lessons 21 - 25

LESSON 21

extern C :

c de function/operator overloading yok. C++ da derleyici çok farklı işler yapıo. Problem isim bulma aslında. derleyiciler farklı dekore ediyor bildirimleri (C farklı C++ farklı). Bu işlemi C kütüphaneleri için yaparız.

Örneğin sen c++ da **void foo();** diye bir fonksiyon yazdın. Bu fonksiyonu derleyici şuna çevirebilir. **void __foo@v(void);** Bunu yapmasının sebebi overloading var.

Örneğin **void foo(int);** diye ikinci fonksiyon daha yazdın. **void __foo@i(int);** derleyici buna benzer bi declarationa çeviriyor. Çağrılarda bunlar gibi oluyor.

AMA C de **void foo();** üzerinde C derleyicisinin bir değişiklik yapma durumu yok yapsa bile C tarzında yapıyor bu dekarasyonu. Sen gidip Bu C derleyicisinin işleyip çevirip linkera bağladığı kodu C++

projesinde çağırırsan C++ projesi C++ fonksiyonu arar yani **__foo@v** bunu arar ama olması gereken sadece **foo(veya C tarzındaki hali).**

```
extern "C"  void foo(void);    // 1. kullanım şekli , her bir pr

extern "C"{    // Bu daha iyi kullanımı !!
   // artık burası için C++ derleyicisi, burayı C nin derlediği
}

// problem şu pure C kodunda bunu yazmamak lazım dolayısıyla şu
// predefined symbolic constants:  nedir ? :  __FILE__  ,  __TI
// __STDC__   : c      ,     __cplusplus     : c++   (bu yoksa

#ifdef __cplusplus
extern "C"{
#endif
```

```
void foo(void);

#ifdef __cplusplus
}
#endif
```

NOT: **narrowing conversion:** double x = 3.5; int e dönüşürken daralması,bilgi kayıbı olması ve 3 kalması.

association (dernek,ortaklık) : 2 sınıfın iş birliği yapması beraber işi yapması. **has-a relationship** ex:

```
class School {
public:
    void enrollStudent(const class Student& student);
};

class Student {
public:
    void joinSchool(const School& school) {   // School 'u kulla
        school.enrollStudent(*this);
    }
};
```

aggregation(toplama) : association içinde , ama burda ki fark burda bir sahiplenme durumu var. biri master biri slave gibi. AMA bu objeler ayrı ayrıykende hayyata kalabilirler. **Birliktelik kurmasalarda ayrı ayrı var olabilirler.** globalde yaratılmış ayrı ayrı 2 objenin birleştirildiğini düşün. **has-a relationship**

Composition(birleştirme) **:** = aggregation. üzerine birde bir nesnenin hayatı bitince diğerininde bitiyorsa **Composition** olur**. (ömürsel birliktelik) , "contains-a"** veya **"part-of" , has-a relationship**

```
class engine{

}

class car{          // kalıtımı buraya yapınca ilerde onun adı:
// copy/move constractorlarda member objecti de yüklemeyi unutma
private:
   engine e;       // has-a relationship    , her araba motor de
   engine e2;
   engine e3;      // unutma önce e sonra e2 sonra e3 hayata gel
}
```

reference qualifiers: modern c++ ile. Sınıf nesnesinin value katogorisi veya const luk durumuna bağlı. sınıfların üye işlevlerinin çağrılabilirliğini kontrol etmek ve belirli referans türlerinde bu işlevleri sınırlamak için kullanılır. Bu, özellikle modern C++'ın getirdiği bazı özelliklerle birlikte, performans artışı ve dilin sunduğu olanakları daha etkili bir şekilde kullanma amacını taşır.

```
class my_class{
   void set(int);
}

int main()
{
   my_class{}.set(12);   // sıkıntı yok derleyici için ama LOGIC

   my_class m;
   my_class{} = m;       //  sıkıntı yok derleyici için ama LOGI
}

// *** yukardaki istenmeyen durumu önleyelim *****************

class my_class{
   void set(int)&;   // bunu sadece artık L value sınıflar çağır
```

```
    void bar(int)&&;  // bunu sadece artık R value sınıflar çağır
    // special member functions içinde geçerli

    // void set(int)&&;   // çağırsan yukardakiyle function over
    // Not: "void set(int); " yapmıcan sentaks hatası olur. refer
    // ama void set(int)const&; ve void set(int)const&&; bunlarda
    // my_class& operator=(my_class&)& = default;   // c++20 de g
}

int main()
{
    my_class{}.set(12);   // hata oldu
    my_class{}.bar(18);   // hata yok

    my_class m
    m.set(15);      // tabiki burda hata yok
    m.bar(86);      // ARTIK HATA
    std::move(m).bar(78);   // hata yok, move R a dönüştürür
}
```

Ne işe yarar ? .

