lessons 51 - 54

DERS 51

```
struct nec{
   int a = 5;
}

int main(){
   nec nec_obj;
   int * ptr = &nec_obj.a;   // ilgili objenin adresi

   int nec::*ptr = &nec::a;   // dikkat! bu objenin türü farklı
   nec nec_obj;
   nec_obj.*ptr = 5;    // işte böyle kullanılır !! a 5 oldu nec
}
```

GÜZEL ÖRNEK:

```
struct OHLC{ // open high low close , USDT/TRY mum grafiği datas
   double open;
   double high;
   double low;
   double close;
}

double get_avarage(const std::vector<OHLC>& candles, double OHLC
{
   double sum{};
   for(const auto& candle : candles){
     sum += candle.*select_dp;
   }
   return sum / candles.size();
}
```

```
int main(){
   std::vector<OHLC> myvec;
   // fill it
   auto mean1 = get_avarage(myvec, &OHLC::low);
   auto mean2 = get_avarage(myvec, &OHLC::high);
}
```

`std::bitset`, C++ programlama dilinde standart kütüphanenin bir parçası olan bir sınıftır. Bu sınıf, sabit boyutlu bit kümelerini temsil etmek için kullanılır. `std::bitset` sınıfı, önceden belirlenmiş bir boyuta sahip bit kümelerini ele alır ve bir dizi bit işlemi yapmayı sağlar.

SMART POINTERS

- EMBEDDED için kritik!! **new** ve **delete** overload ediliebilinir!!

ama !! : derleyici new ve delete expresiionlarını bilio dolayısıyla bu expressionları çağırdığınızda derleyici gidip sınıfın constractor veya destroctorını çağırıp allocation yapma işini kontrol edebiliyor. Bunları overload etsekte unutmaki özel expressionlar. (dolayısıyla malloc ve free den çok varklı, malloc ve free de constractor destroctor çağırma yok!!)

Örneğin :

- new : önce alanı tahsis edip allocation yapar(işte burası overload edilebilinir) sonra sınıfın constractorunu çağırır
- delete : önce destroctor çağırır sonra tahsis edilmiş alanı siler(işte burası overload edilebilinir)

```
void operator delete(void* vp){
   // ARENA allocator ün free si burda !
   asos_free(vp);
}
void* operator new(std::size_t n){
   if(n > 5000){
     throw std::bad_alloc{};   // buda opsiyonel
   }
```

```
   // ARENA allocator ün mallocu burda !
   return asos_malloc(n);
}

// main
my_class x = new my_class(); // önce malloc yaptı sonra constra
delete x;   // önce destroctor çağrıldı sonra free yaptı!
```

set_new_handler(void (*fp)(void))

get_new_handler();

aslında standart kütüphanenin new fonksiyonu tam olarak throw bad alloc yapmıo. sizin kontrolünüzde. Bu şekilde gerçek kod:

```
using new_handler = void (*)(void);

void* operator new(std::size_t n){   // standart kütüphanedeki n
   for(;;){
     void *vp = std::malloc(c);
     if(vp){ return vp; }
     new_handler fptr = get_new_handler();
     if(fptr){ throw std::bad_alloc{} }
     fptr();
   }
}

void my_new_error_capture(void){
   // new allocation işlemi başarısız oldu burda! ne yapılır?
   // - bir şekilde super loopda denediğinden allocationu başarı
   // - set_new_handler(nullptr) yap exaption yapsın
   // - bad_alloc dan kalıtım yoluyla kendi sınıfını yarat ve or
}

int main(){
   // set_new_handler çağırmazsan default da nullptr yani bad_al
   set_new_handler(&my_new_error_capture);
```

```
   //  .... kodlar  ...  new x....
}
```

DERS 52

new my_class[N];

delete[] ;

bu yukardakilerde normal new delete, sadece compiler N * sizeof(my_class) kadar yer tahsis etmesini istiyor operator functions olan new ve delete için. [] varsa yani kullanılmışsa delete de new de [] olmalı !!

placement new: new (x , y z) my_class; yani operator new functionuna fazladan 3 parametre geçtik.

standart kütüphane içinde var bir çok overload: ama sadece bu fonksiyon overload edilemez özel:

void * operator new (size , void*ptr); **ÇOK ONEMLİ ⇒**

```
class my_class{}

int main(){
   unsigned char buffer[sizeof(my_class)];
   my_class x = new(buffer) my_class; // çağrılan: void * operat
// YANI gitti arena allocate edilen yerin adresini tuttu !!!! HA

   delete x; // ASLA YAPMA;
   x->~my_class(); // BUNU YAP !!
}

// istersem kendim overload allocate ederim
void* operator new(std::size_t n , int x , int y , int z){
 // ....
}
my_class test = new(5,6,7) my_class; // oldu
```

not: newlere size_t yi hep derleyici koyuyo sen değil !!

