

int x = 10

pvcat(x++) PR value exp

x++ PR value

--x L value

++x L value

x+5 PR value

Struct Nec {

int x; }

};

int foo();

Struct sin {

static int x; }

pvcat(foo()); PR value

int &bar();

pvcat(bar()); L value

int (&baz());

pvcat(baz()); X value

pvcat(45) PR value

'A' PR value

"di" L value

pvcat(foo()); L value

pvcat(move(x)); X value

pvcat(static_cast<int&>(x)); X value

pvcat(move(foo())); L value

Nec mynec; sin mysin;

pvcat(mynec); L value

Nec& r = Nec{}; sağ taraf değer, sağ taraf refrens

pvcat(r); L value ama bir sin T'de de olmazsa L value exp.

pvcat(mynec.x); L value

pvcat(Nec{}.x); X value

pvcat(move(mysin).x); L value

Casts

PR value \Rightarrow T

L value \Rightarrow T &

X value \Rightarrow T & &

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

1

2

3

4

5

6

7

8

9

0

.

,

:

;

{

}

[

(

)

&

*

#

%

+

-

*

/

\

^

~

<

>

=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

/=

\=

^=

~=

<=

>=

!=

==

+=

-=

*

R value olmalı.

Nec & & r =

Nec x ?

const Nec cx ?

const Nec & & r = Nec { } ;

move(cx) ;

legal

Const S & cx ;

foo (std::move(cx)) ;

T& const T& T&& const T&&

L value	1	2	X	X
const L value	X	1	X	X
R value	X	3	bad	2
const R value	X	2	X	1

S { } ;

struct S { } ;

void foo (S&) { cout << "S&" << "

void foo (const S&) { cout << "const S&" << "

void foo (S&&) { cout << "S&&" << "

void foo (const S&&) { cout << "const S&&" << "

koplama void push_back (const T&); L value - ben alırsam

tarfıma void push_back (T&&); R value - ben alırsam

Myclass m = r ;

Myclass m = std::move (r) ;

class Myclass { }

1.23

int main () {

Myclass m;

Myclass && r = std::move(m); taşınma yok

Myclass y = r; copy ctor çağırıldı. = = = tıpkı L value exp

Myclass y = move(r); move ctor = var

Myclass z;

z = r;

} -> copy

z = std::move(r);

static_cast <(exp)> Ayntı gösterir.

std::move (exp)

Template <typename u> universal ref

void foo (u&&);

T&&

&)

sol taraf referansı sol taraf ref

T&

&&)

eger sağ taraf // in binding

T&&

& // sol & // sağ &

oluşursa L value ref'e bağlanır.

T&& && saf taraf, refine sağ taraf (ref'ye baglanır)
nichts mitgebracht, esleyebilir, nichts mitgebracht (bağlantı)
 \Rightarrow T&& reference collapsing

void func(int x) noexcept (sizeof (int) < 8);
int'in sizeof'u 8'den büyükse noexcept garantisi veriller.

template <typename T>

void func (T x) noexcept (noexcept (x+1));
specifier operator

```
constexpr bool b = noexcept(func(12));  
^  
true
```

~~destructor, copy ctor, assignment~~ bildirmemizle bu durumda move memberler ~~not declared~~ statusine koymus oluyoruz.

* Herhangi bir üye member bildirimini ile copy memberlar deleted statisine düşer

~~8~~ Bir class'in her zaman destructoru var.
indetermined

value initialization int x[3];

R40

(unnamed) value optimization

named return value optimization
NRVO

int ~~88~~ r = 10;

r → bu ifadenin value kategorisi L value
↳ bu ifadenin tipi int

int * p { };

p → bu ifadenin tipi int *

↳ bu " " value kategorisi L value

* tanımlamak üzere

* delete ederek

* default "

C++ 17 sonrası mandatory copy collision değişti

bir nesnenin oluşma sorumluluğunu varsa pr value exp.

* value'ye dönüştürülerek. o zaman result object

* Geçici nesnelerin dönüştürülmesi: Bir işlevin geçici
bir nesnesi return ediyorsa. Kopyalanması yerine
değişiklik hedef nesneye aktarulması istenir

* MyClass func() {

MyClass obj;

// işlemler uygulanır

28 return NRVO ile doğrudan obj nesnesi oluşturular.

