2023.09.25

Private Inheritance

```
class Base
{
    public:
        void bar();
        void baz();
    private:
        void foo();
    protected:
        void pro_func();

};
// private inheritance
class Der : Base // private Base
{
    friend void gf();
    void func()
    {
        foo(); // hatalı olur
        /*
            public private ya da procted' da olsa Base sınfının
            private bölümüne erişemeyiz.
        */
        // private kalıtım olduğu için pro_func'a erişemeyiz
        pro_func() // hatalı

    }

    void der_bar()
    {
        // hatalı değil erişebiliriz
        Der myder;
        Base *o =&myder;
    }
};
void gf()
{
    // hatalı değil erişebiliriz
    Der myder;
    Base *o =&myder;
}

int main()
{
    Der myder;
    /*
        Base sınıfın public interfacesi Der sınıfın artık private
        interfacesindedir.
    */
    myder.bar(); myder.baz(); // hatalı olur çünkü private

    // private kalıtım olduğu için hatalı
    Base* p = &myder;
    Base& r = myder;
}
```

```
class Base
{
    public:
        virtual void vfunc();
}

class Der : private Base
{
    public:
        void vfunc()override; // geçerli public'te
}
```

Private inheritance containement için (composition) bir alternatif oluşturur.

```
// Y1 Sınıfın içinde X1 Nesnesi var
class X1 // containement
{
    public:
        foo();
};

class Y1
{
    public:
        void bar()
        {
            mx.foo(); // yapabilir
        }
    private:
        X1 mx; // member object
};

// Y2 Nesnesinin içinde X2 nesnesi var
class X2  // private inheritance
{
    public:
        void foo();
};

class Y2 : private X2 // base class object
{
    void bar()
    {
        foo();
    }
};
```

Private inheritance hangi zamanlarda tercih ederim:

1) Elemanınım sınıfını sanal fonksiyonunu doğruran override edemem.

   Ancak private taban sınıfın sanal fonkisyonlarını override edebilirim.

2) Elemanınım procted bölümüne erişemem ama private taban sınıfım protected bölümüne erişebilirim.
3) Türemiş sınıf nesnesi olarak ben elemanımın türünden değiim Ancak türemiş sınıfın üye fonkisyonları için ve türemiş sınıfın friendleri için is relation ship var

   Yani :

   Der myder;

   Base *o =&myder;

4) Eğer sınıfımızın bir elemanı bir empty class türünden ise bu sınıfın 1 byte olan storage ihtiyacı aligment gereği sınıf nesnesinin için de padding bytes oluşturabilir.

   Ancak empty class private kalıtımı yaparsak derleyecielr EBO denilen optimazasyonu yaparak o 1 byte dahil etmezler.

```
class Empty
{
    void foo();
    void bar();
};

class Nec
{
    //8 byte çünkü alignment var
    Empty member;
    int max;
}

class Nec1 : private Empty
{
    //4 byte
    int max;
}

int main()
{
    std::cout << "sizeof(Empty) = " << sizeof(Empty) << "\n"; // 1 byte
    std::cout << "sizeof(Nec) = " << sizeof(Nec) << "\n"; // 8 byte
    std::cout << "sizeof(Nec1) = " << sizeof(Nec1) << "\n"; // 8 byte
}
```

Restricted Polymporhsim

```
class Base
{
    public:
        virtual void vfunc();
};

void foo(Base& baseref)
{
    baseref.vfunc();
}

class Der : private Base
{
    friend void f1();
};

void f1()
{
    Der myder;
    foo(myder);
}

void f2()
{
    Der myder;
    foo(myder);
}
```

Protected Inheritance (multi-level inheritance daha sık kullanılır)

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

class Der : protected Base
{

};

class Nec : public Der
{
    // private Base -- Der olsaydı hata olurdu ancak protected olduğu için hata
yok
    void necbar()
    {
        foo();
    };

};
```

Multiple Inheritance

Bir sınıfın birden fazla taban sınıftan tek bir türütme ile oluşturulması.

