lessons 33 - 36

DERS 33

GENERIC PROGRAMING

template: derleyiciye kod yazdırır. türden bağımsız. compile time uzayabilir. kod gizli tutulamayabilir. **specialization** derleyicinin template ile üretiği koda denir.

templatelar bir birlerine overload olurlar.

header da olcak ki derleyici oraya yazsın, clientlar kullansın.

instantiate : örneklendirme , derleyicinin template koda bakıp çoğaltması. bir şablonun belirli bir türle doldurulup gerçek bir kod parçasına dönüştürülmesini ifade eder.

- function template : derleyiciye fonksiyon yazdırıyoruz
- class template : classı derleyiciye yazdırıyoruz.
- variable template : değişken şablonu
- alias template : tür eş isim şablonu
- concept - C++20

```
template <temp_parameter>   // temp_parameter : type , non-type
template <typename T, typename U>   =   template <class T, class
template <typename T, T x>   // x sabit ifade
```

Template kullanırken T nin erişimi:

deduction: çıkarım, c++17 den sonra sınıf içinde tür çıkarımı (**CTAD**) , derleyici çıkarımı.

ex: std::vector<int> vec{5,6}; CDAT → std::vector vec{5,6}; // tür çıkarımı yaptı

explicit template argument: direk yazcaz türü

diğer yol varsayılan tür kullanılır:

ex: template <typename T = int>

DERS 34

function template

```
template <typename T>
void foo(T x , T y){

}

template <typename T, typename U>
void foo2(T x , U y){

}

int main(){
   auto x = 10;     //auto argument deduction
   foo(10, 20);     //template argument deduction , yukardakiyle
   foo(10, 5.6);    // hata ambiguity, türler farklı
   foo2(10, 5.6);   // şimdi çalışır
   foo2<int,double>(45 , 8.9);  // aslında böle
}

/***************/
// ilk derslerde ki & && , && && muhabbeti
template <typename T>
void foo(T&& x){           // universal ref / forwarding ref  ,
    // @xdf152 durumunda x in türü "myclass &"
    // @xdf987 durumunda x in türü "myclass"
}

int main(){
   myclass m;
   foo(m);           //@xdf152 , l value
```

```
   foo(myclass{});   //@xdf987 , r value
}
```

```

auto foo() -> int(*)(int)   // trailing return type
{
   return bar;
}

// sık kullanım şekli:
template <typename T>
auto sum(T x) -> decltype(x.foo())       // trailing return type
{
   // ...
}

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

 auto foo()   //  auto return type               ,    NOT : tra
{
   return 3 * 3;
}
```

DIKKAT!

```
class my_class{
   using val_type = int;
}

template <typename T>
void foo(T x)
{
```

```
   T::val_type y{};   // DIKKAT DERLEYİCİ BUNU STATIK VERI ELEMA
}

// ÇÖZÜM;

template <typename T>
void foo(T x)
{
// derleyiciye bu bir tür ismi diyoruz: typename
   typename T::val_type y;   // sorun çözüldü , derleyici artık
}
```

```
template <typename T, typename R, typename U>
T sum(R x , U y)
{
   return x+y;   // geri dönüş türünü belirleyerek veri kaybını
}

void main()
{
   sum<double>(5 , 8.9);   // diğer türler otomatik bulundu , tü
}
```

Hileli soru: Öyle bir fonksiyon yaz ki içine gelen (int) tür dışında bütün durumlarda hata versin, derleyici bile explicit dönüşüm yapamasın yine hata versin.

```
template<typename T>
void func(T) = delete;

void func(int);

int main(){
   func(12);  // int dışında bütün türler şimdi hata verir
}
```

