

Lambda ifadeleri ve Constexpr:

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6     auto fsquare = [] (auto val) { return val * val; };
7     std::array<int, fsquare(5)> a1; //1
8
9     std::cout << a1.size() << "\n";
10
11    auto f1 = [] (int x) { //2
12        static int cnt = 0;
13        ++cnt;
14        return x * cnt;
15    };
16    //std::array<int, f1(10)> a2; //gecersiz
17    std::cout << f1(20) << "\n";
18
19    auto f2 = [] (int x) constexpr { //3 f2 nesnesinin tanımı gecersiz
20        static int cnt = 0;
21        ++cnt;
22        return x * cnt;
23    };
24
25 }
26
27 //1 fsquare(5) ifadesi C++17 ile birlikte artık varsayılan şekilde constexpr.
28 //Dolayısıyla fonksiyona sabit ifadeleri ile yapılan çağrıdan elde edilen geri dönüş değeri
29 //sabit ifadesi gereken yerlerde kullanılabilir.
30
31 //2 f1 tanımında static yerel değişken kullanıldığı için artık bu lambda constexpr kabul edilmiyor.
32
33 //3 f2 nesnesinin ise tanımı gecersiz. eğer constexpr anahtar sözcüğü kullanılmamasaydı tanım geçerli olacaktı.
```

I static değişken olduğu için, constexpr ifade deger verdiğinde bir static tanı kullanılır. Fakat normal gibi yapmak mümkün.
Bir kereye constexpr yazılık, syntax hatalı olmasına sağlıyor.

```
1 constexpr auto sum = [] (int a, int b) {
2     auto lf = [=] { return a; };
3     auto rf = [=] { return b; };
4     return [=] { return lf() + rf(); };
5 };
6
7 static_assert(sum(1, 2)() == 3);
```

```
1 auto f1 = [] (auto x) constexpr { return x * x; }; //since C++17
2 auto f2 = [] (int x) mutable constexpr noexcept -> int { return x * x; }; //since C++17
3
4 //aşağıdaki tanımlama geçersiz
5 //auto f3 = [] (auto x) constexpr {
6 //    static int cnt = 0;
7 //    ++cnt;
8 //    //...
9 //    return x * x; };
10
11 auto f1 = [] (auto x) { return x * x; }; //implicitly constexpr since C++17
12
13 struct CompilerGeneratedName {
14     template <typename T>
15     constexpr auto operator() (T x) const
16     {
17         return x * x;
18     }
19 };
20
21 
```

↳ lambda expression'ın etrafındaki_QUALIFIERS.

* Noexcept Lambda Expression:

```
int main()
{
    //auto fx = [](int x) {return x + 1; };
    auto fy = [](int x) noexcept {return x + 1; };

    //constexpr auto bx = noexcept(fx(1)); //b is false
    constexpr auto by = noexcept(fy(1)); //b is true
}
```

* Default olarak noexcept değil!

* Lambda, Init Capture:

```
class xyz_19873 {
public:
    xyz_19873(int x) : mx(x) {}
    int operator()(int a)|  

    {  

        }  

private:
    int mx;
};

int main()
{
    int x = 10;  

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

```
int main()
{
    using namespace std;

    auto uptr = make_unique<string>("volkan gundogdu");

    [uptr]() {return *uptr + "can"; }
}
```

→ unique_ptr'nin copy constructor, delete edildi
true syntex hatalı

→ returns. captured değişik, syntax hatalı. return

```
int main()
{
    using namespace std;

    auto uptr = make_unique<string>("volkan gundogdu");

    cout << (uptr ? "dolu" : "bos") << "\n"; → dolu  

    auto f = [uptr = move(uptr)]() {return *uptr + "can"; };

    cout << (uptr ? "dolu" : "bos") << "\n"; → bos,
}
```

→ bu uptr'ler aynı değil!

