C++17 Kullanım Klavuzu

# Structured Bindings

Structured bindings çoklu değişik tipte değerleri girmenizi sağlar.

Örnek olarak

struct **MyStruct**{

int **i**;

std::string **str**;

};

int **main**()

{

MyStruct **temp**{5,"test"};

auto [**v**,**u**] = temp;

std::cout << v << " " << u << " \n";

return 0;

}

Gördüğümüz gibi v ve u değişkenine direk değerleri atama yaptı. Bu bize kodun okunurluğunu artırıcı bir özellik gibi düşünebilirsiniz. Bunu fonksiyonlarda da yapabiliriz.

struct **MyStruct**{

int **i**;

std::string **str**;

};

MyStruct **getStruct**(){

return MyStruct{5,"meaningfull"};

}

int **main**()

{

auto [**v**,**u**] = getStruct();

std::cout << v << " " << u << " \n";

return 0;

}

Yukarda bir fonksiyonun geri dönüş değerini yapısal olarak bağlıyor. Şimdi bir **std::map<>** ile nasıl bir kullanım kolaylığı sağladığına bakalım. Ilk eski yöntemle bir kod yazalım.

std::map<std::string, int> **mMap**{{"key1",1},{"key2",2},{"key3",3}};

for( const auto &**elem** : mMap ){

std::cout << elem.**first** << " " << elem.**second** << "\n";

}

Yukardaki kodda gördüğümüz gibi **std::map<>** içerisinde ki elementler **first** ve **second** diye ayrı ayrı erişmemiz gerekiyor.

Gelelim şimdi C++17 ile gelen kullanım kolaylığına;

std::map<std::string, int> **mMap**{{"key1",1},{"key2",2},{"key3",3}};

for( const auto &[**key**,**value**] : mMap ){

std::cout << key << " " << value << "\n";

}

Kodun ne kadar sadeleştiğine ve okunurluğun ne kadar arttığa yukardaki örnekten bakabilirsiniz.

## Structured bingdings details

Structured bindings daha iyi anlamak için, yeni anonim değişkenler dahil edildiğinin farkında olmak çok önemli. Structured bindings gibi yeni isimler bu anonim değişkenlerin üye/elementlerini referans alır.

**Bir giriş değerine bağlama**

Kesin başlatma şekli aşağıdaki gibidir;

auto [**u**,**v**] = ms;

yukardaki kodda yaptığımızın ayınısı aşağıdaki eski yöntem kodla da halledebiliyorduk.

auto **e** = ms;

aliasname **u** = e.i;

aliasname **v** = e.s;

u ve v dikkat ederseniz e.i ve e.s’nin referansı değil bir kopyası ve ms üyelerine isimleri ile doğrudan erişim yapamadık.

e, structured bindings varlığına bağlılı kadar var olur. Böylece Structured bindins scope sonuna geldiğinde destroy olur.

Sonuç olarak, referans olarak kullanılmadığı zaman initialization değerleri modifiye edildiği zaman bağlı olduğu yer etkilenmez.

MyStruct **ms**{5,"test"};

auto [**u**,**v**] = ms;

ms.**i** = 10; // ms.i 10 yaptığımız halde u halen 5 değerini tutar.

v = "newTest"; // v değiştirmemize rağmen ms.s halen "test" değerinde

Yukardaki örnekte gördüğümüz gibi copy değerlerini aldığı için artık bir birinden bağımsızdır. Yani diğeri üzerin de yapılan değişiklikler mevcut değeri etkilemez.

Structured bindings return value ler bind edildiğin de aynı principle uygulanır.

auto [**u**,**v**] = getStruct();

structured bindings’I const yada non-const referans olarak kullandığımız da her iki değerde aynı adresteki değeri gösterdiğinden birinden deki değişiklik diğerini de etkiler. const referans ta kaynak değiştiği zaman hedef değerde değişir. const olarak tanımladığımız dan hedef değer üzerinde değişiklik yapamayız and non-const olarak tanımladığımız değerlerde her iki yönlü değişiklik yapılabilir ve yapılan değişiklikler diğerinide etkiler. örnekle tanımlayalım.

MyStruct **ms**{5,"test"};

const auto &[**u1**,**v1**] = ms;

ms.**i** = 10; // ms.i 10 yaptığımız zaman u1 değeri de 10 olur, ancak u1 değerinde bir değişiklik yapamayız.

MyStruct **ms**{42,"testTest TETS"};

auto &[**u**,**v**] = ms;

u = 10; // u değerini 10 yaptığımız da ms.i değeride 10 olur.

bu koşullar referanslar da geçerli olduğu gibi move semantics’dede geçerlidir.

MyStruct **ms**{42,"test"};

auto &&[**u**,**v**] = std::move(*ms*);

std::cout << ms.**s** << "\n";// rvalue referans aldığımız için ms.i halen kendini korur.

std::string **a** = std::move(*v*);

std::cout << ms.**s** << "\n"; // ms.s artık bir değere sahip değil çünkü onun değerini a stringine taşıdık.

Ancak yukardaki gibi rvalue referans olarak değilde doğrudan değerini taşırsak artık ms.i ve ms.s de geçersiz bir değere sahip olacaktır.

MyStruct **ms**{42,"test"};

auto [**u**,**v**] = std::move(*ms*);

std::cout << ms.**s** << "\n";// ms değelerini referans ile move etmediğimizden geçersiz değerlere sahiptir.

## Structured Bindingsler nerelerde kullanılabilir

Prensip olarak structured bindingsler structure ların public data member larında, C stili raw dizilerde ve tuple tarzı objelerde kullanılabilir.

* Eğer structure yada sınıfları bütün non-static memberları public ise, bunların hepsini bir isim altından binding yapılabilir.
* Raw arraylar da, her elementi bir isim altında bind edilebilir.
* Tuple benzeri API ler elements adında tanımlamalar olsa bile herhangi bir tipini bir isim altında bind edebiliriniz.
  + Std::tuple\_size<*type*>::value element sayısını geri döndürür.
  + Std::tuple\_element<*idx,type*>::type idx inci elementin tipini geri döndürür.
  + Global yada member olan get<*idx*> idx inci elementin değerini gösterir.

Standard library içerisinde bulunan std::pair<>, std::tuple<> ve std::array<> aynı API lere sahiptir. Eğer structures veya classlar tuple tarzında API lere sahipse kullanılabilirler.