```
class Base1
{
    public:
        Base1(int)
        {
            std::cout << "Base1 ctor\n";
        }
        void foo();
};

class Base2
{
    public:
        Base2(int)
        {
            std::cout << "Base2 ctor\n";
        }
        void bar();
};

class Der : public Base1, public Base2
{
    // Yazılma sırasına göre hayata gelirler ama ctor'daki init sırasına göre
değil.
    // Kalıtımdaki belirtiği sıraya göre
    Der() : Base2(1), Base1(2) {};

}

int main()
{
    Der myder;
    //geçerli
    myder.foo();
    myder.bar();
    //geçerli
    Base1* p1 = &myder;
    Base2* p2 = &myder;
}
```

```cpp
class Base1
{
    public:
        Base1()
        {
            std::cout << "Base1 ctor\n";
        }
        int foo{};
};
class Base2
{
    public:
        Base2()
        {
            std::cout << "Base2 ctor\n";
        }
        void foo(int);
};

class Der : public Base1, public Base2
{
    void bar()
    {
        //ambiguity olur. İ sim arama aşamasında bir öncelik yok
        foo(12.3);
        foo = 10;
        // ancak böyle şekilde hata vermez
        Base1::foo(12.3);
        Base2::foo(12.3);
    }
};
// function overloading
void gfoo(Base1&);
void gfoo(Base2&);

void gbar(Base2&);
int main()
{
    Der myder;

    gbar(myder); // geçerli
    gfoo(myder); // ambiguity var

    gfoo(static_cast<Base1&>(myder));

    myder.foo() // ambiguity isim aramadan kaynaklı
    myder.Base1::foo() // hata yok

}
```

Diamond Formation

```
/*
    Stream -> InputStream & OutputStream -> InOutStream
*/
class Stream {

};
class InputStream : public Stream
{
    public:
        void read(int&);
};

class OutputStream : public Stream
{
    public:
        void write(int);
};

class InOutStream : public InputStream, public OutputStream
{

};


int main()
{
    InOutStream stream;

    int x{};
    stream.read(x);
    stream.write(12);
}
```

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

class Derx : public Base {};
class Dery: public Base {};
class MDer : public Derx, public Dery {};

int main
{
    MDer md;

    Base* p = &md; // ambiguous hatalı
    Base p1 = static_cast<Derx*>(&md);
    Base p2 = static_cast<Dery*>(&md);

    md.Derx::foo();
    md.Dery::foo();
}
```

```cpp
class EDevice
{
    public:
        bool is_open()const;
        {
            return m_flag;
        }
        void turn_on()
        {
            m_flag = true;
        }

        void turn_off()
        {
            m_flag = false;
        }
    private:
        bool m_flag{};
}
class Printer : virtual public EDevice
{
    public:
        void print()
        {
            if (is_open())
                std::cout << "printer is printing..\.n";
            else
                std::cout << "printer cannot print device is off...\n";
        }
};
class Scanner : virtual public EDevice
{
    public:
        void scan()
        {
            if (is_open())
                std::cout << "scanner is scanning..\.n";
            else
                std::cout << "scannner cannot scam device is off...\n";
        }
};

class Combo : public Scanner, public Printer {};
int main()
{
    Combo mydevice;
    // Eğer virtual public EDevice olarak tanımlamazsak böyle:
    mydevice.turn_on(); // ambiguity var
    mydevice.Printer::turn_on();
    mydevice.scan(); // return scanner cannot scan device is off
    mydevice.print(); // return printer is printing

    //Virtual public EDevice  (sVirtual inheritance)
    // Tek bir EDevice nesnesi oluşuyor virtual inheritance ile
    mydevice.turn_on(); // ambiguity yok
    mydevice.scan(); // return scanner is scanning..
    mydevice.print(); // return printer is printing
    mydevice.turn_off();
}
```

```
class Base
{
    public:
        Base (const char *p)
        {
        std::cout << "Base(const char *p) p = " << p << "\n";
        }
};

class Der1 : virtual public Base
{
    public:
        Der1(int) : Base("Der\n") {};
};

class Der2 : virtual public Base
{
    public:
        Der2(int) : Base("Der\n") {};
};

class MulDer : public Der1, public Der2
{
    public:
        MulDer() : Base("Mulder"), Der1(1), Der2(2)
        {
            // Base nesnesi init etmek zorundayız
        };
};
```

RTTI (Run Time Type Information)

- dynamic_cast
- typeid
    - type_info