lessons 26 - 32

LESSON 26

```
class base{
public:
};
class der: public base{
public:
};

void main( ){

   base* px = new der;

   delete px;   // UB , BÜYÜK HATA, base destructor çağrıldı , d
}

// ** ÇÖZÜM **************

class base{
public:
    virtual ~base() {}  //destructor virtual yapılmalı bu durumu
/** diğer çözüm:
    protected:
    ~base() {}    // bu çözümde aşağıdaki main de ki kod(delete
*/
};
class der: public base{
public:
    ~der() {}
};
```

```
void main( ){

   base* px = new der;

   delete px;   // işte şimdi önce ~der()  sonra ~base çağrıldı
}
```

Yukardaki çözüm için **Herb Sutter**'ın önemli ilkesi: polimorfik sınıfların destructorı ya public virtual(yukardaki gibi) yada **protected non virtual** olmalı !!!!!!

Ancak bu problem çok seyrek çıkar. Çünkü smart pointerlar hallediyor. !!

Global fonksiyonlar virtual olamaz. Ama sanki virtual mış gibi bir pattern yapılabilinir. EX:

```
class car{
public:
    virtual ~car() {}
    virtual void start() = 0;
    virtual void stop() = 0;
    virtual car* clone() = 0;
    virtual void print(std::ostream& os)const = 0;
};

class volvo: public car{
public:
   void start() override{}
   void stop() override{}
   car* clone() override{  return new volvo(*this);   }
   void print(std::ostream& os)const override { os << "Im Volvo"
};

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

```
   car* clone() override{  return new bmw(*this);   }
   void print(std::ostream& os)const override { os << "Im bmw";}
};

inline std::ostream& operator<<(std::ostream& os , const car& cr
  cr.print(os);
  return os;
}

void main( ){

   car* cx= new bmw;
   std::cout << *cx << "\n";
   delete cx;
 }
```

"virtual dispatch" (sanal yönlendirme) alternatifi C++ da bir çok yöntem var:

- type erasure
- std::variant
- CRTP (curiosly recurring template pattern) - generic programlama

---

```
class base{
    int x;   // base için de +4 byte yer tutar
    int y;   // base için de +4 byte yer tutar
    void foo(){}         // base için de +0 byte yer tutar
    virtual void bar(){} // base için de +4 byte yer tutar :) !
  virtual void bar2(){} // base için de +0 byte yer tutar :) !!
  virtual void bar3(){} // yukardaki virtual ile aynı
  virtual ~base() {}    // yukardaki virtual ile aynı
}
// sınıf polimorfik olduğund yani (virtual) eklenince 4 byte cla
// derleyici ne yapıyor ? C örneği:
```

sanal işlev çağrısı (virtual dispatch) C örneği: **CHATGPT**

```
#include <stdio.h>

typedef struct {
    void (*start)(void);
} start_t;

typedef struct {
    start_t base;      //   işte bu 4 byte yer tutar (yukarda vi
} volvo;

typedef struct {
    start_t base;      //   işte bu 4 byte yer tutar , yani her
} bmw;

void volvo_start(void) {
    printf("volvo\n");
}

void bmw_start(void) {
    printf("bmw\n");
}

int main() {

    volvo v;
    v.base.start = volvo_start;

    bmw b;
    b.base.start = bmw_start;

    // Polimorfik olarak kullanma
    start_t* vtable[] = {(start_t*)&v, (start_t*)&b}; // bu tabl

    for (int i = 0; i < sizeof(vtable) / sizeof(vtable[0]); ++i)
        vtable[i]->start();  // Sanal dispatch - virtual dispato
```

```
    }

    return 0;
}
```

virtual dispatch zararlar ! KULLANMAK TEHLİKELİ OLABİLİR - GÖMÜLÜDE !

- memory allocation kullanır. HEAP de.
- run time da her polimorfik class için bir sanal fonksiyon tablosu oluşturur.(mem alloction ile) Bu da görünmeyen maliyetlere yol açar. Bu arada ilgili oluşturulan **sanal fonk tablosu**nun indeksleri compile time da biliniyor, yani hangi fonksiyon hangi indekste. Ama allocate edilen memory alanından çağrılan fonksiyonların kendisine ulaşmak ve run etmek ekstra bir maliyet doğuruyor. (dereferencing)
- free etmek memory protection problemleri. **virtual dispatch→** PERFORMANS KATİLİ

devirtualization: derleyici optimizasyonu, derleyici koda bakıo ve burda memory allocationa gerek yok deyip senin virtual dispatch kodunu optimize ediyor.

inherited constructor: (c++ 11)