excaption throw değil **null** dönmesini istiosan C deki gibi:

```
my_class x = new(nothrow) my_class;
if(cx == nullptr){}
```

NEW (ham new çağırmak) sakıncalı neden:

- ilgili nesne pointer, → ile ulaşılıo. dinamik öğe mi static öğe mi belli değil çok sıkıntı! sonuçta pointer.
- nullptr mı belli değil çok sıkıntı!
- delete i ben mi kontrol etcem belli değil. hangi delete bu mu [] yoksa [] yok mu ? belli değil.
- delete edersem Dangling pointer olur mu belli değil. yani ilgili pointerı başka pointer kullanıyorsa?
- yada sen delete edemeden önce senden bağımsız başka fonksiyonlar exaption throw ederse , sen delete edememiş olcan !!
- karmaşıklık seviyesi artar (garbage collector dilde yok c# java gibi değil)

ÖNEMLİ NOT: Embedded: Customizing Dynamic Memory Management in C++ - Ben Saks - CppCon 2020:

Kendi ARENA allocatorını yaratıyorsan eğer, new [] ve delete [] işlemlerini yönetmek zorlayıcı: Bu yüzden bunu yap →

```
class my_class{

   void *operator new(size_t n);
   void operator delete(void *pv)noexcept;
   void *operator new[](size_t n) = delete;             // [] bu
   void operator delete[](void *pv)noexcept = delete;
}
```

TÜM BU SEBEPLERDEN HER ZAMAN SMART POİNTER KULLAN !!

hem nesnelerin hem kaynakların (classların) ömrünü yönetir. extra maliyeti nerdeyse yok, extra işlem maliyeti yok HEP KULLAN !!

unique_ptr : kopyalamaya kapalı pointer, başka kimse sahip olamaz. ama taşınabilir tabi. **exclusive ordership** (tek sahipçilik) (EN ÇOK KULLANILAN)

shared_ptr : paylaşımlı pointer. birden fazla pointer nesneyi gösterir AMA en son delete olan delete olunca nesne gerçekten ölür. Dangling pointer önüne geçer yani!!

(**weak_ptr** , shared_ptr'ın yardımcısı)

```
class my_class{
   int x;
}

int main(){
   std::unique_ptr<my_class> ptr;  // BU boş
   if(ptr){}  // dolu mu boş mu kontrolü, boşsa SAKIN x e ulaşma

//**************************
   {
      std::unique_ptr<my_class> ptr{ new my_class }; // DOLU
      foo();
   }  // PTR nin hayatı burda otomatik bitti !! foo(); exaption

 // ******************************

   auto ptr = std::make_unique<my_class>();  // std::unique_ptr<
}
```

```
class my_class{
   int x;
}

int main(){

   auto ptr = std::make_unique<my_class>();
  // burda ptr DOLU !!
```

```
   auto ptr2 = std::move(ptr); // ptr2 = ptr;  SENTAKS HATASI OL
  // burda ptr BOŞ (çalındı mülkiyeti) , ptr2 DOLU
   auto ptr3 = std::make_unique<my_class>();
   // BURDA ptr2 VE ptr3 DOLU
   ptr3 = std::move(ptr2);
   // ptr3 boşaldı ve ptr2 yi aldı ve ptr3 DOLDU, ptr2 boşaldı
}
```

```
std::unique_ptr<my_class>  bar(){
    auto ptr = std::make_unique<my_class>();

  return ptr;  // burda move YAPMA, pesimistik yaklaşım olur, de
}

void foo(std::unique_ptr<my_class> x){

}

int main(){
   auto up = std::make_unique<my_class>();

   foo(up); // HATA !!
   foo(std::move(up))   //  LEGAL GEÇERLİ !!
   foo(std::make_unique<my_class>())  // LEGAL GEÇERLİ !!
   foo(std::unique_ptr<my_class>());  // LEGAL GEÇERLİ !!


  up.reset();     // up BOŞALDI !!
  up = nullptr;   // up BOŞALDI !!
  up = {};   // up BOŞALDI !!

  up.reset( new myclass() );  // işte böylece unique ptr yeniden
  up = std::make_unique<my_class>();   // yukardakiyle arasında

  my_class *t = up.release();  // bu kaynak free olmadı ama içi
```

```
  auto y = std::move(up);  // destroctor tabiki çağrılmadı

  std::unique_ptr<my_class> y(up.release()); // auto y = std::mo
}
```

HATALAR:

```
int main(){
    auto *p = new my_class;

    std::unique_ptr<my_class> ptr{ p };
    std::unique_ptr<my_class> ptr2{ p };   // HATA !! p 2 farkl
}
```

