lessons 1 - 15

Variable initialization:

```
int x = 5;

int x(5); // x=5; direct initialization

int x(); // func or variable? could be func

int x(3.4);

{} -> uniform

int x{ 5 }; -> list initialization

int x{}; → value initialization , 0 or NULL

int x{3.4} -> compiler syntax error

int x[10]{};
```

```
void foo(){
    // same defs for GLOBAL
    static int x;    // → zero initialization
    static int *x;   // → NULL initialization
    static bool x;   // → FALSE
}
```

NULL → nullptr

ARTIK HEP **nullptr** kullan (tam NULL değil ama NULL'a da dönüşüo , yani başka kontrollerde yapılıo bunla)

VALUE CATAGORIES

```
L value   - adresi olan hersey

example: *ptr

eğer bir expression(ifade) yi L value olup olmadığını test etmek istersen & kullan.

Örnek,

int x;
```

```
&x;   → hata vermedi. O zaman L value

R value    -      L value olmayanlar R value

Universal Value ->    "auto && x;"   ÖZEL dikkat "int && x;" değil karştırma


ref values


int &x -> L value REF
int &&x -> R value REF
auto &&x -> universal REF - hem L hem R value ref ve türden bağımsız !!!!!!!!
-----------------------------
int &x = a;
int &&x = 10;
```

REFERANS SEMANTİĞİ

```
 int a[10]{};
 int (*ptr) = &a;   // dikkat int *ptr = a; bu sadece a dizisinin ilk elemanını gösterir
 int (&ref)[10] = a; // referans semantiği
```

```
int x = 5;
int &y = x; // y artık x oldu (tabiki arka tarafta pointer işlemleri koşuo ama seni karma

const int & x = 10;  //geçerli ama derleyici gizli bir yerde x i yaratır, dikkat x değiş

// int &x = 10; // ERROR
// NOT: int& const x = a;    -> BU SAÇMA referans larda zaten gizli const var aynı yerde,


int g = 5;
int & foo(){ return g;}

foo() = 999; // çünkü artık foo = g referans döndü


//C de böle yapıosan ;
test_create(obj * me, const obj_t * me_)
//C++ de artık böle;
test_create(obj & me , const obj_t & me_)  // bu zaten class member function aslında !! '
```

auto → her türe uyar, AAA (almost always auto)

```
auto x = 42;         // int türünde
auto y = 3.14;       // double türünde
auto name = "John";  // const char* türünde

const int x = 10;
int yy = x;   // ee bu mantıklı ! o zaman aşağıdakide mantıklı
auto y = x ; // burda y int yani constlık düştü



int a[20];
auto b = a; // int*b = a;  , array dikey

const int a[20];
auto b = a; // const int*b = a;  , array dikey


// ref autolar**********************************

const int x = 10;
auto & y = x;  // const int & y = x;


int x[10]{};
```

```
auto &y = x;   // int (&y)[5] = x;              BU SUPERMIŞ !! Diziye direk erişim

int x[10][20]{};
auto &y = x;   // int (&y)[10][20] = x;         Hamallıktan kurtarır


auto& x = "hello"  // const char (&x)[5] = "hello"  , yani bu string için global ram de b

int foo(int); // function
auto& f= foo; // int (&f)(int) = foo ;   auto türü "int(int)" , bu bi fonksiyon ise , f f
```

Function Return Opstion (LIKE it <3)

```
void koo(int *a , double b);

auto foo() ->  void (*)(int* , double)
{


   return koo;
}
```

decltype → derleme zamanında

```
int x = 42;
const double y = 3.14;

// decltype kullanarak değişken türlerini belirleme
decltype(x) a = 10;        // int
decltype(y) b = 2.71828;    // const double


// functions
int foo1();
int & foo2();
int && foo3();

decltype(foo1())  // int
decltype(foo2())  // int &
decltype(foo3())  // int &&
```

using : typedef yerine artık bunu kullan

```
    using my_ptr = int[20];    // typedef int my_ptr[20];
```

```
    using my_ptr = const int*; // typedef const int* my_ptr;
    using my_ptr = int (*)(const char* , char*); // typedef int (*my_ptr)(const char* , c
```