- L R value referansa bağlı çağırma ve atamaları yönetmek için
- @rf897 R ve L value objelere yapılmasını istediğimiz işlemleri ayırmak. (ileri düzey)

DERS 22

namespaces: aynı isimli 2 class veya değişken/func olamaz. namespace kullan. Böyle problemlere global namespace pollution problem denir. **amaç**: isimlerin çakışmasını önlemek.

kendinize özel global scope yaratmak amaç.

- iki farklı namespace birleştirilir. namespace test{} namespace test{} // bu ikisi aynı namespace

- global alanda tanımlanır. namespace içinde namespace olur. Dosyalama gibi düşün.
- **unqualified**(nitelendirilmeyen) ismin, namespace içinde bulunması.

1) **using declaration**: Kendi scopu var, içinde yazıldığı. ex: using std::cout;

2) **using namespace directive**: Kendi scopu var, içinde yazıldığı. ex: using namespace std; , using enum test_enum;(c++20) , using namespace std , my_namespace;

```
namespace my_test{
   int x;
}
int main(){
    int x = 6;
    using my_test::x;    // SENTAKS HATASI, bu bildirim direk i
}
```

```
namespace my_test{
   int x;
}
int main(){
    using namespace my_test;    // BURDA HATA YOK - bildirim gö
    // tanımlanan namespace değişkenlerine tabiki ulaşamazsın(g
    int x = 6;                  // aşağıda artık bu x geçerli t

}
```

3) **ADL(argument dependent lookup)**: bir fonksiyona nitelenmemiş isimle çağrı yapıldığında, fonksiyona gönderilen argümanlardan biri namespace içinde bir tür ise, söz konusu isim o namespace içinde **de** aranır. Friend bildirimindede benzer mantık var.

```
namespace my_test{
   class my_class{}
   int baz(my_class);   // BU ADL ile bulunabilinir oldu , NOT:
```

```
   int bar();
}
int main(){

    bar(); // SENTAKS HATASI,    my_test::bar()   yapman lazımdı

    my_test::my_class obj;   // bu zaten böyle olmalı OK.
    baz(obj); // HATA YOK - ADL devrede. my_test::baz(obj) gibi
}
```

```
namespace other_test_namespace = my_test; // artık other_test_na

//artık kullanılabilinir
other_test_namespace ::bar();   // vs..
using namespace other_test_namespace ;


//örnek;
namespace my_test{
  namespace my_test2{
         namespace my_test3{
          // ...
      void foo();
        }
  }

  namespace view = my_test2::my_test3;  // ****
}

int main()
{
     my_test::view::foo(); // my_test::my_test2::my_test3::foo()
}
```

```
namespace my_test{
  namespace my_test2{
         namespace my_test3{
          int x;
        }
  }
}
// = artık aşağıdaki gibide olur
namespace my_test::my_test2::my_test3
          int x;
}
```

```
namespace my_test{
  inline namespace my_test2{     // bu inline çok mantıklı olma
          int x;
  }
  // artık inline yapınca , x burdada var oldu, my_test2 ekstra
}
int main()
{
     my_test::x; // HATA YOK inline sayesinde
}
```

DERS 23

enum ile operator overloading örneği

```
enum class weeks{sunday,monday.....};

weeks& operator++(weeks& wd)  // bu ++wd;
{
    using enum weeks;
    return wd = wd == saturday ? sunday :
             static_cast<weeks>()(static_cast<int>(wd) + 1);
```

```
}

weeks operator++(weeks& wd, int) // bu wd++; için , unutma 2. pa
{
    weeks temp{wd};
    wd++;
    return temp;
}

std::ostream& operator<<(std::ostream, const weeks& wd)
{
    static const char* const pweeks = {"sunday","monday" ...}
    return os << pweeks[static_cast<int>(wd)];
}

int main()
{
    weeks wd{weeks::sunday};

    for(int i = 0; i < 100 ; i++)
      std::cout << wd++ << '\n';  // weeks operator++(weeks& wd,

     std::cout << ++wd << '\n';   // weeks& operator++(weeks& wd
}
```

nested class:

```
class my_class{
public:
   class test{    // bu defination yani tek başınayken tür gibi

   }
private:
```

```
   test x;    // obje bu, şimdi yer tuttu
}
```

```
class my_class{
private:
   class test{
      void bar();
   }
public:
   static test foo();
}

int main(){
        my_class::test x = my_class::foo();  // HATAAA!! my_clas
        auto x = my_class::foo();            // legal: bak acces
        x.bar();                             // legal
}
```

pimpl idiom , handle-body idiom , cheshire cat , compiler firewall , opaque pointer

pimpl idiom: yukardakiler aynı? sınıfın private bölümünü client kodlardan gizlemeye yönelik. headera bakanda görmesin yani. sebebi copile süresi hızlansın çok header içinde header yazmayalım: örnek;

```
#include "A.h"
#include "B.h"
#include "C.h"

class my_class{
private:
  A a;     // BU elemanları nasıl gizleriz? gizlersek eğer yukar
  B b;     // clientlar yani kütüphaneyi kullanmak isteyenler bi
  C c;
}
```

