

*LTI'den devam:

- Dynamic Cast Devam:

- Dynamic cast, referansla yapılmamalı. Çünkü nullptr var ama null ref yok !!
- Eğer downcast, başlangıçta bad cast exception verir.

- Type ID Operatorü:

- Type id operatörünün ürettiği değer ismi typeid olan bir simit tarafından const referanstır.
(std::name_type)

- typeid türünden class'ı bir deklare ediyorsanız. Çünkü default constructor'ı yok. Copy ctor'u da deleted.
- Tek yolu → type id kullanmak.

```
int ival{};
```

```
const std::type_info& x = typeid(ival);
```

- Type id iki şekilde kullanılır:
1. operand bir tür / class / primitive olabilir.
 2. operand bir expression olabilir.

*Hatırlatma: Un-evaluated Context:

- Derleyicinin işlem kodu üretmediği bağlam
- Bunlar sizeof, decltype, typeid

- Base class polimorfik değilse, typeid hep base class'ı döner. Eğer polimorfikse, runtime'da typeid değişir de.

```
int main()
{
    for (int i = 0; i < 100; ++i) {
        Car* carptr = create_random_car();
        std::cout << *carptr;
        std::cout << typeid(*carptr).name() << "\n";
    }
}
```

```
void car_game(Car& cr)
{
    std::cout << cr << "\n";
    cr.start();
    cr.run();

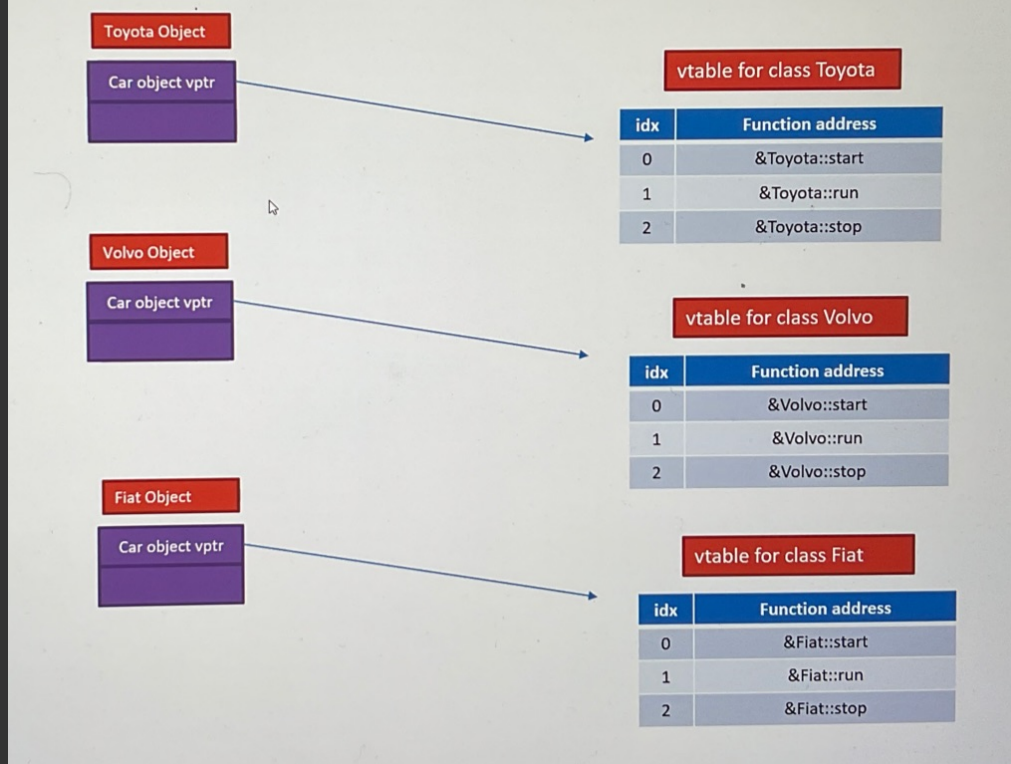
    if (typeid(cr) == typeid(Volvo)) {
        static_cast<Volvo&>(cr).open_sunroof();
    }

    cr.stop();
}
```

→ ya bir ref / ya da default pointer operand olarak verilebilir.