REFERENCE COLLAPSING (referansa referans olma durumlarının hangi tür olacağı kararı verme)

```
using lref = int&;   // typedef int my_ptr[20];
using lref2 = int&&; // typedef int my_ptr[20];

int y{};

lref &x = y;
lref &&x = 10;
lref & &&x = y;
lref2 &&x = Y;
lref2 &x = y;

KURAL ->
T&  &   => T& (l value)
T& &&   => T& (l value)
T&& &   => T& (l value)
T&& &&  => T&& BU FARKLI   REF (R value)
```

auto && x; UNIVERSAL referans herseye bağlanıo. L da R value de olur

Funstions inits SUPER özellik, compile timeda oluo

```
void foo(int , int = 5);

foo(6);  // foo(6,5);
```

constexpr - const yerine artık bunu kullan. gizli inline taşır. compiler optimize yetkisi alır.

```
const int  x = 10;
int a[x]; // bunun çalışması için x e değer verilen ifade önemli

int r = 5;
const int  y = r;
int a[y]; // ERROR    , işte bu olmasın istiyorsan "constexpr"

constexpr int  x = 10;   // bunu yaparsan yukardaki problem olmaz
constexpr int  x = r;    // ERROR, sabit exprassion olmalı


constexpr int test(int x){  // SUCCESSS
```

```
   int y = x;
   return y*y*x;
}

constexpr int test(int x){  // ERROR , sıkıntı VAAAAR
   static int y = x;        // static yapamazsın
   return y*y*x;
}

/** "constexpr" asıl olay şu , run timeda değil compile timeda değerler belli olsun,
    örneğin yukardaki fonksiyonu "constexpr" ile oluşturunca, derleyici fonksiyonu komple
    zamanında alıp bir sabit ifadeye çevirio yani fonksiyon artık fonksiyon değil sabit d
    Tabiki bunun kararını vermesi için fonksiyonun girdileride sabit olmalı !! eğer değil
    BU DEVRİM.. **/

// örnek

constexpr int ndigit(int x){
    return out;
}

int main(){

char array[ndigit(1541512)]; // hata yok çünkü fonksiyon girdisi sabit ifade ve fonksiyon

constexpr int abc = 54848;
```

```
char array[ndigit(abc)]; // yukardaki ile aynı durum

int abc2 = 11;
char array[ndigit(abc2)]; // OLMADI hata yok kodda ama artık değer run timeda işlem sonuc

}
```

ODR (one defination rule). Bir fonksiyonun bildirimi bir den çok olabilir. AMA bu fonksiyonın tanımı 1 kere yapılmalı.

```
#include test.c

int foo(){
 return x;
}

#include test2.c

int foo(){                  // foo 2 kez yazıldı olmaaaaz linker hatası, Dikkat derleyici
 return x;
}
```

```
*******************************

int foo();
int foo();           // ama burda sıkıntı yok bildirimi istediğin kadar yap istediğin her
```

inline → en önemli optimizasyon

```
inline int foo(int x){
  return x*x;
}

int foo(int x){
  return x*x;
}

// bu yokardaki iki fonksiyonda nerdeyse aynı, inline olmayan fonksiyonu derleyici optimi
// veya inline olan fonksiyonu derleyici inline değilmiş gibi inline hiç yokmus gibi yani


// ODR tanımını inline parçalayabilir

#include test.c

inline int foo(){
```

```
 return x-5;
}

#include test2.c

inline int foo(){                  // ODR ihlal edilmedi inline çünkü. TEK şartla fonksiy
 return x-5;
}


// Bu bize şunu sağlar inline fonksiyonlar; .cpp ler artık iptal. .h yani headarda tüm fo
// YANİ sadece h olan kütüphaneler oluşuyor only header


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

static vs inline   (in header)

static foo(){} // in header
aynısını tüm headerlarda fonksiyonları static yaparakta yaparız ODR ihlal olmaz.çümkü "in
Yani her headerı çağırınca birden fazla o static yaptığın fonksiyon oluşuyor. adresi fela

inline foo(){} // in header
AMA inline yaptığın fonksiyonlar sadece bir kere yaratılır. static gibi her headırı çağır
inline yaparsak o fonksiyon sadece 1 kere yaratıldı yani tek bir adresi var.
```

```
dikkat: C++ 17 den sonra global yerel değişkenlerde inline yapılıo ve ODR bozulmuo.
header içinde ->  inline int x=5;    -> for c++ 17 ve sonrası için


Dikkat classlarda ODR bozmaz


Dikkat constexpr fonksiyonlar iplictly inline dır ve yine ODR kuralını bozmaz.
heaadır içinde -> constexpr int foo(){return c;}  sıkıntı yok gizli inline var -> aynı c
```

DERS 7

enum

```
enum colors : int { red , greed};     // bu özellik modern c++ da

enum class colors { red , greed};     // bak bu class felan değil kafa karıştırma, bu tü

enum class CLR { red , greed};

colors::red;
CLR::red;     // bu ikisi eski enumda hata verirdi artık scopeları var ve böle çağrılır.
```

```
CLR myclr {CLR::red};
int x = myclr;                          // hata !! eskiden değildi. classı silersen enumdan
int x = static_cast<int>( myclr );    // çözüm bu yukardakinin

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

c++ 20 de gelen özellik

using enum CLR;           dersen, artık o scopda CLR::red demene gerek yok,   direk red d
```