```
class car{
public:
    car(int);
    car(int , int);
};

class volvo: public car{
public:
    using car::car;  // inherited constructor - artık bütün car
/** diğer uzun çözüm
  volvo(int x):car(x){}
  volvo(int x , int  y):car(x, y){}
```

```
**/
};
```

covariance (variant return type) - bir pattern gibi, bir önceki dersteki araba örneğindeki clone() işlemi de buna örnek aslında. O örnekde clone fonksiyonlar “car*” verine örneğin “volvo*” da dönderebilir !!.

```
class car{
public:
    virtual int foo();
};

class volvo: public car{
public:
    double foo()override{} // HATA , int olmalıydı, covariance
};

// covariance yukardaki problemi çözer, pointer (*) yada & olma

class B {}

class D: public B {}

class car{
public:
    virtual B* foo();
    virtual B& bar();
};

class volvo: public car{
public:
    D* foo()override;
    D& bar()override;
};
```

RTTI:(Run-Time Type Information) bir nesnenin dinamik türünün run time da öğrenilmesi. Önceki örneklerde;

```
void car_game(car* p)
{
   // sorun şu buraya gelen araba volvo mu bmw mi? Bu aslında k
   // bu fonksiyonda bu objenin bu özelliği denetlenmemeliydi! a
   // dynamic_cast ile
}
```

NVI (non virtual interface pattern) - Herb Sutter: polymorphism kullanılarak sınıfın davranışlarını kontrol etmek ve genişletmek amacıyla kullanılır.

```
class base{
public:
  void foo(int x)//non virtual
  {
     // örneğin burda x i kontrol et ve vfoo çağır
     vfoo(x);
  }
private:
   virtual void vfoo(int);
}

class der: public base{
public:
   void vfoo(int)override;  // bak derived class base clasın pri
}

der d;
d.foo(); //  foo da çağrıldı, override edilen der::vfoo da çağri
```

final contextual keyword:

final class - bu classdan kalıtım yapmayın demek!

final override - bu fonksiyonu artık kimse override edemez demek

```
class test final{ // der sınıfından kimse kalıtım yapamaz demek
public:
}

class base{
public:
  virtual void func();
}

class der final: public base{  // der sınıfından kimse kalıtım y
public:
   void func()override final;// @ft145, artık kimse override ede
}

class mer: public base{
public:
   void func()override;   // HATA oldu @ft145 sayesinde, @ft145
}
```

DERS 27

private interitance

- has-a relationship ama tamda değil. dışardan yaratılan ve kullanılan ifadelerde has-a relationship gibi ama türetilmiş sınıf içinde is-a relationship gibi. **@madx485**
- private aynı public de ki gibi erişilemez türemiş sınıf için.
- base classın public ve protected fonksiyonları , türemiş sınıfın **private** bölümüne eklenir!!
- virtual dispatch de ki ki kurallar burdada geçerli, override , final , virtual
- containment (composition yani has-a relationship) için bir alternatif oluşturur.
- **derived** class override ı kendi içinde istediği bölümde (public , private , protected) yapabilir tüm kalıtımlarda !!

```
class base{
public:
   void foo();
protected:
   void foo2();
}

class der: private base{
private:
// base::foo() buraya eklendi
// base::foo2() buraya eklendi
public:
  void bar(){    // aynı şekilde friend'ler içinde geçerli bu @m
     //@madx485
      der x;
      base *p = &x;   // ÇALIŞIR
      base &p = x;    // ÇALIŞIR , artık is-a relationship gibi
   }
}

int main(){
   der x;
   base *p = &x;   // HATA
   base &p = x;    // HATA , artık is-a relationship değil !!
}
```

```
class engine{
}

class car{
// - override diye bişey yok, e.foo();  erişim böyle
// - engine in protected bölümüne erişemez
// - araba motora sahip ama araba motor değil. motor objesi ve a
private:
```

