

Tekrar:

- Structured Binding :

```

struct Data {
    int a, b, c;
};

int main()
{
    Data mydata = { 1, 4, 5 };
    auto [x, y, z] = mydata;
}

```

C'den de gidiyor. public, aggregate bir struct.

herhangi bir ek kod olmadan structured binding kullanılır.

```

#include <iostream>

struct Data {
    int a, b, c;
};

Data get_data()
{
    return { 4, 8, 7 };
}

int main()
{
    auto [x, y, z] = get_data();

    std::cout << "x = " << x << "\n";
    std::cout << "y = " << y << "\n";
    std::cout << "z = " << z << "\n";
}

```

Birinci şekilde

→ Structured binding, arrayler için de geçerlidir.

```

#include <iostream>

int main()
{
    int a[] = { 2, 7, 9, 13 };

    auto [x1, x2, x3, _] = a;
}

```

3 elemen
biri boş gelir.

Ancak aynı scope'da farklı da underscore olursa syntax hatalı !!

```

#include <iostream>

struct Nec {
    int x, y, z;
};

Nec foo()
{
    return { 1, 2, 3 };
}

int main()
{
    auto [x, y, z] = foo();
    auto f = [x](int a) {return a * x; };
}

```

* Cpp 20'den sonrası, structured binding ile init edilen elementler, lambda ifadelerinin capture olabileceği gibi kullanılabilir.

+ Cpp 17'de syntax hatalı

```

class MyClass {
public:
    MyClass() = default;
    int a{}, b{};
    friend void foo();
private:
    int c{};
};

void foo()
{
    MyClass m;
    ...
    auto [x, y, z] = m;
}

```

friend bildirimi olduğu için, z' init edilebilir. Bu da Cpp 20'de geldi. Yalnız private'ye erişemeyecektir.

→ Reference Wrapper tütün Containerlar:

```
1 // #include <iostream>
2 // #include <functional>
3 // #include <vector>
4
5
6 using iref = std::reference_wrapper<int>;
7
8 int main()
9 {
10     using namespace std;
11     int x{}, y{ 10 }, z{ 20 }, t{ 30 };
12
13     vector<iref> vec{ x, y, z, t }; → reference wrapper
14
15     for (auto& r : vec) { → tütün INT vector.
16         ++r; → Avantaj: direkt range-based
17     } → fer loop iie turndegileri degistirili.
18
19     std::cout << "x = " << x << "\n";
20     std::cout << "y = " << y << "\n";
21     std::cout << "z = " << z << "\n";
22     std::cout << "t = " << t << "\n"; → Turndegiler
23                                     | oratti.
24
25 }
26
```

→ Reference Wrapperlar'a Fonksiyonlar da bağlanabilir.

```
3 int foo(int x)
4 {
5     return x * x + 5;
6 }
7
8 int main()
9 {
10     using namespace std;
11
12     reference_wrapper rf = foo;
13
14     rf(10)
15
16 }
17
18
19
20
21
22 }
```

* Function Adapters:

```
std::bind
mem_fn
not_fn

std::function //general function wrapper

std::invoke
```

→ Fonksiyon adaptörleri nedir?

- Bindler bir callable, dip, bize callable etkinlikler bir wrapper.
- Ünitenin 3 argümanlı bir callable, 2 elemenli argümanlı, hole getirme kizi, std::bind
- invoke.

* Std::Bind:

```
#include <functional>

void func(int x, int y, int z)
{
    std::cout << "x = " << x << " y = " << y << " z = " << z << '\n';
}

int main()
{
    using namespace std::placeholders;
    auto f = std::bind(func, _3, _1, _2);
    f(10, 20, 30);
}
```

Argümanların yerini tutuyor : -1, -2, -3 gibi...

$x = 30$
 $y = 10$
 $z = 20$ ye bind edildi

1.
2.
3.

```
#include <functional>

void func(int x, int y, int z)
{
    std::cout << "x = " << x << " y = " << y << " z = " << z << '\n';
}

int main()
{
    using namespace std::placeholders;
    auto f = std::bind(func, 50, 20, 30);
    f();
}
```

Aynı şekilde bind ile sabır değerler atıp, f'e argüman göndermeden de çalıştırırsınız.

