7.Hafta

> 05_08_2023

İçindekiler

Space Ship Operatörü

Bu operatörü çoğunlukla doğrudan kullanmıyoruz. Kullandığımız yer kendi türlerimiz için bu operatörü overload ettiğimiz yerler. Artık operatörler arasında bir ayrım yapılıyor ve bu ayrım:

1. `==` ve `<=>` operatörü
2. `!=`, `>`, `<=`, `>=`, `>` operatörleri

- Primary operatörler ***reversable*** (*derleyici uygun fonksiyon bulunmaması durumunda operandları yer değiştirebiliyor*).
- Secondary operatörler ise **rewritable** (eğer x!=y yok ise bunu !(x==y) olarak değiştirebilir.)

Bu da kendi sınıflarımız için `!=` değili yazmamıza gerek yok `==` yazmamız yeterli olacak. Aynısı diğer operatörleri yazmak yerine `<=>` operatörünü yazmamız yeterli olacak.

- Öğeler bildirim sırasına göre karşılaştırılır.

- `==` fonksiyonu için aşağıdaki örtülü dönüşümler yapılıyor.

```
class Myclass {
public:
    Myclass(int x = 0) : mx{x} {}
    bool operator==(const Myclass &other) const
    {
        return mx == other.mx;
    }
private:
    int mx;
};

int main()
{
    Myclass x{10}, y{20};
    auto b1 = x == y; // x.operator==(y)
    auto b2 = x != y; // !(x.operator==(y))
    auto b3 = x == 5;
    auto b4 = 5 == x; //C++20 öncesi bu legal değildi, derleyici 5 ==
x'için x.operator==(5) çağrısına dönüştürüyor.
}
```

- Sadece `<=>` default edilirse diğer 6 karşılaştırma işlemini de yapabiliyoruz

```
class Myclass {
public:
    Myclass(int x = 0) : mx{x} {}
    auto operator<=>(const Myclass &other) const = default;
    //auto operator==(const Myclass &other) const = default; bunu da
default ediliyor
private:
    int mx;
};

int main()
{
    Myclass x{10}, y{20};
    bool b{};
    b = x == y; // x.operator==(y)
    b = x != y; // !(x.operator==(y))
    b = x != 5;
    b = 5 != x;
    b = x == 5;
    b = x < y; // x<=> < 0 şeklinde de yazabilirdik fakat bu şekilde daha
okunaklı
}
```

- Default edilmiş durumda derleyici yazdığı fonksiyonu, eğer öğelerin karşılaştırılması `noexcept` garantisi veriyorsa fonksiyonda `noexcept` garantisi veriyor.

```
template <typename T>
class Type{
public:
    [[nodiscard]] virtual std::strong_ordering
     operator<=>(const Type &)const requires(!std::same_as<T,bool>)=
default;
};
```

- Geri dönüş değerinin `auto` olması generic kod kısmında işimize yarıyor. Burada geri dönüş değerini trail return type ile belirleyebiliriz.

```
template <typename T, typename U>
class Myclass {
private:
        T mx;
        U my;
public:
    auto operator<=>(const Myclass& other) const ->
std::common_comparison_category_t<decltype(mx<=>mx),decltype(my<=>my)>
    {
        if(auto res = mx<=>other.mx; res != 0)
            return res;
        return my<=>other.my;
    }
};
```

- Vectorler için Lexicographical comparison yapılıyor.

```
int main()
{
    using namespace std;
    vector v1(100'000, 5);
    vector v2(5, 6);
    bool b = v1 < v2; // false

    //lexicographical comparison için 2 farklı container kullanabiliyoruz.
    list mylist{3,6,7,9,1};
    vector myvec{3,6,8,1};
    cout <<boolalpha <<lexicographical_compare(mylist.begin(),
mylist.end(), myvec.begin(), myvec.end());//true
}
```

- C++20 ile öyle durumlar varki range'leri spaceship operatörü ile karşılaştırmamız gerekebilir, bunun için lexicoğraphical_compare_three_way algoritması eklendi ve bu fonksiyonun geri dönüş değeri `<=>` operatörünün geri dönüş değeri.

```
std::ostream& operator<<(std::ostream& os, const std::strong_ordering& so)
{
    return so == 0 ? os << "equal" : so < 0 ? os << "less" : os <<
"greater";
}

int main()
{
    list mylist{3,6,7,9,1};
    vector myvec{3,6,8,1};
    auto result = lexicographical_compare_three_way(mylist.begin(),
mylist.end(), myvec.begin(), myvec.end());//true
    cout << result << "\n";
}
```

Lambda İfadeleri, C++20 ile Gelen Yenilikler ve Tipik Idiomlar

- C++11 standartı ile dile eklendi.

Lambda ifadesi karşılığında bir local sınıf oluşturuyor ve ifade kategorisi PR-Value. Derleycinin oluşturduğu sınıf türüne `closure-type` deniyor. Nesneye ise `closure object` deniyor. Nested function'ın C++ karşılığı. Fonkisyon ihtiyacı olan nokta ile fonksiyonun arasındaki mesafeyi kapatıyor. Fonksiyona nerede ihtiyaç var ise orada tanımlıyoruz. Derleyicinin optimizasyon olasılıkları da optimize edilmiş oluyor.