DERS 35

class template

```
template <typename T>
class my_class
{

}

template <typename T>
class test
{
     T bar(T x);
}

int main(){
  test<my_class<int>> x;  // iç içe mümkün
}



/*********/

// test.hpp file

template <typename T>    // SAKIN UNUTMA ! BUNLAR CPP file da ol
T test<T>::bar(T x)
{
  // ...
}
```

bir fonksiyondan iki farklı değer nasıl dönderirsin

```
template <typename T,typename U>
class pair                      // bu pair std içinde var std::p
```

```
{
  T first;
  U second;
}

pair<int,double> foo()
{
   pair<int,double> x;
   x.first = 5;
   x.second = 8.7;
   return x;
}
```

```
std::pair x{ 5 , 9.8};  // CTAD C++ 17 , tür çıkarımı yapılıo
```

```
template <typename T,typename U>
std::pair<T, U> my_make_pair(const T& t ,const U& u ) // bu göst
{
  return std::pair<T,U>(t,u);
}

template <typename T,typename U>
std::osteam& operator>>(std::ostream os , const std::pair<T,U>&
{
   return os << '[' << p.first << "," p.second << ']';
}

// main
auto x = std::make_pair(9 , 8.9);
auto y = std::make_pair( std::make_pair(8 , 7.8) , std::make_pa

cout << y; // @ref48
```

```
template <int U>      // CONST SABIT OLMAK ZORUNDA
class my_class
{

}

int main()
{
    int x = 6;
    my_class<x> y;  // HATA
    my_class<6> y1;  // BÖLE OLCAK
}
```

```
template <typename U>
class my_class
{
   void foo();
}

template <>
class my_class<int>   // explicit specialization / full specializ
{
  // yani  my class ın int değer alan template ini elimizle yazd
   void bar(); // diikat et yukardaki foo , ama burda bar var KA
}
```

DERS 36

```
template <typename U , typename T>
class my_class  // primary template
{
   void foo();
```

```
}

template <typename T>
class my_class<int , T>  // partial specialization
{
   void bar();   // ilk typenami int olanları bu template yazar
}

template <typename T>
class my_class<T, T>  // partial specialization , iki arguman ti
{
   void foo();
}
```

perfect forwarding

problem şu;

```
void foo(T x){
 // ..
}

void func(T x){  // tabi problem domaininde çoğunlukla bu çağıra
   foo(x);
}

int main(){

    func(arg);   // bu verilen argümanın, func aracılığı ile foo
// arg L veya R value olabilir , const olabilir etcc. hiç bir öz
// Çözüm ; perfect forwarding
}

// çözüm:
template <typename T>
void func(T&& x){
```

```
  foo(std::forward<T>(x)); // forward: R ı R L i L  yani value l
}
```

alias template: using in önemi !!

```
template<typename T>
using ptr = T;

int main(){
   ptr<int> x = 10;
}

/*********/

template <typename T , int N>
struct array{
   // .....
}

// bu arrayi diyelim çoğunlukla int kullanıosun o zmaan->
template<int N>
using int_array = array<int,N>;

// artık int_array ile N verip yap iş sentaks kolaylaşsın
// int_array<50> x;
```

```
template<typename T>
constexpr T pi = T(3.1466558665585565);

template<typename T>
T get_circle_area(T radius){
   return pi<T> * radius * radius;
}

int main(){
```

```
    get_circle_area(8.9);
}
```

variadic template: template<typename ...types>

```
template <typename ...VALS>
class my_class{
   // .....
   static constexpr auto x = sizeof...(VALS); // compile time da
}

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

template <typename ...VALS>
void func(VALS ...args){
   // argümanların kullanılma işi recursive olabilir, args tek b
}

// pack expansion
void foo(int x , double , y , char* x){}

template <typename ...VALS>
void func2(VALS ...args){
    foo(args...);   // bir fonksiyona bunu gönderirsek derleyici
}

int main(){
   func2(5 , 2.7 , "selam");  // foo(5 , 2.7 , "selan")  derleyi
  // aynı şey func çağırma değil, func2 içinde class yaratma vs
}
```