lessons 40 - 50

DERS 40

LAMBDA EXPRESSION:

- derleyiciye sınıf yazdırır ve geçici objeye dönüştürülür(PR value)
- dönüştürülen obje: closure object , tipi : closure type
- std algoritmalarına algoritmalarına argüman göndermek sıklıkla kullanılan yöntemi
- C++ 20 ye kadar lambda classların constractorları delete statusundeydi C++ 20 den sonra değişti (ÖNEMLİ FARK)
  lambda syntax logic sırası:
  [] : lambda introducer , içine yazılan ifadelere: lambda captures denir
  () : derleyisınıfı yazdı, bu ifade varsa içine function yazacak , function çağrı operator fonksiyonunun parametre parantezi bu oluyor !!
  içine normal fonksiyon gibi parametre değişkenleri yazılır, içi boşsa yazılmasada olur
  {} : sırada gelen parantezlerin içine fonksiyonun kodu yazılır

örnek kod:

```
int main(){
[]{}();     // hiç bişey yapmayan kod :) , sondaki () kadar laml
}
```

yukardaki expresion için yazılan kod:

```
class xyz555_845{           // verilen isimden derleyici soru
mlu
public:
    void operator()()const{  // derleyici defult const yapar
    }
}
```

```
int main(){
    xyz555_845{}();   // {}();   bu oldu işte :)
}
```

Bunlar gelebilir :

- {/** code **/} // eğer func parametre almıosa : []{} böylede yazılır , derleyici anlıor
- constexpr{/** code **/} // default da dönüşler constexpr fonksiyonların eğer derleyici aksini anlamazsa, bu sebeple kendimizi const olmasını isteyip garanti ediyoruz, normalde zaten constexpr !! kendimizi koruyoruz yani aksi durumda sentaks hatası olsun diye
- noexcept{/** code **/} // LAMBDA excaption dönmicek demek
- mutable{/** code **/} // LAMBDA introducer içinde ki tanımlanan veri mutable olcak, değiştirilebilir (() operator overload func hep const olduğundan)
- []{/** code **/} // değer yok () içinde kaldırabilirsin
- > type{/** code **/} // LAMBDA fonksiyonun geri dönüş typı değiştirilebilinir

examples:

```
int main(){
    [](int x){ return x*x; } // fonksiyon operator çağrı fonksiy
    [](int x){ return x*x*0.5; } // dönüş double oldu mesela

    auto foo = [](int x){ return x*x; }; // foo const ta olur ->
    int test = foo(8); // test = 64 oldu
}
```

exmaples , en sık kullanım:

```
template <typename F>
void func(F foo){
```

```
  auto x = foo(120);
}

int main(){
  func( [](int x){ return x*x; } );
}
```

examples:

```
int main(){

    auto value = [](int x){ return x*x; }(4);   // value nin ti
}
```

- statik ömürlü nesneleri lamda fonksiyonları içinde direk kullanabiliyoruz
  examples:

```
int main(){

    int t_x = 5 ;
    auto value = [](int x){ return x * t_x; };   // HATA, local

    static int t_x = 5 ;  // Globalde olsa olur
    auto value = [](int x){ return x * t_x; };   // DOĞRU

    // diğer çözüm: capture close
    int t_x = 5 ;
    auto value = [t_x](int x){ return x * t_x; };   // DOĞRU, @4

    auto test = [t_x](){ ++t_x; }; // sentaks hatası çünkü const
    auto test = [t_x]()mutable{ ++t_x; };   // işte böyle yapars
}
```

@4878' nin kodu:

```
class xyz555_845{
    public:
    xyz555_845(int r):mx(r){}
    void operator()()const{     // dikkat bunu derleyici const y



    }
private:
     int mx;
}
```

super ozellik:

```
int main(){

    int a = 5 ,b = 9 , c = 7;
    auto test = [a,b,c](int x){ return x*a*b*c }; // doğru soru
    auto test = [=](int x){ return x*a*b*c }; // yukardakiyle ay

}
```

```
int main(){

    int a = 5;
    auto test = [a]()mutable{ a += 5; };
    test();
// a hala burda 5
    auto test = [&a](){ a += 5; }; // mutable a gerek yok çünkü
    test();
// a şimdi burda 10 olur

}
```

```
int main(){

    int a = 5 ,b = 9 , c = 7;
    auto test = [&](int x){ return x*a*b*c }; //local deki tüm d
    auto test = [=,&a](int x){ return x*a*b*c }; //şimdi sadece
    auto test = [&,a](int x){ return x*a*b*c }; // şimdi hepsi r
}
```

```
int main(){

    auto test = [](int x)->double{  // normal de int yapcaktı ,
                  return x*x;
                };
}
```