Her ne durumda element sayısı yada member data lar Structured binding declare edilirken isim sayıları bire bir uyması gerekmektedir. Yani tuple içerisinde 3 tip değişkeniniz var ise structed binding declare ederken 3 tane isim belirtmeniz gerekmektedir. Array tanımlarken array içerisinde 4 element var ise structured bindings te 4 adet yeni isimde değişken koymamız gerekmektedir. Daha az yada daha fazla isim altında değişken koyamayız.

## Structerlar ve Classlar

Şimdiye kadar gördüğümüz örnekler basit anlamda structered binding kullanımına örnekti. Ancak bu sınıflar inheritance gibi durumda biraz kısıtlayıcı durumları vardır. Örnekle bakalım.

struct **B**{

int **a** = 1;

int **b** = 2;

};

struct **D1** : B {

};

struct **D2** : B {

int **c** = 3;

};

int **main**()

{

auto [**x**,**y**] = D1{}; // Bu satır Başarılı bir şekilde yürütülür. OK

auto [**k**,**l**,**m**] = D2{}; // Ancak bu satır compile-time error verir. !!!ERROR

return 0;

}

Yukardaki örnekte gördüğümüz gibi inherite yaptığımız D2 de yeni bir değişken tanımladığımızdan parent sınıftaki değerleri göremediğinden k,l,m değerlerini atama yapmaya çalıştığımızdan compile başarısız olacaktır. Yeni bir değer kullanmadan inheritance yaptığımız da herhangi bir problem ile karşılaşmayacağız.

## Raw Arrays

Aşağıdaki kod C stili bir arraydır;

int **arr**[] = { 47 , 11 };

auto [**x**,**y**] = arr; // element sayısı ile declare edilen isim sayısı aynı olduğundan bir problem yoktur. OK

auto [**z**] = arr; // burda element sayısı ile declare edilen element sayısı birbirini tutmadığından hata alırız. ERROR

yukardaki örnekte gördüğümüz gibi array içerisinde ki eleman sayısını bilmek zorunlu. Eleman sayılarını bilmediğimiz durumlarda kullanamayız. Ancak C++ da array sayısını referans olarak döndüren bir özelliğe sahip olduğundan, fonksiyonlar da geri dönüş değerine size da atayabildiğmizden kullanılabilir.

auto **getArr**() ->int(&)[2]

auto [**x**,**y**] = getArr();

Yukardaki örnekte ki gibi bir return tipinde bir size belirtirsek structured binding kullanabilir ancak bu tuple tarzı bir yaklaşım olduğu için bir sonraki bölümde inceleyeceğiz.

## Std::pair, std::tuple ve std::array

Structured binding genişletilebilir bir mekanızması vardır, yani bunu herhangi bir bir tipinde de kullanabiliriz. STL kütüphanesinde bunlar std::pair, std::tuple ve std::array dır.

### std::array

Aşağıdaki örnekte getArray() fonksiyonu 4 elementli bir std::array geri değerine sahiptir.

std::array<int,4> **getArr**(){

return std::array<int,4>{1,2,3,4};

}

int **main**()

{

auto [**k**,**l**,**m**,**n**] = getArr();

return 0;

}

K,l,m ve n değerlerine array içerisindeki değerler copy olarak bağlanır. Fonksiyonumuz temporary bir değer döndürdüğü için yukardaki gibi kullanılır ancak temporary olmayan durumlarda referansta alınabilir.

std::array<int,4> **arr**{1,2,3,4};

auto &[**k**,**l**,**m**,**n**] = arr;

k = 10; // arr dizisinin içinde ilk değerde 10 olur.

### std::tuple

aşağıdaki örnek kodda da 3 farklı tipte değere sahip bir tuple;

std::tuple<char,float,std::string> **getTuple**(){

return std::tuple<char,float,std::string>{'1',2.0,"test"};

}

int **main**()

{

auto [**a**,**b**,**c**] = getTuple();

return 0;

}

Gördüğünüz gibi a değeri **char**, b değeri **float** ve c değeride **std::string** olarak değerini döndürür.

Std::pair

Başka bir örnekte, associative/unordered containerların da insert fonksiyonunun geri dönüş değerine direk olarak structured binding yapılabilmesi. Insert fonksiyonunun geri dönüş değeri olan std::pair first ve second değerlerini yerine kullanılabilmesi;

std::map<int,std::string> **coll**;

auto [**pos**,**ok**] = coll.insert({42,"test"});

if( ok ){

std::cout << pos->**first** << " " << pos->**second** << "\n";

}

C++17 den önce kullanılan yöntem de aşağıdaki gibidir.

std::map<int,std::string> **coll**;

auto **ret** = coll.insert({42,"test"});

if( !ret.**second** ){

// do something

}

**pair ve tuple’la bağlanmış olan structured binding değerlerine yeni değer atama**

structured binding declare edildikten sonra oluşturulan değerlere daha sonra aynı isimlerle yeniden declare edemezsiniz.

std::map<int,std::string> **coll**;

auto [**a**,**b**] = coll.insert({42,"test"}); // OK

auto [**a**,**b**] = coll.insert({55,"test2"});// a ve b tanımlandığından yeniden declare edilemez.

Yukarda Gördüğünüz gibi arka arkaya declare tanımlamaları yapamazsınız. Buna loop yaptığımız yerlerde ihtiyacımız olabilir. Yani a değerini tekrar kullanmak istersiniz, yeni bir insert methodu çağırdınız da tekrar auto [a,b] denilemez bunun yerine std::tie(a,b) methodunu kullanılmalıdır.

auto [**a**,**b**] = coll.insert({42,"test"}); // OK

std::tie(*a*,*b*) = coll.insert({55,"test2"});// a ve b tanımlandığından yeniden declare edilemez. std::tie() kullanmamız gerekir.

## Tuple Benzeri Bir API yi Structured Binding İçin Tasarlama

Standard Kütüphanenin std::array<>, std::pair<> ve std::tuple structured binding API leri gibi herhangi bir veri tipini structured binding bağlayacak şekilde tasarlayabiliriz.

### Sadece Okuma Modunda(Read-Only) Structured Binding

Aşağıdaki Customer class’ının structured binding için nasıl yazıldığına bir bakalım.