* delete edilmiş copy ctor olduğunda mandatory copy
collision olacaktır ama NRVO syntax hatalı olacak

NRVO

- a) halen bir optimizasyon
- b) derleyici yapmak zorunda değil
- c) debug modunda optimizasyon yapayabilir.
- d) kodun legal olması için diliin kurallarına göre legal şekilde regulülür. copy yada move ctor olmasının gereklisi.

C++17 öncesi:

- foo (Myclass $\{ \}$) move copy ctor çağrılacak
Myclass \cdot x = Myclass (Myclass $\{ \}$); ama şuan sadece ctor çağırılacak
move, copy ctorların bulunmasına gerek yok!

* Member olduğunda geçici nesne kullanmak verimlilik astıracı daha az ctor çağrıları.

- * auto val = Myclass $\{ \}$.foo(); mandatory RVO
- * bir fonksiyonda 1 den fazla nesne varsa NRVO bekleyemeyiz veya kalkılım varsa garanti edilemez.
- * Eğer pointer, ref'ler size konusunda değilse derleyicinin yazdığı move assignment, ctor izimizi görev ettir.
- * Fonksiyon ile nesne return iedərətən move kullanmak copy ellision'u engeller. Kullanınmak daha doğru
- * bir nesne döndüren fonksiyonun peri dönüs deyərini const yapmak move memberlərin çağrılmasını engeller.

template <typename T>
void func(T) noexcept(std::is_nothrow_constructible<T>());
* Destructor default olarak noexcept olarak tanımlıdır.

m_str(1000, 'A') fill ctor

m_str{1000, 'A'} initialization list ctor

Move ctor default'un hatalı olabileceği durumlar:

* birbirine bağlı data memberler

* pointerlar veya smart pointerlar.

Perfect Forwarding

template <typename T>

void foo(T&& arg)

// eğer T türü referans türü ise

bar(arg)

// eğer T türü referans türü değil ise

bar(std::move(arg))

template <typename T>

void foo(T&& arg)

{

bar(std::forward<T>(arg))

}

arg L-value referans ise yine L-value expression
oluyor // R // lvalue // rvalue // h //

* Bana gelen isteyen constsa bende o argumenti
diğer fonksiyona (bar) const olarak göndermiş oluyorum.

return std::forward(arg) ← lvalue

→ R value ref (move ctor)
push-back → const L value ref (copy ctor)
emplace-back → Perfect forward ile ctor ediyor.

template <typename T, typename U>
void call-foo (T &&t, U &&u)
{
 foo (forward<T>(t), forward<U>(u));
}
* Variyedik ; *

template <typename ... Args>
void call-foo (Args &&... args)
{
 foo (forward<Args>(args)...);
}

int x=4;

decltype(auto) y = x; y'ın int türündedir

decltype(auto) y = 12; y'ın int türündedir

int& foo();

decltype(auto) y = foo(); int &

decltype(auto) y = (x); int &

*+i ⇒ Value katex → L value

(Return

auto f = [] (auto func, auto &&... args) → decltype(auto) {
decltype(auto) ret = func(std::forward<decltype(args)>(args)...);
if constexpr (is_rvalue_reference_v<decltype(ret)>) {
return move(ret); → move xvalue returned by
else
return ret; } } → f() to the caller
↳ return the plain value or lvalue reference

→ Base classin kullanıldığı her yerde Derived class'ta
kullanılabilirlik ilkesi solid

```
class Base {  
public: Base() {} = default;  
void foo();  
void bar(); } ; → using Base::base;  
class Der : public Base {} ;
```

Protected kalitim ile private kalitim arasında ciddiye
alınabilecek tek fakt;

Protected kalitiminden taban sınıfın public bölümünden
fareniş sınıfın private bölümünde değil

protected bölümünde eklenir.

Bu durumda multi-level inheritance ile yeni işaretlenen
sınıflar bu öğelerle erişebilir.

template <typename>

Alias template

using RemoveReference_t = typename RemoveReference<T>; type;

template <typename T>

RemoveReference_t <T> func(T);

Fonksiyonun içinde parametre deşifterin kopyasını
sıkırtıp kullandıktan sonra parametreleri normal almak daha
mantıklı olabilir. void func(std::string str)
Böylece mandatory copy ellisten tetiklenebilir.
Daha iyi giderilen kodunlığını bir daha kullanmayı

auto ile template type deduction (çalıştırma) arasında tek fark
auto yaInitializer list parametre görevi biliken
template' e gerekmiyoruz!

