

* Tekrar:

```
#include <memory>
#include <string>
#include <iostream>

int main()
{
    using namespace std;

    auto uptr = make_unique<string>(100, 'a');

    cout << uptr->data() << '\n'; → 100 tane a (void*) ile alınan adres
    cout << *uptr << '\n'; → 100 tane a adres
    cout << (void*)uptr.get() << '\n'; → uptr'nın adresi

    cout << uptr << '\n'; → uptr'nın adresi
}
```

* Eğer, bir fonksiyon, dinamik ömrüde bir nesnenin (kaynegin) mülkiyetini almak isteniyorsa, → general olarak sink adı verilir

```
struct Nec {
    Nec() { std::cout << "Nec default ctor this : " << this << '\n'; }
    ~Nec() { std::cout << "Nec destructor this : " << this << '\n'; }
    void func()
    {
        std::cout << "Nec::func() this = " << this << '\n';
    }
};

void foo(Nec* np)
{
    np->func();
    delete np; → sink function → Farklı unique_ptr kullanmak data işaretçisi
}

int main()
{
    auto p = new Nec;
    std::cout << "p = " << p << '\n';

    foo(p);

    p = nullptr;

    std::cout << "main devam ediyor"
}
```

```
Nec() { std::cout << "Nec default ctor this : " << this << '\n'; }
~Nec() { std::cout << "Nec destructor this : " << this << '\n'; }
void func()
{
    std::cout << "Nec::func() this = " << this << '\n';
}

void foo(std::unique_ptr<Nec> up)
{
    up->func();
}

int main()
{
    foo(up: std::unique_ptr<Nec>{new Nec()}); r value expression olduğu için more semantics oluyor
    std::cout << "main devam ediyor\n";
}
```

* Factory Functions: - Eger bir fonksiyonun amaci, **nesne oluşturup, oluşturulan nesnenin otomatik olarak silinece**, buna bir nesne kullanılsın gibi bir **factory function**

```
// factory function

std::unique_ptr<Nec> create_nec()
{
    auto uptr = std::make_unique<Nec>();
    uptr->func();

    return uptr;
}
```

```
int main()
{
    std::cout << "main basladi\n";
    {
        auto uptr = create_nec();
    }

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

* Her ne kadar bu fonksiyonun return değeri **unique_ptr** olsa da, biz tek seferde **akasyonu** kullanmak sonunda **düşürüriz**.

```
std::unique_ptr<Nec> create_nec()
{
    auto uptr = std::make_unique<Nec>();
    uptr->func();

    return uptr;
}

int main()
{
    std::cout << "main basladi\n";
    {
        std::shared_ptr<Nec> sptr = create_nec();
    }

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

Shared_ptr'nın unique_ptr
acılımına var. Böylece unique_ptr'ın
shared_ptr halinde kullanılabılır.

* Pass Through Functions: Fonksiyon her **unique_ptr** alır, **return değer sine unique_ptr**

```
16
17 std::unique_ptr<Nec> fpass(std::unique_ptr<Nec> uptr)
18 {
19     uptr->func();
20     return uptr;
21 }

22
23
24 int main()
25 {
26     auto up = std::make_unique<Nec>();
27     up = fpass(std::move(up));
28     std::cout << "main devam ediyor\n"; move ömrüne syntek hatalı,
29     (void)getchar(); cinsîye kopyalanmış hatalı
30
31
32
33 }
```

*Unique Ptr Tipik Hatalar:

1. Bir dinamik olarak tanımlanmış adresle (denk болу unique_ptr - gerekli bir oluşturulması) → Dangling Pointer

```
using namespace std;

auto p = new Nec;
{
    auto up1 = unique_ptr<Nec>{ p };
    auto up2 = unique_ptr<Nec>{ p };
}

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

2. Unique-Ptr boyunca dereference edilmesi. → Undefined Behavior

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

    unique_ptr<string> up1;
    unique_ptr<string> up2{};
    unique_ptr<string> up3=nullptr;

    cout << (up1 ? "dolu" : "bos") << "\n";
    cout << (up2 ? "dolu" : "bos") << "\n";
    cout << (up3 ? "dolu" : "bos") << "\n";
}
```

↳ Herhangi bir değer yok.

