*RTTI'dan devam:

- **Dynamic Cast Devam:**
  - Dynamic cast, referansla yapılmamalı. Çünkü nullptr var ama null ref yok !!
  - Eğer downcast, başarısız olursa bad cast exception verir.

- **Type ID operatörü:**
  - Type id operatörünün ürettiği değer resmi type_info olan bir sınıf türünden const referanstır. (std namespace)
  - type_info türünden class'ı biz oluşturamayız. 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. operandı bir tür / class / primitive olabilir.
2. operand bir expression olabilir.

***Hatırlatma: Unevaluated Context:**
- Derleyicinin işlem kodu üretmediği bağlam
Bunlar sizeof, decltype, type_id

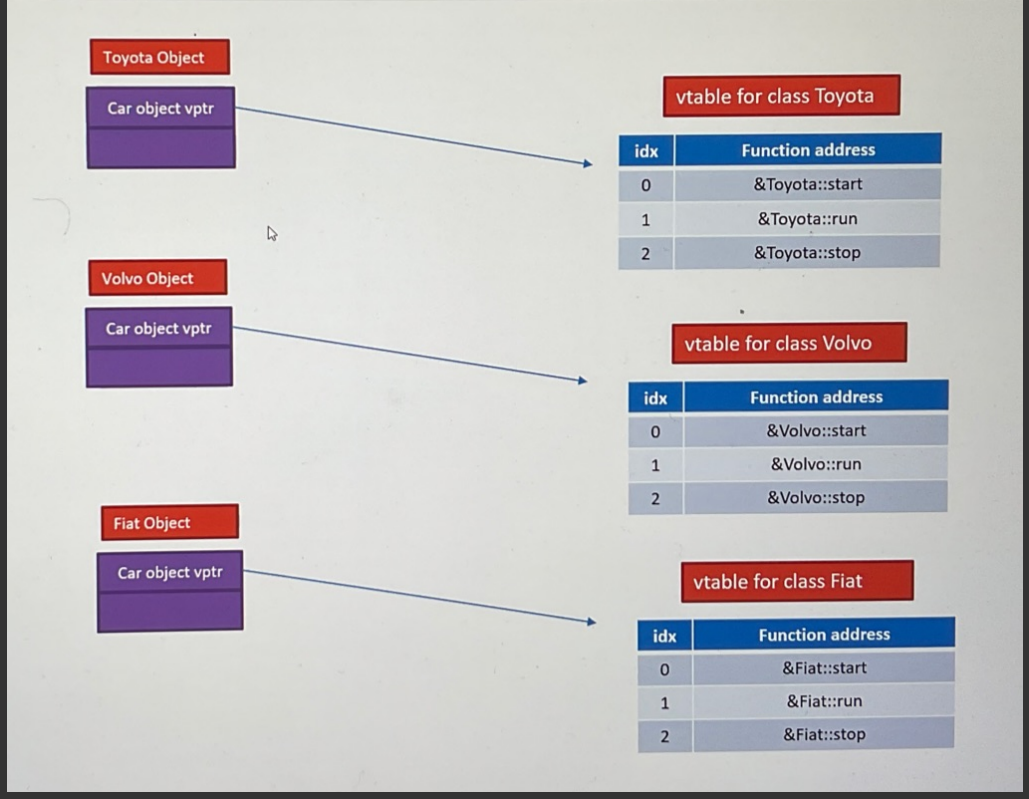
- Base class polimorfik değilse, type id hep base class'ı döner. Eğer polimorfikse, runtime da type id değişiyor olabilir.

```
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();
}

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

yan bir ref / ya da dereft pointer operand olarak verilir.

→ static cast yeterli çünkü, artık bu ifadenin içerisinde volvo olduğu biliniyor



* Multiple Inheritance:

- Birden fazla taban sınıf ile türetilen sınıflar.

```
class A {
};

class B {
};

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

eğer yazılmasa private olacaktı

→ Her C bir A'dır, her C bir B'dir.

→ Constructor çağrılma sırası, isimlerde 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

```cpp
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);
}
```

→ Ambiguity.

→ Multiple inheritance'da, isim arama sırası yoktur. Inherit edilen tüm classlara bakılır.

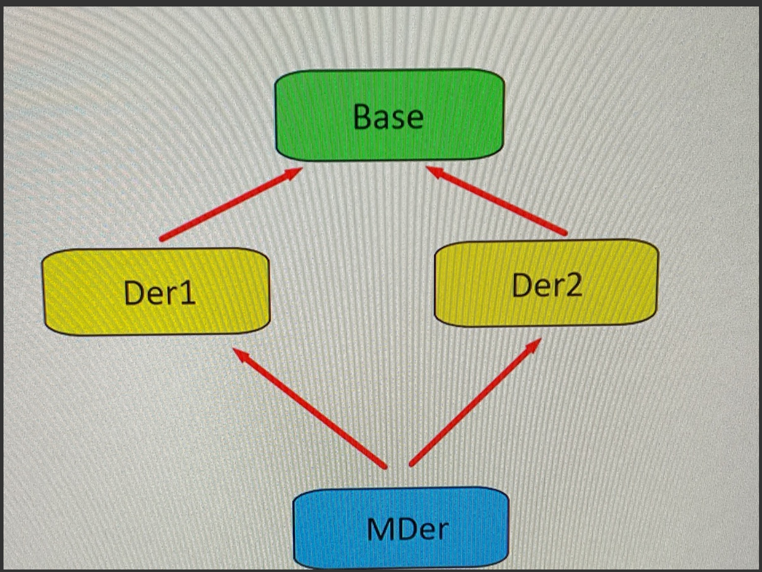
→ Bu yüzden iki base class'ın aynı ismi taşıyan verilikleri sahip olmamalı

→ Bu ambiguity'nin çözümü, ismi qualified name olarak kullanmak.

```cpp
int main()
{
    C cx;

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

```cpp
    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, iki kez Base inherit etmiş oldu

→ Çözümü, virtual inheritance

```cpp
class Der1 : virtual public Base {

};

class Der2 : virtual public Base {

};

class MDer : public Der1, public Der2{

};
```

→ Tek bir base class olması garantilendi

- iostream sınıfı buna bir örnek.
- gerekirse aç bir daha incele

→ Multi level inheritance'da yalnızca kendi base class'ımızı init edebiliriz. → Indirect base class init

Virtual inheritance'da ise tepe birim o ilk base'i init edebilirim. → Direct base class init