* initializer list, copy constructor gelir

remove = Bir cengedeki belirli deşere sahip öğeleri
siliyor (lojik siliyor)

remove_if = Unary predicate in true değer verdiklerini
lojik silmeye yapıyor.

unique = Ardışık aynı deşere sahip öğelerin sayısını
bire indiriyor.

String_view → basının nerası dangling pointer.

f dinamik anırlı string'in bağlı olduğu nesne silirse

+ string nesnesidir. sonra bir yazı ilave ederiz reallocation yapılır

1- dangling pointer oluşturur. (null pointer)

2 Null terminated byte streama bükleyen bir yere

programının bir şekilde string_viewden bir adresi görmesi

(dereference) → pointer

~(T) sau { T } + ~pointer yazılışı

int (int, int)

* - FUNCST GİBİ Bir işlevsel makrosudur.

"function signature" içeresi bir karakter dizisi olmalıdır.

* Unary left fold

template <typename ... TS> ~~...~~ ellipsis

```
auto sum (const TS& ... args)
{ return ( ... + args);
  (((p1 + p2) + p3) + p4) + p5 }
```

* Unary ~~left~~ right fold

template <typename ... TS>

```
auto sum (const TS& ... args) {
```

```
return ( args + ... );
```

```
p1 + (p2 + (p3 + (p4 + p5))))
```

* template <typename ... TS>

```
int subtract (int num, TS ... args) {
```

```
return ( num - ... - args ); } Binary left fold
```

```
subtract (100, 50, 20, 7)
```

```
((((100-50)-20)-7) = 23
```

unordered
associative

Sequence Containers associative Containers Containers

+ C array

std::set

std::unordered_set

+ std::vector

std::multiset

std::unordered_multiset

+ std::deque

std::map

std::unordered_map

* std::list

std::multimap

std::unordered_multimap

* std::forward_list

* std::string

* std::array

İşlevsellik (mehmetçik, organizatör) İşlevsel fonksiyonlar

Nasıl işlevsellik (mehmetçik, organizatör) işlevsellik (mehmetçik, organizatör)

İşlevsellik (mehmetçik, organizatör) işlevsellik (mehmetçik, organizatör)

İşlevsellik (mehmetçik, organizatör) işlevsellik (mehmetçik, organizatör)

template <typename InIter, typename OutIter>

OutIter Copy (InIter beg, InIter end, OutIter destbeg)

{

 while (beg != end) {

 *destbeg++ = *beg++

 }

 return destbeg;

Vector<int> iVec {1, 3, 5}

list<int> ilist (3)

Copy (iVec.begin(), iVec.end(), ilist.begin());

copy (iVec.rbegin(), iVec.rend(), ilist.begin());

* Bit container'ın sonundaki 2'yi silen kod

int val = 2;

if (auto riter = find (iVec.rbegin(), iVec.rend(), val); riter != rend())

{ iVec.erase (prev(riter.base())); }

* Girilen isimden kaç tane olduguunu dindiren kod

Count (slist.begin(), slist.end(), name);

* Count_if syntax template <typename InIter, typename Upred>

int Count_if (InIter beg, InIter end, Upred f)

int cnt {0};

while (beg != end) {

 if (f(*beg)) {

 cnt++;

 beg++;

* std:: back_inserter fonksiyonu, konteynirlara element eklemeyi

bağlayacaktır. Bu, mevcut bir konteynirin sonuna elementler eklenet iken

dinamik olarak büyüyen bir konteynir kullanmanız gereken durumlarda özellikle kullanışlidır.

```

if (auto iter = find_if (svec.begin(), svec.end(),
    [key] (const string & s) {
        return s.find (key) != string::npos;
    } ;
    iter != svec.end()) {
        cout << "bulundu" << *iter;
}

```

eğer find-if fonksiyonu bir işleve uygun değil bulursa
 bu işlemin konumunu gösteren "iter" döndürür.
 eğer bulamazsa svec.end() döndürür. (2017-2018)

* std::transform amaç giriş elemanlarını başka bir
 aralığa dönüştirmek.

```

transform (ivec.begin(), ivec.end(), ovec.begin(),
    [ ] (string &s) { ... });
return reverse (s.begin(), s.end()); }