Örnek sorular

```
int g = 99;
auto fx = [=]{return g+1;}
auto fy = [g=g]{return g+1;}

int main()
{
    g = 500;
    cout << fx() << "\n"; //100
    cout << fy() << "\n"; //100
}
```

```
int main()
{
    auto x = {static int x{}; return ++x;}
    decltype(x) y;
    decltype(x) z;
    cout << y() << y() << z() << "\n";
    cout << y() << z() << z() << "\n";
}
```

```
int main()
{
    const int x = 10;
    //auto f = [x]()mutable{++x;}; Sentaks hatası.  Capture close'da x ismi
kullanılmış
    //copy-capture ve burada sınıfın veri elemanı const, sınıfın üye
fonksiyonu non-const olmasına rağmen
    //sınıfın üye elemanı const olduğu için sentaks hatası oluşuyor.

    //auto g = [x = x](){++x;};  Sentaks hatası.
    // Bunun legal olması için mutable olması gerekiyor, sınıfa bir veri
elemanı koyacak ve buradaki x'ile initialize edilecek.
    //sınıfın üye fonksyionu const olduğu için burada sentaks hatası
oluşuyor.
    auto h = [x = x]()mutable{++x;};
}
```

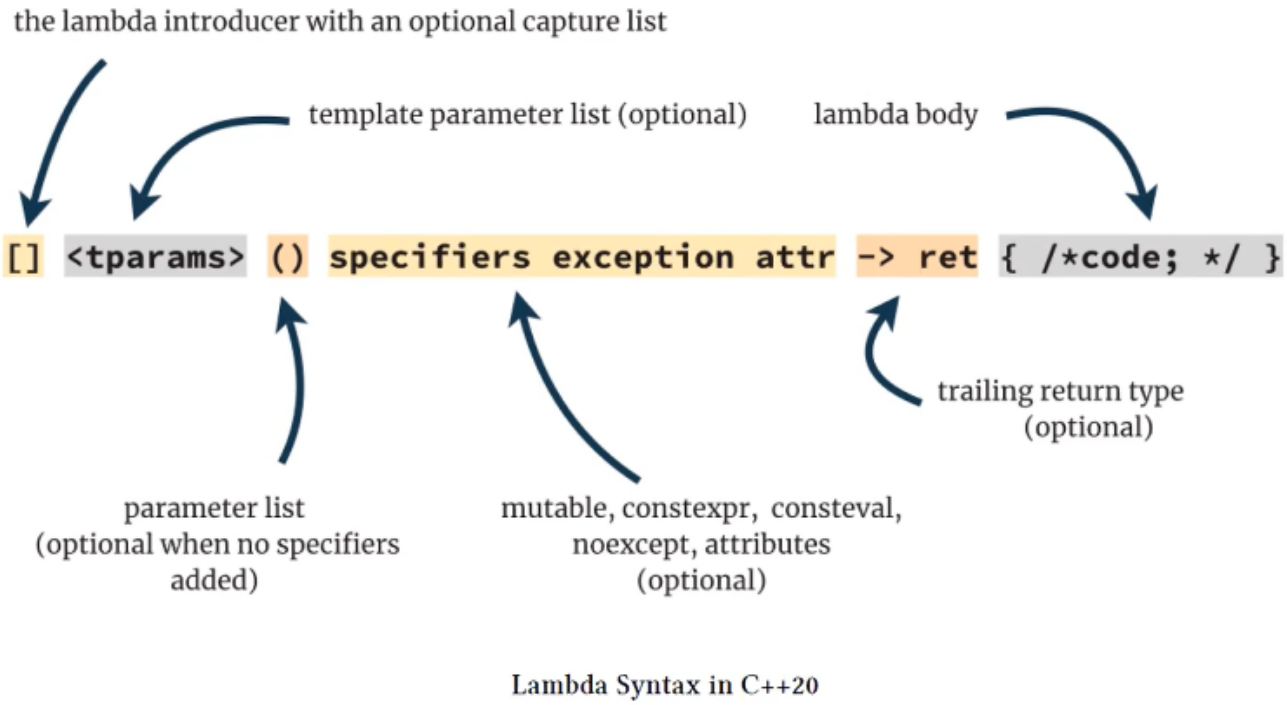
```
int main()
{
    int x = 4;
    auto y = [x = x + 1, &r = x]()
    {
        r+=2 ;
        return x*x;
    }();
    cout << x << " " << y << "\n"; // 6 25
}
```

```
int g = 5;
int main()
{
    auto f = [](int x = ++g){return x*x;};
    auto x = f();
    auto y = f();
    cout << x << " " << y << " " << g <<"\n"; // 36 49 7
}
```

```
double i{};
int main()
{

    auto f = [i =0 ]()-> decltype(i) {return 1;}();
    cout << is_same_v<decltype(f), int>;
}
```