→ static cast yeteri yeterli, artık bu if'in içerisinde volvo olduğu garanti

```
int main()
{
    for (int i = 0; i < 100; ++i) {
        Car* p = create_random_car();
        car_game(&cr:*p);
        getchar();
        delete p;
    }
}
```



* Multiple Inheritance:

- Birden fazla taban sınıf ile derive edilen sınıflar.

```

1
2 class A {
3     ...
4 };
5
6
7 class B {
8     ...
9 };
10
11 class C : public A, public B {
12     ...
13 };
  
```

eğer yoklarsa private olacaktır

→ Her C bir A'dır, her C bir B'dir.

→ Constructor çağrılma sırası, ilkete belirtilen sıra, önce A sonra B

```

class C : public A, public B {
public:
    C() : B(), A() {
    }
};
  
```

Önemli olan bu sıra değil

Önemli olan bu sıra

→ A ve B complete type olmalı


```

class A {
public:
    void foo(int, int);
};

class B {
public:
    virtual foo(int);
};

class C : public A, public B {
public:
};

int main()
{
    C cx;

    cx.foo(1, 2);
    // cx.foo(1);
}

```

void A::foo(int, int)
Search Online
Function definition for 'foo' not found.
Show potential fixes (Alt+Enter or Ctrl+.)

→ Multiple inheritance, isim arama sırası yoktur.
Inherit edilen tüm classlara bakar.

→ Bu yüzden Her base class'ın aynı isim taşıyan veritabanı
Sarıp olmalıdır.

→ Bu **ambigüite'nin çözümü**, isim qualified name. diye
kullanmak.

```

int main()
{
    C cx;

    cx.A::foo(1, 2);
    cx.B::foo(1);
}

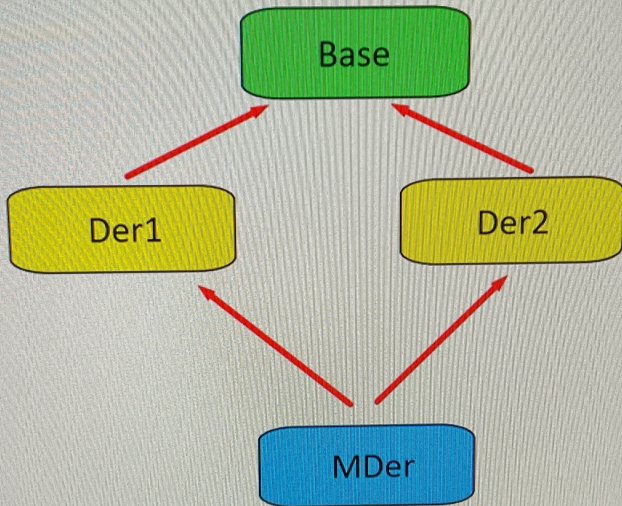
```

```

void func()
{
    B::foo(12);
    A::foo(3, 5);
    static_cast<A*>(this)->foo(3, 4);
    static_cast<B*>(this)->foo(12);
}

```

* Diamond Formation:



→ Burada MDer, Her iki Base inherit ettiği oldu

→ Çözümü, virtual inheritance

```

class Der1 : virtual public Base {
};

class Der2 : virtual public Base {
};

class MDer : public Der1, public Der2 {
};

```

Tek bir
base
class olması
garantilendi

- İostream sınıfı buna
bir örnek.
- gerekirse ac bir başka örnek

→ Multi level inheritance **virtualized** kendi base class'ımızı init edebiliriz. → **Indirect base class init**
Virtual inheritance'da ise **teke birim o ilk base'ı init edebiliriz.** → **Direct base class init**