*** std::function:** - Bir callable'ı sarmalar

- Fakat bir bellek overhead'i var. Bazen de dinamik memory alloc yapıyor.

```cpp
using namespace std;

auto fn = [](double d) {return d * d + .3; };

cout << "sizeof(decltype(fn)) = " << sizeof(decltype(fn)) << '\n';

std::function f = fn;
cout << "sizeof(f) = " << sizeof(f) << '\n';

cout << fn(5.8763) << "     " << f(5.8763) << '\n';
}
```

(sizeof(decltype(fn)) → 1 byte; std::function f = fn; → CTAD kullanımı)

- Lambdanın size 1 byte iken, recenin compiler'ı function için 40 byte kullandı!

*** Generic / Generalized Lambda Expression:** Cpp 14

```cpp
using namespace std;

auto get_size = [](const auto& c) {
    return std::size(c);
};

vector<int> ivec(30);
list mylist{ 34., 56. };

std::cout << get_size(ivec) << '\n';
std::cout << get_size(mylist) << '\n';
int ar[20]{};
std::cout << get_size(ar) << '\n';
}
```

- Yazılan fonksiyon template olduğu için, farklı farklı türler için kullanılabilir.

- Bir diğer avantajı ise, daha az verbose bir kod yazmamızı sağlıyor.

```cpp
using namespace std;

vector<pair<string, string>> pvec;

for (int i = 0; i < 1000; ++i) {
    pvec.emplace_back(pair{ rname(), rtown() });
}

sort(pvec.begin(), pvec.end(), [](const pair<string, string>& x, const pair<string, string>& y) {
    return pair{ x.second, x.first } < pair{ y.second, y.first }; });

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

(pair<string, string> → auto, pair<string, string> → auto; gereken yere const auto'r yazıp generalized lambda yaptık)

- Verbose'luk azalınca hata yapma riski azalıyor!

- Generic lambdalar universal reference parametreye sahip olabilir. → Perfect Forwarding

```cpp
int main()
{
    auto fn = [](auto &&f)
}
```

→ Cpp 20 de eklenen syntax

```cpp
int main()
{
    auto fn = []<typename T>(T &&f)
}
```

```cpp
void foo(std::string&)
{
    std::cout << "L VALUE REF\n";
}

void foo(const std::string&)
{
    std::cout << "const L VALUE REF\n";
}

void foo(std::string&&)
{
    std::cout << "R VALUE REF\n";
}


int main()
{
    auto fn = [](auto&& s) {
        foo(std::forward<decltype(s)>(s));
    };

    string name{ "ali" };

    fn(std::move(name));
}
```

• Generic overload çağrıldı!

```cpp
using namespace std;

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

int main()
{
    auto fn = [](auto &&...args) {
        print(std::forward<decltype(args)>(args)...);
    };

    fn(23, 7.6, "osman", Date{ 6, 8, 2023 });
}
```

*** C++ Lambda Story → Bartłomiej Filipek ***

→ Cpp 23 hariç, lambdaların geçmişini anlatan kitap!

Lambda Expression as Function Return Statement:

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

    std::string s;
};

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

auto → Cpp 14

[s = s] → Init capture kullanmadan s'yi capture etmedi!

Noexcept Lambdas:

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


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

    func(fn);

}
```

- noexcept olup olmamasına göre farklı kod derleniyor olacak!

Stateless lambdaların Implicit Olarak Fonksiyon Adresine Dönüşmesi:

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

    int (*fp)(int) = fn;

    cout << fp(34) << "\n";
}
```

```cpp
void foo(int (*fp)(int))
{
    ///

    int x = fp(20);;

    //code
}


int main()
{
    foo([](int x) {return x * 7; });
}
```

- Pratikte bize, C API'lerini kullanma imkanı sunuyor!
- Callback function olarak kullandık!

Positive Lambda Idiom:

→ Nasıl Legal:

```cpp
char c = 'a';

+c
```

- C' dilinde de,
  Bu ifade de "+" olmasaydı: L value
  " "+" olunca: R value

→ Stateless integral promotion da var!
+C ifadesinin türü: int

```cpp
int main()
{
    int (*fp)(int) = foo;

    auto val = +fp;
}
```

Aynı şekilde, pointer türlerini de operand olarak kullanabiliriz!

```
int foo(int);

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

+'nın operandı olmalı, lambda kullanabilmek için, lambdayı fonksiyon adresine dönüştüren tür dönüştürme operatör fonksiyonu çağırılıyor.

```
int foo(int);

class Myclass {
public:
    operator int()const;
};

int main()
{
    Myclass m;

    //+m
    +m.operator int()

}
```

• Operatör int fonksiyonu olmasaydı, tm ifadesi syntax hatası olayı alırdı!

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

• + ifadesi olmadan, f'nin türü: closure type
• + ifadesi varsa, f'nin türü: function pointer

+ open std.org → Cpp 17-20 farklarını gösteriyor!