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

    unique_ptr<string> up1;
    unique_ptr<string> up2{};
    unique_ptr<string> up3=nullptr;

    cout << (up1 ? "dolu" : "bos") << "\n";
    cout << (up2 ? "dolu" : "bos") << "\n";
    cout << (up3 ? "dolu" : "bos") << "\n";

    cout << *up1; ↳ Run time error
}
```

↳ Exception throws!

→ Her zaman istenilen yapanın öncesi sorgula dolu mu değil?

3. Unique Ptr dolu ama null işaretinden boş olabilir. → release edilip → resource leak

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

    auto up = make_unique<string>("necati ergin");

    auto ptr = up.release(); ↳

    delete ptr; ↳ Mükemmel sonibr silmek zorunda
```

4. Release halledilmesi gereken yerde get kullanıp, get'in return değeriyle yeni bir unique_ptr oluşturulması. → Dangling Pointer / Runtime error

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

    auto up = make_unique<string>("necati ergin");
    ↳ up destructor'i çağırılmış.

    auto ptr = up.get();

    delete ptr; ↳ Double deletion
```



* Shared Ptr:

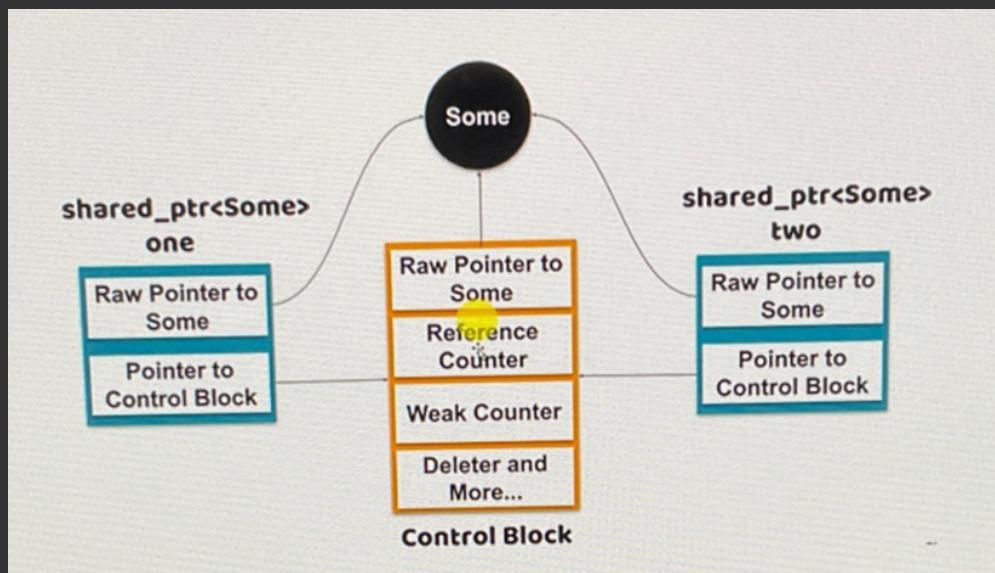
include <memory>

- shared ownership - Paylaşımı pointer
- Bir kaynaktan, birden fazla pointer ile gerekliyse, fazla kodları etmemek.
- Yükselen riskler - resource leak (krim delete edilecektir)
 - bir resource kullanırken başka kodun onu delete etmesi
- shared pointer Reference Counting kullanır.

→ Pointeler, kendilerinin
son mı, değil mi? bunu biliyor

Günümüzde en çok pointer delete edilir.
Kullanıcı da obecte etmeli.

→ Fakat bu konuya müzakere



Microsoft Visual Studio Debug Console

```

sizeof(string*) = 4
sizeof(uptr) = 4
sizeof(sptr) = 8 → size daha yüksek, çünkü hem string* hem de control block
  tutuyor
  
```

- Sınıfın copy constructor'ı, copy assignment'ı, delete edilmeye bu safer!

* Shared Ptr Oluşturma:

- boş oluşturabilir. Default / Value / Nullptr init

- Operator bool ile boş / dolu sorusunu yanıtlayabilir.

```

int main()
{
    using namespace std;

    shared_ptr<Date> sp1;
    shared_ptr<Date> sp2{};
    shared_ptr<Date> sp3{nullptr};
  
```

- Unique Ptr da olduğu gibi, Dinamik bir sınıfın adresini verirerek, nüfusa erişebilir.

```

int main()
{
    using namespace std;

    shared_ptr<Date> sp1{ new Date{1, 5, 1978} };

    cout << "sp1 : " << (sp1 ? "dolu" : "bos") << '\n';
}
  
```

```
#include <memory>
#include <string>
#include <iostream>
#include "date.h"
```

```
int main()
{
    using namespace std;
    shared_ptr<Date> sp;

    {
        shared_ptr<Date> sp1{ new Date{1, 5, 1978} };
        {
            auto sp2 = sp1;
            {
                auto sp3 = sp2;
                sp = sp3; → sp'ye hala nıktır.
            }
        }
    }

    std::cout << "main devam ediyor\n";
    (void)getchar();
}
```

- Bu senaryoda obektler 'a' hâlinde deklar edilmiş

* use_count fonksiyonu:

- shared pointer'ın kaç kez kullanıldığından fonksiyon.