derleyici yazdığı lambda class'ın fonksiyonunu template ile yazması için , C++14:

```
int main(){

    // @4485 : burda derleyicinin yazdığı temsili class
    auto test = [](auto x)->double{  // artık classın fonksiyonu
                  return x*x;
                };
}

class xyz555_845{
    public:
    template<typename T>
    void operator()(T x)const{     // @4485
        return x*x;
    }
}
```

Stateless Lambda: (veri elemanı yok, herhangi bir capture içinde elemanı yok) , C++20 de etkili oluyor constractor delete edilmediği için lambda classta

decl
positive lambda:

```
int main(){

      auto test = [](auto x)->double{  return x*x;  };  // bunu
    auto tesx = +[](auto x)->double{  return x*x;  }; // bu + sa
    decltype(test); // türü çıkarır, C++20 de bunu decltype bile
}
```

C++20 supanallah iş yapmış, mesela double float etc için
template<double x> // BU c++ 20 ye kadar hata, çünkü sistemden sisteme bunun formatı parse etme şekli değişebiliyordu, IEEE754 zorunlu oldu ve sorun çözüldü! C++20

```
template<auto x>
class my_class{
}

int main(){

    my_class<[]{}>  m;  // sentaks hatası yok C++ ile, yukardaki
}
```

DERS 41

algoritmalara devam

- any_off , all_off , non_off
- find_if
- remove_copy , reverse_copy , replace_copy (bunların copy yazmayanlarıda var, copy ler bi yere kopyalayıp işlemleri o şekilde yapıyor !)

Not: çoğu loop içeren algoritmada mantık aynı, ilk değer iteratorun .begin(), ikinci değer .end() üçüncü ise function pointer ile func istio yani LAMBDA
tabiki geri dönüşleri farklı veya içincü parametrede conteiner ve dördüncüde LAMBDA olanlarda var

Konteynırlar , veri yapıları , vector en iyisi!!

std:array - fixed array - C de ki dizinin aynısı!! , interface i güzel sadece, veya standart vectorle mesela bazı functionları mesela iterator ?

- insert
- push_front
- push_back
  Yukardakilerden birini vectorde veya diğer konteynırlarda çağırırken dikkatli ol copy çağrılıyor çünkü genelde ama sen içinde std::move yaparsan kaynak çalınır ama kopyalama olmaz işlem maliyeti düşer !!! tabi algoritmana bağlı !!

C++'da `push_back` ve `emplace_back` vektör sınıfının eleman eklemek için kullanılan iki farklı yöntemdir. İkisi de vektörün sonuna yeni bir eleman eklemek için kullanılır, ancak farklı şekillerde çalışırlar. **MÜLAKATLARDA FARKI ÇOK SORULUR**

```
std::vector<int> myVector;
myVector.push_back(42);   // COPY VEYA MOVE İLE İLETİLİR
/***********/
struct MyStruct {
    int value;
    std::string text;
    MyStruct(int v, const std::string& t) : value(v), text(t) {
};
std::vector<MyStruct> myVector;
myVector.emplace_back(42, "Hello");  // Elemanı doğrudan vektör
```

Kısacası, `push_back` elemanın kopyasını vektöre eklerken, `emplace_back` elemanın kendisini doğrudan vektör içine oluşturarak ekler. `emplace_back` genellikle performans açısından daha etkili olabilir, özellikle elemanın oluşturulması için özel bir yapılandırıcı veya sınıf fonksiyonu kullanılıyorsa.

