

→ Döfika 12'deki örnekte move vs copy !!

### \* Function Try Block:

```
void func()
{
    try {
        // all code of func
    }
    catch (const std::exception& ex) {
    }
}
```

→ Function try block constructor da kullanılır. Çünkü normalde constructor da exception oluydu, neyse hata gelirdi.

→ Constructor içinde exception handle için

```
#include <chrono>

class Member {
public:
    Member(int x) {
        if (x % 2 == 0)
            throw std::invalid_argument{ "ctor argument should be odd!" };
    }

    ~Member()
    {
        std::cout << "Member destructor\n";
    }
};

class Nec {
public:
    Nec() = default;

    Nec(int x) try : mx{ x }
    {
    }
    catch (...) {
        std::cout << "exception caught in Nec ctor\n";
    }
    // default olarak rethrow eder
    // (.throw;) ile aynı

    ~Nec()
    {
        std::cout << "Nec destructor\n";
    }
};
```

bu x burada da kullanılabilir.

→ sağ tarafta bu şekilde

→ default olarak rethrow eder  
[(.throw;) ile aynı]

### → Private Inheritance:

- Taban sınıfın private bölümü, türetilmiş sınıfa her zaman kapalı.

public kalıtımı	
taban sınıf	türetilmiş sınıf
public	public
protected	protected
private	xxxxxxx

- Taban sınıfın public bölümü, türetilmiş sınıfın private bölümüne atılır.



```

public:
    void basefunc_public();
protected:
    void basefunc_protected();
};

```

```

class Der : private Base {
public:
    void f()
    {
        basefunc_public();
        basefunc_protected();
    }
};

```

Base'den gelenler bizim private bölümümüzde, class içi erişilir ancak dışarıya erişilemez.

```

int main()
{
    Der myder;
}

```

myder.baseptr - public - protected } → senden notları

```

class Base {
};

```

```

class Der : private Base {
};

```

→ "is A" relationship clientler için yok

```

int main()
{
    Der myder;
    Base* baseptr = &myder;
    Base& baseref = myder;
}

```

private inheritance'da artık derived class, base ptr, base ref kullanılır. yer de kullanılır. implicit conversion yok X. public olso olurdu.

```

class Base {
};

class Der : private Base {
    friend void g();
    void func()
    {
        Der myder;
        Base* baseptr = &myder;
    }
};

void g()
{
    Der myder;
    Base* baseptr = &myder;
}

int main()
{
}

```

→ Fakat "is a" relationship'ı member fonksiyonlar ve friend deklarasyon ne "class içi"nde sağlayabiliriz.

- Türetilmiş sınıf nesnesi içinde taban sınıf nesnesi var
- türetilmiş sınıfın implementasyonu taban sınıfın public ve taban sınıfın protected işlevlerini kullanabiliyor
- taban sınıfın private bölümüne erişemiyorduk

taban sınıfın public ve taban sınıfın protected öğeleri türetilmiş sınıfın private bölümüne ekleniyor  
türetilmiş sınıfın client'ları taban sınıfın public öğelerini görmeyecekler

private inheritance (çok büyük çoğunlukla) containment yoluyla composition'a bir alternatif!!!

is a relationship yok

Containment da yok!

- private kalıtımında taban sınıfın sanal fonksiyonlarını override edebiliyoruz!!!
- private kalıtımında taban sınıfın protected fonksiyonlarını çağırabiliriz!!!
- Türetilmiş sınıfın üye fonksiyonlarında ve türetilmiş sınıfın friend'lik verdiği fonksiyonlarda is-a relationship söz konusu



## \* Composition By Composition Örneği:

```
class A {  
public:  
    void f1();  
    void f2();  
};  
  
class Nec1 {  
private:  
    A ax;  
};  
  
class Nec2 : private A {  
};
```

Bu iki gösterim aynı anlama geliyor.

• Her ikisinin public bölümü, birim private bölümünde kendi public interface'imize, aynı şekilde bile, farklılar tanımlayabiliriz.

• Alternatif olarak "using Base: fuction" ile birim public kısmımıza erişilebiliriz.

## - Protected Inheritance:

- Temel sınıfın public bölümü, türetilen sınıfın protected bölüme elverişli.

## \* Templates:

→ Derleyiciye kod yazdırmak = Template Code

```
function template (fonksiyon şablonu)  
class template (sınıf şablonu)  
variable template  
alias template (tür eş isim şablonu)  
variadic template
```

→ modern Cpp önerisi  
→ modern Cpp ile geldi.

```
template <class T>  
template <typename T>
```

Aynı anlama geliyor.

- a) type parameter (template tür parametresi)
- b) non-type parameter (template sabit parametresi)
- c) template parameter (template parametresi)