DERS 53

smart pointerın **delete** ini özelleştirelim

```
//c++ 17 öncesi
struct my_deleter{
   void operator()(std::string* p) const noexcept
   {
      std::cout << "my delete works";
      delete p;
   }
}

 void my_other_delete(std::string* p)
 {
      std::cout << "my delete works";
      delete p;
 }

int main(){
// aşağıdakilerin hepsi aynı işi yapar !!
```

```
    std::unique_ptr<string, my_deleter> uptr{ new string{"test"}}

    std::unique_ptr<string, decltype(&my_other_delete)> uptr{ new

    std::unique_ptr<string, void(*)(std::string*)> uptr{ new stri


    auto f_del = [](string* p){
       std::cout << "my delete works";
       delete p;
     };

    std::unique_ptr<string, decltype(f_del)> uptr{ new string{"te

    std::unique_ptr<string, decltype(f_del)> uptr{ new string{"te
}
```

SMART POINTER SADECE new ve delete den ibaret değildir !!!: AŞAĞIDAKİ KODU İYİ ANLA SMART POINTER HER YERE UYGUN, POINTER OLAN !!

```
// C api si, handle/obj pattern uygun
struct data_x{
   int a;
};

data_x* create_api();
use_api_for_somothing(data_x *me);
delete_api(data_x*me)

int main(){
    // c / c++ de ki normal kullanım
    data_x obj = create_api();
    use_api_for_somothing(obj);  // bu arada exeption throw olu
    delete_api(obj);

   // ama artık yukardaki işi yapma !!SMART POINTER kullan
```

```
   auto x_delete = [](data_x * me){    delete_api(me);  };
   std::unique_ptr<data_x , decltype(x_delete)>   obj( create_a
   use_api_for_somothing(obj.get());
}  // hayatı burda sona erdi
```

array tipi smart pointerlar;

```
class my_class{  }

int main(){

   std::unique_ptr<my_class>   ptr( new my_class[4] ); // UB , H

   auto x_delete = [](my_class * me){    delete[] me;  };
   std::unique_ptr<my_class, decltype(x_delete)>   ptr( new my_c

   std::unique_ptr<my_class[]>   ptr( new my_class[4] ); // HEP
   auto ptr = make_unique<my_class[]>(4) // veya HEP BÖYLE YAP
}
```

Bu hatayı yapma doğru anla

```
int main(){
   std::Date* p = new Date{4 , 8 , 1995};
   {
      unique_ptr<Date> upx(p);
   }  // burda p delete oldu
   unique_ptr<Date> upy(p);  // dangling pointer oldu HATA !!!!
}
```

shared_ptr , unique_ptr içinde var, eşitlenebilinir !! shared_ptr iconstractorlardan biri unique_ptr alıor.

kavramsal analiz: shared_ptr ilgili pointerın 4 bytılık adresini sarmalarken ekstra 4 bytelıkda kontrol bloğu allocate(new) eder. bu control bloğunun içinde deleter function ilgili adres ve counter vardır. her yeni shared_ptr objesinde bu işlem

yapılır. Dolayısıyla aynı adresi gösteren shared_ptr objeleri içindeki control bloğunda counter artar ve azalır. Son shared_ptr ilgili pointerın hayatını bitirebilir ancak.

```
int main(){
   std::shared_ptr<Date>   sptr( new Date(5,7,1785));// control
// tabi burda derleyici optimizasyon yaptı ve tek seferde Date s
// eğer zaten allocate edilmiş bir data ile shared ptr yaratırsa

   auto xptr = sptr; // counter 2 oldu
   {
      auto xptr2 = xptr;   // counter 3 oldu
   }// counter 2 oldu

   std::cout << xptr.use_count();    // counterı bunla gör

   // kendi deletimizi yazmak istersek, constactorına parametre
   std::shared_ptr<Date>   sptr( new Date(5,7,1785) , my_delete_

}// tüm hayatlar burda bitti
```

HATALAR:

```
int main(){
    auto *p = new my_class;

    std::shared_ptr<my_class> ptr{ p };
    std::shared_ptr<my_class> ptr2{ p };   // HATA !! p 2 farkl

  std::shared_ptr<my_class> ptr2{ ptr }; // böyle yapcan bunda s
}
```