Hızlı Temel Tekrar



Lambda Syntax in C++20

- Optional bir kısmı C++20 ile geldi.

```
int main()
{
    [](){}; //en basit lambda ifadesi
    [](){}(); //fonksiyon çağrı operatörünü böylede yazabiliyoruz.
    []{}(); //yukarıdaki ile aynı

    [](int x){return x*6;}; //burada fonksiyon çağrısını yapmıyoruz. ; ile
geçici nesnenin ömrü burada bitiyor.
    [](int x){return x*6;}(10);//bulunduğu yerde doğrudan çağırmak.
Immidiately invoked function expression
    // bu fonksiyonu birden fazla kez çağırmak istersek closure-type'ı bir
değişkene atamamız gerekiyor.
    auto f = [](int x){return x*6;}; //isimlendirilmiş bir function-object
yapabiilriz.
    f(10);
    f(20);
    // İsim vererek kodu okuyana yardımcı olunuyor ve genelde const
semantiğine uyulması gerekiyor.
}
```

- İsimlendirilmiş lambda ifadeleri const olarak tanımlandığında optimizasyon ihtimali artıyor.
- En sık kullanılan senaryo da lambda ifadesini bir fonksiyonu argüman olarak gönderiyoruz.

```
template <typename F>
void func(F f);
```

```
int main()
{
    func([](int x){return x*6;});
}
```

- Bazı durumlarda lambda ifadesi'nin türünü bilmemiz gerekmiyor fakat bazı durumlarda bu bilgiye erişmemiz gerekebilir.

```
int main()
{
    auto f = [](int x){return x*6;};
    decltype(f) g = f; //f'nin türünü decltype ile elde edebiliyoruz.
    cout << typeid(f).name(); //lambda ifadesinin türünü öğrenmek için
typeid kullanabiliriz.
    g(10);
}
```

- Closure objectlerin türlerinin aynı olup olmaması bazı durumlarda kritik olabiliyor.

```
int main()
{
    auto f1 = [](int x){return x*6;};
    auto f2 = [](int x){return x*6;};
    is_same_v<decltype(f1), decltype(f2)>; //false
    auto f3 = f1; //f1'in türüne göre f3'ün türü belirleniyor.
    is_same_v<decltype(f1), decltype(f3)>; //true
}
```

- auto type-deduction'da dikkat edilmesi gereken closure-type çıkarımı ile bu türden çağırılan fonksiyonun geri dönüş değeri olabilir.

```
int main()
{
    auto f = [](int x){return x*6;};
    auto f = [](int x){return x*6;}();

}
```

Sentaks Öğeleri

`[](){}`

- lambda fonksiyonu *default* olarak `const`.
- Eğer return ifadesi yok ise geri dönüş değeri default olarak `void`. Geri dönüş değerini ise çıkarım yoluyla elde ediyoruz.

- `mutable` ile *non-const* üye fonksiyon olarak tanımlayabiliyoruz.

> semantik tarafında `mutable` kullanımı programcıya semantik olarak o üye veri elemanının problemin domaini ile alakası yok, sınıf nesnesini değiştirmeyen fonksiyonlar olarak işliyor.

- Fonksiyon içerisinde birden fazla return statement olması ve bu return statement'lerin geri dönüş değerlerinin türleri farklı olması sentaks hatası

```
int main()
{
    auto f = [](int x)
    {
        if(x < 10)
            return x*6;
        else
            return 7.0;
    };
}
```

- Bu durumda geri dönüş değerinin ya hepsinin aynı olması gerekiyor, ya da *trailing return* type kullanmak.
- Ara bloğun içine artık bir kısıtlama yok.

```
int main()
{
    auto f  =[](){
        static int x = 10;
        ++x;
        cout << x << "\n";
    }
    f();
    f();
    f();
}
```

- Aynı ifadeyi tekrar edip birden fazla lambda ifadesini tanımlarsak ve bu fonksiyonlar içerisindeki static yerel değişkenler birbirinden farklı.

Parametre değişkenleri:

- C++14 ile default argüman alabiliyor. `[](int x = 10 ){}`
- Yeniliklerden biri generalized lambdalar. C++14 sonrası için bu parametrenin member template şeklinde yazdırabiliyoruz.

```
/*
fonksiyon:

class xyasdf
```

```
{
public:
    template<typename T>
    T operator()(T x) const
    {
        return x*6;
    }
};
*/


int main()
{
    auto f = [](auto x){return x*6;};
}
```

- Gönderilen birden fazla argümanın türü aynı olmak zorunda değil. C++20 ile gelen *familiar template syntax* dile eklendi.

