2023 12 04

std::unique_ptr

```
class Myclass;
int main()
{
    using namespace std;
    unique_ptr<Myclass>; // incomplete type olarak kullanabiliriz
}
```

```
/*
    uptr.release(): sarmaladığı adresi döndürür mülkiyeti bırakır (dtor çağırmaz)
    uptr.get(): sarmaladığı adresi döndürür

*/
```

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

    auto uptr = make_unique<string>("alican korkmaz");
    // ikiside aynı
    cout << uptr << "\n";
    cout << uptr.get() << "\n";
    // tanımsız davranış
    auto p = uptr.get();
    unique_ptr<string> upx(p);
    // geçerli
    auto p = uptr.release();
    unique_ptr<string> upx(p);
}
```

```
template<typename T>
struct DefaultDelete
{
    void operator()(T* p)
    {
        delete p;
    }
};
template<typename T, typename D = std::default_delete<T>>
class UniquePtr
{
    public:
        ~UniquePtr()
        {
            if (mp)
            {
                D{}(mp);
            }
        }
    private:
        T* mp;
}
```

```cpp
#include <iomanip>
struct SDeleter
{
    void operator()(std::string* p)const noexcept
    {
        std::cout << std::quoted(*p) <<  " delete ediliyor\n";
        delete p;
    }
}
void fdeleter(std::string *p)
{
    std::cout << std::quoted(*p) << " delete ediliyor\n";
}
int main()
{
    using namespace std;

    {
        //1
        unique_ptr<string, SDeleter> uptr{new string{"tamer dundar"}};
        //2
        unique_ptr<string, decltype(&fdeleter)> uptr{new string{"tamer dundar"}};
        //3
        auto fdel = [](string *p)
        {
            std::cout << std::quoted(*p) <<  " delete ediliyor\n";
            delete p;
        };
        unique_ptr<string, decltype(fdel)> uptr{new string{"tamer dundar"}};


        //4
        decltype([])string *p)
        {
            std::cout << std::quoted(*p) <<  " delete ediliyor\n";
            delete p;
        };
        unique_ptr<string, decltype([])string *p)
        {
            std::cout << std::quoted(*p) <<  " delete ediliyor\n";
            delete p;
        };> uptr{new string{"tamer dundar"}};

    }

    std::cout << "main devam ediyor\n"
    /*
        tamer dundar delete ediliyor\n
        main devam ediyor
    */
}
```

Neden kendi deleter'imiz kullanmalıyız:

- Delete etmek dışında kaynağı başka bir şekilde sonlandırmak istersek.
- Delete ederken yanında başka işlemler yapmak istersek.

```cpp
#include<cstdio>
int main()
{
    auto fdel = [](std::FILE* f)
    {
        std::cout << "file is being closed!\n";
        std::fclose(f);
    };

    std::unique_ptr<std::FILE, decltype(fdel)> uptr{fopen("melike.txt"), "w"  };
    fprintf(uptr.get(), "Emre Bahtiyar");
}
```

```cpp
// C-style Ctor Dtor
// unique_ptr ile herhangi bir kaynağı sarmalayabiliriz.
struct Data
{

};

Data * createData(void);
void do_something(Data *);
void do_this(Data *);
void do_that(Data *);
void destroyData(Data *);

int main()
{
    auto fdel = [](Data *p)
    {
        destroyData(p);
    };

    unique_ptr<Data, decltype(fdel)> uptr(createData());

    do_something(uptr.get());
    do_this(uptr.get());
    do_that(uptr.get());
    destroyData(uptr.get());
}
```

```
struct Nec
{
    Nec()
    {
        std::cout << "Nec default ctor this : " << this << "\n";
    }

    ~Nec()
    {
        std::cout << "Nec default ctor this : " << this << "\n";
    }

    char buf[256]{};
}

int main()
{
    using namespace std;
    // tanımsız davranış 4 defa ctor çağrılır ama bi kere dtor çağrılır
    unique_ptr<Nec>  uptr(new Nec[4]);

    // 4 kez ctor çağrılır 4 kez dtor çağrılır
    auto fd = [](Nec *p) {delete []p;};
    unique_ptr<Nec, decltype(fd)> uptr(new Nec[4]);

    // unique_ptr için partial spec  for array pointer
    unique_ptr<Nec[]> uptr(new Nec[4]);
    auto up = make_unique<Nec[]>(5);

}
```

```
int main()
{
    Date *p = new Date{3, 12, 1872};

    {
        unique_ptr<Date> upx(p);
    }
    // tanımsız davranış
    unique_ptr<Date> upy(p); // p dangling pointer oldu
}
```

```
// unique_ptr -- container'da tutma
int main()
{
    using namespace std;

    vector<std::unique_ptr<Date>> dvec;

    dvec.reserve(10);
    for (auto i = 1; i < = 10; ++i)
    {
        dvec.push_back(new Date {i ,i 2000 +i});
    }
}
```

```cpp
int main()
{
    using namespace std;
    vector<unique_ptr<Date>> dvec;

    auto uptr = make_unique<Date>(1, 5, 1986);

    dvec.push_back(move(uptr));
    dvec.push_back(make_unique<Date>(1, 5, 1986));

    dvec.emplace_back(new Date{5, 6, 1965});