::

```
::x -> scope resolution operator , binary operator

:: sayesinde derleyici "name lookup" yani isim aramaya bakarken, LOCAL alanda isim aramay

örnek:

int main(){
   int printf = 10;
   printf("hello");   // hata -> int printf bulunur
}

int main(){
```

```
   int printf = 10;
   ::printf("hello");   // hata YOK , printf artık local de değil globalde aranır ve bulu
}

x::y ise -> x "namespace" yada "class name" olmalı ve onun içinde çağrılan ismin name loc
```

Compiler Not

```
Derleyici proses steps;
 1 - Name lookup     - ilgili exprasion ismi geçerlimi var mı
 2 - context control - syntax uygun mu, kod hatası var mı , dil kuralları
 3 - access control  - public private bakılıo izin var mı o isme ulaşmaya
```

C++ → () içinde ifade tanımlanıor

```
int test(void){
   return 1;
}

int main(){

  while / if / switch( int x = test() )
  {
  }
```

```
if( int x = test() ){}

int x = test;
if(x){}              //  bu yukardaki ile aynı anlam scope leakage engellenio yukarda

}

*****************************
if(x = test(); x > 10){    // buda c++17 den sonra
  x = 6
}
else
{
  x = 8;   // bak x burdada geçerli çok iyi
}
```

type-casts , olay şu diyelim sen kodda (double*) tür dönüştürmeyi böyle yaptın. Kodda bunu nasıl arıcan, tüm tür dönüştürmeleri?. O yüzden isim vermişler aşağıdaki gibi. ve compiler controlünde.

```
static_cast
const_cast
reinterpret_cast
dynamic_cast            (bu sonra öğrencez ileri seviye)
```

```
xxxxx_cast<type>(expression)
static_cast<int>(x)

int x = 0;
double dval = static_cast<double>(x);

void* d = &x;
int* ptr = static_cast<int*>(d);

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

int x{5};
const int *f = &x;
int * ptr = const_cast<int*>(f);      // constlar için

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

double x{5};
char * ptr = static_cast<char*>(&x);         // hata
char * ptr = reinterpret_cast<char*>(&x);    // ÇÖZÜM bu , reinterpret_cast: Bu operatör,
// Ancak, dikkatli kullanılmalıdır, çünkü bu operatör tip güvenliğini sağlamaz. Eğer yanl
```

```
[[xxxx]]    -> bunlar compilera direktifi , compilera göre değişir
[[nodiscard]]   çok kullanılıo

örnek:

[[nodiscard]]
int foo(){ return 5; }

foo();    // derleyici uyarı vercek çünkü geri dönüşünü kullanmadın
```

Function overloading - **DERS 8**

```
//Function Overload - compile time da oluor.
//Aynı isme sahip bir den fazla fonksiyon yazabilme özelliği

int foo(double);
int foo(int);      // işte bu ikisi overload

int foo(int*)
int foo(int[])    // bu func redecleration , aynı func ikiside , overload değil tabiki


"ambiguity" (belirsizlik) // sentaks hatası, uygun tür değişkenin olmaması
```

```
int foo(int,double); // bu fonksiyonun signutere(imza) sı    "(int,double)" dönüş değil

************
void foo(int*)
void foo(int)

void main(){
   foo(NULL);  foo(0); //  bak eğer burda NULL = 0 deseydik  foo(int) çağrılırdı, "nullpt
   foo(nullptr);       //  foo(int *) çağrılırdı
}

******************
void foo(int&)    // L valueler buna
void foo(int&&)   // R valueler buna

void main(){
   int x;
   foo(x);        //  void foo(int&)
   foo(56);       //  void foo(int&&)
}

 void foo(const int&)    // r da alır l de alır
```

DERS 9

Classes

```
----- header file  ----
class my_class{
// burası default private, struct olsa public
public:
   void foo();


}

inline void my_class::foo()  // heeader içinde ODR uygun olması için inline olmalı, const
{

}


------------------------------


----- header file  ----
class my_class{
// burası default private, struct olsa public
```

```
public:
    void foo()    // burda gizli "inline" var yukardaki gibi aslında ama inline gizli, ir
        {

        }


}

*******

----- header file  ----

class my_class{
// burası default private, struct olsa public
public:
   void foo();
   int x;
}

----- cpp file  ----

int x;

 void my_class::foo()  // heeader içinde ODR uygun olması için inline olmalı
{
```

```
  int x;

  my_class::x = x + ::x;   // hiç sıkıntı yok, ::x(global) , x (local) , my_class::x(clas
  // veya this->x ,  my_class::x  yerine
}

void main{
  my_class cl;

  cl.foo();             // aşağıdakiyle aynı
  cl.my_class::foo();   // sıkıntı yok, kalıtımda kullancaz, burdaki kullanım sacma, ama
}
```