#include <string>

#include <utility>

class **Customer**{

private:

std::string **first**;

std::string **last**;

long **val**;

public:

**Customer**(std::string **f**, std::string **l**, long **v**)

:**first**(std::move(*f*)),**last**(std::move(*l*)),**val**(v)

{

}

std::string **getFirst**() const {

return **first**;

}

std::string **getLast**() const {

return **last**;

}

long **getValue**() const {

return **val**;

}

};

Yukarda Gördüğünüz sınıf standard bir sınıf. Bir sonraki sayfa da structured binding bağlanabilmesi için bazı temple implemantasyonlar yapmamız gerek.

Şimdi Bu Sınıfın tuple benzeri bir API sini oluşturalım.

#include "customer1.hpp"

#include <utility>

auto [f,l,m] dediğimiz zaman 3 adet değişkenimiz var. bu değişken sayısını derleyicinin eşleştirebilmesi için hemen bir alt satırdaki kodu tanımlamız gerekir. Derleyici 3 adet geri dönüş değerinin varlığını buradan bilir. Eğer eşletirme yapamazsa;

error: C3449: the number of identifiers must match the value of 'std::tuple\_size<Customer>::value' in a structured binding of tuple-like type

yukarda gördüğünüz gibi bir hata verir.

template<>

struct std::**tuple\_size**<Customer>{

static constexpr int **value** = 3;

};

Ikinci elementin tipini long olduğunu söylüyoruz burada. Yani sırasıyla f,l ve m tiplerinin declrtype yaparken derleyici 2 inci elementin tipinin long olduğunu buradan anlar. Biz burada long değilde başka bir değişken tanımlarsak m aliasname o şekilde ayarlayacak ve mevcut değerini assignment yapacak. Eğer cast yapamadığı bir tip koyarsak yani customer içerisinde long olarak tanımladığımız value derleyivi auto olarak structered binding yaparken tipinin ne olduğunu hemen alttaki satırdan bilir. Bu kod satırıran varsayılan template argumanı 2inci index teki tipe bağlar. Eğer biz burda long olarak değilde std::string olarak bir tanımlama yapsaydık kırmızı renkteki kod hatasını alacaktık.

error: C2440: 'initializing': cannot convert from 'long' to 'std::string &&'  
note: Reason: cannot convert from 'long' to 'std::string'  
note: No constructor could take the source type, or constructor overload resolution was ambiguous

ama şu tipler arasında hata vermemektedir; float -> double, int to long gibi. Bu tipleri kendi arasında compiler cast etmektedir.

template<>

struct std::**tuple\_element**<2,Customer>{

using **type** = long;

};

Hemen alltaki kod satırındada get<0> veya get<1> diye çağırdığımız da index indexindeki veri tipleri için tanımlama içermektedir. Yani 0ıncı ve 1inci indexteki veri tiplerinin tamamı std::string tipinde veri olarak tanımlamış olduk.

template<std::size\_t **index** >

struct std::**tuple\_element**< **index**,Customer> {

using **type** = std::string;

};

Aşağıdaki kod satırırandada get ile çağırdığımız da index ile gelen veri ile hangi veriyi geri döndüreceğimizi seçiyoruz.

template<std::size\_t **index**> auto **get**( const Customer &**c**)

{

static\_assert(**index** < 3);

if constexpr(**index** == 0 ){

return c.getFirst();

}else if constexpr( I == 1 ){

return c.getLast();

}else{

return c.getValue();

}

}

Aşağıdaki kod satırıda değişken adetlerimizin her biri için geri dönüş değeri belirken template fonksiyonlarımızı tanımladık. Bu kod satırını tanımlama şartımız yok çünkü üstte bulunan kodun aynı işlevi yerine getirior.

template<> auto **get**<0>( const Customer &**c** ) { return c.getFirst();}

template<> auto **get**<1>( const Customer &**c** ) { return c.getLast();}

template<> auto **get**<2>( const Customer &**c** ) { return c.getValue();}

Şimdi Ana Fonksiyonumuz içerisinde daha önce tanımladığımız API için bir örnek kod çalıştıralım;

#include "lang/structeredbind1.hpp"

using namespace std;

int **main**()

{

Customer **c**("Tim","Starr",42);

auto [**f**,**l**,**v**] = c;

std::cout << "F: " << f << " L: " << l << " V: " << v << std::endl;

std::string **s** = std::move(*f*);

std::cout << "S: " << s << "\n";

std::cout << c.getFirst() << " " << c.getLast() << " " << c.getValue() << "\n";

std::cout << "F: " << f << " L: " << l << " V: " << v << std::endl;

return 0;

}

Gördüğünüz gibi artık auto [] şeklinde Customer sınıfını kullanabiliriz. Bunu loop içerisinde de kullanabiliriz. Ancak şuana kadar yazdığımız kodda sadece geriye dönen değerdeler okuma modu şeklide hazırladık. Yani Customer sınıfının herhangi bir üyesinde değişiklik yapamayız.

### Structured Binding Yazma(Write-Access) Erişimi

Bir önce ki örnekte sadece Customer sınıfında sadece member üyelerinin verilerini okuyabiliyorduk. Şimdi bu verilerin nasıl değiştirilebildiğine bakalım. Öncelikli olarak Customer sınıfına referans değerlerini geri döndüren fonksiyonlar ekleyelim.

class **Customer**{

private:

std::string **first**;

std::string **last**;

long **val**;

public:

**Customer**(std::string **f**, std::string **l**, long **v**)

:**first**(std::move(*f*)),**last**(std::move(*l*)),**val**(v)

{ }

const std::string **getFirst**() const {

return **first**;

}

const std::string **getLast**() const {

return **last**;

}

const long **getValue**() const {

return **val**;

}

std::string &**getFirst**(){

return **first**;

}

std::string &**getLast**(){

return **last**;

}

long &**getValue**(){

return **val**;

}

};

Referans değerlerini yazdıktan sonra tuple benzeri apilerinde de bazı implementasyonlar yapmamız gerekmektedir.

template<>

struct std::**tuple\_size**<Customer>{

static constexpr int **value** = 3;

};

template<>

struct std::**tuple\_element**<2,Customer>{

using **type** = long;

};

template<std::size\_t **Idx**>

struct std::**tuple\_element**<Idx,Customer> {

using **type** = std::string;

};

template<std::size\_t **I**> auto **get**( const Customer &**c**)

{

static\_assert(I < 3);

if constexpr(I == 0 ){

return c.getFirst();