Özünde → @784545: (tüm emplace ve push(insert) içeren std algoritmaları farkı bu)

```
template <typename F>
class Vector{
   void push_back(const T& r){}            // R de alır L de al
   void push_back(T&& r){ std::move(r); }  // R alır , adresin

  template <template ..args>
  void emplace_back(args&& ...args)   // işte bu üniversal value
  {
    T(std::forward<args>(args)...);  // yani ilgili base vector
  }
}
```

DERS 42

vector devam

ranged base for loop: iteratore açılıyor

```
for( auto x: expr(container .. ) )  // copy , @dd7874

for( const auto& x: expr(container .. ) ) // adres taşıyor , per

for(auto iter = r.begin() ; iter ≠r.end() ; ++iter){
    auto x = *iter;// @dd7874 yukardakinin derleyicinin çevirdi
}
for(auto iter = r.begin() ; iter ≠r.end() ; ++iter){
    const auto &x = *iter; // @789785 yukardakinin derleyicinin
}
```

DERS 43

std devam

DERS 44

STD devam - deque
sıralama algoritmaları.

DERS 45

- sıralama alg, binary tree vs ..
- list

DERS 46

- kuyruk, stack, set ...

DERS 47

`std::set` C++ dilinde bir standart kütüphane koleksiyon türüdür ve sıralı, benzersiz elemanları içeren bir veri yapısı sunar. `std::set` içindeki elemanlar otomatik olarak sıralanır ve her eleman yalnızca bir kez bulunabilir. arka planda **binary tree** yapıor. `std::multiset` yukardakinden farkı benzersiz değil yani aynı değerden birden fazla olabilir içinde.

`std::map`, C++ standart kütüphanesinde bulunan bir veri yapısıdır ve bir anahtar-değer çiftlerini depolayan bir ilişkisel konteyner türüdür. Her bir anahtar, benzersiz bir değere eşlenir.**multimap** benzersiz değil. buda **binary tree.**

'std::unordered_map ve unordered_set' hash tabledır. hashtable da hash ler indexe dönüşüor. stringi sayıya dönüştürmek olay. hashtable avantajı: "Constant time" (sabit zaman) da bulunur veri.

`std::hash` C++ standart kütüphanesinde bir özel sınıftır ve hash fonksiyonları oluşturmak için kullanılır. Bu sınıfın örnekleri, belirli bir veri tipinin (örneğin, bir sayı, bir dize veya özel bir tür) hash değerini üretebilir.

DERS 48

functinal std.

`std::reference_wrapper` sınıfı, başka bir nesnenin referansını kapsayan bir işlev nesnesi (wrapper) sağlar. Bu sınıf, referansları işlev nesneleri aracılığıyla geçirme ve işlev nesneleri içinde saklama gibi senaryolarda özellikle yararlıdır. `std::reference_wrapper` C++11 standardı ile birlikte eklenmiştir ve özellikle öğe taşıma olasılığı olmayan bağlamlarda kullanışlıdır.

`std::ref`, C++ standart kütüphanesinde yer alan bir işlevdir ve referansları işlev nesneleri aracılığıyla geçirmek için kullanılır. Bu işlev, `std::reference_wrapper` sınıfını

kullanarak bir referansın işlev nesnesine (function object) dönüştürülmesini sağlar.

Not: `reference_wrapper` içerde pointer taklidi yapılıyor.

```
int &x[20]; // HATA referansla dizide yapamazsın
std::vector<int&> x; // HATA bu hatayı önlemek için:
std::vector<std::reference_wrapper<int>> x;

int ival = 50;
std::reference_wrapper<int>  hold = std::ref(ival);
int k = hold.get();
//std::reference_wrapper<int> hold = &ival;  BU HATA TABIKI
```

DİKKAT: diyelim senin fonksiyonun adres almıyor. ilgili input veri de çok büyük boyutu. Copy by value den kaçınmak için std::ref yap !!

```
struct data{
 char x[1000000];
}

void foo( data x ){
}

int main(){
  data d;
  foo(d);  // YAPMA MALIYET ÇOK
  foo(std::ref(d)); // SUPER BOYLE TAŞI , std::cref -> const olu
}
```