DERS 10

```
THIS , bu bizim C deki   void foo(obj*me);   nin "me" si "this" :)   Direk classın adresi

----- header file  ----

class my_class{
```

```
public:
   void foo()
    {
        this   // direk classın adresi
    }
    static void foo2()   // static ancak static i çağırır
    {
       this     // hata , static de olmaz
  }
}

******
DİKKAT this PR (R) value

class my_class{

public:
   void foo();
   int x;
}
void my_class::foo(){
    my_class m;
    this = &m;     // this L value olsaydı bu çalışırdı. AMA R value bu sentaks hatası o
   *this = mz;     // bu çalışıor bak L yaptık.

   this->x = 5;
```

```
   my_class::x = 5   // bu sadece look up bakıo yani direk x bu
   x = 5;   // bu yukardaki 3 ü de aynı
}

********************
this keywod
this pointer

*this nesnenin kendisi
```

this neden kullanılır örnekler:

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

// c file

void test2(my_class *p){

}

void test1(my_class &r){
```

```
}
void my_class::foo(){
    test1(*this);
    test2(this);            // bizim c de ki object mantığı
}
```

this neden kullanılır örnekler: **BU ÇOK İYİiiii**

```
class my_class{

  public:
    my_class& foo();
    my_class& bar();
    my_class& baz();
}

// c file

void my_class::baz(){
    // ..
    return *this;
}
```

```
void my_class::bar(){
    // ..
    return *this;
}

void my_class::foo(){
    // ..
    return *this;
}

int main () {

  my_class ME;

  ME.bar().baz().foo();    //  "chaining" zincirleme

  count << x << y << "test";    // "chaining" ref@789

  count.operator<<(x).operator<<(y).operator<<("test");  // ref@789
}
```

this benzer mantık yukardaki ile

```
class my_class{

  public:
    my_class* foo(){return this}
    my_class* bar(){return this}
    my_class* baz(){return this}
}


int main () {

  my_class ME;

  ME->bar()->baz()->foo();
}
```

const functions

```
---- C de:

void foo(const  my_obj * me);
```

```
----------C++ da

class my_class{

  public:
    void foo()const ;    // gizli const my_class * this(me)
}
```

bu classların aslında hepsinin içindeki fonksiyon bizim c de ki gibi gizli objeye sahip "me" gibi

hatta class sizeoffu nu fonksiyon hiç etkilemio , sadece içindeki değişkenler belirlio aynı C de ki struct :) yani o fonksiyonlar aslında hep globalde , bizim c de ki gibi “me” almıs compiler yapıo bu işi. 😀

```
class my_class{

  public:
    void foo()const ;
    void bar();
}

int main () {
```

```
  const my_class ME;

  ME->foo();   // sıkıntı yok
  ME->bar();   // sentaks hatası Neden mi (const my_class* this) gizli elemanı const olma
}

----  C version


void foo(const obj * me) ;
void bar(obj * me);


int main () {

   const my_class ME;

   foo(ME);   // sıkıntı yok
   bar(ME);   // sentaks hatası
}


************************************
class my_class{
```

```
  public:
    my_class* foo()const ;        // const dönmeli
    const my_class* foo()const ;  // bu doğru
}
```

En çok kullanılan overloading

```
class my_string{

  public:
    void foo()const ;
    void foo();        // overloading
}

int main () {

   const my_string me;
   me.foo();          // void foo()const ;   obje değişime kapalı, okuma için

   my_string me2;
   me2.foo();   // void foo();  obje değişime açık
}
```

“**mutable**”

```
class my_string{

  public:
    void foo()const
        {
            debug_count++;     // bu sentaks hatası fonksiyon const değiştiremezsin.
//ama öyle durumlar var ki const fonksiyon objenin anlamsal açıdan fark yaratmayan
//değişkenlerini belki test için değişmesi gerekecek o zaman ne yapacan? ÇÖZÜM AŞAĞIDA
        }

    int debug_count;
}

-- ÇÖZÜM ---

class my_string{

  public:
    void foo()const
        {
            debug_count++;     // şimdi çalışır
        }
```

```
    mutable int debug_count;  // "mutable" : değişken
}
```

DERS 11

constructor, destructor , kural: static olmayacak, const olmaz, member function olacak(yani global değil sınıfın içinde), sınıfın adı olacak. Birden fazla olabilir.(func overloading)

```
class my_string{

  public:
    my_string();      // constructor , default constructor argüman almıo
    my_string(int);   // constructor , overload olabilir
        ~my_string();     // destructor
}

void main{
    {
     my_string  m;
     // ...
    }  // burda gizli  "~my_string" çağrıldı
}
```

special member functions: derleyici bunların kodunu bizim için yazabiliyor. Bu fonksiyonları sen bir class yazınca derleyici gizli yazıo yani.