```
  engine e;   // has-a relationship   ,  member object , contai
  engine e2;  // burda 2. motor eklemek mümkün ama aşağıda priva
}

// ********** vs ***************

class engine{
}

class car : private engine{  // base class object
private:
// - override ederim
// - engine in protected bölümüne erişir
// - arabının ve motorun adresi objeleri dönüştürülebilir oldu,
// - has-a ralationship gibide ama içinde is-a relationship var
}
```

containment'ı private interitance a tercih etmek daha iyi bir çözüm olur genelde !!

containment'ı kullanmayıp private kullanmak için sebepler;

- override istiyorsunuzdur
- protecteda ulaşmak istiyorsunuzdur
- yada derived içinde is-a relationship gibi davransın istiyorsunuzdur

EBO - empty base optimization; eğer sınıfın bir elemanı empty byte(boş yani 1 byte ise, boş class veya içinde sadece fonksiyonlar olan class 1 bytedır) ise private inheritance yapınca derleyici onu optimize eder ve derived classın boyutu küçülür. Containmentda bu olmaz.

Polymorphism : poli (çok) + morfizmosn (biçim) = **çokbiçimcilik**

Herb Sutter - Restricted(sınırlı) Polymorphism Pattern; çok sık kullanılan bir örüntü değil !

```
class base{
public:
   virtual void foo();
```

```
}

class der: private base{
    friend void f1();   // pattern şu ki derived class base'e er
}

void foo(base& p){
   p.foo();
}

void f1( ){
   der  d;
   foo(d);      // ÇALIŞIR
}

void f2( ){
   der  d;
   foo(d);    // sentaks HATA sı
}
```

protected inheritance; private da ki kuralların çoğu geçerli

- fark; private inheritance da taban sınıfın üyeleri türetilmiş sınıfın private üyeleri oluyordu,

protected inheritance da ise, taban sınıfın üyeleri türetilmiş sınıfın **protected** üyeleri oluyor.

- **Multilevel inheritance** da daha çok **protected inheritance** tercih edilir.

```
class base{
public:
    void foo();
}

class der: protected base{
```

```
    // base::foo(); bu der'in protected kısmına eklendi.
}

class Mder: public der{
    void bar(){
        foo();       // işte HATA YOK ! protected sayesinde ulas
    }
}
```

Multiple Inheritance; yukardaki multilevel farklı bu farklı !!! **Multilevel yukardaki**

Ne demek; bir sınıfı oluştururken, türetirken bir den fazla taban sınıf kullanmak. !!

Tek li kalıtımda ki çoğu kural burdada geçerlidir. 2 aynı base class implement edilemez.

```
class base1{
   void foo();
    virtual void  foo2();
}
class base2{
  void bar();
  virtual void  bar2();
}

// class der: public base1 ,/** burayı boş bırakma**/ base2{// b
class der: public base1 , public base2{ // önce base1 yazdın, ön

    void bar2() override{}
    void foo2() override{} // kurallar aynı!
}

void test(base1& x){}
void test(base2& x){}  // func overloading
```

```
int main()
{
   der x;

   x.bar();   // kurallar aynı!
   x.foo();

   base1 * p1 = &x;  // kurallar aynı!
   base2 & p2 = x;

   test(x);   // HATA , ambiguous
   test(static_cast<base1&>(x));   // böyle yapcan
}
```

STD IO stream de böyle !! output ayrı class , input ayrı class ve stream içinde **Multiple Inheritance** olur.

DDD: dreadful diamond of derivation - diamond formation

```
class stream{
   void foo();
}
class input_stream : public stream{  // @fcd485 bak burda bu str
}
class output_stream : public stream{ // @fcd486 bak burda bu str
}
// not: @fcd486 ve @fcd485 ya farklı değil aynı stream objesi iç
class io_stream : public input_stream  , public output_stream {
     // io_stream içinden diren stream classına ulaşamıoz suan a
     // ve her stream objesi input ve output için farklılar -> c
}
/** bu yukardaki yapı işte ELMAS formasyonu **/
int main()
{
  io_stream  der;
```

```
  stream* s1 = static_cast<input_stream*>(&der); // böyle ulaşı

  der.input_stream::foo();    // böyle ulaşılır
}
```

virtual inheritance: yukardaki örnekte @fcd485 ve @fcd486 ile üretilen stream objelerinin aynı root a bağlanmasını istiyorsan: Objeyi(stream) hiyararşi içinde en son alan yarattırır, kontrol eder aslında!