```

* std::foreach syntax

template <typename Iter, typename Ufunc>

Ufunc for_each (Iter beg, Iter end, Ufunc f)

while (beg != end) {

f(*beg++); }

return f; }

* structor binding

```
auto [iter_min, iter_max] = minmax_element(svec.begin(),
```

```
auto ip = *ip.first; svec.end());
        ip = *ip.second;
```

pair<vector<string>::iterator, vector<string>::iterator> ip

(2017-2018) 2. sınıf 1. Dönem 1. Ünite

* replace_if islevi, verilen bir konteynırın belirli bir aralığın üzerinde dolayarak, belirli bir koşulu sağlayan öğeleri belirli bir değere değiştirdi.

replace_if(numbers.begin(), numbers.end(), isOdd, 0)

tek sayıları 0'la değiştirecektir.

* std::remove_copy islevi belirli bir değeri silmek ve yeni bir aralık oluşturmak için kullanılır.

remove_copy(ivect.begin(), ivect.end(), back_inserter(dest), 3);
ivect içindeki 3 değerleri hariç diğerlerini dest'e kopyalar.

* std::remove_copy_if (svec.begin(), svec.end(), back_inserter(dest))
[] (const string& s) { return s.contains('a'); };

* std::iterator_traits kullanmanız gereken durumlar genellikle iteratorların programlamada kendi iterator türlerini oluştururken ihtiyaç duyduğunuz. iterator özellikleri erişmek istediğiniz durumlarda.

std::iterator_traits<vector<int>>::iterator

iteratorın Kategori türüdür. $\text{input_iter} \leftarrow \text{output} \Rightarrow \text{iterator_category}$

→ işaret ettiğim elementin tipi $\leftarrow \text{value_type}$

iteratorın işaret ettiğim elementin $\Rightarrow \leftarrow \text{difference_type}$

$\Rightarrow \text{pointer}$

$\Rightarrow \text{reference}$

list<int> myList { 1, 3, 5, 7 }

vector<int> ivect { myList.begin(), myList.end() }

list'ı vector'e çevirdik range vererek

* vec.pop_back(); vec.erase(vec.begin());
↓ ↓
Son öğeyi siler. İlk öğeyi siler.

* vec.erase(vec.begin(), next(vec.begin(), 3));
↓
Baştan 3. karakteri siliyor.

* silme
vec.clear();

vec.erase(vec.begin(), vec.end());

vec.resize(0);

vec = {};

vec = vector<string> {};

* vector<int>{3, 6, 8}; ivec ilistten kopylse true döndürür.

list ilist {3, 6, 8};

lexicographical_compare(ivec.begin(), ivec.end(), ilist.begin(), ilist.end())

* auto y = [=] (int x) {return x*x;}; (10);
↓
Eğer buradaki scopelerde
bir veri ya da hepsi
bildirmek için

parametreler işlenir yapılışı
baş

parametreye
geçilen değer.

[=] → capture all

[x] → x bildirili

[&x] → x'e değer atanırsa scopelerde o değerle gelir.

| int x=10;
| [val = x*6] | → initializer capture

[] () mutable {code} → lambda ifadesinin const olduğunu kaldırır.

[] () → int {code} → dönüş yapacağı değer int'tir. qikanm!

[] () noexcept {code}

[] () constexpr {code} → fonksiyonun constexpr olması engelliyor
bir ifade varsa syntax hatalı olur.

* Generalized lambda expression

```
auto f = [] (auto &&... param) {
    print (std::forward<decltype(param)>(param)...);
}
f(12, 3.4, 45f, "necati");
```

~~int (fp)(int) = [] (int x) {return x*6;};~~
fp(20);

* auto f = +[] (int x) {return x*5*3;}

positive lambda idiom, f function pointer form olur.

* erase remove idiom

```
iVec.erase(remove(iVec.begin(), iVec.end(), 3), iVec.end());
auto = std::erase(iVec, 3) C++20
```

↳ kag deger sildiyse doldurulur.

* Vector bellekte ardışık bir bloktı depolandığından, rastgele erişmek daha hızlıdır. Ancak deque daha esnek bir yapıya sahip olduğu için baş ve sondan element eklemeliğinde işlevleri daha hızlı olabilir.