}

else if constexpr( I == 1 ){

return c.getLast();

}else{

return c.getValue();

}

}

template<std::size\_t **I**> decltype(auto) **get**( Customer &**c** ){

static\_assert(I < 3);

if constexpr(I == 0 ){

return c.getFirst();

}

else if constexpr( I == 1 ){

return c.getLast();

}else{

return c.getValue();

}

}

template<std::size\_t **I**> decltype(auto) **get**( Customer &&**c** ){

static\_assert(I < 3);

if constexpr(I == 0 ){

return std::move(c.getFirst());

}

else if constexpr( I == 1 ){

return std::move(c.getLast());

}else{

return std::move(c.getValue());

}

}

Yukardaki kodda sadece referans, move referans geri döndüren template fonksiyonları tanımladık sadece. Şimdi nesnemizi tekrar yapalım.

int **main**()

{

Customer **c**("Tim","Starr",42);

auto &[**f**,**l**,**v**] = c;

std::cout << "F: " << f << "\n";

f = "Arnold";

std::cout << "c.getFirst() " << c.getFirst() << "\n" << "F: " << f ;

return 0;

}

Dikkat ederseniz auto &[] diye kodumuza başladık. Bu kod satırını kullandığımız da aşağıdaki fonksiyon çağırılacaktır.

template<std::size\_t **I**> decltype(auto) **get**( Customer &**c** )

Eğer auto &&[] diye kodumuzu yazsaydık bu seferde aşağıdaki fonksiyon çağırılacaktı;

template<std::size\_t **I**> decltype(auto) **get**( Customer &&**c** )

main fonksiyonunun içerisindeki çalıştırdığımız kodun çıktısı aşağıdaki gibidir.

F: Tim  
c.getFirst(): Arnold  
F: Arnold

# if ve switch koşullarını özel başlatmalar

If ve switch ifadelerinde daha öncede sadece bir koşul kullanabiliyorken C++17 ile birlikte bir koşulunda yanı tanımlamalarda yapabiliriz.

if( State **status** = checkStatus() ; status == State::OK ){

std::cout << "Durum OK";

}else{

std::cout << "Durum HATALI";

}

Aslında status verisi if dışında tanımlanmış bir veri değil. Önceden bunu if satırından önce tanımlamamız gerekiyordu. C++17 ile birlikte artık buna ihtiyacımız kalmadı. Gördüğünüz gibi yukardaki gibi tanımlayabiliriz.

## if koşul içerisinde tanımla

if koşul tanımladığımız herhangi bir değişken tanımlayabiliriz. Bu tanımladığımız değişken if veya else scope alanını sonunda destroy olur.

if( std::ofstream **strm** = getLogStrm(); coll.empty() ){

strm << "<no data>";

}else{

for( const auto &elem : coll ){

strm << elem << "\n";

}

}

Yukardaki kod satırında tanımladığımız strm, if else scope bittikten sonra destructor çağırılır.

Başka bir örnekte std::lock\_guard<> kullanımında bazı kurallara bağlı olarak işlem yapılırken if içerisinde direk tanımlayabiliriz.

if( std::lock\_guard<std::mutex> **lg**{collMutex}; !coll.empty() ){

std::*cout* << coll.front() << "\n";

}

Yukardaki örnek class template arguman deduction dan dolayı (tip indirgeme) aşağıdaki gibi kullanılabilir.

if( std::lock\_guard **lg**{collMutex}; !coll.empty() ){

std::*cout* << coll.front() << "\n";

}

Yani arada std::lock\_guard<> template içerisinde değişkenimizin tipinin std::mutex olarak tanımlamamıza gerek yok. Yukardaki her iki kod aslında aşağıdaki ile aynıdır.

{

std::lock\_guard<std::mutex> **lg**{collMutex};

if ( !coll.empty() ){

std::*cout* << coll.front() << "\n";

}

}

Küçük bir farkla sadece lg adında bir değişkeni if statement ın içerisinde kuralla birlikte aynı yerde tanımladık, for loop içerisinde ki tanımlama gibi düşünebilirsiniz.

Bütün tanımladğımız değişkenler bir isim belirtilmesi zorunludur. Yoksa nesne yaratıldığı gibi hemen arkasına destroy olur ve kullanılamaz. Buda run time anında bir hata olur.

Isim olarak sadece \_ karakterini kullanmak aslında yeterli ama okunurluğu azalttığından pek tercih edilmez, aşağıdaki örnek gibi

std::lock\_guard<std::mutex> **\_**{collMutex};

3üncü bir örnektete std::map<> içerisine bir çift ekleme yapıp bunun sonucunuda structured binding yaparak tekrar sonucu kontrol edebiliriz.

std::map<std::string,int> **coll**;

if( auto [**pos**,**ok**] = coll.insert({"new",42}); !ok ){

// insert yapılamadığından kod buraya düşer

const auto &[**key**,**val**] = \*pos;

std::cout << "bu değer zaten var\n";

}

Bu kodu c++17 den önce aşağıdaki gibi yazabilirdik.

std::map<std::string,int> **coll**;

auto **ret** = coll.insert({"new",41});

if( !ret.**second** ){

const auto &**elem** = \*(ret.**first**);

std::cout << "bu değer zaten var\n";

}

## switch koşul içerisinde tanımlama

Switch koşullandırma alanı içerisinde aynı if koşul ile aynı şekilde kullanabilir. Koşul içerisinde tanımladığımız yeni değişkenimiz switch scope alanı bittiğinde destructor otomatik olarak çağırılır.

using namespace std::filesystem;

switch (path **p**(name); status(p).type() ) {

case file\_type::not\_found:

break;

case file\_type::directory:

//

break;

default:

break;

}

## NOTE

If ve switch koşulları içerisinde değişken tanımlama önerisi ilk olarak *Thomas Köppe* tarafından önerilmiştir. Daha detaylı bilgi için: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0305r1.html> linkinden bakabilirsiniz.

# inline variable

C++ en güçlü yeteneklerinden biriside header-only kütüphanelerinede destek verebilmesidir. Ancak bu C++17 kadar, eğer kütüphane tarafından global bir değişken/nesneye ihtiyaç duymadığında mümkündü.

