2023 12 06

std::weak_ptr

```cpp
#include <memory>
#include <string>

using namespace std;

int main()
{
    /*
        shared_ptr ile oluşturduğumız weak_ptr ile shared_ptr'in kaynağının
        sonlanıp sonlamadığıa bakabiliyoruz.
    */

    shared_ptr<string> sptr(mew string{"tamer dundar"});
    cout << "sptr.use_count() = " << sptr.use_count() <<"\n";
    weak_ptr wp = sptr;
    // sptr.use_count() sabit kalır
    cout << "sptr.use_count() = " << sptr.use_count() << "\n";

    weak_ptr wp1 = sptr;
    cout << "wp.use_count() = " <<  wp.use_count() <<"\n";
    cout << "wp1.use_count() = " <<  wp1.use_count() <<"\n";

    auto wp3 = wp2;
    cout << "wp3.use_count() = " <<  wp3.use_count() <<"\n";

    // hepsi 1 çıkar.
}
```

```cpp
//  weak_ptr --expired and lock
int main()
{
    shared_ptr<string> sptr(mew string{"tamer dundar"});
    weak_ptr wp = sptr;
    // true dönerse kaynak sonlamış, false verirse kaynak halen hayatta
    wp.expired()

    // kaynak hayattaysa shared_ptr döndürür, hayatta değilse nullptr döndürür
    auto spx = wp.lock()
    // cpp 98s
    if (shared_ptr<string> spx1 = wp.lock(); spx != nullptr)
    {

    }

    shared_ptr sp(wp); // kaynak sonlamışssa  bad_weak_ptr throw eder
}
```

```cpp
struct A
{
    std::shared_ptr<B> bptr;
    A()
    {
        std::cout << "A default ctor this = " << this << "\n";
    }
    ~A()
    {
        std::cout << "A default ctor this = " << this << "\n";
    }
};

struct B
{
    std::shared_ptr<A> bptr;
    B()
    {
        std::cout << "B default ctor this = " << this << "\n";
    }
    ~B()
    {
        std::cout << "B default ctor this = " << this << "\n";
    }
};

int main()
{
    using namespace std;

    shared_ptr<A> spa(new A);
    shared_ptr<B> spa(new B);
    // dtor çağrılmaz
    spa->bptr = spb;
    spb->aptr = spa;

}
```

CRTP (Curiously Recurring Template Pattern)

```cpp
// CRTP (Curiously Recurring Template Pattern)
using namespace std;
//CRTP Base
template <typename T>
class Base
{
    // taban sınıf türemiş sınıfa interface sağlar ve taban sınıf bu interfacete
    // türemiş sınıfın fonksiyonlarını kullanabilir.
    void func()
    {
        static_cast<Der*>(this).foo();
    }
};

class Der : public Base<Der>
{

};
```

```
class Nec{};
int main()
{
    Nec *ptr = new Nec;

    // tanımsız davranış dangling pointer oluşturur
    shared_ptr<Nec> sp1(ptr);
    shared_ptr<Nec> sp2(ptr);

    cout << sp1.use_count() << '\n'; // 1
    cout << sp2.use_count() << '\n'; // 1

    // tanımsız davranış yok
    shared_ptr<Nec> sp3(sp2);
    cout << sp3.use_count() << '\n'; // 2
}
```

Eğer bir sınıfın üye fonksiyonu içinde shared_ptr ile hayatı kontrol edilien *this nesnesini gösteren shared_ptr'nin kopyasını çıkartmak isterseniz sınıfınızı CRTP örüntüsü ile kaltım yoluyla std::enable_shared_from_this sınıfından elde etmelisiniz.

```
class Nec  : public std::enable_shared_from_this<Nec>
{
    public:
        void func()
        {
            /*
                reference count 2 olmaz hala 1 olur. tanımsız davranıştır
            */
            shared_ptr<Nec> spx(this);
            cout << spx.use_count() << "\n";
        }

        void bar()
        {
            auto spx = shared_from_this();
            cout << "spx.use_count() = " << spx.use_count() << "\n"; // 2
        }
};

int main()
{
    shared_ptr<Nec> sptr(new Nec);

    cout << "sptr.use_count() = " << sptr.use_count() << "\n"; // 1
    sptr->func();
    cout << "sptr.use_count() = " << sptr.use_count() << "\n"; // 1

    auto p = new Nec;
    try{
        p->bar(); // exception throw eder
    }
    catch(const std::bad_weak_ptr& ex) {
        std::cout << "exception caught: " << ex.what() << "\n";
    }

}
// container'da shared_ptr tutma
```

```cpp
#include <list>
using namespace std;
int main()
{
    list<shared_ptr<Date>> mylist;

    mylist.push_back(make_shared<Date>(2, 6, 1999));
    mylist.push_back(make_shared<Date>(2, 6, 1993));
    mylist.push_back(make_shared<Date>(21, 11, 1997));
    mylist.push_back(make_shared<Date>(7, 10, 2004));

    for (auto ptr : mylist)
    {
        cout << *ptr << "\n";
    }

    vector<shared_ptr><Date>> myvec(mylist.begin(), mylist.end());

    // vectordeki shared_ptr değiştirsem listteki shared_ptr değişir

    sort(myvec.begin(), myvec.end(), [](auto p1, auto p2)
    {
        return *p1 < *p2;
    }

    for (auto sp : myvec)
    {
        cout << *sp << '\n';
    }
}
```

```cpp
using svector = std::vector<std::string>
class NameList
{
    public:
        NameList() = default;
        NameList(std::initializer_list<std::string> list> : sptr(new
svector{list}}
        {

        }
        void add(const std::string& name)
        {
            sptr->push_back(name);
        }

        void remove(const std::string& name)
        {
            sptr->erase(std::remove(sptr->begin(), sptr->end(), name), sptr-
>end());
        }

        size_t size()const
        {
            return sptr->size();
        }

        void print()const
        {
            for(const auto& s : *sptr)
            {
                std::cout << s << " ";
            }
            std::cout << "\n";
        }

        void sort()
        {
            std::sort(sptr->begin(), sptr->end());
        }
    private:
        std::shared_ptr<svector> sptr;
}

int main()
{
    // x, y ve z 'nin veri elemanı olam shared_ptr aynı vektörü gösterir.
    NameList x{ "ali", "gul", "eda", "naz"};
    NameList y = x;
    NameList z = y;

    x.add("nur");
    y.add("tan");

    std::cout << "listede " << x.size() << "isim var\n";
    z.sort()
    x.print();
    y.remove("gul");
    z.print();
}
```

```cpp
// Bir sınıf için dinamik bellek yönetimini özelleştirme
class Myclass
{
    Myclass()
    {
        std::cout << "default ctor this : " << this << "\n;
    }

    ~Myclass()
    {
        std::cout << "dtor this : " << this << "\n";
    }

    // operator new ve delete static yazmasak bile static üyedir
    void* operator new(size_t)
    {
        std::cout << "Myclass::operator new n:" << n << "\n";
        auto p = std::malloc(n);

        if (!p)
        {
            throw std::bad_alloc{};
        }

        std::cout << "address of the allocated block is " << p << "\n";
    }
    void  operator delete(void* vp)noexcept
    {
        std::cout << "Myclass::operator delete vp:" << vp << "\n";
        std::free(vp);
    }

    void func()
    {
        std::cout << "Myclass func() this = " << this << "\n";
    }

    private:
        unsigned char buf[512]{};
};

int main()
{
    auto p = new Myclass;
    delete p;
}
```