DERS 54

std::weak_ptr: shared_ptr nin yardımcısı. shared ptr den türer. counter arttırmaz. amacı gözlemcilik. **amaç**: shared_ptr nin hayatını kontrol etmek. kaynağı

shared_ptr verdimi vermedimi weak_ptr kontrol edebilir. weak_ptr kaynak geri vermez sadece gösterici / sınayıcı gibi düşün.

```
std::shared_ptr<Date>   sptr( new string("hello"));

std::weak_ptr wp1 = sptr;  // shared_ptr counter 1
std::weak_ptr wp2 = sptr;  // shared_ptr counter hala 1

// weak_ptr dereferans edilemez

if( wp1.expire() ){
  // kaynak yani shared_ptr sonlandırılmış(reset olmus/ kaynağı
}else{
 // ilgili shared_ptr dolu/ hayatta
}

auto test = wp1.lock();// shared_ptr eğer hayattaysa shared_ptr
if( test != nullptr ){
  // ilgili shared_ptr hayatta
  // ve counterı 1 arttı "test" artık shared_ptr yi paylaşıor
}else{
 // ilgili shared_ptr hayatı bitmiş
}
```

BU HATAYA DİKKAT !!! :

```
class dog{
 // std::shared_ptr<dog> my_friend;   @7845  ÖLÜMCÜL HATA OLURDU
// eğer ana root obje dışında hyaratılmış ise genelde obje içinc
  std::weak_ptr<dog> my_friend;  //  @9845 böyle yap
  std::string<dog> name;

  dog(std::string _name){name = _name;}

  void make_friend(std::shared_ptr<dog> x){
      my_friend = x;
```

```
   }

  void show_my_friend( )const{
       if(!my_friend.expired()){  //shared_ptr hayata gelmiş mi
          std::cout << my_friend.lock()->name; // gerçek shared_
       }
   }
}

int main(){
    std::shared_ptr<dog> sp1(std::make_shared<dog>("name1"));  /
    std::shared_ptr<dog> sp2(std::make_shared<dog>("name2"));

    sp1->make_friend(sp2);
    sp2->make_friend(sp1);  // eğer  @9845 değil @7845 açılsaydı
// bir birlerini loopa sokar ve counter hiç bir zaman 0 olmaz ve
}
```

CRTP, "Curiously Recurring Template Pattern" ifadesinin kısaltmasıdır ve C++ dilinde bir şablon desenidir. Bu desen, türemiş sınıfın kendi türünü şablondan elde etmesine dayanır. CRTP, şablon parametresi olarak kendi türünü (türemiş sınıfın türünü) kullanarak şablon özel durumu oluşturur.

CRTP'nin temel fikri, bir temel sınıfın şablonlu bir türe sahip olması ve türetilmiş sınıfın bu temel sınıfı şablondan kalıtmaktır. Bu, türetilmiş sınıfın, temel sınıfın şablon parametresini kendi türüyle özelleştirmesine olanak tanır.

CRTP:

```
// CRTP base       -- kullanıcıya bilgi vermek için yaz
template <typename Derived>
class Base {
public:
    void foo() {
        /* SUPER OLAY: taban sınıf türetilmiş sınıfa bu şekilde
        static_cast<Derived*>(this)->implementation();
// veya
```

```
        static_cast<Derived&>(this).implementation();
    }
};

class DerivedClass : public Base<DerivedClass> {
public:
    void implementation() {
        // code ..
    }
};
```

STD yi **CRTP destekler ⇒**

```
class nec : public std::enable_shared_from_this<nec>{  // çözüm

  void foo()  // problem şu, @ref475 nin kopyasını bu fonksiyon
  {
    std::shared_ptr<nec>  sptr2(this);  // ÖLÜMCÜL HATA aynı adı

    auto spx = shared_from_this();
    spx.use_count();   // işte şimdi burda @ref475 counter 1 dah
  }
}

int main(){
   std::shared_ptr<nec>  sptr(new nec); // @ref475
   sptr->foo();
}
```

CRTP Aynı dinamik ömürlü nesneyi birden fazla konteynırda tutabilmemizi sağlar !! Ve konteynırlardada farklı işlemler yapmamızı sağlar .

NEW ve DELETE classlar için Önemli Özellik : EMBEDDED için önemli

```
class my_class
{
```

```
   static void* operator new(size_t n){}   // ÇOK ONEMLİ! bu yaz
   static void operator delete(void* vp)noexcept{} // ÇOK ONEMLİ
}

int main(){
   my_class * obj = new my_class;
   delete obj;


   my_class * obj = ::new my_class;  // NOT: bu global new overl
   ::delete obj;  // globali bulur
}

// dolayısıyla global new ve delete operator overload functionla
// class içinde var ise çağrılmaz!! ve classdan classa özelleşti
```

Embeddedda Kullanılır:

```
class my_class
{
   static void* operator new(size_t n){
     // mbuf ın kullanılmayan yerlerini mflag lere bakarak bul
   }
   static void operator delete(void* vp)noexcept{
      // ilgili adresin mflag ını temizle
   }

   // böylellikle heap hiç kullanılmamış oldu !!
   static unsigned char mbuf[1000];
   static bool mflag[1000];
}
```