C++17 den itibaren, global değişken/nesne tanımlarken cpp dosyalarına header dosyasını include ettiğimizde tekrar tekrar değer atamak zorunda değiliz. Bunu header dosyası içerisinde inline olarak tanımlayarak yapmak artık mümkün. Bu özellik bize kodun hem okunurluğunu hemde kod üzerinde ki hakimiyetimizi artırır. Şimdi eskiden nasıldı ona bakalım;

class **MyClass** {

static std::string **name** = ""; // C++17 den önce böyle bir tanımlama yapamayız

};

Yukardaki tanımlama c++17 den önce geçerli değil. Geçerli olabilmesi için “name” kullanıcağımız cpp dosyasında değer atamamız gerekir veya başka bir yöntemde “name” değişkenini const olarak tanımlamamız gerekir. C++17 önce bunun nasıl yapıldığına bir bakalım.

class **MyClass** {

static std::string **name**;

};

// \*.cpp dosyası içerisinde  
std::string MyClass::**name** = "";// C++17 den önce böyle yapabiliyorduk

eğer const olarak tanımlarsak aliasname(std::string) den önce const olarak her iki satırda da tanımlamamız gerekir. Buradaki bize problem çıkaran olay MyClass içerisinde tanımladığımız name değişkenine header dosyası içerisinde atama yapamıyor olmamız. Bunu bu header dosyasını kullandığımız cpp dosyasının içerisinde yapma zorunluğu. Başka bir cpp dosyasına bu header dosyasını include ettiğimiz de tekrar değer ataması yapmamız gerekli. Buda derlektedikten sonra link ederken hangi static değeri link edileceğine derleyici karar veremediğinden linker error hatası alacaksınız. Yani bir cpp dosyasında name=”Ahmet” dediniz. Başka bir cpp dosyasında da name=”Mehmet” dediniz. Linker hangisi ile devam ediceğini bilemez ve hata ile size geri dönüş yapar.

Şimdi c++17 de bu sorunun nasıl giderildiğine bir bakalım.

class **MyClass** {

static inline std::string **name** = ""; //C++17 tanımlama

};

Yukardaki koddan gördüğünüz gibi artık c++17 ile hem sade hem kontrolü kolay ve derleyici tarafından tek bir değer ile bağlandığından hata almayız.

C++17 den önce kullanabileceğimiz durumlar aşağıdaki gibidir;  
Static const integral ları( double, bool,int…) aşağıdaki gibi kullanabiliriz;

class **MyClass** {

static const bool **checked** = false;

};

Inline olarak static bir local değişkeni geri döndürebiliriz.  
class **MyClass** {

inline std::string **getName**() {

static std::string **name** = "";

return name;

}

};

Static bir değişkeni normal olarak kullanabiliriz.  
class **MyClass** {  
 std::string **getGlobalObject**(){  
 static std::string **name** = "";  
 return name;  
 }  
};

C++17 ile template olarak tanımlama yapabilirmiz.  
class **MyClass** {

template<typename T = std::string>

T myGlobalObject = "std::string";

};

Base aldığımız bir bir sınıftan static üye ile birlikte kalıtım alabiliriz.  
template<typename **T**>

class **BaseClass**{

static std::string **name**;

};

//myclass.cpp

template<typename **T**>

std::string BaseClass<T>::**name** = "Başlangıç Değeri";

//myclass.cpp

class **MyClass** :public BaseClass<void> {

};

Yukardaki gördüğünüz yaklaşımların tamamı geçerli ve c++17 önce kullanılıyordu. Bu kodun okunurluğunu azalltığı gibi farklı işlemler için farklı methodlar kullanmak zorundayız. Global değişkenler tanımladıkları yerden kullanılacağı yere kadar değer atanmaları yapılmıyor.

# Aggregate extensıons

C++ da değişkenlere verilerimizi atarken tek yol var oda aggregate initialization, buda bize bir yapıda bize birden fazla değikene veri atamamıza olanak tanır.

struct **Data**{

std::string **name**;

double **value**;

};

Data **c**{"test",15.5};

Yukardaki örnekteki gibi nesnemizi oluştururken doğrudan değerlerimizi/verilerimizi atayabiliyoruz. C++17 den itibaren inherit yaptığımız sınıflarda base sınıflara da artık değer/veri atayabiliyoruz.

struct **MoreData** : public Data {

int **newData**;

};

MoreData{{"test",15.5},44};

Yukardaki örnekte gördüğümüz gibi artık aggregate initialization base sınıf/struct larada veri atamamızı çift curly brace kullanarak atamamıza izin veriyor. Fakat burda çift curly brace kullanma zorunlu değil. Direk olarakta atayabiliriz. Aşağıdaki örnekteki gibi.

MoreData{"test",15.5,44};

Aggregate initialization özelliği olmadan önce kodumuz aşağıdaki gibi tanımlamak zorundaydık;

struct **MoreData** : public Data {

int **newData**;

**MoreData**(const std::string &**\_name**, double **d**, int **i**)

:Data{\_name,d},**newData**{i}{

}

};

## extended aggregate ınıtıalization kullanımı

Sınıflardan üretilmiş C style structure lar memberlarına yeni veri yada operation ekleme tipik uygulaması aşağıdaki gibidir;

struct **Data**{

const char\* **name**;

double **value**;

};

struct **PData** : Data {

bool **critical**;

void **print**() const {

std::cout << "[" << **name** << "," << **value** << "]\n";

}

};

PData **data**{{"test1",6.778},false};

data.print();

yukardaki örnekte base class/struct veri aktarırken çift curly brace kullanmamız gerekiyor.

Artık bu değişkenlere veri ataması yapmak zorunda değiliz. Şöyleki değişkenimizi tanımladık ve iç değişkenlere bir değer atamadığımızdan c++17 bizim başlangıç değerleri atayacaktır. Şimdi nasıl olduğuna bir bakalım.

PData **a**{}; // zero-initialization bütün elementler

PData **b**{{"msg"}}; // {{"msg",0.0},false} ile aynı sonuçlar

PData **c**{{},true}; // {{nullptr,0.0},true} ile aynı sonuçlar

PData **d**; // herhangi bir değer ataması yok, değerlerin ne olduğu bilinmiyor.

Boş curly brace kullanımı ile hiç bişey kullanmamanın arasındaki farklara bir bakalım;

* Curly brace kullandığımız da eğer değişkenlere bir değer atamamışsak derleyici onlara başlangıç değerleri verecektir. Yani 0 veya nullptr gibi
* Ancak hiç bişey kullanmadığımız derleyici herhangi bir atama yapmadığından tam olarak değişkenin ne sakladığını bilemeyiz yani aslında bu güvenli olmayan bir programlama tekniğidir ve yazılımcılara tarafından hiç tercih edilmez.