```
using namespace std;
shared_ptr<Date> sp;

{
    shared_ptr<Date> sp1{ new Date{1, 5, 1978} };
    {
        auto sp2 = sp1;
        {
            auto sp3 = sp2;
            sp = sp3;
            her biri  
= 1 ← [ std::cout << "sp.use_count() = " << sp.use_count() << "\n";
            std::cout << "sp3.use_count() = " << sp3.use_count() << "\n";
            std::cout << "sp2.use_count() = " << sp2.use_count() << "\n";
            std::cout << "sp1.use_count() = " << sp1.use_count() << "\n";
        }
        std::cout << "sp2.use_count() = " << sp2.use_count() << "\n";
    }
}

std::cout << "main devam ediyor\n";
(void)getchar();
```

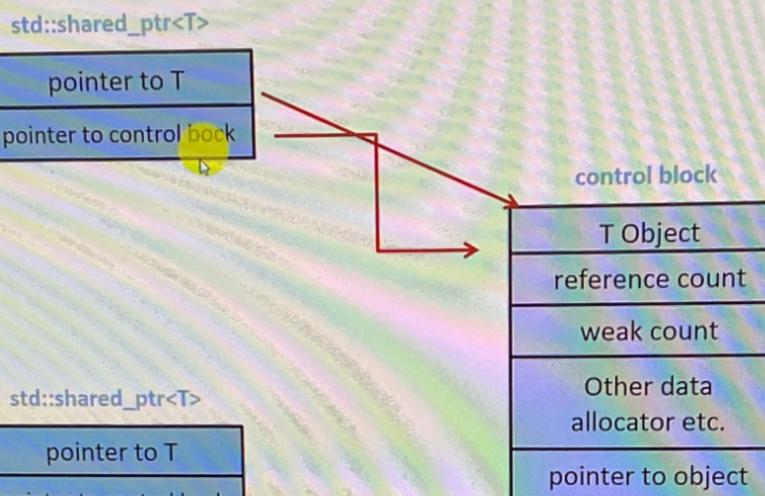
Artık 3. Aşırı sp3 destroy edildi!

* make_shared: same as make_unique

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

    auto sp1 = make_shared<Date>(5, 8, 1998);
    auto sp2 = make_shared<Date>(4, 4, 1944);

    sp1 = sp2; → Fakat bu da sp2'nin kaybolması sp1'e kaybolur.
    → sp1'nin destructor çağırılır. Sp1'i gösteren 1 pointer varsa:
}
```



* Reset Fonksiyonu:

- Eğer arguman geçerlizse, boyutunu sondurur.
- Eğer adres, geçerse kendini kaybetmeye, adresi gelen kaynaktır.

unique_ptr

→ her resette / nullptr atadığında, use count azalır

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

    auto sp1 = make_shared<Date>(12, 12, 2012);
    auto sp2 = sp1;
    auto sp3 = sp2;
    auto sp4 = sp2;

    std::cout << "sp4.use_count() = " << sp4.use_count() << "\n";
    sp1.reset();
    sp1 = nullptr;
    sp1 = {};
    (void)getchar();
}
```

* Shared_ptr Array Specifikasyonu:

```
class Myclass {
public:
    void foo(){}
    unsigned char buffer[32]{};
};

int main()
{
    using namespace std;

    {
        shared_ptr<Myclass[]> sptr(new Myclass[5]);

        sptr[2].foo()

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

* C++ 11'de olmadığını, alors serin eklediğini belirtti.

* Get Fonksiyon / Insertesi:

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

    auto sp = make_shared<string>("necati engin eray goksu Musa sertkaya");
    cout << sp << "\n"; → Daha önceki nesnenin adresi
    cout << sp.get() << "\n"; → Daha önceki nesnenin adresi
    cout << (void *)sp->data() << "\n"; → Yozunun tuttuğu adres
}
```

* Operator Bool:

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

    auto sp1 = make_shared<string>("necati ergin");
    auto sp2 = make_shared<string>("necati ergin");

    cout << boolalpha;

    cout << (sp1 == sp2) << "\n"; → false kaynak farklı
    cout << (*sp1 == *sp2) << "\n"; → true stringler aynı
    sp2 = sp1;
    cout << (sp1 == sp2) << "\n"; → true

    → Her ne zaman da sp1'in kaynağını bırdan
       sp2'nin kaynağını aldı.
```

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

    auto sp1 = make_shared<string>("necati ergin");
    auto sp2 = make_shared<string>("necati ergin");

    cout << boolalpha;

    cout << (sp1->data() == sp2->data()); → false
    ↴
    data, string'in dosyası. Andreler farklı olduğu için
    false
```

* Deleter:

- unique_ptr'da olduğu gibi, 2. bir template parametresi yok. → Type Eroded (?)
- fakat constructor'a kendin delete fonksiyonunu geçesilir.