```
default ctor
destructor
copy ctor
move ctor      (C++ 11)
copy assigment
move assigment (C++ 11)
```

Derleyicin bunların kodunu bizim için yazmasına → “default” etti denir.

RAII → idomu : "Resource Acquisition Is Initialization” : Temel fikir, bir nesne oluşturulduğunda, o nesnenin yapılandırıcı metodunun çağrılması ve kaynakların bu yapılandırıcı içinde edinilmesidir. Nesne yok olduğunda ise yıkıcı metodun çağrılması ve kaynakların serbest bırakılmasıdır.

Bir nesnenin storage katagorisi şunlardan birdir. Nesnenin hayatını tutuo.

```
automatic storage class          -> örneğin fonksiyon içinde class çağırdın (static class
static storage class             -> global member functions
dynamic storage class
thread-local storage class
```

```
class my_string{

  public:
    my_string()
    {
      // .. default ctor
    }
  private:
    int x;
}

void main(void){
   my_string m;    // defaul init  // .. default ctor ,  AMA x e ilk değer atanmadı çöp d
   my_string m2{}; // value init   // .. default ctor , x e 0 atandı {} sayesinde !!
   my_string m3(); // hataa!! bu fonksiyon tanımıı !!
}
```

```
class my_string{

  public:
    my_string(int x_)
    {
       x = x_;
```

```
    }
  private:
    int x;
}

void main(void){
   my_string m(10);       // direct init , en çok bu kullanılır ama {} daha iyi
   my_string m2 = 20;     // copy init      , bu kötü bence
   my_string m5 = {20};   // copy list init , az kullanılıo
   my_string m3{50};      // direct list init     , yani 3 ü de doğru ama {} bu daha iyi
// içi int olmassa hata verirdi derleyici {} sayesinde.
}
```

Constractor init list, Başlatıcı liste: ile ilk değer verme

```
class my_string{

  public:
    my_string()
    {
       y = 6;
       x = 5;
    }
```

```
  private:
    int x , y;
}

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

class my_string{

  public:
    my_string() : x(5) , y(6)     // veya my_string() : x{5} , y{6}
    {

    }
  private:
    int x , y;
}


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

class my_string{

  public:
    my_string(int _x , int _y) : x(_x) , y(_y)
    {
```

```
    }
  private:
    int x , y;
}
```

const member için **kritik yer**

```
class my_string{

  public:
    my_string( )
   {
      // istersen const değişkenlere burda değer ver yine sentaks hatası zaten çöp value
   }
  private:
    const int x ;       // SENTAKS HATASI , const yaptın değer default atandı olmaaaaz
    int &r;             // referansta default init edilemez
}

****************************
class my_string{

  public:
    my_string( int &_r )  : x{50} , r(_r) // işte çözüm , böle yapcan constuda, hata gitt
```

```
    {
    }
  private:
    const int x ;
    int &r;
}
```

Constractor Başlatıcı liste: DURUM özeti şu ki sen eğer constractor içinde {} scope içinde ilk değerleri verirsen class içindeki başka objeler senin gerçek objelerinden önce bile başlayabilir ve maliyeti çok olur. Bunun için hep **Başlatıcı liste** kullanki senin objelerin hayata gelir gelmez o değer ile gelsin. Yukardaki const örneğine iyi bak onu kavrasan kompilerın nasıl çalıştığını anlıcan.

```
class my_string{
  public:
    my_string( ) : x{50} , y(78)   /// öncelik bu değerler, aşağıdakiler değil
    {

    }
  private:
    int x = 5;     // (inline initialization)
    int y{6};      // default member init,
}
```

DERS 12

delete : böyle bir fonksiyon var ama bu fonksiyona yapılan çağrı sentaks hatası olacak!!

```
class my_string{
  public:
    my_string(int) = delete;
  private:
    void foo(int) = delete;
}

void foo_global(int) = delete;
```

```
class my_string{
  public:
    my_string() = default;    // derleyici bizim yerimize bildirir. special member functi
}
```

ex:

```
class my_string{
  private:
    const int x;    // const int x = 10;     yine hata , const değerler inline initilatic
}

void main(){
     my_string m;    // SENTAKS HATASI
}

//Derleyicinin yukardaki kod için implicitly yazdığı kod:

class my_string{
    my_string() = delete;         // çünküü delete yazdı sebebi -> const un ilk değeri yo
  private:
    const int x;
}
```

```
class my_string2{
  public:
    my_string2();    // declered but not defination , PROBLEM
}
class my_string{
   // @xxx  burda implicitly declered delete var yani -> my_string() = delete;
```

```
   my_string2 M2;
}

void main(){
     my_string m;    // bu durumda ne olur_?  my_string2 constractorun içeriği yok? ceva
}
```

special member functions → copy constructor : verdiğin nesneye aynı şekilde değerin içeriğini kopyalar yani, apayrı bir nesne yaratmak için tabi

```
class my_string{
 public:
 my_string()
 {

 }

 my_string(const my_string&)    // user defined copy constructor , bunu biz yazmasak der
 {

