2023.09.15

Operator Overloading Enum

```cpp
enum class Weekday
{
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
Weekday& operator++(Weekday& wd)
{
    using enum Weekday; // Cpp20
    return wd = wd == Saturday ? Sunday :
    static_cast<Weekday>(static_cast<int>(wd) + 1);
};
Weekday operator++(Weekday& wd, int)
{
    Weekday temp {wd};
    ++wd;

    return temp;
}

std::ostream& operator<<(std::ostream&, const Weekday&)
{
    static const char* const pwdays[] = {"Sunday", "Monday",
                "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};

    return os << pwdays[static_cast<int>(wd)];
}

int main()
{
    Weekday wd { Weekday::Friday};

    // ++wd; operator overloading ile olur

}
```

Nested classes

```cpp
class Myclass
{
    class Nec; // Nested classes

    //Nested type or type member
    typedef int value_type;
    using value_type = int;
    enum Color {Blue, Red, Green};
}
```

- Nested type or type memberları class'ın içinde tanımladığımız da class scope'ta oluyor namespace scope yerine
- Class scope sayesinde erişim kontrollu kazanmış oluyoruz.
- Logic açından bulunduğu class ile ilişkisel olduğunu belirtmiş olduk.

```
//nec.h
class Nec
{
    public:
        class Erg;
    private:
        void foo(Erg);
        Erg bar();
}

void Nec::foo(Erg e)
{
    // Hata yok
}
Erg Nec::bar()
{
    /*
    Hatalı çünkü Erg class scope'ta aranmaz ama
    Nec::Erg Nec::bar() diye tanımlasak hata almayız
    */
}
```

```
struct ValueType{};

class Myclass
{
    void foo()
    {
        ValueType = x; // int türünde olur class scope'ta aranır
    }
    ValueType mx;  // struct ValueType türünde int türünde değil
    typedef int ValueType;

    class Nec; // incomplete type
}

class Nec {}; // Class'ın içinde tanımlanan Nec ile bu Nec farklı
```

```cpp
class Myclass
{
    static void mbar();
    int mx;
    class Erg
    {
        public:
            void bar();
        private:
            void foo()
            {
                // Modern cpp ile geldi
                Myclass::mbar(); // mbar private olmasına rağmen erişebildik
                auto sz = sizeof(mx)
            }
    };
    void func()
    {
        Erg e;
        e.foo(); // kendi nested class'ımızın private bölümüne erişemez
    }

};
```

```cpp
class Myclass
{
    class Nested
    {

    };

    public:
        static Nested foo();
};

int main()
{
    /*
        Myclass::Nested access controlden kaynaklı hatalı oluyor
        ama auto x hata yok çünkü adını yazmadık Myclass::Nested
        access kontrol isim aramayla ilgili eğer adını yazmayıp auto kullanırsam
        acces kontrole takılmayız.
    */
    Myclass::Nested x = Myclass::foo(); // hatalı
    auto x = Myclass::foo();
}
```

```cpp
//system.h
class System
{
    public:
        class Element
        {
            void foo();
            /*
                foo fonksiyonun tanımı inline olarak
                sınıfın içinde yapabiliriz

                System sınıfın içinde foo fonksiyonun tanımını
                yapamayız

                system.cpp içinde tanımlayabiliriz void System::Element::foo();
            */

            Element& operator=(const Element&);
            Element(const Element&);
        };

    void Element::foo(){}; // hatalı
}

//system.cpp

void System::Element::foo(); // böyle tanımlayabiliriz.
System::Element& System::Element::operator=(const Element& other)()
{
    return *this;
}

System::Element::Element(const Element&)();
```

Design Pattern (pimpl idiom)

Diğer isimleri:

- pimpl idiom (pointer to implementation)
- handle-body idiom
- cheshire cat
- compiler firewall
- opaque pointer

Bu pattern sınıfın private bölümünü client kodlardan gizlemeye yönelik

Neden sınıfımızın private bölümünü gizlemek isteriz?

1. Compilation time kısalır çünkü A, B classları için başlık dosyalarını include etmek zorundayız eğer private gizlersek include etmeyiz ve time kısılır.
2. Elemen eklemek, çıkarmak veya tanımlama sırasını değiştirmek tamer.h include eden kodların hepsi yeniden derlenmesi gerekir. Eğer elemanları gizlersek tamer.h yapılabilecek değişikler tamer.h kullanan kodları derlenmesine gerek kalmıcak
3. implementation hakkında bilgi vermemek

Pimpl Idiom örnek

```
// tamer.h
class Tamer
{
    public:
        Tamer();
    private:
        A ax;
        B bx;
        int ival;
}

// pimpl idiom uygulaması (eski - tip)
// tamer.h
// Maliyeti olan bir işlem
class Tamer
{
    public:
        void foo();
        void bar();
    private:
        class pimpl;
        pimpl *mp; // smart pointer kullanmak daha mantıklı
}

// tamer.cpp

struct Tamer::pipml
{
        A ax;
        B bx;
        int ival;
}

Tamer::Tamer() : mp{new pipml{}};

void Tamer::foo()
{
    mp->ax;
    mp->bx;
}
```

Sınıfın veri elemanlarını başka bir sınıf üzerinden kullanıyoruz yani pimpl sınıfını kullanıyoruz Tamer class'nın data memberlerina erişmek için

Inheritance (Kalıtım)

Kalıtımda kaynak olarak kullanılan sınıfa base (taban). Üretilen sınıfa da derived (türemiş) ismi verilir. (Cpp'da)

3 Ayrı katılım kategorisi var:

- public inheritance
- private inheritance
- protected inheritance

Multiple Inheritance: Bir sınıf tek bir kalıtım işlemiyle birden fazla taban sınıfın interfacesini alabilir.

Kalıtımı mekanizması ile bir sınıf oluşturmamız için bir complete type a ihtiyacımız var.

```
class Base;
class Der : public Base{}; // public private protected gibi anahtar sözcükleri kullanıabilir

class Der : Base {}; // private olarak Base classının inheritance ettik
struct Der : Base {}; // public olarak Base classının inheritance ettik
```

```cpp
class Base
{
    public:
        void foo();
        void bar();
        void baz();
};

class Der : public Base
{

};

int main ()
{
    Der myder;
    myder.foo();
    myder.bar();
    myder.baz();

    Der der;

    Base* p = &der;
    Base& br = der;
}

// Bu fonksiyonlara Base sınıfını kalıtım yoluyla elde etmiş sınıflar
//arguman olarak verilebilir
void base_func_ptr(Base* p);
void base_func_ref(Base& p);
```