Capture Close

- Derleyiciye lambda ifadesi ile bir sınıf kodu yazdırdığımızı biliyoruz. Parametre parantezi içinde `auto` kullandığımızda bunun template member fonskiyonu yaptığımızı biliyoruz. İlave bir sentaks kullanmak durumunda sınıfın bir veri elemanı olmuyor ve bu durumda bizim derleyiciye veri elemanı ekleyip bunu veri elemanı için constructor yazılmasını istiyoruz.
- Sınıfa eklenecek veri eleman referans olmayan herhangi bir türden olabilir. Burada bazı avantajları elde etmek için referans semantiğini kullanmak isteyebiliriz.
- Fonksiyon içerisinde statik ömürlü değişkenleri doğrudan kullanabiliriz. Bunları capture etmeye çalışırsak sentaks hatası oluşuyor.
- nesne'nin const olması ve sabit bir değer ile ilk değerini alması durumunda capture etmemize gerek yok.

```
int main()
{
    int val{35};
    auto f = [val](int a){return val*a;}; //copy capture
}
```

- `val` derleyicinin oluşturduğu sınıfın non-static veri elemanı.
- Birden fazla değeri virgüller ile ayrılmış liste ile capture edebiliriz.
- `=` tokenini kullanarak tüm görünen yerel değişkenleri capture edebiliyoruz.
- capture edildiğinde direkt array-decay oluşmuyor, diziyi direkt alıyoruz gibi düşünebiliriz.
- Init capture ifadesi kullandığımızda ise array-decay oluşuyor.

```
int main()
{
    int a[4]{};
```

```
    auto f = [a]{
        std::is_same_v<decltype(a), int*>; //false
        std::is_same_v<decltype(a), int[4]>; //false
    }
    auto f1 = [a = a](){
         std::is_same_v<decltype(a), int*>; //true
    };
}
```

- capture edilen nesnenin değerini değiştirebilmemiz için `mutable` olması gerekiyor.

```
int main()
{
    int x = 10;
    auto f = [x]()mutable{return ++x;};
    f();
    f();
    f();
}
```

- Nesnenin kendisini kullanmak istiyorsak ve o nesneyi değiştirmek isteyebiliriz, bu durumda yakalamayı referans semantiği ile yazmamız gerekyiyor.

```
int main()
{
    int x = 10;
    auto f = [&x](){return ++x;};
    f();
    f();
    f();
    cout << x << "\n";
}
```

- Bir kısmını referans bir kısmını copy capture edebiliiz

```
int main()
{
    int x{}, y{}, z{};
    auto f =[=, &z]{}; //x ve y capture, z referans semantiğii ile
    auto f = [&]{}; //hepsi referans semantiği ile
    auto f = [=]{}; //hepsi copy capture ile
}
```

- Özellikle gerekmiyorsa referans ile capture etmemeye çalış. Bazı durumlarda farkında olmadan *dangling reference* oluşturabiliriz.

```
auto foo()
{
    int x;
    return &x; //geri dönüş değeri int * olurdu ve otomatik ömürlü bir
nesnenin adresini dönmüş olucaktı.
}

auto fo()
{
    std::vector vec{1,2,3,4,5};
    return vec;
}

auto foo(int i)
{
    auto f = [&](int x){ x*i};
}
int main()
{
    using namespace std;
    auto fn = foo(10);
    auto val = fn(290);
    cout << val << "\n";
    //bu kod undefined behavior oluşturuyor.
}
```

- Bir sonraki derse soru, aşağıdaki closure type için const sol taraf referans değeri kullanılabiliyor mu?

```
int main()
{
    using namespace std;
    vector<int> ivec(10000, i);
    auto f = [ivec]{    }; //burada kopyalama yapıyoruz ve maliyeti çok
yüksek

}
```

Lambda init capture:

- C++14'ten önce unique_ptr'ı taşımak istediğimizde taşınamıyruz bunun için artık lambda init capture kullanıyoruz.

```
int main()
{
    auto uptr = make_unique<string>(10'000,'a');
    //auto f = [uotr](){} Sentaks hatası çünkü unique_ptr kopyalanamaz
    auto f = [&uptr](){}; //bu durumda unique_ptr'ı yakalıyoruz
    f();
```

```
    cout << (uptr ? "dolu" : "bos");
    auto f1 = [uptr = move(uptr)]{}; //bu durumda unique_ptr'ı taşıyoruz
    cout << (uptr ? "dolu" : "bos");
}
```

Sınıfın üye fonksiyonları içinde capture

- Bunu yapmak için üye fonksiyon hangi nesne için çağırıldıysa onu capture etmemiz gerekiyor. Bunun için biri `this` pointer'ını capture etmek.
- Copy-all ile capture edilebilirdir fakat C++20 ile DEPRECATED hale geldi.
- Referans yoluyla capture edilmesi hale mümkün.

```
struct Myclass{
    int mx;
    void func()
    {
        auto f = [this]{return mx;};
        auto f1 = [this]{return this->mx;};
    }
};
```

- Dangling referans oluşmaması için ve fonksiyon içinde *this'in bir kopyasını kullanmak için

```
struct Myclass{
    int mx;
    void func()
    {
       auto f = [copy_this = *this]{auto val = copy_this->mx;};
    }
};
```