→ return kısmındaki uptr → class member
[] içindeki uptr → captured.

```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>

int main()
{
    std::vector<std::string> svec{ "selahattin", "muratcan", "polat", "kahraman", "nurullah" };
    const std::string suffix{ "can" };

    auto iter = find_if(svec.begin(), svec.end(), [&suffix](const std::string& s) {
        return s == "murat" + suffix;
    });
}
```

lambda mit capture

```
int main()
{
    std::vector<std::string> svec{ "selahattin", "muratcan", "polat", "kahraman", "nurullah" };
    const std::string suffix{ "can" };

    auto iter = find_if(svec.begin(), svec.end(), [str = "murat" + suffix](const std::string& s) {
        return s == str;
    });
}
```

+ Neca mülakat sorusu:

```
#include <iostream>

int g = 99; capture all by copy

auto fx = [=] {return g + 1; };

int main()
{
    g = 500; copy değerinden bir sınıf oluştur
    std::cout << fx() << "\n";
}
```

→ Cevap 99

```
#include <iostream>

int g = 99;  

auto fx = [=] {return g + 1; };

int main()
{
    int g = 500;  
    std::cout << fx() << "\n";
}
```

Fırtınası
cevap 100 çünkü globaldeki g kullanılmış

```
1 #include <iostream>
2
3 int g = 99; burada sınırlı var elemenler olur
4
5 auto fy = [g = g] {return g + 1; }; g = 99 olucak
6
7 int main()
8 {
9     g = 500; buradaki 500 atemsi
10
11    std::cout << fy() << "\n";
12 }
```

→ Cevap 100

buradaki 500 atemsi
sınırlı var elemenini değiştirmeye!

* Lambda Expression and Perfect Forwarding:

```
using namespace std;
```

```
int main()
{
    [](auto &&x)
    {
        std::forward<decltype(x)>(x)
    }
}
```

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

int main()
{
    auto f = [](auto &&... param) {
        print(std::forward<decltype(param)>(param)...);
    };
    f(12, 3.4, 4.5f, "necati ergin");
}
```

* C++ 20 ile Lambda Expression:

```
int main()
{
    using namespace std;

    auto f1 = [](int x) {return x + 5; };

    decltype(f1) f2;
}
```

```
int main()
{
    using namespace std;

    auto fcomp = [](int a, int b) {
        return abs(a) < abs(b);
    };

    set<int, decltype(fcomp)> myset;

    //myset.insert(123);
}
```

→ Syntax Notes: Coklu set'in template code'u decltype(fcomp) terinden bir nesne olusturur. Bu nesne de default constructor eder. Bu nesne default constructor olmadiği için gizli notları.

→ Aşağıdaki kod C++ 20 de legal. Çünkü default ve copy ctor var.

```

int main()
{
    using namespace std;

    int x = 5;
    auto f1 = [x](int a) {return a + x; };

    decltype(f1) f2;
}

```

→ Cpp 20'de lambda, ref'lerin yerine neler!

→ C++'da class yapısına, stateless lambda ren gerek bilmedi!

```

1 #include <iostream>
2 #include <set>
3 #include <type_traits>
4
5 int main()
6 {
7     using namespace std;
8
9     auto f1 = [] (int a) {return a + 5; };
10    auto f2 = [] (int a) {return a + 5; };

12    is_same_v<decltype(f1), decltype(f2)>
13
14    false!!!
15 }

```

→ İstediğiniz lambda ifadesi. Oluşanlar arasında, olsun bir degeri, bir de lambda ifadesinin true/false!

```

auto f1 = [] (int a) {return a + 5; };
decltype(f1) f2;
is_same_v<decltype(f1), decltype(f2)>

```

↓ Sıradır oynu tor!

* Cpp 20 ile lambda ifadeleri, unevaluated context'ta (decltype gibi) kullanılabilir hale gelmiş!