```
class stream{
   void foo();
}
class input_stream : virtual public stream{    // tek/ortak stre
}
class output_stream : virtual public stream{   // tek/ortak stre
}

class io_stream : public input_stream  , public output_stream {
}
/** bu yukardaki yapı işte ELMAS formasyonu / stream lar ORTAK
int main()
{
  io_stream  der;

  stream* s1 = &der;   // bak burası nasıl oldu şimdi yukardaki

  der.foo();   // bak burası nasıl oldu şimdi yukardakinden farl
}
```

NOT: Bu örnekte io_stream classının constractorı aslında tek/ortak olan stream classının constractorını init ediyor. !! input_stream VEYA output_stream DEĞİL.

RTTI - Run-Time Type Information: yukarda neden lazım anlattık. soru şu; car_game() içinde hangi araba modeli geldi ona göre belki farklı fonksiyon çağırcaz. Nasıl yaparız?

- **dynamic_cast**

- **typeid:** bir sınıf türünden nesneye eriştirir, **type_info**

```
 class car{
public:
    virtual ~car() {}
    virtual void start() = 0;
    virtual void stop() = 0;
    virtual car* clone() = 0;
    virtual void print(std::ostream& os)const = 0;
};

class volvo: public car{
public:
   void autopilot();       // volvoya otonom mod eklenmiş !!
   void start() override{}
   void stop() override{}
   car* clone() override{  return new volvo(*this);   }
   void print(std::ostream& os)const override { os << "Im Volvo"
};

class bmw: public car{
public:
   void start() override{}
   void stop() override{}
   car* clone() override{  return new bmw(*this);   }
   void print(std::ostream& os)const override { os << "Im bmw";}
};

void car_game(car* p)
{
  //TABAN SINIFTAN TÜREMİŞ SINIFA DOWN CASTING YAPMAK GEREK, BU
  volvo* bx = dynamic_cast<volvo*>(p); // bu bildiğin conteiner_
    if(bx){
          // code...
       bx->autopilot();
```

```
    }
}

void car_game(car* p) // exception handling
{
  try{ // yukardaki gibi nullptr bakmıosan diğer çözüm bu !!
     volvo* bx = dynamic_cast<volvo*>(p); // bu bildiğin contei
       bx->autopilot();
  }catch(const std::exception& ex){
     std::cout<< "fault" << ex.what();
  }
}
```

- dynamic_cast: çalışma zamanında taban sınıftan, türemiş sınıfa dönüşümün yapılıp yapılamayacağını sınamak için. maliyeti var, virtual gibi! **dynamic_cast yapabilmen için , ilgili sınıf polimorfik olmalı !! (virtual func olmalı)**
auto x = dynamic_cast<bmw * >(car_ptr)
auto x = dynamic_cast<bmw & >(car_ref)
if(x != nullptr) // nullptr olur hata ise

DERS 28

unevaluated context: işlem kodu üretilmeyen bağlam. mesela **sizeof()** veya **decltype()** içinde ki kod işlem yapmaz ve **typeid()** de öyle.

typeid() : type_info nesnesi üretir . bütün türler için derleme aşamasında oluşturulur bir kez. eğer projeye kütüphaneyi dahil edersen. compiletime da olur. sınıf polimorfik ise dinamik türü esas alır. polimorfik değil ise statik tür esas alınır.

```
class test_c{}

int main(){
   test_c cx;
   std::cout << typeid(cx).name();  // türü in olsa outputta mes

   if( typeid(cx).operator==(typeid(test_c)) ){  // işte tip ka
```

```
    }
}
// output:    "class test_c"
```

```
void car_game(car* p) // dynamic_Cast alternatifi oldu
{
  if(typeid(*p) == typeid(volvo)){
          // code...
       static_cast<volvo*>(p)->autopilot(); // tür dönüşümü içi
    }
}
```

Template Method: Çok sık kullanılan bir pattern:

```
class base{
public:
  void bar(){
    // code ..
    foo();
  }
private:   // sanal fonksiyonlar privateda saklanır !
  virtual void foo();  // burası bir interface oldu, türemiş sı
}
```

STL (Standard Template Library) - std

sequence containers:

- vector
- deque
- list
- forward_list

- string
- array

associative containers:

- set
- multiset
- map
- multimap

unordered associative containers:

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

DERS 29 | DERS 30

the c++ standart library OKU

derste string , vector , vs anlatıldı

DERS 31

Dikkat programlama hatası değil bu ! programlama hatasını yakalayan birim → **assertions(c de ki “assert()”),**

exception handling run time hatalarını bulur !

exception handling:programın normal akışının bir istisnanın (exception) neden olduğu bir durumu ele almasıdır. İstisnanın ortaya çıkması durumunda, programın normal çalışma akışından sapmaktan ve hatayı ele alarak programın kontrolünü düzenlemek mümkündür.

Diyelim bu aşağıdaki fonksiyonlar bir birini çağırıyor, ve f5() fonksiyonunda hata oldu ama bu hatanın asıl çıktığı kontrol edilmesi veya değiştirilmesi gereken yer f1() içinde.

f1() ⇒ f2() ⇒ f3() ⇒ f4() ⇒ f5() , f5() de çıkan hatanın f1() de karşılığı var ve direk hata durumunda oraya gitsin istiyoruz →

Bu durumda ne yapcan, sürekli hepsinin returnünü bir birine bağlamak mümkün olmayabilir. çözüm: **exception handling, try-catch** !!

veya her fonksiyon return ile hata kodu dönemez mesela **constractor member functionların'ın geri dönüşü yok**!! çözüm: **exception handling, try-catch-throw** !!

```
try{
   f1();
}
catch(int) // burda tür dönüşümü yok illa int göndercen
{
}
catch(long)
{
}
// eğer bu catch türlerinden her hangi biri değil başka bir tür
// uncaught exception olur, bu da std::terminete fonksiyonunu ça
// set_terminate(my_func_ptr) yaparsanız abort değil senin istec
```

```
void my_abort(){
  // log
   exit(EXIT_FAILURE);
}
void f2(){
  throw 1;  // eğer bunları kapsayan try catch yoksa direk termi
// normalte: throw (exprassion)
// throw dönüşü genelde referans ile yapılır ki objeler copy olı
}
void f1(){
  f2();
}
int main(){
```

```
  set_terminate(&my_abort);

  try{
      f1();
    }
  catch(int){  // burdaki parametreyi kullanmıcaksan isim verme

    }
  catch(char){  // burdaki parametreyi kullanmıcaksan isim verme

    }
  catch(...){  // tüm hataları bu yakalar

    }
}
```

```
//std::string içinde ki hata bu aşağıdaki classları miras alır,
//-> out_of_range -> logic_error -> excaption
void f2(){
 std::string str{"test123"};
  auto c = str.at(400);  // 400. indexe erişim yok, hata
}
void f1(){
  f2();
  f3();  // bunda exception yok
}
int main(){

  set_terminate(&my_abort);

  try{
      f1();
    }
  catch(const std::exception& ex){ // catch(const std::logic_er
      // print ex.what()
```

```
    }
}
```

DERS 32

```
void f3()noexcept   // bu bildirim noexcept specifier: ben throw
{
    throw std::exception{};  // eğer burda excaption çıkarsa , b
}

// bu bilgi generic programalmada çok önemli, noexcept operator
void f4()noexcept(true) = void f4()noexcept // noexcept(true) de
void f5()noexcept(false) = void f4()        // noexcept(false) d
void f6()noexcept(sizeof(int) > 2);         // işte bu şekilde b

/****/

void f7(int x)noexcept{}
void f8(){int y}

void f9()noexcept(f7(12)); // bu noexcept(true) çünkü f7 true
void f9()noexcept(f8(12)); // bu noexcept(false) çünkü f8 false,

/****/

class mclass{
   void f1(int x)noexcept;
}

void f2(mclass mx)noexcept(mx.f1(12)); // f2 noexcept olur, yuka

// note: noexcept de sizeoff ve decletype gibi yan etkisi yok ya
// destractor lar her zaman noexcept !!
// move constractor / swap functions / deallocations(free) noexc
```