	C++11/C++14	C++17	C++20
[this]	&(*this)	&(*this)	&(*this)
[*this]	X	*this	*this
[this, *this]	X	X	X
[&]	Implicit &(*this)	Implicit &(*this)	Implicit &(*this)
[&, this]	&(*this)	&(*this)	&(*this)
[&, *this]	X	*this	*this
[=]	Implicit &(*this)	Implicit &(*this)	Deprecated Implicit &(*this)
[=, this]	X	X	&(*this)
[=, *this]	X	*this	*this

- C++17 ile `*this` kullanabiliyoruz.

```
struct Myclass{
    int mx;
    void func()
    {
       auto f = [*this](){auto val = mx;};
    }
};
```

constexpr Keyword'ü

- C++17 standartı ile lambda fonksiyonu eğer `constexpr` olma koşulunu sağlıyorsa `constexpr` oluyor.

```
int main()
{
    auto f = [](int x){return x*5  ;};
    constexpr auto val = f(10);
}
```

- **constexpr** keywordünü kullannarak eğer bu durumu bozan bir koşul olduğunda sentaks hatası oluşturuyor.

```
int main()
{
    auto fsquare = [](auto val){return val*val;};
```

```
    std::array<int, fsquare(5)>a1;
    cout << a1.size() << "\n";

    auto f1 = [](int x){static int cnt = 0; ++cnt; return x*cnt;};
    //std::array<int, fsquare(5)>a1; sentaks hatası
    auto x = f1(10);
    //auto f2 = [](int x) constexpr{static int cnt = 0; ++cnt; return
x*cnt;}; sentaks hatası
}
```

- fsssquare(5) ifadesi c++17 ile birlikte artık varsayılan şekilde constexpr.
- Dolayısıyla fonksiyona sabit ifadeleri ile yapılan çağrıdan elde edilen geri dönüş değeri sabit ifadesi gereken yerlerde kullanılabilir.
- f1 tanımında static yerel değişklen kullanıldığı için artık bu lambda constexpr olamıyor.
- f2 nesnesinin ise tanımı geçersiz, eğer constexpr anahtar sözcüğü kullanılmasaydı tanımlı olucaktı.

> 13_06_08_2023

---

std::function

- Fazladan bir bellek alanı kullanıyor ve bazı durumlarda dinamik bir bellek alanıda allocate edebiliyor.

```
#include <functional>
int main()
{
    using namespace std;
    auto fn = [](double x){return x*x+.3;};
    cout << "sizeof(decltype(fn))= " << sizeof(decltype(fn)) << "\n"; // 1
byte
    std::function f = fn; // C++17 ile dile eklenen CTAD.olmasaydı
std::function<double(double)> yazmamaız gerekirdi
    cout << "sizeof(decltype(f))= " << sizeof(decltype(f)) << "\n"; // 1
byte
    cout << fn(5.8763)<<  " " << f(5.8763) << "\n";

}
```

- Bir nesne ile denediğimizde allocation'u görebiliyoruz

```
void *operator new(std::size_t sz)
{
    std::cout << "new called with size " << sz << "\n";
    if(sz==0)
        ++sz;
    if(void *p = std::malloc(sz))
        return p;
    throw std::bad_alloc{};
```

```
}

class Myclass
{
    unsigned char buf[512]{};
}

int main()
{
    Myclass m;
    auto fn = [m](int x){return x*x;};
    cout << "sizeof(decltype(fn))= " << sizeof(decltype(fn)) << "\n"; //
512 byte
    std::function f = fn; // C++17 ile dile eklenen CTAD.olmasaydı
std::function<double(double)> yazmamaız gerekirdi
    cout << "sizeof(decltype(f))= " << sizeof(decltype(f)) << "\n"; // 40
byte fakat burada 512 byte bellek alanı allocate ediliyor.
}
```

- Kullanılması gereken durumlar haricinde kullanma.

Generalized Lambda

```
int main()
{
    auto get_size = [](const auto &c)
    {
        return std::size(c); //c-arrayleri için bu şekilde yazmamız
gerekiyor.
    };
    vector<int> ivec(30);
    list ml{1,2,3,4,5};
    cout << get_size(ivec) << " " << get_size(ml) << "\n";
    cout << get_size("ali") << "\n";
    int ar[20]{};
    cout << get_size(ar) << "\n";
}
```

- generic lambda ile normalde yanlış bir şekilde çıkarılacağı yerde, onun doğru şekilde çıkarılmasını sağlıyor.
- Bazı temalarda birden fazla kez farklı türler için aynı kodu yazmamız ürettirebiliyoruz.
- Yazım kolaylığı sağlıyor.

```
int main()
{
    using namespace std;
    vector<pair<string, string>>pvec;
    for(int i = 0; i< 1000; i++)
```

```
        pvec.emplace_back(pair{rname(), rownt()});
    //sort(pvec.begin(), pvec.end(), [](const pair<string,string>6x, const
pair<string,string> &y{ return pair{x.second, x.first} < pair{y.second,
y.first}; } ) );
    sort(pvec.begin(),pvec.end(), [](const auto &x, const auto &y){ return
pair{x.second, x.first} < pair{y.second, y.first}; } );

    for(cons auto &[name, town]: pvec)
        cout << name << " " << town << "\n";
}
```