* vector'in bellek bloğu genizlendiğinde geçerli olan iteratörlerini korur. Ancak deque'da bellek blokları parçalara ayrıldığında, yeni bloklar eklenip çıkarıldığında iteratörler geçerliliğini yitirir.

* partial_sort (vec.begin(), next(vec.begin(), n), vec.end()); ilk n'ci degerde kalar olan kismi sıralar

* vector<string> destVec(10); {shablon} std::sort () []

partial_sort_copy (vec.begin(), vec.end(), destVec.begin(), destVec.end()); [] en küçük 10'u destVec'e kopyaladı {shablon} () ()

* ~~A~~ nth-element (vec.begin(), vec.begin+5000, vec.end());
5000. eleman medyani gösterir. Diğerleri sıralı doğrudır.

partition partition algoritması bir aralıktaki elemanları belirli bir koşula göre böler ve bu koşulu sağlayan elemanlar ~~bir~~ taraflına yerleştirirken, koşulu sağlamayanları diğer taraflına yerleştirir.

partition-copy (vec.begin(), vec.end(),
back_inserter(x), back_inserter(y),

[] (const string & s) { içinde i olalar x'te düşer.
return s.contains('i'); } ; y'de

* ① is_sorted (begin(a), end(a))

② is_sorted (begin(a), end(a), greater<>{})

1. küçükten büyüğe sıralıysa 1 döndür.

2. büyükten küçüğe 1 1 1 1 .

* make_heap (ivec.begin(), ivec.end());

heap yapısını büyükten küçükne + nadir - sağlama - tarihi

pop_heap en büyük veya en küçük öğeyi, kık düşünden çıkararak en büyük deperi back'e getirir. heap' gerekir

push_heap

heap yapısına yeni bir öğe ekler ve heap özelliğini koruyarak heap yapısını günceller

* `fwlist.insert_after (iter, -1);`

iter'in olduğu kısma '-1' ekler.

`fwlist.erase_after (iter);`

iterin olduğu kismi siler.

`fwlist.insert_after (x, before.begin(), {444, 13});`

Normal begin'i çağırırsan front ardına verileri ekler.

en başa eklemek istiyorsan before.begin()'i çağırır.

* `Set<string, bool> myset;`

Sette her veriden 1 adet olurdu →

`auto retval = myset.insert (name);`

↙

`pair<set<string>::iterator, bool>`

Retval. second → veri eklenmeye true!

* ~~Toplum~~ emplace_hint(): verilen bir pozisyon

izaretçisine göre bir öğe eklenmeye şansılık optimize

edilmiş bir eklene işlemi gerçekleştiriliyor. set ve map'te

`Myset.emplace_hint (myset.begin(), 45);` Kullanılabilir

olması gereken yere 45 ekler.

*

`Set<int> Myset {3, 6, 8, 12, 78, 89, 234};`

`auto beg = myset.find (8); auto end = myset.find (80);`

`auto x = Myset.erase (beg, end); x = 3 für 8, 12, 78 silindi.`

* String-view bir fonksiyon geri döndür deşerit
yapılmamalıdır. dangling pointer

* string str(10, 'a');

string-view sv{str};

str.append(100, 'x');

cout << str << "\n";

* SV.remove-prefix(mio(sv.find_first_not_of("a"), sv.size()));

* foo("sinan"sv);

* string-view sv {"recati ergin"};

cout << sv.starts-with("recati") << "\n";

cout << sv.ends-with("gin");

* std::optional Bir bardağın dolu olmasa lader boş
olması da doğal.

optional<string> opt;

opt = "sinan"

Vesige sahib mi? → has-value
→ bool

cout << *opt << "\n";

opt.value() → boşsa bad-optialAccess exception giderir.

opt.value-or("sid") → boşsa sid gelmiş doluysa dolu olan
değer döner.
düğünden dopo semantik

* başta iken Ø veriyi olur edmez.

Ø

kontrol şartları doğrulanır ve hata tespit

auto op = make_optional<Date>(11, 07, 21)

auto op2 = optional<Date>(in-place, 11, 07, 21)

op.reset(); == op=nullopt; = op={};

op.emplace(11); optional nesnesini destroy etti ve
11 ile ctor çağırır.