`std::bind`, C++'ın standart kütüphanesinde yer alan bir öğedir ve işlev nesnelerini ve fonksiyonları bağlamak (bind) için kullanılır. `std::bind` işlevi, bir işlev veya işlev nesnesi çağrısı sırasında belirli parametreleri veya parametre değerlerini bağlamak için kullanılır, böylece işlevin çağrılması önceden belirlenmiş parametrelerle yapılabilir.

```
int foo(int x , int y , int z){}

int main(){
   foo(20,30,40);

   auto f = std::bind(foo, 20 , 30 , 40);
   f(); //  foo(20,30,40); demek !!

// SUPANALLAH :
  auto fx = std::bind(foo, std::placeholders::_1 , 30 , 40);
  fx(50); // foo(50, 30 , 40); demek !!

 auto fx = std::bind(foo, std::placeholders::_1 , std::placehol
  fx(50); // foo(50, 50 , 50); demek !!

 auto fx = std::bind(foo, std::placeholders::_1 , std::placehol
  fx(50,40); // foo(50, 40 , 40); demek !!

 auto fx = std::bind(foo, std::placeholders::_1 , 10 , std::pla
  fx(50 , 40); // foo(50, 10 , 40); demek !!
}
```

DERS 49

`std::function` C++ standard library template sınıfıdır ve genel olarak fonksiyon nesnelerini (function objects) ve fonksiyon göstericilerini taşımak için kullanılır.

```
class nec{
   void test(int)const{}
   static void stest(int){}
}
void foo(int x){
}
void bar(int x){
}
```

```
int main(){

std::function<void(int)> f = foo;// DOĞRU

f = bar; // DOĞRU

f = nec::stest;  // DOĞRU

f = nec::test; // büyük HATA !!!! gizli me var !!! olmaz !!
//doğrusu;
// SUPER BİLGİ !!!!!!!!!!!!!!!!!
std::function<void(const nec&,int)> fx =  &nec::test; // DOĞRU
}
```

```
void bar(std::function<int(int>)); // ÇOK GÜÇLÜ

int f1(int);

struct nec{
   static int f2(int);
}

struct f3{
   int operator()(int)const;
}

int test(int , int , int);

int main(){

    auto f5 = [](int x){return x*x;};

    auto f6 = std::bind(test, std::placeholders::_1, std::placeh

    bar(f);        // hepsi doğru !!
```

```
    bar(nec::f2);
    bar(f3{});
    bar(f5);
    bar(f6);
}
```

std::array anlatıldı, dinamik değil, stack de aynı array

```
"test123"  -> const char*
"test123"s -> std::string
```

DERS 50

`std::tuple` C++ programlama dilinde, birkaç farklı türdeki değerleri tek bir veri yapısı içinde gruplamak için kullanılan bir şablondur.

```
std::tuple<int, double, std::string> myTuple(42, 3.14, "Hello");

auto myTuple = std::make_tuple(42, 3.14, "Hello");
```

classların fonksiyon pointerlar: ÇOK ÖNEMLİ , .* ve ->* operatoru

```
class nec{
  int foo(int , int);
  int foo2(int , int);
  static int baz(int, int);
}

int main(){
   int (nec::*fp)(int , int) = &nec::foo;
   fp = &nec::foo2;

   nec x;
   fp(&x, 12 , 15);  // HATAA !! olmazz..
   (x.*fp)(12,15);   // işte böyle çağrılır, yeni operator => .
```

```
   int (*fptr)(int, int)= &nec::baz;
}
```

```
class nec{
  int foo(int );
  int foo2(int );
}

using nec_fp = int (nec::*)(int);

int main(){
   std::vector<nec_fp> my_vec { &nec::foo };

   my_vec.push_back(&nec::foo2);

   nec x;

   for(auto fp : my_vec){
      (x.*fp)(10);
   }
/*******/
   nec *y = new nec; // pointerda iş biraz değişir

   for(auto fp : my_vec){
      ((*y).*fp)(10);   // çok karışık !!! ama doğru, kolay yol
   }

   for(auto fp : my_vec){
      (y->*fp)(10);   // yukardakinin aynısı, daha okunaklı oldu
   }

   for(auto fp : my_vec){
      std::invoke(fp, y , 10)   // yukardakinin aynısı, daha oku
```

```
   }
}
```

yukardaki sentaks .* ve ->* vs karışık. **std::invoke** kullan aynı işi yapar. !!

```
class nec{
public:
  void func(){
    (this->*mfp)(34); // incoke lada yapılır , std::invoke(mfp,
  }

  int foo(int );
  int foo2(int );
private:
  int (nec::*mfp)(int) = &nec::foo;
}
```