- Bir başka örnek

```
void f1(vector<shared_ptr<string>>&svec)
{
    sort(begin(svec), end(svec), [](const shared_ptr<string> &x, const
shared_ptr<string> &y){return *x < *y;});
    for_each(begin(svec), end(svec), [](const shared_ptr<string> &sp){cout
<< *sp << "\n";});
    cout << "-------------------\n";
}

void f2(vector<shared_ptr<string>>&svec)
{
    sort(begin(svec), end(svec), [](const auto &x, const auto &y){return *x
< *y;});
    for_each(begin(svec), end(svec), [](const auto &sp){cout << *sp <<
"\n";});
    cout << "-------------------\n";
}

int main()
{
    vector<shared_ptr<string>>svec;

    svec.emplace_back(new string {"deneme1"});
    svec.push_back(make_shared<string>("deneme2"));
    svec.emplace_back(make_shared<string>("deneme3"));
    f1(svec);
    f2(svec);
}
```

- Bir önemli tema da generic lambda'nın perfect forwarding yapabilmesi.

```
void foo(std::string &)
{
    cout << "lvalue\n";
}
void foo(std::string &&)
```

```
{
    cout << "rvalue\n";
}
void foo(const std::string &)
{
    cout << "const lvalue\n";
}

int main()
{
    auto f = [](auto &&x){ foo( std::forward<decltype(x)>(x));};
//universal referans parametre
    //C++20 ile familiar template syntax ile aynı şeyi yapabiliyoruz.
    std::string name{"deneme"}
    const std::string name1{"deneme1"}
    f(name);
    f(name1);
    f(std::string{"deneme2"});
    auto y = f(x);
    cout << y << "\n";
}
```

- Variadic template de olabilir

```
template <typename...Args>
void print(Args&&...args)
{
    std::initializer_list<int>{((std::cout << std::forward<Args>(args) <<
'\n'),0)...;};
}

int main()
{
    using namespace std;
    print(2,4.5,"denem",string{"fffff"});

    auto fn =[](auto && .....args)
    {
        print(std::forward<decltype(args)>(args)...);
    }
    fn(2,4.5,"denem",string{"fffff"});
    return 0;
}
```

lambda init capture

- move only bir türü lambda init capture bir tür zourunluluk

```
int main()
{
    using namespace std;
    auto uptr = make_unique<string>("deneme");
    auto f = [uptr = move(uptr)]{cout << *uptr << "\n";};
    if(uptr)
        cout << "up dolu"
    else
        cout << "up bos"
}
```

- Bazı durumlarda ilave olarak aşağıdaki gibi bir kod içerisinde s değerini s değişkeninden alıyor.

```
struct Baz
{
    auto foo() const
    {
        return [s=s]{std::cout << s<< std::endl;};
    }
    std::string s;
};

int main()
{
    const auto f1 = Baz["xyz"].foo();
    const auto f1 = Baz["abc"].foo();
    f1();
    f2();
}
```

- Lambda story kitabında tanıtılan bir temada

```
int main()
{
    using namespace std;
    vector<string>svec(1000);
    //
    string str{"deneme"};
    //...
    auto iter = find_if(svec.begin(),svec.end(), [&str](const string&s){
        return s == str + "den"s;
    }); //3.dereceye geçilen unary predict'ın true değer veren yeri
döndürücek
    //bunun yerine
    auto iter = find_if(svec.begin(),svec.end(),  [&str](const string&s){
        return s == str + "den"s;
    });
}
```

Noexcept olup olmaması

Bazı generic kodlarda fonksiyonun exception throw edip etmemesinde göre kod seçimini yapıyor olabilir.

```
template<typename T>
void func(F&&f)
{
    if constexpr(std::is_nothrow_invokable_v<F, int>)
        cout << "no throw";
    else
        cout << "may throw";
}

int main()
{
    auto fn = [](int x){return x*5;};
    func(fn);
    auto fn1 = [](int x)noexcept{return x*5;};
    func(fn1);
}
```

- lambda fonksiyonunu aşağıdaki gibi çağırabiliyoruz.

```
int main()
{
    auto val = [](int x ){return x*5}(9);
    auto val1 = [](int x ){return x*5}.operator()(6);

    cout << "val = " << val<< "\n";
}
```

- Eğer lambda stateless ise bu operator fonksiyonun bir tür dönüştürme operatör fonksiyonu var ve bu

```
class xyz_13
{
public:
    int operator()(int x)const
    {
        return x*5;
    }
    using ftype = int(*)(int);
    operator ftype()const;
};
```

- Bu da bize pratikte stateless lambdalar fonksiyon adresi türüne implicit olarak dönüşebiliyor.

```
int main()
{
    auto fn = [](int x){return x*5;};
    int (*fp)(int) = fn.operator();
    cout << fp(2)<<"\n";
}
```