 }

 ~my_string()
 {
```

```
 }
}

void foo(my_string obj)
{
   // copy burda
}

/*void foo(const my_string& obj)          // bunla karıstırma bak yukardakini, tabikide
{

}*/

void main(){
    my_string m;   // my_string() çağrıldı
  foo(m)        // my_string(const my_string&) çağrıldı
}  // ~my_string() çağrıldı
```

```
void main(){

    my_string m;   // ctor
    my_string  m2 = m;  // copy ctor
    my_string  m3(m);// copy ctor
    my_string  m4{m};// copy ctor
```

```
    auto m5 = m;// copy ctor
    auto m6(m);// copy ctor
    auto m7{m}; // copy ctor

}
```

DERS 13

special member functions → copy assigment : bunuda derleyici yazıo tabiki

```
my_string m , mx;

m = mx;   // burda copy assigment çağrıldı, m in copy assigmentı ex:     m.operator = (mx
```

Olay şu önce **destructor**(~my_string) çağırıor sonra **copy constructor**(my_string(const my_string&)) çağırıor. KODu:

```
class my_string
{
 public:

/** constructor **/
 my_string(const char*p) : mlen(std::strlen(p)) , mp( static_cast<char*>(std::malloc(mlen
 {
    cout << "loaded" ;
```

```
    std::strcpy(mp,p);
 }

/** copy constructor **/
 my_string(const my_string& other): mlen(other.mlen) , mp( static_cast<char*>(std::malloc
 {
    std::strcpy(mp, other.p);
 }

/** copy assigment **/
 my_string& operator = (const my_string& other)   // bu c de legal olamaz çünkü C de "x =
 {
      if(this == &other)    // m = m;   kendisini kendisine eşitlersen diye KORUMA, yoksa
         return *this;

      /** destructor ***/
      cout << "freed copy" ;
      std::free(mp);

      /** copy constructor **/
      mlen = other.mlen;
      mp = static_cast<char*>(std::malloc(mlen+1)) ;
      std::strcpy(mp, other.p);

      return *this;
 }
```

```
/** destructor ***/
 ~my_string()
 {
    cout << "freed" ;
    std::free(mp);
 }

 int mlen;
 char *mp;
}
```

special member functions → move constructor : bunuda derleyici yazıo tabiki, R value alıo, kaynak çalıo , değerini değiştirebilio

special member functions → move assignment: bunuda derleyici yazıo tabiki, R value alıo, kaynak çalıo, değerini değiştirebilio

```
class my_string
{
 public:
        /** constructor **/
         my_string(const char*p);
        /** destructor ***/
```

```
          ~my_string();


        /** copy constructor **/
         my_string(const my_string& other);

        /** move constructor **/
         my_string(my_string&& other);       // not aynı zamanda copy constructor ve move

        /** copy assigment **/
         my_string& operator = (const my_string& other);

        /** move assigment **/
         my_string& operator = (my_string&& other); // not aynı zamanda copy assigment ve
}
```

my_string m1 = m2; // olay şu m2 için derleyici öyle bir karar veriyor ki m2 yi yok edip m1 e atabilior. O yüzden R value ve const değil girişi move ların

```
class m_class{
}
```

```
void func(const m_class& me)
{
}
void func(m_class&& me)
{
}

void main(){
    m_class m;
    func(m);                  // void func(const m_class& me) çağrılır çünkü L value
  func(m_class{});          // void func(m_class&& me) çağrılır çünkü R value (sonra öğre
  func(static_cast<m_class&&>(m)); // void func(m_class&& me) çağrılır çünkü R value işte
  func(std::move(m)); // void func(m_class&& me) çağrılır R oldu , YUKARDAKİNİN AYNISI !
}

YAN OZETI: Move doestn move, move L value yü R valueye dönüştürüo, R ı da R a !!!!
std::move "constexpr" bir func, compile timeda yani !!
std::move(x) = static_cast<m_class&&>(x)
```

Efective modern C++ oku

DİKKAT!!

```
class m_class{
}

void func(const m_class& me) // BAK bu hem R value hem L valueleri alabilio const olduğu
{
}
void func(m_class&& me)
{
}

void foo(m_class&& r){
  // yani bu "m_class&&" tür R veya L value değil, R veya L value olması için bişeyin ona
  func(r);  // void func(const m_class& me) ÇAĞRILIR -> isimlerin oluşturduğu ifadeler he
}

void main(){
    m_class m;
    foo(std::move(m));
}
```

EN ÇOK KLULLANILAN PATTERN=