Optional kullanım alanı söyle bir varlık söz konusu tı onun

bir değerinin olması bekler olmaması da son derece normal

+ fonksiyonun geri dönüş değerinin optional olması

+ " " parametre -- -- =

+ Sınıfın veri elemanları,

const degrupluğu lifetime optimizasyon

scope leakage.

taşıma semantigi

Varın fonksiyonları

Variant <int, double, float, char> var{};

hs lcs₀ Alternative <int>(var) → true döndürür

var.index() → 0 döndürür.

Variant <int, string, Date> var{emplace_index<2>,

10, 5, 1988};

{emplace_types<Date>} ;

get<2>(var) *get metodunu sağla mühendis*

get<Date>(var) = (Bu, 12, 2020);

var.emplace<string>(10, 'A');

perfect forwarding, eski değeri destroy etmesi

~~Monostate~~ basen variant'te nulladığımız alternatiflerin bir veya birden fazla default constructoralı oluyor ama biz yine de variant oluşturmak istiyoruz, özellikle ilk elementin default ctor'u olmalı ki variant nesnesi defter edebilmek için

variant < monostate, int, double, string> ux;

ux = monostate {};

ux = {};

ux.emplace<monostate>();

ux.emplace<0>();

~~Template <typename T>~~

Void operator() (const T& t) const {
||

Void operator() (const auto& t) const

Variant, virtual dispesialı adayıdır. yerine nulladığında verimlilik sağlar. ama yeni sınıflar sık sık ekleneneceğse virtual class tekrar düşünülebilir.

any a {};

Cout << any->int(a) << "\n";

~~badcast~~ ↴ int olmadığı için exp. throw exception

bad-any-cast

any {};

any-cast < int &>(a) = 67;

Cout << any-cast < int >(a) << "\n";

make_any<Date>(4, 4, 1844); // = any a {in-place-type< Date>, 5, 5, 1855};

any $\&x \in \{46\}$

```
if (any_cast<double>(&ax)) {
```

```
    cout << "double tutuyor";
```

```
else {
```

```
    cout << "double tutmuyor";
```

```
}
```

any $\&y \in \{43, 43\}$

```
if (auto ptr = any_cast<double>(&ay)) {
```

```
    cout << "double tutuyor";
```

```
    cout << "Value" << *ptr;
```

```
for (volatile int i = 0; i < 1000; ++i) { }
```

\hookrightarrow derleyici bunu silmez

Compiler optimizasyonu

İşlemci //

Caches (cashing) mekanizması //

* for-each'in geri dönüş değeri callableıdır.

\Rightarrow world says just ignore the tail

using `tptype = std::tuple<int, double, long, string>;`
int main () {

`tuple<size_t <tptype> → 4`

`tuple-element-t < 3, tptype> → string`

`array <double, 20> ar {1, 5., 21.} 3`

`cout << get <0> (ar); → 1`

`tuple-fonksiyonları UL array'de kullanılır` 
`interface'nı.`

`auto [x, y] = make_tuple (10, 4, 5);`

`lambda init capture`

`auto f = [&x=x] { return x*2; };`

`↳ c++17 vb kullanıyorsun x'in referansı`

`çokyla capture etmek için bunu kullanıgersün`

`auto f = [x] { return x*2; } C++20 ✓`

`namespace {`

`template <>`

`struct tuple-size <Person> : integral-constant`

`<size_t, 3u> {};`

`or`

`template <> struct tuple-element <0, Person>`

`{ using type = int; };`

	Equality	Ordering
Primary	$= =$	$< = >$
Secondary	$!=$	$<, >, <=, >=$

Strong-ordering → equal, equivalent, less, greater
 Weak-ordering → \sim
 Partial-ordering → ordered, \approx , \sqsubseteq , \sqsupseteq

$$S \leqslant T \Rightarrow S \leqslant T \geqslant 0 = S \geqslant T$$

↓
-1

peri
don't
stop
↓
1

$$T \leqslant S =$$

auto operator \leqslant (const Nec&) const = default;

C++ 20 ile geldi noexcept sunsa sterlegimin
operatörlerde noexcept [nodiscard] vbolsa diğerlerde söyle
dur. \leftarrow operatorde default edilebilir.

auto result = lexicographical_compare_three-way(mList.begin(), mList.end(),
myVec.begin(), myVec.end());

* Bir C++ sınıfındaki varı üyesinin mutable olarak işaretlenmesi, bu üyelerin sınıfın const niteliğli üye fonksiyonlarından bile degistirilebilmesini sağlar.