Positive Lambda Idiom'u

- İşaret operatörünün operand'ı pointer olabiliyor.

```
int foo(int);
int main()
{
    char c = 'a';
    +c; //bu ifade R-Value oluyor, integral promotion'a neden oluyor ve bu
ifadenin türü int.
    int x = 10;
    int *ptr = &x;
    +ptr;//pointer'da olabiliyor.
    inf (*fp)(int) = foo;
    auto val = +fp;
}
```

- Bu da bize stateless lambda'nın function-pointer türüne dönüşüyor ve

```
int main()
{
    +[](int x){return x*5};
}
```

- Aşağıdaki örnek çalıştırıldığında ilk for-each'te yeni bir pair yatırılıyor. Burada for-each algoritmasında pairlerden bir kopyalama oluyor. Fakat ikinci for-each algoritması çağırılıdığında generialized lambda kullanıldığında koyalama yerine refereans ile aynı adresler ekrana yazdırılıyor

```
int main()
{
    std::map<std::string, int> numbers{{"one",1}, {"two",2}, {"three",3}};

    for(auto mit = numbers.cbegin(); mit != numbers.cend(); ++mit)
        cout << mit->first << " " << mit->second << "\n";
    std::for_each(numbers.cbegin(), numbers.cend(), [](const
std::pair<std::string, int> &p ){cout << p.first << " = " << p.second <<
"\n";});
    //
    std::for_each(numbers.cbegin(), numbers.cend(), [](const auto &p ){cout
```

```
<< p.first << " = " << p.second << "\n";});
}
```

IFI (Immidiately Invoked Function Expression)

```
int main()
{
    [](int x){cout << x*5 << "\n";}(10);
}
```

- Bunun bir faydası initialization const nesneye ilk değer vermemiz zorunlu ve bu ilk değer bir hesaplamayla elde edililyor olabilir.
- Böylece const' i olan değere geri dönüş değeri ile verebildik.

```
const int i = [&] {
    int i = some_default_value;
    if(someConditionIsTrue)
    {
        Do some operations and calculate the value ıf i;
        i = some calculated value;
    }
    return i;
}();
```

- Bunu free function yapabilirdik, fakat burada lokal değerleri argüman olarak göndermemiz gerekiyordu.

```
int main()
{
    int a = 5;
    int b = 4;
    const int x = [&]{return a+b;}();
}
```

- 2.tipik kullanılıdığı yer constructor initialization list içerisinde kullanmak.

```
struct Foo{
    int baz;
    Foo(int bar) : baz([&]{
        //complex initialization of baz.
    }())   {}
};
```

- C++17 ile dile eklenen bir genel fonksiyon çağırıcısı var ve burada fonksiyon çağrı operatör fonksiyonu fark etmke zor olduğunda inovke fonksiyonuna lambda ifadesine callable olarak veriyoruz.

```
int main()
{
    int x = 10;
    int y = 20;
    const int a = std::invoke([&](){
        auto val = x + y;
        //....
        ++val;
        return val;
    });
}
```

- Bir başka kullanımı senaryosu ise static yerel değişkene değer veren bir lambda ifadesi olarak kullanılarak sadece constructor içerisindeki kod bir kere çalıştırılacak. Bu static değişkenler ise thread-safe.

```
class Myclass
{
public:
    Myclass(int x, int y)
    {
        static auto _ {[]{
            std::cout << "bu kod bir kere çalisir\n"; return 0;
        }()};
    }
};

int main()
{
    Myclass m ;
    Myclass m1 ;
    Myclass m2 ;
}
```

- Overload resolution'da kullanılabilir. Transform algoritmasına (range'te ki öğeleri bir callable gönderiyor ve diğer range'in verilen iterator değerinden itaberen kopyalamak.). Burada overload resolution'ı yapılamıyor. Kodun legal olması için fonksyion adresini tür değiştirme operatörü kullanarrk dönüştürmek, c tarzda tür değiştirme operatörü ile yapabiliriz.
- Fakat daha güzel bir çözüm lambda ifadesini kullanmak, böylece generic lambda ile kullanabililyoruz.

```
int f(int);
int f(double);
```

```
int f(long);
int main()
{
    std::vector<int> xvec(100, 5);
    std::vector<int> yvec(100);
    transform(xvec.begin(), xvec.end(), yvec.begin(), f);
    transform(xvec.begin(), xvec.end(), yvec.begin(), static_cat<int(*)
(int)>(f));
    transform(xvec.begin(), xvec.end(), yvec.begin(), (int(*)(int))(f));
    transform(xvec.begin(), xvec.end(), yvec.begin(), [](int a){return
f(a);});
    std::invoke([]{f(10);});
}
```

C++20 ile Gelen Araçlar

- Unevulated context'te lambda kullanımı

```
int main()
{
    auto f = [](int x){return x*5;};
    decltype(f) x; //c++20 illle artık setaaaaks hatası değil.
}
```

- Lambdaların default constructor ve copy-assignment fonksiyonlarının delete ediliymiyor olması

```
int main()
{
    auto f = [](int x){return x*5;};
    decltype(f) x; //c++20 illle artık setaaaaks hatası değil.
}
```