```
class m_class{
    m_class() = default;

    mclass(const m_class&){
    // COPY
    }
    mclass( m_class&&){
    // MOVE
    }
}

void func(const m_class& me)
{
  m_class m(me);       // mclass(const m_class&) çağırdı
}

void func(m_class&& me)
{
  m_class m(std::move(me));  // mclass( m_class&&) çağırdı   , KAYNAK BURDA ÇALINDI
}


void main(){
    m_class m;
    func(std::move(m)); // MOVE
```

```
  func(m);  // COPY
}
```

NOTE NEDEN MOVE: diyelim senin classın içinde dinamic değişkenler var mesela string !!

eee bu classı yani objeni process sırasında kullanırken çeşitli eşitleme atamalar vs yapıyorsun, bi yere gönderiorsun objeni,

peki o zaman bu dinamik değişkenler sürekli her atamada **copy** yaparsa ne olur, verimi düşürür sürekli cop yaparsa. AMA move öyle değil. direk var olan adresi atanacak yeni objeye verip adresi çıkıo işin içinden. kopyalama yok,

ALL special member functions → derleyici böle yapıo

```
class my_string
{
 public:

/** constructor **/
 my_string(const char*p) : mlen(std::strlen(p)) , mp( static_cast<char*>(std::malloc(mle
 {
    cout << "loaded" ;
    std::strcpy(mp,p);
 }
```

```
/** copy constructor **/
 my_string(const my_string& other): mlen(other.mlen) , mp( static_cast<char*>(std::mallo
 {
    std::strcpy(mp, other.p);
 }

 /** move constructor **/
 my_string(my_string&& other) :
 mlen(std::move(other.mlen)) , mp( std::move(other.mp) )  // otherın destructorı çağrıla
 {
   other.mlen = 0;   // dikkat kaynağı çalınanı temizledik !!
   other.mp = nullptr; // unutma move yapmak destructor engellemez, bu yüzden böle yaptı
 }

/** copy assigment **/
 my_string& operator = (const my_string& other)   // bu c de legal olamaz çünkü C de "x =
 {
      if(this == &other)    // m = m;   kendisini kendisine eşitlersen diye KORUMA, yoks
         return *this;

      /** destructor ***/
      cout << "freed copy" ;
      std::free(mp);

      /** copy constructor **/
      mlen = other.mlen;
```

```
      mp = static_cast<char*>(std::malloc(mlen+1)) ;
      std::strcpy(mp, other.p);

      return *this;
}

/** move assigment **/
my_string& operator = (my_string&& other)   // BAK movelarda kopyalama yok görüyon demi
{
   if (this != &other) {

      /*** chat GPT : my destructor
         mlen  = 0;
         if(mp)
           std::free(mp);  //destructor
      **/


       mlen = std::move(other.mlen);
       mp = std::move(other.mp);


    /*** chat GPT : other fake destructor
         other.mlen  = 0;
         other.mp = nullptr;
      **/
```

```
    }
    return *this;
 }

/** destructor ***/
 ~my_string()
 {
    cout << "freed" ;
    if(mp)
      std::free(mp);
 }

 int mlen;
 char *mp;
}
```

```
// Senin yazdığın class
class my_class{

}

//Derleyicinin yazdığı class
```

```
class my_class{
  public:
    my_class() = default;
    ~my_class() = default;
    my_class(const my_class&) = default;
    my_class(my_class&&) = default;
    my_class& operator=(const my_class&) = default;
    my_class& operator=(my_class&&) = default;
}
```

EZBER TABLOSUNU EZBERLE DERS 14

DERS 14

move-only type class - copy'ler delete , movelar hayatta

```
class my_class{
public:
   my_class();
   my_class(const my_class&) = delete;              // bak unutma bu R da L de alıo
   my_class& operator = (const my_class&) = delete; // bak unutma bu R da L de alıo
   my_class(my_class&&);                            // bu olmasada olur , R alıo sadece
```

```
   my_class& operator = (my_class&&);               // bu olmasada olur , R alıo sadece
}
```

NOT: **ASLA MOVE memberları delete etme**, ANLAMSIZ. zaten move olmassa copy memberlara düşüo çağrıda “const class&” R ve L alması sayesinde. Ama move memberları silerseniz hata olabilir. Move member delete edersen copy memberlar otomatik iptal olmuş oluor.

copy var move yok

```
class my_class{
public:
   my_class();
   my_class(const my_class&);
   my_class& operator = (const my_class&) ;
  // move'u yazmana gerek yok, copy açık  , compiler kendi move ları delete edio, isterse
}
```

move-only type dada durum yukardakiyle aynı yani

```
class my_class{
public:
   my_class();
   my_class( my_class&&);
   my_class& operator = (my_class&&) ;
```

```
  // copy'u yazmana gerek yok, copy i compiler silio zaten, istersen elinle yap delete
}
```

temporary object - geçici nesne → PR value (r value)

```
class my_class{
    public:
    my_class()
    {
        std::cout << "create\n";
    }
    my_class(int x)
    {
        std::cout << "int create\n";
    }
     ~my_class()
    {
        std::cout << "delete\n";

