lessons 15 - 16

LESSON 15

copy elision: (C++ 17 ve sonrası ; **mandatory copy elision**)

C++ 17 den önce bu işlem için derleyici copy çağrıyordu ama artık C++17 ve sonrasında bu şekilde → **my_class{}** tanımlanan ifadeler artık bir reçete gibi çalışıyor, yani ulaşacağı yere göre optimize oluyor. Copy değil, direk init olmasını sağlıyor gibi düşünebiliriz, ulaştığı objenin. Durumdan duruma göre de değişebiliyor.

Eski(C++17 öncesi) optimizasyondu artık dil kuralı oldu.

```
class my_class{
    public:
    my_class( );

};

void foo( my_class m){

}

int main(){
    foo(my_class{});
// C++ 17 den önce şuna dönüşüodu => { my_class m ; foo(m); } gi
// C++ 17 den sonra bu artık reçete gibi çalışıyor yani sadece i

   const my_class &x = my_class{};  //  yani my_class{} bir nesn
```

```
}
```

RVO - return value optimization ; copy elision ile mantık aynı C++ 17 den sonra bu işlem çok kullanışlı oldu.

Eski(C++17 öncesi) optimizasyondu artık dil kuralı oldu.

```
class my_class{
    public:
    my_class(int , int);

};

my_class foo(){
   return {6,7};  // yine copy yok reçete gibi çalışıyor, al bu
}
```

NRVO - named return value optimization. Bu hala optimizasyın, GÜZEL OPTİMİZASYON

```
my_class func()
{
    my_class m;

    /** process ***/

    return m; // burda L value to X value dönüşümü yapıo derleyi
}

int main(){
    my_class mx = func();    // normalde move oluo, ama optimiza
    my_cla-ss mx2 ;
```

```
    mx2 = func();              // bu olmaz ! bunda optimize eder
}
```

Dikkat UNUTMA ki şuana kadar konuştuğumuz **copy ve move assinment** gibi işlemler **allocation yapılmış** vector, string gibi objeleri taşımakta avantaj. int **x[1000]** i herkes mecbur hamallık ile taşımalı. Burdada devreye **copy elision**un girebilio olması güzel !!

dynamic storage class : **new** , **delete**

new: expression(not operator) , **operator new(size_t)** foksiyonu çağrılır , new my_class , new int ... **malloc** göre avantajı **Exception Throw** olması. NOT: **new(std::nothrow)** Exception Throw dönmez nullptr döner.

delete: ptr→~my_class() ve **operator delete(ptr)** çağrılır**!**

```
// new my_class -> çağrılan global fonksiyon = void* operator ne

std::vector <void*> test;

try{
   while(1)
     test.push_back(operator new(1000));
}
catch(const std::exception& ex){
   //exception caught
}

**********************
//new my_class   türü pointer (my_class*)
new my_class();  <=>  static_cast<my_class *>(operator new(sizeo
```

```
class my_class{
    public:
    my_class( );
    int x;
};

int main(){
   my_class* m = new my_class;
   my_class* m = new my_class(); // içine varsa argumanları yaz
   my_class* m = new my_class{};
   auto *    m = new my_class;
   auto      m = new my_class;   // 5 i de aynı !!

   // delete demezsen , destroctor çağrılmaz -> resource leak ,

   m->x;   // pointera ok ile ulaşılır


   delete m;  // hayatı bitti -> operator delete   , free(m) ->
}
```

memory leak: malloc yaptın , alloction yapılan alanın **free edilmemesi**

resource leak: genellikle programın çalışması sırasında açılan kaynakların (bellek, dosya, soket, veritabanı bağlantısı gibi) serbest bırakılmaması. YANİ **destroctor ÇAĞRILMAMASI. (destroctor içinde free olabilir tabi)** kaynakların geri verilmemesi.

NOT: artık modern C++ da çok kullanılmaz. **smart pointer** bu işi halledio bizim yerimize.

smart pointer: bir sınıf türünden nesne. **operator overloading** sayesinde.(ilerde görcez). (my_class*) tanımlanan sınıflara **pointer-like class** denio. Eğer pointer-

like sınıfları kontrol etmek istiosan bu işi smart pointer otomatik halledio. **std::unique_ptr** , **std::shared_ptr.**

RAW pointer/naked pointer: bizim daha önce bildiğimiz kontrol edilmesi gereken pointerlar. malloc , free gibi.

operator delete ve **operator new** , BU GLOBAL FONKSIYONLAR NASIL IMPLIMENT EDİLMİŞ

```
void * operator new(std::size_t sz)
{
   auto vp = std::malloc(sz);
   if(!vp)
      throw std::bad_alloc{};

   return vp;
}

void  operator delete(void * vp)
{
    std::free(vp);
}
```

```
int *p = new int[1000];  // memory allocation like malloc

delete[] p;   // dikkat et bak class object gibi değil array si

*************

my_class * p = new my_class[10];

delete[] p;   // class dizi , sondan başa dizinin hayatı biter ,
```