Aggregate olan bir struct/class tan aggregate olmayan bir struct/class türetebiliriz.

struct **MyString** : std::string {

void **print**(){

if(empty() ){

std::cout << "unspecified value\n";

}else{

std::cout << c\_str() << "\n";

}

}

};

Hatta multiple derive bile yapabiliriz ve değerleri/verileri değişkenlere atayabiliriz. Örnekleyerek bakalım;

template<typename **T**>

struct **D** : std::string , std::complex<T>

{

std::string **data**;

};

Yukarıdaki veri tiplerini aşağıdaki gibi kullanabiliriz;

D<float> **s**{{"Merhaba"},{4.5,6.7},"dünya"}; // OK C++17

D<float> **t**{"Merhaba",{4.5,6.7},"dünya"}; // OK C++17

std::cout << s.**data**; // outputs: "Merhaba"

std::cout << static\_cast<std::string>(s);// outputs: dünya  
 std::cout << static\_cast<std::complex<float>>(s); // outputs: (4.5,6.7)

initializer listesinde veriler base struct/class takie declarion sırasına göre atama yapar.

## aggregate tanımlama

C++17 den itibaren aggregate tanımlaması

* Arraylar içinde yapılabilir.
* Veya *sınıfı tipi* (class,union, veya struct ) ile;
  + *Explicit* constructıor yada kullanıcı tanımı olmayan
  + Constructor inherite yapılmayan
  + *Private* yada *protected* non-static veri üyeleri olmadan
  + *virtual* fonksiyonlar olmadan
  + *virtual, private* yada *protected* olarak inherita edilecek base class olmadan

yukardaki şartlar sağlandığı takdirde yapılabilir.

C++17 de *std::is\_aggregate* adında bir type trait ile veri tipinizin aggregate edilip edilemeyeceğini kontrol edebilirsiniz.

D<float> **s**{{"Merhaba"},{4.5,6.7},"dünya"};

std::cout << std::is\_aggregate<decltype(s)>::value << "\n"; // output: true(1)

# Mandatory copy elısıon yada passıng unmaterialized objecs

Bu bölüme iki açıdan bakılabilir:

1. teknik olarak C++17 belirli şartlar altında mandatory copy elision için yeni bir kurala giriş yaptı. Fonksiyonlara parametrelerini değer geçirirken veya geri dönüş değeri döndürürken kopyalamayı ortadan kaldıran önceki seçenekler artık zorunlu hale geliyor.
2. Sonuç Olarak başlatma için materyalleştirilmemiş nesnelerin değerlerinin etrafında dolaşmakla ilgileniyoruz.

## mandatory copy ellision for temporaries

Ilk standartından beri, herhangi bir copy constructor çağırılmadığı zaman C++ bazı belirli durumlarda copy operation’u programın çalışmasını etkilese bile izin veriyordu. Bunlardan bir tanesi yeni bir nesne initialize edilirken temporary nesneyi kullanması gibi. Bu özellikle temporary nesnenin bir fonksiyona parametre olarak gönderilmesi yada geri dönüş değeri olarak kullanılmasında olur. Örneğin;

class **MyClass** {

public:

**MyClass**(){}

};

void **foo**( MyClass **param**){

}

MyClass **bar**(){

return MyClass(); // geri dönüş değeri geçiçi

}

int **main**(int **argc**, char \***argv**[])

{

foo(MyClass()); // geçici nesne parametre olarak kullanıldı.

MyClass **x** = bar(); // Geçici geri dönüş değeri local nesne için kullanıldı.

foo(bar()); // geçici geri dönüş değeri fonksiyona parametre olarak kullanıldı

}

Ancak bu optimisazyonlar zorunlu olmadığından,nesne kopyalama copy yada move constructor’dan doğrudan yada dolaylı olarak sağlanabiliyordu. Bu, copy/move constructorlar kullanılmasa bile varlardı. Copy yada move constructor olmasaydı zaten yukardaki kod derlenemezdi. C++17 den beri copy/move constructorlar artık zorunlu. bir fonksiyonun scope alanı içerisinde oluşturduğunuz nesne geri dönüş değeri kullanırken derleyici nesnenin mutlaka copy/move constructor yada copy/move assigment özelliğine ihtiyaç duyacaktır.

MyClass **bar**(){

MyClass **obj**;

return obj; // copy/move özelliğine ihtiyaç var

}

Bu zorunlu özelliğin ana faydaların biri daha iyi copy performansı sağlamasına yardımcı olur. Ancak move semantic nesne copy maliyetini ciddi anlamda düşürür.

Diğer faydalarında biride factory fonksiyonu kullanırken daima çalışıyor olmasıdır, çünkü copy yada move operasyonu sınıf tarafından kabul edilmese bile geri nesne değeri döndürür. Örnek:

template <typename **T**, typename... **Args**>

T **create**( Args&&... **args**){

return T{std::forward<Args>(args)...};

}

Yukardaki fonksiyon atomic değerlerde copy move desteklemese bile çalışır.

int **main**(int **argc**, char \***argv**[])

{

auto **i** = create<int>(42);

std::unique\_ptr<int> **up** = create<std::unique\_ptr<int>>(new int{42});

std::atomic<int> **ai** = create<std::atomic<int>>(42);

}

Diğer bir özelliği de doğrudan move constructor iptal edilmiş sınıflarda copy constructor çalışır. Örnek:

class **CopyOnly**{

public:

**CopyOnly**(){}

**CopyOnly**(int){}

**CopyOnly**( const CopyOnly &) = default;

**CopyOnly**( CopyOnly&& ) = delete; // move iptal edildi.

};

CopyOnly **ret**(){

return CopyOnly{42}; // C++17 den Dolayı Çalışır

}

CopyOnly **x** = 42; // C++17 den Dolayı Çalışır

X ataması C++17’den önce geçerli değildi. Çünkü 42 değeri copy olarak geçici nesneye atanır ve oradan da move olması gerekir.

# lambda extensions

Lambda fonksiyonları ilk olarak C++11 ile geldi ve C++14 de başarılı bir şekilde hikayesine devam etti. Lambda fonksiyonları yeri geldiğinde arguman gibi kullanarak özel operasyonlarımızı yapmamızda kolaylık sağlar.

