2023.09.27

RTTI (Run Time Type Information)

- dynamic_cast
- typeid
    - type_info

upcasting : türemiş sınıftan taban sınıfa yapılmasına denir.

downcasting : taban sınıftan türemiş sınıfa yapılmasına denir.

Dynamic_cast

dynamic_cast<target tope>(variable)

Mercedes *p = dynamic_cast<Mercedes*>(car_ptr)

Mercedes &p = dynamic_cast<Mercedes&>(car_ref)

Eğer cast başarılı olmazsa p nullptr olur

```
class Base
{
    public:
        virtual ~Base() = default;
};

class Der : public Base
{

};

void foo(Base* baseptr)
{
    /*
        Dynamic cast olabilmesi için base sınıfı polymorfik olması gerekir
        yani en az bir tane virtual fonksiyonu olması gerekiyor.
    */
    Der* derptr = dynamic_cast<Der*>(baseptr);

};
```

Dynamic Cast Usage:

```
void car_game(Car* carptr)
{

    Tesla *tp = dynamic_cast<Tesla*>(carptr)

    if (tp)
    {
        tp->autopilot();
    }
    // böyle yapmak daha havalı
    if (Tesla *tp = dynamic_cast<Tesla*>(carptr))
    {
        tp->autopilot();
    }
}

int main()
{
    for (int i = 0; i < 1000; ++i)
    {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}
```

```
// Reference semantiği ile (Not Null Ref yok)
void car_game(Car* carptr)
{
    try
    {
        Tesla &tp = dynamic_cast<Tesla&>(*carptr); // down_casting
        tr.autopilot();
    }

    catch (const std::exception&ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
    /*
        Dynamic_cast kullanımında hedef tür referans türü ise
        down-casting güvenli bir şekilde yapılamıyor ise standart
        kütüphanemizin std::bad_cast sınıfı türünden exception throw edilir.

    */

}
int main()
{
    for (int i = 0; i < 1000; ++i)
    {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}
```

Typeid

```
/*
    typeid bir operatordür

    typeid(*ptr);
    typeid(x);
    typeid(10);
    type(Tesla);
    type(int);

    type info sınıf türünden const bir nesneye referans

    her ayrı (distinct) tür için bir type_info nesnesi var
*/
```

```
#include <typeinfo>
int main()
{
    int x = 10;

    typeid(x); // bu ifadenin türü typeinfo
    auto y = typeid(x); // syntax hatası çünkü typeid copy ctor delete edilmiş.

    //legal
    const auto &r = typeid(x);
    auto &r2 = typeid(x);
}
```

Unevaluated Context

```
int main()
{
    int x = 10;
    auto sz = sizeof(x++);
    decltype(x++) y = x; // unevaluated context
    std::cout << "x = " << x << "\n";
    // x = 10 çünkü sizeof unevaluated context

    int a[10]{};
    auto s = a[30] // tanımsız davranış
    auto sz = sizeof(a[30]); // tanımsız davranış değil


    int *ptr{};
    auto x = *ptr; // tanımsız davranış
    auto szz = sizeof(*ptr); // tanımsız davranış değil

    int x  = 12;
    const auto&r = typeid(++x); // unevaluated context
    std::cout << "x = " << x << "\n"; // x = 12

    int *ptr{};
    const auto&p = typeid(*ptr); // tanımsız davranış değil

}
```

```cpp
class Nec
{

};

int main()
{
    Nec mynec;
    // derleyiciden derleyiciye çıktı değişir o yüzden karşılaştırma yapmamak
lazım
    std::cout << typeid(Nec).name() << "\n";

    typeid(mynec).operator==(typeid(int)) // false
    typeid(mynec).operator==(typeid(Nec)) // true
}
```

```cpp
class Base
{
    virtual ~Base(){};
};

class Der : public Base {};

int main()
{
    Der myder;
    Base *ptr = &myder;

    /*
    Eğer Base polymorfik bir class ise 1 döner ama değilse 0 döner
    Burada Base polymorfik o yüzden 1 döner
    */
    std::cout << (typeid(*ptr) == typeid(Der)) << "\n";
}
```

```cpp
void car_game(Car *ptr)
{
    if (typeid(*ptr) == typeid(Audi))
    {
        std::cout << "This a Audi\n";
    }

    if (typeid(*ptr) == typeid(Tesla)) // run-time'da yapılır
    {
        static_cast<Tesla *>(ptr)->autopilot();
        // dynamic_cast'deki gibi Teslanın alt sınıfları bu if'e girmez
    }
}
int main()
{
    for (int i = 0; i < 1000; ++i)
    {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}
```

static_cast in class

```
class Base{};
class Der : public Base {};
class Nec {};

int main()
{
    Nec mynec;

    //legal değil
    Der *derptr = static_cast<Der*>(&mynec);

    // legal
    Base mybase;
    Der *derptr = static_cast<Der*>(&mybase);
}
```

Note: Virtual Function Table'da typeid için yer ayrılıyor

1) **RTTI hiç kullanılmazsa ilave bir maliyet var mı?**

Var çünkü run-time'da her türlü typeid nesneleri oluşturuluyor ama derleyiciye bağlı olarak biz run-time type nesnesini kullanmacağım dersek RTTI disable edersek bize maliyet oluşturmadan kapatırız

2) **Büyük bir hiyerarşi var ve typeid ya da dynamic_cast kullancaksın ikiside işimizi görüyor. Hangisinin maliyeti düşüktür?**

Typeid'nin maliyeti daha düşük çünkü typeid classların türemiş sınıflarını kontrol etmez ancak dynamic_cast türemiş sınıfları kontrol eder.

typid(*ptr)

polimorfik tür söz konusunda olduğunda eğer typeid operatörünün operandı olan ifade dereference edilmiş pointer ifadesi ise pointer değerinin nullptr olması durumunda:

std::bad_typeid sınıfı türünden exception throw edilir. bad_typeid sınıfı std::exception sınıfından kalıtım yoluyle elde edilir.

```
void car_game(Car *ptr)
{
    try
    {
        if (typeid(*ptr) == typeid(Tesla))
        {
            static_cast<Tesla *>(ptr)->autopilot();
        }
    }
    catch (const std::bad_typeid& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
        // nullptr
    }
}

int main()
{
    Car* ptr{};
    car_game(ptr);
    // ptr null olduğu için expection verir
}
```

Taban sınıfların dtor'ları ya public virtual ya da protected non virtual olmalı. Ancak NVI kullanarakta tanımlayabiliriz.

Template Method

```
// template method
class Figther
{
    virtual void attack();
    public:
        void fight()
        {
            attack(); // savaşçıyı göre customize edilecek
        }

};
```

Non Virtual Interface ile:

- Taban sınıf kendi kontrollerini dayatabiliyor
- Ortak kodları bi yere toplayabiliyoruz
- implemantasyon ile interfacesi ayırmış oluyoruz

```
class Figther
{
    public:
        void attack()
        {
            //invariant'lar
            attack_impl();
            //invariant'lar
        };
    private:
        virtual void attack_impl() = 0;
}
```