KABACA **smart_pointer** , NASIL IMPLIMENT EDİLMİŞ

```
class my_smart_pointer{
public:
    my_smart_pointer(my_class *p):pm(p){}
   ~my_smart_pointer(){delete pm;}

private:
   my_class*pm; // objenin türden bağımsız olduğunu düşün
}
```

static class members

- **static** bir bildirim. **non defining declaration**. Yer ayrılmıo sınıfta, linkerda yapılıo. yani global gibi düşün.

```
// .h

class my_class{
   static int mx;  // class veriable, assambly kodda global değ
   // global değişkenler ile farkları;
   // - class scope bu class ile erişiosun :: , namespace avanta
   // - public private oluor , accessibility
}

// .cpp

int same_mx = 0;  // aynı mx böle

int my_class::mx{};  // initilation , class da yazıldı burda sta

**********
int main(){
   my_class m1 , m2;
```

```
    m1.mx = 56;
    m2.mx = 78;    // bu m1 'inde m2 ninde mx'i aynı. 1 tane glob
}
```

DERS 16

C için de C++ içinde önemli bilgi !!

test.h içindeyiz →

```
//#include "test2.h" // EĞER BUNU YAPARSAN saadece "test2_obj" d

struct test2_obj;  //incomplite type #include "test2.h"  içinde


struct test1_obj{
     struct test2_obj *  x;  // bu başka headerda ki bambaşka bi
 }
```

Modern C++17 ile

```
class my_class
{
   // static int x = 5; // hata
   // static std::Vector<int> x{1,2,3}; // bu bildirim hata, çağ
    inline static std::Vector<int> x{1,2,3}; // Bu kabul, artık
}
```

Modern C++ all

```
class my_class
{
    static const int x = 5; // olur
    static constexpr int x = 5; // olur, zaten inline bu
}
```

```
class my_class
{
    static int x;

    my_class(int k): x(k){}   // HATAAA!! bu constroctor list nd

    void foo()const{
        x = 5;   // bak hata yok const fonksiyon bile değiştirel
    }
}
```

```
class my_class
{
    int x;
    static int y;
}

int my_class::y = x;  // HATA, bu x class namespace arama yani n
```

class static member functions:

```
class my_class{
     static void foo(){   // global function gibi düşün, gizli p
         // this yok
```

```
     }

     static void bar()const{} // gizli parameter olan "obj* me"

     static void bar2(){
        x = 6;    // HATA, ortada my_class objesi yok, this(me)
        y = 5;    // DOĞRU tabiki, 2 side global.
        foo2();   // HATA tabiki, bu kimin objesi belli değil a

        my_class m;
        m.foo2();   // bu şekilde kullanılır.
    }

     void foo2(){
          bar2(); // DOĞRU tabiki
     }

     int x ;
     static int y;
}

void main(){

       my_class m;
       m.bar2(); // doğru ama yani m objesinin hiç bir vasfı yo
}
```

class static member functions vs global functions

- namespace veya class space, global de yok böle bişi
- public , private vs..

```
class ali{

  static int foo(){return 3;}
  static int x;
}

int foo(){return 2;}

int ali::x = foo();   // x 3 olur. ali:: isim aramadan class iç
int ali::x = ::foo(); // şimdi 2 olur
```

```
//header
class ali{
    static int foo();
}

//cpp
int ali::foo(){   // burda static olmaz, zaten class içinde bil

}
```

NERELERDE KULLANIRIM: globali neden kullanırsan o yüzden. veya c de file içi static leri. ama güzelliği namespace var. erişim kontrolü var.

```
class ali{
    // olay şu; bu x, sınıfın tüm elemanları ve diğer yaratılan
    static std::vector<int> x;
}
```

named constructor - bu bir pattern, static ile, fabrika fonksiyon

```
class ali{
public:
      static ali create_object();// constractorı gizlemek
}

void main(){
   auto x = ali::create_object(); // içerisinde constroctor çağı
}
```

```
class factory{
public:
  static factory create_usb(int port)
  {
     return factory(port);
  }
  static factory create_ethernet(const char* ip)
  {
     return factory(ip);
  }
private: // dikkat private, constractorları gizledik
  factory(int x);
  factory(const char* str);
}

int main(){
   auto usb = factory::create_usb(5);
   auto eth = factory::create_ethernet("192.168.1.1");
}
```

Kitap: object orianted design patterns: https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612 OKU!!

Singleton pattern

```
class singleton{
public:
    singleton(const singleton&) = delete; // copyler kapalı
    static singleton* get_instance()
    {
       if(!mp)
          mp = new singleton();

       return mp;
    }
  void user_func1();
  void user_func2(); // vs ..
private:
   inline static   singleton *mp{};
   singleton();
}

int main(){
    singleton::get_instance()->user_func1(); // obje 1 kere yara
}
```

Meyers singleton , simple , thread safe

```
class singleton{
public:
    singleton(const singleton&) = delete; // copyler kapalı
    static singleton& get_instance()
    {
       static singleton obj;

       return obj;
    }
```

```
  void user_func1();
  void user_func2(); // vs ..
private:
   singleton();
}

int main(){
    singleton::get_instance().user_func1(); // obje 1 kere yarat
}
```