```

int main()
{
    using namespace std;

    using lenset = set < string, decltype([](const auto& l, const auto& r) { return l.size() < r.size(); }) >;
    lenset namelens = { "Ayse", "Zehra", "Murat", "Bilal", "Sumeyye", "Nur", "Cemil" };

    for (const auto& s : namelens)
        std::cout << s << "\n";

    /* set<int> iset = { 9, -3, 12, -5, 40, -8, 77, -75 };
    print(iset);

    using abs_set = set<int, decltype([](const auto& l, const auto& r) { return abs(l) < abs(r); }) >;
    abs_set abs_vals = { 9, -3, 12, -5, 40, -8, 77, -75 };
    print(abs_vals); */
}

```

* Templatized Lambda: * Cpp 20 ile eklenmiştir!

↳ auto'nun farklı ne?

```

5 //templatized lambda
6
7 class xyz {
8 public:
9     template<typename T, typename U>
10    auto operator()(T x, U y)
11    {};
12
13 int main()
14 {
15     auto f = [](auto x, auto y) {
16         return x + y;
17     };
18
19
20
21
22 }

```

→ Fonksiyona gönderilen argümanlar, aynı olmaz zamanında değil

```

auto f = []<typename T>(T x, T y) {
    return x + y;
};

```

f(12, 4.5);

↳ auto ne yazdırır, bu sınırları kırıktır ama yok!

→ *git hub
orange bold*

```
#include <iostream>
#include <vector>

#include <list>

int main()
{
    auto f1 = [] (int x, int y) {return x + y; };

    auto f2 = [] (auto x, auto y) {return x + y; };

    auto f3 = [] (auto x, decltype(x)y) {return x + y; };

    auto f4 = []<typename T>(T x, T y) { return x + y; };

    auto f5 = []<typename T>(std::vector<T> x, std::list<T> y) { return x + y; };
}
```

* Üye Fonksiyonlar İçinde Lambda:

```
struct Nec {
    void foo()
    {
        auto f = [] (int a) {
            return a * (mx + my);
        };
    }

    int mx{}, my{};
};
```

→ Sınıfın data memberlerini bu şekilde lambda ifadesinde kullanmak sentence hatası!

→ *this capture edilmeli!* Bu nün bir hata yoktur!

```
struct Nec {
    void foo()
    {
        auto f = [this] (int a) {
            return a * (mx + my);
        };

        std::cout << f(3) << "\n";
    }

    int mx{10}, my{20};
};

int main()
{
    using namespace std;
    Nec mynec;
    mynec.foo();
}
```

Capture this

```
struct Nec {
    void foo()
    {
        auto f = [&] () {
            mx++;
            my++;
        };

        f();
    }

    int mx{10}, my{20};
};
```

Capture all by reference

```

struct Nec {
    void foo()
    {
        auto f1 = [this] {return ++mx; };
        auto f2 = [&] {return ++mx; };
        auto f3 = [=] {return ++mx; }; //deprecated
        auto f4 = [=, this] {return ++mx; }; //C++20

        f1();
        f2();
        f3();
        f4();
    }

    int mx{10};
};

```

deprecation dersinin neden hizli capture edigini
 this olsun! Niye? Cok acik olmasagini sen deprete etti!
 Dogru kullanim Cpp 20 icin bu!

* Capturing *this: Cpp 17 de geldi! ANLATIMADIM //

* this'i capture ettigimizde ~~closure~~ içinde closure type'ı "this'in logicosu" hesaptiliyor. This'in kendisi capture edildiği senaryoda, nesne eger silinirse dangling reference olurdu!

* mutable olmali!

```

void foo()
{
    auto f1 = [*this]()mutable
    {
        mx += 100;
    }
}

int mx{10};
};

int main()
{
    using namespace std;
    Nec mynec;
    mynec.foo(); Cevap: 10
}

```

* Space ship / Three way comparison Operator <=>:

- include <compare> → Cpp 20
- Konsol tercihimizde return overload edilebilir.
- Special member functionlar gibi, default edilebilir!

```

class Myclass {
public:
    auto operator<=>(const Myclass&) const = default;
};

```