C++17 lambda fonksiyonlarının daha fazla yerde kullanılmasına olanağını artırmıştır.

* Constant tanımlamarda ( compile time )
* Lambda thread içerisinde kullanırken copy bir nesneye ihtiyaç duyduğumuzda.

## constexpr lambda

C++17’den beri lambda fonksiyonlar doğrudan constexpr olarak tanımlanabiliyor. Bu lambda fonksiyonlarının compile time anında kullanabileceğimiz anlamına gelir. ( sadece: literal tipler, static olmayan, sanal olmayan, try catch içermeyen ve new delete olmayan tanımlamalarda)

Mesela aşağıdaki kodun sonucunu compile time anında kullanabilirsiniz.

auto **squared** = []( auto **val** ){

return val\**val*;

}

std::array<int,squared(5)> **a**; // C++17 de bu std::array<int,25> a;

Yukardaki örnekte compile time anında hesaplanabildiğinden std::array<> dizisini oluşturabiliriz. array dizisi tanımlanirken boyutu sabit olarak belirlenmelidir. C++17 dolaylı olarak constexpr olarak tanımlanır.

auto **squared2** = []( auto **val** ){

static int **calls** = 0;

return val\**val*;

}

std::array<int,squared2(5)> **a**; // ERROR: static değer compile time anında kabul edilmez;

std::cout << squared3(5); // Bu Satır Çalışır.

Constexpr değerini biz kendimizde tanımlayabilirniz.

auto **squared2** = []( auto **val** ) constexpr {

static int **calls** = 0;

return val\**val*;

}

Ayrıca geri dönüş değeride belirtebiliriz;

auto **squared2** = []( auto **val** ) constexpr ->int {

static int **calls** = 0;

return val\**val*;

}

## this değerini lambda copy olarak geçirme

class **C**{

private:

std::string **name**;

public:

**C**(){}

void **foo**(){

auto **l1** = []{ std::cout << *name* <<"\n";} // ERROR

auto **l1** = []{ std::cout << this->name <<"\n";} //ERROR

}

};

Yukardaki kodda lambda sınıf scope alanını capture etmediğinden dışardan herhangi bir veriye erişemez.

C++11 ve C++14 de yukardaki örnek aşağıdaki gibi çalışır hale getirilebilir:

class **C**{

private:

std::string **name**;

public:

**C**(){}

void **foo**(){

auto **l1** = [this]{ std::cout << name <<"\n";}; // OK

auto **l2** = [=]{ std::cout << name <<"\n";}; //OK

auto **l3** = [&]{ std::cout << name <<"\n";}; //OK

}

};

Burdaki oluşabilecek sorun değişken yada alanın referans yada pointer olarak capture ettiğimizde nesne ömrünü tamamladığında lambda bu nesne erişirken hata verecektir. Özellikle thread içerisinde tanımlanan değer olarak düşünün.

C++14 de bu sorun copy yapılarak aşılabilir:

auto **l1** = [**thisCopy**=\*this]{ std::cout << thisCopy.name<<"\n";};

Aşağıdaki gibi bir kafa karıştırıcı kod karşımıza gelebilir:

class **C**{  
private:  
 std::string **name**;  
public:  
 **C**(){}  
 void **foo**(){  
 auto **l1** = [&,**thisCopy**=\*this]{  
 std::cout << thisCopy.name<<"\n"; // Kopya name  
 std::cout << name<<"\n"; // C sınıfındaki name  
 };

}

};

Mesela aşağıdaki örneği inceleyelim;

class **Data**{

private:

std::string **name**;

public:

**Data**(const std::string&**s**) : name(s){}

auto **startThreadCopyOfThis**() const {

using namespace std::literals;

std::thread **t**([\*this]{

std::this\_thread::sleep\_for(3s);

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

});

return t;

}

};

int **main**(int **argc**, char \***argv**[])

{

std::thread **t**;

{

Data **d**{"c1"};

t = d.startThreadCopyOfThis();

}

t.join();

}

t threadini yarattıktan sonra scope içerisinde D sınıfından bir nesne yaratıyoruz be bu t thread’inin d nesnesinin fonksiyonunda arama yapıyouruz scope dışında t threadini join ediyoruz. Şimdi burada lambda capture içerisinde copy olarak nesneyi almasaydık, scope dışına çıkınca nesne destrol edileceğinden tanımsız bir alanda t çalışmak isteyecek belirlenemeyen bir sonuç bize geri döndürecek yada hata verekcektir.

# Yeni attrıbute’ler ve özellikleri

C++11 attribute tanımlayarak derleyici uyarılarını kapatıp açabiliyorduk. C++17 ile birlikte yeni attribute’ler eklendi.

## [[nodiscard]]

Bazen fonksiyonlarda geri değer döndürüp bu değeleri kullanmadığımız zaman compiler bize uyarı verecektir. Bu uyarı daha da belirgin olarak compilerin göstermesi için tanımlayabiliriz.

[[nodiscard]] int **topla**( const int &**a** ){

return a+2;

}

int **main**(int **argc**, char \***argv**[])

{

topla(5);

return -1;

}

Yukardaki örnekte [[nodiscard]] kullanmadığımız zaman msvc 2022 derleyicim bana uyarı vermiyor ancak şimdi uyarı veriyor.  
Memory Leak olmasını engeller.Geri dönüş değerini kullandığımız da bilinmeyen bir davranış göstermesini engeller. Gereksiz yük olmasını engeller.

## [[maybe\_unused]]

Bu attribute ile bazen kullandığımız fonksiyon yada parametrelerin başına yazarak derleyicinin bu satırda hem optimisazyon yapmasını hemde derleyici de kullanılmayan diye başlayan uyarılarını kapatmamıza olanak sağlar.

[[may\_unused]] attributesini [[no\_discard]] ile birlikte kullanamayız. Çünkü birbirlerini zıt kavramlar gibidir. Yani bir fonksiyonu tanımladınız [[no\_discard]] ile ve bir yerde bu fonksiyondan geri dönen değeri [[maybe\_unused]] diyemeyiz. Mantıksal olarak hatalıtır zaten.

