adresindeki Point nesnesi delete ediliyor\n";
    delete p;
}

int main()
{
    using namespace std;
    I
    {
        //unique_ptr<Point, void (*)(Point *)> uptr{new Point{4, 7, 9}, del_point};
        unique_ptr<Point, decltype(&del_point)> uptr{new Point{4, 7, 9}, del_point};
        std::cout << uptr << "\n";
        cout << *uptr << "\n";
    }

    std::cout << "main devam ediyor\n";
}

```

```

int main()
{
    using namespace std;

    auto fdel = [] (Point* p) {
        std::cout << p << " adresindeki Point nesnesi delete ediliyor\n"; I
        delete p;
    };

    I
    //unique_ptr<Point, void (*)(Point *)> uptr{new Point{4, 7, 9}, del_point};
    //unique_ptr<Point, decltype(&del_point)> uptr{new Point{4, 7, 9}, del_point};
    unique_ptr<Point, decltype(fdel)> uptr{new Point{4, 7, 9}}; //C++20
    std::cout << uptr << "\n";
    cout << *uptr << "\n";
}

```

→ C++ 20'de lambdanın `closes` type imzamını zorunlu değil.

Pma Epp 17'de { new Point< ->, fdel }

```

typedef void* WindowHandle;

WindowHandle create_window(void);
void destroy_window(WindowHandle);

int main()
{
    I
    using namespace std;

    auto fdel = [] (WindowHandle handle) {
        destroy_window(handle);
    };

    unique_ptr<void, decltype(fdel)> uptr{ create_window() };
}

```

Her zaman new ile oluşturulan dynamic object'i kontrol etmeli zannede olabilir.

C API'sini de böyle kontrol edebiliriz
unique_ptr ile