çözüm; ÇOK İYİ <3 , bu işin maliyeti sürekli new delete işi !! smartpointer daha mantıklı, ilerde görcez.

```
// header
class my_class{
public:
  my_class();
  ~my_class();
private:
   class pimpl;
   pimple * mp;
}

//.cpp
class my_class::pimpl  // struct?
{
 A a;
 B b;
 C c;
}

my_class::my_class(): mp{new pimpl{}}
{

}

my_class::~my_class() {
    delete mp;
}
```

INHERİTANCE(MİRAS , KALITIM)

SOLID. OOP

is-a relationship , sınıf hiyararşisi (iç içe kalıtımlar) , multiple interitance

parent class = super class = **base class(c++)**

child class = subclass = **derived class(c++) → türetilmiş olan sınıf bu, yani başvurular bunda**

- public interitance ⇒ OOP diğer diller ile ortak kısım
- private inheritance
- protected inheritance

complete type ihtiyacımız var. yani class defination değil kendisi olmalı.

NOTE: bir fonksiyon çağrısı yapınca derived classdan. Önce namelookup kendi CLASS içindeki fonksiyonları tarar, bulursa direk çıkar (**function overloading olmaz base class ile**). Bu işi public private bile değiştirmez. En son base class içindeki namelookup bakılır.

```
class base{
private:
// burdaki private alanına derived child class ulaşamaz , buras
protected:  // kalıtım için var
// burdaki protected alanına derived child class ulaşabilir
public:  // herkese açık
  void bar();
}
          //   base object
class der: public base{  // veya private base, veya protected ba

   // bar()   -> direk burda kullanılır

   // base bs;   -> member object olurdu has-a relationship

   void bar();  // function overloading olmaz bu iki classın sc
}

// struct da default kalıtım public

int main(){
```

```
   der d;
   d.bar();

   base *p = &d;
   base& cr = d;   // ikiside legal, C den anladın o işi, ortak

// yani "der"i hayata getirince "base"de hayata gelio, yani "siz

   base c = d; // bunu YAPMA iyi değil. object slicing
}
```

DERS 24

```
class base{
    public:
      void bar(double);
}

class der: public base{

 private:
   void bar(int);
}

int main(){

   der d;

// bu inheritance olmasa , private içinde func overloading olsa
   d.bar(3.4);  // HATA namelookup baktı private bu dedi HATA ve
// d.base::bar(3.4);
}
```

YUKARDAKİ HATANIN ÇÖZÜMÜ:

```
class base{
    public:
      void bar(double);
}

class der: public base{
   using base::bar;    // ** enjekte ettik, bildirmiş gibi oldul
 private:
   void bar(int);
}

int main(){
   der d;
   d.bar(3.4);  // ÇALIŞIR, base::bar(3.4); oldu
}
```

Multilevel Inheritance

```
class A{

}
class B: public A{

}
class C: public B{

}
// C için B -> direct base class
// C için A -> indirect base class
```

```
class base{
    public:
      base(int); // constractor
}
```

```
class der: public base{
}
int main(){
   der d;  // sentaks hatası !! der in ctorunu derleyici delete
}
```

```
class base{
    public:
      base(){ cout<<"base c-tor"; }
    // base(int , int)             // @784 olsaydı
    ~base(){ cout<<"base d-tor"; }
}
class der: public base{
//  der():base(5,6){        // @784 olsaydı
      der(){   // burda gizli ":base(){"    var aslında derleyi
        cout<< "der c-tor";
        }
    ~der(){ cout<< "der d-tor"; }
}
int main(){
   der d;
}
/**   OUT:
    base c-tor
    der c-tor
    der d-tor
    base d-tor
*/
```

```
// special member functionları bırak hep derleyici yazsın (copy
int main(){
   der d;
   der d2(d); // BURDA der için copy c-tor çağrılır ama base iç
   der d3 = std::move(d);  // BURDA der için move c-tor çağrılı
```

```
   d2 = d3;            // BURDA derleyici base i de der i de cop
   d2 = std::move(d3); // BURDA derleyici base i de der i de mov
}

der(const der& other):base(other){}          // @xd485   ( other
```

```
class car{
    public:
      void start(){} //@xxdf123
}

class audi:public car{
    public:
      void start(){}
}

class reno:public car{
    public:
      void start(){}
}

void car_game(car * ptr){
   ptr->start();
}

int main(){
      reno r;
      audi a;

      car_game(&r);   // car yani @xxdf123 çağrılır
      car_game(&a);   // car yani @xxdf123 çağrılır
}

// virtual dispatch , run time polymorphism ********************
class car{   // (polymorphic  class)
```

```
    public:
      virtual void start(){} //@xxdf123
}

class audi:public car{
    public:
      void start(){}    //@auid123
}

class reno:public car{
    public:
      void start(){}    //@reno123
}

void car_game(car * ptr){
   ptr->start();
}

int main(){
      reno r;
      audi a;

      car_game(&r);   //   şimdi @reno123 çağrılır
      car_game(&a);   //   şimdi @auid123 çağrılır
}
```