```
int main()
{
    using namespace std;
    const auto f = [](string* p) {
        std::cout << "custom deleter\n";
        delete p;
    };

    {
        shared_ptr<string> sptr{ new string("necati ergin"), f };
    }

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

BU şekilde custom lambda geçer.

* Shared_Ptr'ın Contentless Sınaması:

```
using sptr = std::shared_ptr<std::string>;

int main()
{
    using namespace std;

    list<sptr> mylist;

    for (int i = 0; i < 20; ++i) {
        mylist.push_back(make_shared<string>(rname() + ' ' + rfname()));
    }

    for (const auto& sp : mylist) {
        std::cout << *sp << "\n";
    }

    vector<sptr> myvec(mylist.begin(), mylist.end());
    sort(myvec.begin(), myvec.end(), [] (const auto& sp1, const auto& sp2) {return *sp1 < *sp2; });

    std::cout << "\nsıralanmış biçimde\n";
    for (const auto& sp : myvec) {
        std::cout << *sp << "\n";
    }
}
```

* Shared_ptr as Class Members:

```
(Global Scope)
using svec = std::vector<std::string>;
```

```
class NameList {
public:
    NameList(std::initializer_list<std::string> list) : mp{new svec{list}} {}
```

```
    void print() const
    {
        for (const auto& s : *mp) {
            std::cout << s << " ";
        }
        std::cout << "\n";
    }
private:
    std::shared_ptr<svec> mp;
};
```

```
int main()
{
    NameList male_list{ "ferhat", "volkan", "safa" };
    NameList female_list{ "esra", "demet", "aleyna", "fadime" }; } → class de
    male_list.print();
    female_list.print();
}
```

```
male_list.print();
female_list.print();

auto m1_list = male_list; } → Ayn
auto m2_list = male_list; liste
auto m3_list = male_list; } exception.

m3_list.print();

m1_list.add_name("hakan");
m2_list.add_name("samet");
m3_list.add_name("necati"); } tem liste
                            exception.

male_list.print();

m1_list.sort();

m2_list.print(); } ←

auto f1_list = female_list;

f1_list.add_name("canan");
f1_list.add_name("yelda");

}
```

* Kaynak sınıfları da oluştur. Aynı sınıf içerisinde farklı nesneler, aynı logikin yerlilikleri. Exception throws da silme gerekli.

* weak_ptr:

- Venedir bir smart pointer değil!
- Shared_ptr sınıfının bir yedekası. Shared_ptr nesnesiyle weak_ptr ilişkisine etkileşimde onun referans sayısını artırmaz
- weak_ptr'den shared_ptr elde edilebilir.

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

    auto sp1 = make_shared<Date>(1, 2, 1992);
    auto sp2 = sp1;

    std::cout << "sp1.use_count() = " << sp1.use_count() << "\n"; } ←

    weak_ptr<Date> wp = sp1;

    std::cout << "sp1.use_count() = " << sp1.use_count() << "\n"; } → use count yine 2
    (void)getchar(); } ← use count değişti mi.

}
```

* weak_ptr referansı edilemez XX

* Sınıfın bool özellikleri expired fonksiyonu var. → weak_ptr, hedefi getiren shared_ptr'in kaynakları kullanımda mı / hedefini kontrol ediyor. if expired, return true.

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

    auto sp = make_shared<Date>(1, 2, 1992);
    weak_ptr<Date> wp = sp; } ← 5 overloads
    cout << boolalpha << wp.expired() << "\n";
    cout << wp.use_count() << "\n";

    sp.reset();

    cout << boolalpha << wp.expired() << "\n";
    cout << wp.use_count() << "\n";

    (void)getchar();
}
```

* weak_ptr Lock Fonksiyonları

- Onu olusturan `shared_ptr`'in return eder. Eger o logik verilmisse / destroy edilimisse, nullptr dandarır.

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

    auto sp = make_shared<Date>(1, 2, 1992);

    weak_ptr<Date> wp = sp;
    I

    sp.reset();

    if (auto spx = wp.lock()) {
        cout << *spx << "\n";
        cout << "use_count = " << spx.use_count() << "\n";
    }
    else {
        std::cout << "kaynak geri verilmis\n";
    }

}
```

* Array, multimap, kismalar, gizlilikler gibi devar etmeli.

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

    auto sp = make_shared<Date>(1, 2, 1992);

    weak_ptr<Date> wp = sp;

    //sp.reset();

    if (!wp.expired()) {
        shared_ptr<Date> spx(wp);
        cout << *spx << "\n";
    }

}
```

* Eger expired oldumissa, yani shared_ptr olusturuldu, Eger bos bir weakptr'dan shared_ptr olustursak, exception throw eder.