* A Functor Example:

```
class Functor {
public:
    int operator()(int x, int y) const
    {
        return x * x + y * y;
    }
};

int main()
{
    using namespace std;

    Functor f;

    cout << f(5, 7) << "\n";
}
```

→ Functor class da curly braces olduğu için, std::Bind'a argüman gerekmez.

* Lambda Expression Example:

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

    auto f = [](int a, int b) {
        return a * b + 23;
    };

    std::cout << f(3, 5) << "\n"; I

    auto fn = bind(f, _1, _1);

    std::cout << fn(20) << "\n"
}
```

→ Reference Wrapper Kullanım Açıklaması:

```
void func(int& x, int& y) I
{
    x *= 5;
    y *= 10;
}

int main()
{
    using namespace std;

    int a = 5;
    int b = 7;

    auto fn = std::bind(func, ref(a), ref(b)); I

    fn();

    std::cout << "a = " << a << "\n";
    std::cout << "b = " << b << "\n";
}
```

bind içe, arguman olarak reference olan bir callableı
bağlamak istiyorsun.

↳ Eğer örnekleri gibi ref(a), ref(b)
demeseyseniz, değerleri kopyalanırdı. XX
⇒ x=5
y=7 kalmadı.

↳ ref(a), ref(b) diyecek, reference wrapper
arguman olarak geçersin, a=25
b=35 elide ederis.

```
void myprint(std::ostream& os, int x, int y)
{
    os << "(" << x << ", " << y << ")\n";
}

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

    auto fn = bind(myprint, ref(cout), _1, _2); I

    fn(45, 90);
}
```

→ ostream, kopyalanma kapalıdır. !!!

→ Bu yüzden ref(cart) şeklinde gecmeli!
ref olmadan kullanması → syntax hatalı X

→ Bind kullanım örneği:

```
#include <string>
#include <vector>
#include "nutility.h"
#include <algorithm>
#include <iterator>
#include <functional>

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

    vector<string> svec;
    rfill(svec, 50, rname);
    print(svec);
    vector<string> destvec;

    //copy_if(svec.begin(), svec.end(), back_inserter(destvec), bind(greater{}, _1 "necmettin"));
    copy_if(svec.begin(), svec.end(), back_inserter(destvec), [](const string& s) {return s > "neco"; });

    print(destvec);
}
```

vektördeki string'in placeholder 1. elemen
neco/necmettin'den boyut stringler destvec'e kopyılır

* Mem_fn Adapter: - Bir sınıfın tipe fonksiyonu olur, olsa bir sınıf nesnesi göndermeye, bu nesne adresini gönderdiğimiz fonksiyon çağırılır.

```
#include <functional>

class Nec{
public:
    Nec(int x = 0) : mx{x} {}
    void print()const
    {
        std::cout << "(" << mx << ")\n";
    }

    void set(int val)
    {
        mx = val;
    }
private:
    int mx;
};

int main()
{
    Nec mynec{ 5 };

    mynec.print();
    mynec.set(36);
    mynec.print();
}
```

```
void print()const
{
    std::cout << "(" << mx << ")\n";
}

void set(int val)
{
    mx = val;
}
private:
    int mx;
};

int main()
{
    using namespace std;

    Nec mynec{ 5 };

    auto fn = mem_fn(&Nec::print);

    fn(mynec);           I

    auto fnx = mem_fn(&Nec::set);

    fnx(mynec, 20);
```

- Uzune lambda ifadeler ile yapılıbırradı.

* Mem_fn örneği:

```

Nec(int x = 0) : mx{x} {}

// void print()const
{
    std::cout << "(" << mx << ")\n";
}

void set(int val)
{
    mx = val;
}
private:
    int mx;
};

int main()
{
    using namespace std;

    vector<Nec> nvec;
    Irand myrand{ 0, 1000 };

    for (int i = 0; i < 100; ++i) {
        nvec.emplace_back(myrand());
    }

    for_each(nvec.begin(), nvec.end(), mem_fn(&Nec::print));
}

```

Nec::print'i
global fonksiyon
bir for-each
attı.

* Member Functionları Gözden Geçirdik:

```

class Myclass {
public:
    static int foo(int x)
    {
        std::cout << "int Myclass(int x) x = " << x << "\n";
        return x * x;
    }
};

int main()
{
    auto fptr = &Myclass::foo;
    int (*fp)(int) = &Myclass::foo;

    std::cout << fptr(10) << "\n";
    std::cout << fp(20) << "\n";
}

```

Aynı parametrik yapıda bir fonksiyon
pointeri

- Bir sınıfın static işlev fonksiyonun adresinin türüyle
benzer şekilde bir fonksiyonun adresi olsa
burası da farklıdır.

- Ayrıca bu okurum non static member functionlarda
farklı.. pointerler kullanır değil.

```

class Myclass {
public:
    Myclass(int val = 0) : mx{val} {}

    int foo(int x)
    {
        return x * mx;
    }

private:
    int mx;
};

int main()
{
    auto fp1 = &Myclass::foo;
    int (*Myclass::*fp2)(int) = &Myclass::foo;

    auto deduced_type
}

```

Function
pointer
boyne
tutulur.

```

class Myclass {
public:
    int f1(int);
    int f2(int);
    int f3(int);
    int f4(int);
    /**
};

using mf_type = int(Myclass::*)(int);

int main()
{
    //int (Myclass:: * fa[])(int) = {
        mf_type fa[] = {
            &Myclass::f1,
            &Myclass::f2,
            &Myclass::f3,
            &Myclass::f4,
        };
    }

    /*auto fp1 = &Myclass::f1;
    int(Myclass:: * fp2)(int) = &Myclass::f1;
    mf_type fp3 = &Myclass::f1;*/

    //int(Myclass:: * *fpp)(int) = &fp1;
    //mf_type *fpp = &fp1;
}

```

* Non static member function pointerla o fonksiyonun çağrımı operator:

• \cdot^* = elmnzde \cdot^* this olurken kullandıralı nesne versyj kullanımı = (object, \cdot^* fp)(args)
 \rightarrow^* = elmnzde \rightarrow^* this olurken kullandıralı nesne adresi versyj kullanımı = (object->pointer \rightarrow^* fp)(args)

→ Operator precedensi var. Bkz. operatörlerin precedensi. () operatöründen sonra devam!

```

int main()
{
    auto fp = &Myclass::foo;

    Myclass m{ 47 };
    std::cout << "&m = " << &m << "\n";
    gerekli vermek için parantez ekle

    auto val = (m.*fp)(32);
    I fp'nin gösterdiği fonksiyonun nesne "this"
    std::cout << "val = " << val << "\n";
}

```

* fp = non static member function pointer

```

auto fp = &Myclass::foo;

Myclass m{ 10 };
std::cout << "&m = " << &m << "\n";
Myclass* pm = &m;

auto val = (pm->*fp)(5);
I
std::cout << "val = " << val << "\n";

```

→ Bu kısım bitti #04'tür 2.33.57'ye geldim. Daha sonra devam eder.

+ std::invoke:

- Az önceki bölümdeki kodu kullanın.
- Arguman olarak, bir tane callableı ve bu callableın dözen. argumentları alır ✓

```
#include <iostream>
#include <functional>

int sum(int x, int y, int z)
{
    return x + y + z;
}

int main()
{
    using namespace std;

    int a{ 3 }, b{ 5 }, c{ 987 };

    //auto val = sum(a, b, c);
    auto val = invoke(sum, a, b, c);

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

```
main.txt (Global Scope)

#include <functional>
#include <string>

class Pred {
public:
    Pred(char c) : mc{c} {}

    bool operator()(const std::string& s) const
    {
        return s.find(mc) != std::string::npos;
    }

private:
    char mc;
};

int main()
{
    std::string name{ "mustafa" };

    Pred f{ 'a' };

    std::invoke(f, name);
}
```

→ Member function pointerlerden çok daha okunur ve güzel