* Bir diziyi capture ederken array diley yoktur.

range base for loop

Struct Nec {

```
int mx[], my[]; → kullanma  
void foo() {  
    auto f = [this] (int val) {  
        return val + mx[0] + my[0];  
    };  
}
```

lambda ifadesi default olarak constexpr bildiriliyor ama son harici olarak constexpr yazarsan ve senin ifaden constexpr olmayan bir ifadeyse syntax hatası olursun.

lambda ifadelerinin en büyük faydalari:

Local dözyede ihtiyacın olduğu yerde kullanılması böylece kodun daha iyi takip edilmesi;

```
auto fn = [] <typename T> (T t) { t += f }  
familiar template syntax
```

```
auto fn = [] (auto&& s) {  
    forward <decltype(type(s))>(s); }  
forwarding constructor
```

Sınıfın fonksiyon adresi tırına dönüzen yapan fonksiyon

```
auto f = [] () { }; closure type → f  
dözyeyinin bildiği bir tür.
```

```
auto f = +[] () { }; int (*i)(int x) + i
```

Struct Foo {

int bar;

(33) Foo (int bar) : bar{[&] {

// Complex initialization of bar

```
} () } }
```

Lambdalar artık default ctor ve copy assignment fonksiyonlarının delete edilmişken şimdi delete ediliyor.

* auto fn = [] <typename T> (T&& x) {
 foo(std::forward(T)(x));
}

auto fn = [] (auto&& x) {
 foo(std::forwarddecltype(x)(x));
}

auto callfoo = [] <typename ...Args>
(Args&& ...args) {foo(forward<Args>(args)...);};

auto fn = [] (auto x) {return x;x;};
fn.operator() <double> (3);

Lambda fonksiyonunu double parametre ile çağırma yöntemi:

[[fill][align]]	[sign][#][0]	[width][[-precision]]	[type]
↓ doldurma karakteri right left internal	↓ tamsayıların şareti type ile beraber kullanılır	↓ heading zero yazma alan genişliği	↓ noktadan sonra Bewerat sayısı türden ne oldupunu anlatıyor

cout << format ("|{:12}|", x); sola dayalı 12 tane yer kollar
(":{<12}", x); sola dayalı "
(":{>12}", x); ortaya "
(":{:-12}", x); "-" "<" 12 tane - karakteri "
(":>{", x, width})
(":+{",) ignoreler yazır
(":+07d", x)

String name {"TUNAHAN"}

cout << format ("{:24.4}", name); sağda doğru TUNA 24 yerlesdir
 ("{:24.4}", name); sola " " " " " "

String str;

int x = 345655;

format_to(back_inserter(str), ".{:^16x}", x);

Cout << "length :" << str.length() << "\n";

Cout << str << endl;

Custom type'ler için

a) std::formatter sınıfı kullanma

g++ -c main.cpp

g++ -c kutuphone.cpp

g++ -o main.exe main.o kutuphone.o

MingW32-make

const init global you da statik ömürli nesnelerin
 desenle zamanında ilk değer almalarını istiyorsak, gallent.
 Zamanında değer almasını istemiyorsak sadece static
 initialization olsun istiyorsak kullanabiliriz. C++20

constexpr bi değişkene atılan constexpr değer dindren faktisine

constexpr int foo (int x) {return x*5; }

constexpr int x = foo(5);

const init int y = foo(6);

x++ ; illegal

y++ ; legal

const init = constexpr + const

+ fonksiyon const değer dindren

ile mutable hale getirilemeyez

`[[nodiscard]]` bu fonksiyonun geri dönüş değerini kullanılmaması
desteyici uyarı versin.

`[[nodiscard ("message")]]` int foo (int);

(void) foo(2); → discard, uyarı mesajı çıkmaz

class [[nodiscard]] Myclass {};

Myclass foo(); → fonksiyon tanımı

foo(); → uyarı mesajı verir.

Template Özellikleri

* function template

* class template

* variable template

* alias template

* concept

typename template <typename T>

Concept Integral = std::is_integral_v<T>;

template <typename T> / template <Integral T>

requires Integral <T> foo(T); / void func(T);

template <typename T>

Concept as_int = integral || is_convertible_v<T, int>;

template <typename T>

requires as_int<T> void foo();

foo<long>(); ✓

foo<Nec>(); X - Nec'in int() operatörü olsaydı