    }
};

int main() {
//main 1 test
```

```
  {
    my_class x();
    std::cout << "test\n";
  }
/*********************/
//main 2 test
    my_class();                  //  VEYA my_class{} , my_class(6) , my_class{8}
    std::cout << "test\n";

/*********************/
//main 3 test
//ref @789
  {
    /** life extension **/
    const my_class& r = my_class{};    // main 1 test ile aynı !!! const ref obje kaptı d
    std::cout << "test\n";
  }
}
/**
main 1 test OUT:
create
test
delete
*****************************
main 2 test OUT:
create
```

```
delete                                  //  !!!!!!!!!!!!!! obje yaratıldığı gibi yok edilc
test
**************************
main 1 test OUT:
create
test
delete
*/

void func(const my_class&){  // veya void func(my_class me){
}
int main() {
   my_class m(12);  // Scope leakage
   func(m);         // BUNU YAPMA ARTIK , Scope leakage

   func(my_class{12});   // BUNU KULLAN -> mandatory copy elision özelliği C++ 17 den sor

   my_class& r = my_class{12};   // error , BU hata PR value olduğundan

   const my_class& r = my_class{12};    // BU çalışır R value consta bağlanır -> BU durur

   my_class&& r = my_class{12};         // BU çalışır R value bağlanır   aynı  ref @789
}
```

moved-from state , taşınmış nesne durumu

taşıma semantiği gereksiz kopyalamanın önüne geçer !!

```
std::string func( ){

}


int main() {

   std::string s = func();    // move assign çağrılır çünkü R value bak bu fonksiyon ve d
   std::string s = std::string(100000,'A'); // move assign çağrılır çünkü R value bak bu

   std::string s2
   {
      std::string sx(20000,'A');
      s2 = std::move(sx) ;   // move assign çağrılır
      // sx is in moved_from state : tam bu noktada sx hala hayatta!! sx } da desctroto
      // sx tekrar atama yapıp kullanabilirsin
    }// sx öldü

}
```

Örnek **moved-from state**

```
int main() {

   using namespace std;

   ifstread ifs{"dosya_yolu/notlar.txt"}
   string sline;   // satır satır tutacak
   vector<string> svec; // satırları tutacak

  // satır satır koşuyor
   while(getline(ifs,sline)){      // dikkat et sline sürekli dolduruluo her loopda
    svec.push_back(sline);         // dikkat burda copy yapıon sline ı vectore gereksiz
  }

  // DOĞRUSU
   while(getline(ifs,sline)){           // move'un sline ı yok etmemesi sayesinde bu işle
    svec.push_back(std::move(sline));   //  işte şimdi gereksiz kopyalama önlendi ama sl
  }
}
```

conversion constructor : dönüştüren kurucu işlev

```
class my_class{
    public:
    my_class()
    {
        std::cout << "create\n";
    }
    my_class(int x)
    {
        std::cout << "int create\n";
    }
     ~my_class()
    {
        std::cout << "delete\n";

    }
};

int main() {

   my_class mx;  //@x145

   std::cout << "test1\n";

   mx = 5;   // hata yok -> my_class(5) -> @x948 öldü   , BUNLAR TEHLIKELI İŞLERR YAPMA E
```

```
   std::cout << "test2\n";
}

/**
out:
test1
int create
delete      -> @x948 öldü     , yani derleyici geçici başka bir nesne oluşturdu ve move y
test2
delete      -> @x145 öldü
**/
```

function overload resolution about **conversion constructor**

bunlar kaybediyor sırayı:

- variadic conversion
- user-defined conversion - type cast operator functions - pointerdan boola dönüşüm felan , BAZEN BU DÖNÜŞÜMLER TEHLIKELI OLUO,ONLEMENIN YOLU → **explicit ctor**
- standart conversion - double dan inte felan

explicit ctor : conversion constructor önler. derleyiciye akıllı ol işimize karışma onu bunu dönüştürme demek demek. Örtülü dönüşüm yapma demek !! , tek elemanlı ctor ları her zaman **explicit yap. !!**

```
class my_class{
    public:
    my_class()
    {
        std::cout << "create\n";
    }
    explicit my_class(int x)
    {
        std::cout << "int create\n";
    }
     ~my_class()
    {
        std::cout << "delete\n";

    }
};

int main() {
   my_class   MX;
   MX = 5;    // bak conversion constructor da bu geçerliydi artık HATA thanks for explic
   MX = static_cast<my_class>(5); // BU hilesi hata yok !!

   my_class m{5};   // SIKINTI YOK
```

```
   my_class m2{5.6};   // HATA !!!
}
```

NOT: bir sınıfın tek parametreli ctor larını (aksi yönde bir durum olmadıkça) HEP **explicit YAP !!!!**

explicit ise copy assigmentlar felan olmuor

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

};

my_class foo(){
   return {6,7};  // HATA Yok ama my_class explicit olsaydı hata olurdu !!!
}
```