}
```

```cpp
void fsink(std::unique_ptr<Date> uptr)
{
    std::cout << *uptr << '\n';
    // Date sınıfı bu scopeta yok olur
}

int main()
{
    using namespace std;
    fsink(make_unique<Date>(3, 6, 1987));
    cout << "main devam ediliyor\n";
}
```

```cpp
// pass-through
std::unqiue_tr<Date> pass_through(std::unique_ptr<Date> uptr)
{
    std::cout << *uptr << '\n';
    return uptr;
}

int main()
{
    using namespace std;
    auto up = pass_through(make_unique<Date>(3, 6, 1987));
    // uptr main scope sonunda sonlanır
    cout << "main devam ediyor\n";
}
```

std::shared_ptr

```cpp
// std::shared_ptr -- bir nesnenin birden fazla sahibi olabilir
int main()
{
    using namespace std;
    std::cout << "sizeof(unique_ptr<string>)" = << sizeof(unique_ptr<string>) <<
"\n"; // 4
    std::cout << "sizeof(shared_ptr<string>)" = << sizeof(shared_ptr<string>) <<
"\n"; // 8
}
```

```cpp
template<typename T>
class SharedPtr
{
};
int main()
{
    using namespace std;

    shared_ptr<Date> sp1(new Date{ 1, 2, 1998});
    {
        auto sp2 = sp1;
        // sp2  burda biter ama dtor çağrılmaz
    }
    std::cout << "main devam ediyor\n";
    // sp1 burda sonlanır ve dtor çağrılır
}
```

```cpp
// use_count()
int main()
{
    using namespace std;
    shared_ptr<Date> sp1(new Date{ 1, 2, 1998});
    auto sp2 = sp1;

    //use count = 2
    cout << "use count = " sp1.use_count() << "\n";
    cout << "use count = " sp2.use_count() << "\n";

    {
        auto sp3 = sp2;
        cout << "use count = " sp3.use_count() << "\n"; // use count = 3
    }

    cout << "use count = " sp2.use_count() << "\n"; // use count = 2
}
```

```cpp
void* operator new(std::size_t n)
{
    std::cout << "operator new called n = " << n << "\n";

    void* vp = std::malloc(n);
    if (!vp)
    {
        throw std::bad_alloc{};
    }
    std::cout << "the adress of the allocated block is: " << vp <<"\n";

    return vp;
}

struct Nec
{
    char buf[512]{};
};

void foo()
{
    std::cout << "foo cagrildi\n";
    auto pnec = new Nec;
    std::shared_ptr<Nec> sptr(pnec);
    // burda önce Nec için blok açılıyor
    //sonra shared_ptr'in kontrol bloğu için yer açılıyor
}

void bar()
{
    std::cout << "foo cagrildi\n";
    auto pnec = new Nec;
    std::shared_ptr<Nec> sptr(pnec);
    // burda Nec ve kontrol bloğu için ayrılan yer aynı anda açılıyor
    // derleyici optimazyon yapıyor
}

int main()
{
    foo();
    bar();
}
```

```cpp
// type erasure -- shared_ptr
class Myclass
{
    public:
        ~Myclass()
        {
            std::cout << "Myclass dtor\n"
        }
};

struct MyclassDelete
{
    void operator()(Myclass* p)const
    {
        std::cout << "the object at the adress of " << p << "is being deleted\n";
        delete p;
    }
};

int main()
{
    using namespace std;

    {
        shared_ptr<Myclass> sptr(new Myclass, MyclassDeleter{});
    }

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

```cpp
// unique_ptr to shared_ptr
int main()
{
    using namespace std;
    auto uptr = make_unique<Date>(1, 1, 2024);
    shared_ptr<Date> sptr(move(uptr));

    /*
        std::shared_ptr kontrol bloğu ne zaman oluşur:
        - eğer default dtor ile oluşturulduysa  kontrol bloğu oluşmaz
        - eğer 2. ya da daha fazla shared_ptr oluşturulduysa kontrol bloğu
oluşmaz.
        - unqiue_ptr'den shared_ptr dönüştürüyorsak kontrol bloğu oluşur
        - 1. kez shared_ptr oluşturuyorsak kontrol bloğu oluşur.
    */
}
```

```cpp
// shared_ptr fonksiyonları
int main()
{
    using namespace std;
    shared_ptr<Date> sp1(new Date{1, 1, 2024});

    auto sp2 = sp1;
    auto sp3 = sp2;
    // üçünde aynı pointer sarmalar yani get fonksiyonları aynı adresi döndürür
    cout << sp1.get() << "\n";
    cout << sp2.get() << "\n";
    cout << sp3.get() << "\n";

    if (sp) // boş mu dolu mu operator bool ile yapılabilir

    sp1.reset(); // mülkiyeti bırakıyor unique_ptr'deki release gibi
}
```

```cpp
void foo(std::unqiue_ptr<std::string>)
{
    // reference sayıcı artıcak sonra fonksiyon sonlanınca azalcak

}

void bar(std::unique_ptr<std::string>& r)
{
    // reference sayıcı artmıcak
    // eğer shared_ptr değerini değiştirmiceksek, reset yapmıcaksak
    // fonksiyon parametresini reference yapmaya gerek yok
}
```