static binding - early binding : derleme zamanında yani compile time da hangi fonksiyonun çağrılacağı demek

dynamic binding - late binding : run time zamanında hangi fonksiyonun çağrılacağı demek (**yukardaki örnek !**) derleyici ilk ürettiği kodda hangi fonksiyon çağracağını bilmio yani run time da anlıo!

DERS 25

Taban Sınıf Fonksiyonlarının Özellikleri:

- 1-türemiş sınıflara hem interface hem impementasyon veren fonksiyonlar

- 2-türemiş sınıflara hem interface hem default impementasyon veren fonksiyonlar. **(polymorphic class)** - yukardaki örnek **virtual function. ex : "**virtual void start(){ }"
- 3-türemiş sınıflara sadece interface veren, impementasyon vermeyen fonksiyonlar. **(abstract class) - ex: "**virtual void start() = 0;" **pure virtual function.** Türemiş sınıf bu startı yazmaz ise syntax hatası.

function overriding: türemiş sınıfın bu interface i alarak , taban sınıf implemantasyonunu istemeyip kendi implemantasonunu yapmasına**(2.madde)** veya boş interfacei alıp yazmasına**(3.madde)** denir !

```
class airplane{
public:
   void stop(){   ..... }  // madde 1 ,
   virtual void fly(){ ... } // madde 2 , türemiş sınıf dilerse
   virtual void fly2(){ ... } // private virtual functionlarda
   virtual void land() = 0; // madde 3 , kesinlikle override edi
};

class airbus : public airplane{
public:
  void fly()override{}    // override , burda ki yazılan  "overr

  // int fly(); // ERROR, fly yukarda virtual olduğu için artık

  int stop(); // virtual değil hata yok serbest

  virtual void fly2(){ ... }  // yeniden override edilebilinir
};

class airline:public airbus{
public:
    void fly2()override{}   // airplanedan değil airbus dan aldı
}
```

contextual keywords:

override : derleyiciye kontrol et, override yapıyorum diyoruz! kesin kullan derleyici kontrol edio.

final: madde 2 için. ex: void fly() override final { ... } artık overrida yapmaya kapalı. diğer classlar yapamaz

```
class base{
public:
    virtual void foo(){std::cout << "base";}
};

class der:public base{
private:
     void foo(){std::cout << "der";}
};

 void test(base*f){
     f->foo();    // ÇAĞIRIR BAŞARILI !! -> der
 }

int main() {
    der d;
    d.foo(); // SENTAKS HATASI, private çünkü !!
    test(&d);
}
/** OUTPUT:
der             // bak private olmasına rağman !!! NEDEN? çünkü
*/
```

isim arama statik türe bağlı yapılır. erişim kontrolüde öyle. ama virtual fonksiyonlar değil.

not: global func / static func / static member func ve special member functionlar **virtual olamaz.** Ama constractorları virtual yapmak bir ihtiyaç olabilir! Bunun için çeşitli patternler bulunmaktadır**:**

virtual constructor idiom / clone idiom

```
class car{
public:
    virtual void start() = 0;
    virtual void stop() = 0;

    virtual car* clone() = 0;
};

class volvo: public car{
public:
   void start() override{}
   void stop() override{}

   car* clone() override{  return new volvo(*this);   }
};

class bmw: public car{
public:
   void start() override{}
   void stop() override{}

   car* clone() override{  return new bmw(*this);   }
};

/** bu fonksiyon, çalışma zamanında , kullanıcı tarafından seçil
void car_game(car* p){

   car* px = p->clone();   // CLONE PATTERN

   px->start();
}
```

"**virtual dispatch**" (sanal yönlendirme)(yukardaki örnek),Temel amacı, bir sınıfın türetilmiş sınıfları arasında dinamik bağlama (dynamic binding) sağlamaktır. Taban

sınıf hayata geldiğinde **derived class hayatta değil.**

Bir başka sık kullanılan pattern; Taban sınıf üye fonksiyonlar içinde **virtual dispatch** uygulanmaz. aşağıdaki örnekde bu nedenle her zaman **@gth485** çağrılır

```
class car{
public:
   car(){
       start();    //@gth485
    }
    virtual void start(){
         //@gth485
    }
};
```