Immediately Invoked Function Expression (IIFE):

```
int main()
{
    [](int x) {return x * 5; }(12)
}
```

↳ herhangi bir değişken ismi olmadan, bulunduğu yerde çağırdık!

↳ Faydalı sağladığı noktalar: - initialization (özellikle const nesneler için)

```
const int x = a > b ? exp1 : expr2;
```

const ifade en fazla bir ternary ifade ile init edilebilir. Başka bir expr bağlanamaz!

```
const int i = [&]{

    int i = some_default_value;

    if(someConditionIstrue)
    {
        Do some operations and calculate the value of i;
        i = some calculated value;
    }

    return i;

} ();
```

↳ Const ifadelere bu şekilde hesaplama sonrası değer atayabiliriz!

```cpp
int a = 34;
int b = 345;

const int x = [&]() {
    int val = a * b;
    ++val;
    ////
    |
    return val;
}();
```

(const int x) → burada tanıt

x = ~~ → burada değer ver

→ Önce tanıt, sonra değer ver yapmaktan daha iyi! → Yandan kodlarda dikkat et!

- IIFE idiomi: yalnızca bir kez çağrılması istenen fonksiyonlarda da kullanılabilir!

```cpp
class Nec {
public:
    Nec()
    {
        static auto _{ [] {
            std::cout << "bu kod yalnizca bir kez calismali"; return 0;
        }() };
    }
};

int main()
{
    Nec n1;
    Nec n2;
    Nec n3;
    Nec n4;
}
```

- static auto _ (ad) değişkenine ilk değer ataması lambda ile yapıldı!

NOT/: Statik yerel değişkenler, fonksiyon ilk kez çağrıldığında init edilir. Eğer çağrılmazsa, hayata gelmez! Thread safe'tir.

* Andrei Alexandrescu → Modern C++ Design

← Bu kitabı önerdi hoca.

Yine Design Pattern için: C++ Software Design → fena değilmiş,

* Dimitri Nesteruk? kitabı bayağı kötüymüş!

* Generalized Lambda vs Overload Resolution in Algorithms:

```cpp
void func(int) { putchar('i'); }
void func(double) { putchar('d'); }
void func(long) { putchar('l'); }

using namespace std;

int main()
{
    vector<int> vec(20);

    //...
    //for_each(begin(vec), end(vec), func); //gecersiz
    for_each(begin(vec), end(vec), (void (*)(int))func);
    putchar('\n');
    for_each(begin(vec), end(vec), [](auto x) {return func(x); });
}
```

(func) → bu predicate, içine gönderilen range'in türüne göre overload resolution'ı çalıştıramaz X → Ambiguity!

*C++ 20 sonrası Lambda ifadeleri ve iddiaları:

• Cpp 20 öncesi lambdaların copy assignment'i ve default ctor deleted!

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

    decltype(f) x;
}
```

→ Cpp 20'den itibaren legal!

→ Fakat Stateless'ler zaten bu evrene geldi!

```
using namespace std;

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

set<int, decltype(f)> x(f);
}
```

Custom Compare

Kendi custom comparatorlarımızı (lambda ile oluşturulan), bu şekilde belirtmek zorundayız! (Cpp 20 öncesi) → yoksa syntax hatası!

```
using namespace std;

decltype([](int x) {return x * 5; }) f;

cout << f(213);
```

→ Cpp 20 ile bu şekilde stateless lambda kullanılabilir!

```
set<int, decltype([](int a, int b) {
    return a > b;
    })> myset{ 4, 7, 2, 4, 2, 1, 8, 34, 3 };
```

• Attributelar C++ 20'de de kullanılamıyor!

```
int main()
{
    auto myLambda = [](int a) [[nodiscard]] { return a * a; }; // gecersiz
}
```

* Familiar Template Syntax: - Cpp 20

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

Cpp 20 ile →

```
int x{ 0 };

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

Cpp 20'ye kadar, lambda introducer'dan sonra, argüman listesi bir şey yazılamıyor!

```
int main()
{
    auto fn = []<typename T>(const std::vector<T> &vec) {
        return vec.size();
    };

    fn(12);
}
```

→ Artık vektör dışında çağrı yapmak sentaks hatası!

- Non-type parametreler için de bu syntax'ı kullanabiliriz.

```
auto fn = [] <class T, int n>(T(&ra)[n]) {
    for (auto& t : ra)
        t += 10;
};

int a[]{ 1, 3, 6, 8 };

fn(a);

for (auto val : a)
    std::cout << val << " ";

}
```

* Capturing this: . cpp 20 ile, this'in copy [=] capture'ı deprecate edildi! ⟶ Bir önceki dersle daha ayrıntılı işledik!

```
class Myclass {
public:
    void foo()
    {
        int a = 10;

        auto f = [=, this] {return a * (mx + my); };  //invalid in C++17 valid in C++20
    }
private:
    int mx, my;
};
```

capture *this by reference!

capture all by copy