## [[falltrhough]]

Bu attribute lerde switch içerisinde break kullanmadığımız da ikici case durumuna düşürerek iki durumlarını da çalışmasını sağlar.

auto **control**{1};

switch (control) {

case 1:

std::cout << "case 1\n";

[[fallthrough]];

case 2:

std::cout << "case 2\n";

break;

case 3:

std::cout << "case 3\n";

break;

default:

break;

}

## [[deprecated]]

Bu attribute de bundan sonra desteklenmeyecek ama bir süre daha kullanımına devam edilecek değerler ve fonksiyonlar da kullanılır.

# diğer dil özellikleri

## nested namespace

2003’te ilk defa C++ komitesi aşağıdaki gibi bir nested namespace kabul etti.

namespace **A**::**B**::**C**{

}

Yukardaki kodun aynı işlev aşağıdaki gibidir.

namespace **A**{

namespace **B**{

namespace **C**{

}

}

}

Inline namespaceler desteklenmemektedir.

## tanımlanmış expression evulation order

std::string **s** = "I heard it even works if you don't believe";

s.replace(0,8,"").replace(s.find("even"),4,"sometimes").replace(s.find("you don't"),9,"I");

Şimdi yukardaki işlem sırasına baktığımız mantıksal olarak en sonda ve en içteki operasyonlar başlayarak ilerlemesi gerekir. Bu C++17 den önce bunun bir garantisi yoktu ancak C++17 bu işlem sırası beklendiği şekilde garanti etti.

int **i** = 0;

std::cout << ++i << " "<< --i << "\n";

Yukardaki kod satırında çıktının nasıl olucağının bir garantisi yoktur. Çıktı 0 0 yada 1 0 olabilir. Bazı derleyiciler bunu önlesede bu bir standart değildi. Ancak bu C++17 ile birlikte bir standart haline geldi ve çıktının 1 0 olması garanti edildi.

new int(5);

int tipinde 5 adet hafızada bir operasyon yapmadan önce hafızada yer açıldığının C++17 ile artık garantisi var.

void **print10Elems**( const std::vector<int> &**list** ){

for( int **i** = 0 ; i < 10 ; i++ ){

std::cout << "value: " << list.at(i) << "\n";

}

}

int **main**(int **argc**, char \***argv**[])

{

std::cout << \_\_cplusplus << "\n";

try {

std::vector<int> **vec**{1,2,3,4};

print10Elems(vec);

} catch (const std::exception &**e**) {

std::cout << "EXCEPTION: " << e.*what*() << "\n";

}catch(...){

std::cout << "EXCEPTION UNKNOWN" << "\n";

}

return -1;

}

C++17den önce Çıktı:

*value: 1  
value: 2  
value: 3  
value: 4  
EXCEPTION: invalid vector subscript*

C++17 ve sonrası için:

*value: 1  
value: 2  
value: 3  
value: 4  
value: EXCEPTION: invalid vector subscript*

Şeklindedir.

## exceptıon tıp özellikleri

C++17’den itibaren exception handling özellikleri farklı fonksiyon tipi haline geldi. Aşağıdaki örnekten bakalım;

void **f1**();

void **f2**() noexcept; // farklı tipte

C++17’den önce bu fonksiyonlar aynı özellikte bir fonksiyondu ancak artık aynı özellikte değil.

void (\***fp**)() noexcept; // pointer fonksiyon herhangi bir fırlatma yapmaz

**fp** = f2; // OK

**fp** = f1; // ERROR

noexcept ile noexcept olmayan fonksiyonlar artık birbirlerine eşitlenemezler. İnheritance yaparkende aynı durum geçerlidir.

class **Base**{

public:

virtual void ***foo***() noexcept;

};

class **Derived** : public Base{

public:

void ***foo***() override; // override olmaz

};

**Şartlı exception tipleri**

void **f1**();

void **f2**() noexcept;

void **f3**() noexcept(sizeof(int)<4); // f2 yada f1 ile aynıdır

void **f4**() noexcept(sizeof(int)>=4);// f3 ile aynı değildir.

**sizeof(int) 4 bytelık bir alanı kaplıyor ise**void **f3**() noexcept(false);  
kaplamıyor ise:  
void **f3**() noexcept(true);  
f2 ile aynıdır.

**Template fonksiyonlarda**

template<typename **T**>

void **call**( T **op1**, T **op2**){

op1();

op2();

}

void **f1**(){}

void **f2**() noexcept{}

int **main**(int **argc**, char \***argv**[])

{

call(f1,f2); // ERROR, fonksiyonların tipi aynı değil

return -1;

}

# class template argument deduction

C++17 den önce class template’larda template tip parametresini belirtmeden tanımlama yapamazsınız. Mesela aşağıdaki örnekte **double** belirtmeden tanımlama yapamazsınız.

std::complex<double> **v**{ 1.2 , 3.4 };

yada aşağıdaki gibi **std::mutex** ikinci kez tanımlamayı pass geçemezsiniz;

std::mutex **mutex**;

std::lock\_guard<std::mutex> **lg**(*mutex*);

C++17 den itibaren, bu kısıtlamalar karşısında her zaman zorunda olduğu tip belirteçleri için rahat olun. *class template argument deduction* sayesinde template parametresinde eğer constructor deduce yapıyorsa artık parametre tipini belirtme zorunda değilsiniz.

Şimdi aşağıdaki gibi declare edebilirsiniz;  
std::complex **v**{ 1.2 , 3.4 }; // std::complex<double> ile aynı

Yada uygulamasını aşağıdaki gibi yapabilirsiniz;  
std::mutex **mutex**;

std::lock\_guard **lg**(*mutex*); // std::lock\_guard<std::mutex> lg(mutex);

Hatta aşağıdaki container sınıflarında da tip değişkenlerini deduce yapmanıza gerek yok;

std::vector v1{1,2,3};  
std::vector v2{"hello","world"};

## sınıf şablonlarda tip çıkartımları

Sınıf şablonlarında tip çıkartımını, argümanı constructor doğrudan göndererek yapabiliriz. Bu yöntem bütün initializion yöntemleri kullanılabilir.

std::complex c1{1.1,2.2};

std::complex c2(1.1,2.2);

std::complex c3 = 2.2;

std::complex c4 = {2.2};

c3 ve c4 de kullanımı mümkün, çünkü std::complex<> tek bir arguman ile initialization yapılabilir. Bu şekilde kullanıldığında 2inci değişken de birince değişkenin değerini alır.

namespace std {

template<typename T>

class complex {

constexpr complex( const T& re = T() , const T& im = T() );

...

}

}