- Greater bir functor object ve biz burada kendi karşılaştırma fonksiyonumuzu closure object olarak ifade etmeye çalıştığımızda

```
int main()
{
    using namespace std;
    //set<int, std::less<int>> myset; //aşağıdaki bildirim ile aynı
    set<int> myset;
    auto f = [](int x, int y){return abs(x) < abs(y);};
    // set<int,decltype(f)>x; C++17 ile sentaks hatası çünkü burada closure
object'in default ctor'u delete edildiği için
    // Onun için:
    set<int,decltype(f)>x(f); //bu şekilde tanımlamamız gerekiyor.
    x.insert({4,-5,7,-3,5,-2,9,1,-1});
    for(int i : x)
```

```
        cout << i << " ";
}
```

- 2.bir problem ise f nesnesini tek bir yerde kullanılıyor ve scope'unu sınırlamak istiyoruz.
- Bir sınıfın veri elemanını bir closure-type türünden yapmak istiyoruz:

```
class Myclass
{
public:
private:
    decltype(f) mf;
};
```

- Aynı zamanda un-evulated context'te lambda ifadesi kullanılabiliyor.

```
int main()
{
    constexpr auto s2 = sizeof([](int x ){return x*5;});
}
```

- Bunun faydası:

```
int main()
{
    decltype([](int x){return x*5;}) f;
    cout << f(213);
}
```

- Bir önceki set örneğini:

```
int main()
{
    using namespace std;
    set<int,decltype([](int x, int y){return abs(x) < abs(y);})>x;
    x.insert({4,-5,7,-3,5,-2,9,1,-1});
    for(int i : x)
        cout << i << " ";
}
```

- unique_ptr'nin 2.template parametresi `default delete` opreatörünü alıyor, biz bir çok durumda C apisini sargılamak için custom-deleter kullanılaiblir.

```
int mian()
{
    using namespace std;
    auto f = [](int *p)
    {
        std::cout << p <<" delete called\n";
    }
    unique_ptr<int, decltype(f)>uptr(f);
}
```

- Bunun yerine aşağıdaki gibi decltpye() içerisinde lambda ifadesini yazabiliriz.

```
int main()
{
    using namespace std;
    unique_ptr<int, decltype([](int *p){std::cout << p <<" delete
called\n";
    delete p;
    })>uptr;
}
```

- Attributelare doğrudan kullanılamıyor.

Familiar Template Syntax

- Generic lambdalarda templateee sentaksının yazılması.
- lambda introduce'urdan sonra artık yazabiliyoruz, Fonksiyon şablonlarında özellikler burada da var.

```
int main()
{
    // [] ()
    auto f = []<typename T>(T x){return x;};

}
```

- Aynı türden çıkarım yapılması gerekiyor. Birden fazla olanak var.

```
int main()
{
    auto f1 = [](int x, int y){return x + y;};
    auto f2 = [](auto x, auto y){return x + y;};
    auto f3 = [](auto x, decltpye(x)y){return x + y;};
    auto f4 = []<typename T>(T x, T y){return x + y;};

    cout<< f1(10,20) << "\n";
    cout<< f2(10,20) << "\n";
```

```
    cout<< f3(10.,20) << "\n";
    cout<< f4(10.,2.5) << "\n";
    //cout<< f4(10.,20) << "\n"; geçersiz
    //cout<< f4(10,20.) << "\n"; geçersiz
}
```

- Specialization için bir sentaks:

```
int main()
{
    auto f = []<typename T>(std::vector<T> &r ){
        return r.size();
    }
    vector<int>ivec(34);
    vector<double>dvec(77);
    cout << f(ivec) << "\n";
    cout << f(dvec) << "\n";
}
```

- Non-type parametreler için de kullanabiliriz.

```
int main()
{
    auto fn = []<class T, int n>(T(&ra)[n])
    {
        for(auto &t: ra)
            t +=10;
    }
    int a[] {1,3,6,7};
    fn(a);
    for(auto val : a)
        cout << val << " ";
}
```

- Variadic template parametre paketini de kullanabiliriz.

```
template <typename ...Args>
void foo(Args&&...args);

int main()
{
    using namespace std;
    auto fpush = []<typename T>(std::vector<T>&x, const T &val)
    {
        x.push_back(val);
    };
    vector <int> ivec;
```

```
    fpush(ivec, 10);
    auto calll_foo = []<typename ...Args>(Args && ...args)
    {
        foo(std::forward<Args>(args)...);
    };
}
```

- Burada explicit olarak tür belirtmemiz için değişik bir sentaks var:

```
template <typename T>
void foo(T);

int main()
{
    auto fn = [](auto x) {return x*x;};
    foo(34); //çıkarımdan istifade edebiliriz veya explicit olarak
çağırabiliriz.
    foo<double>(10); //foo'nun double açılımını çağıracağız.
    fn.operator()<double>(12);
}
```

```
int main()
{
    auto fn = []<int n>(){int a[n]{};return a;};
    fn.operator()<20>();
}
```