***Structured Binding Devam:**

- Bileşenlere sahip yapıları, bileşenlerine ayırmak için kullanılır.

- Range based for loop gibi, structured binding de bir syntactic sugar!

```cpp
using namespace std;

std::ofstream ofs( "out.txt" );
if (!ofs) {
    std::cerr << "out.txt dosyasi olusturulamadi\n";
    exit(EXIT_FAILURE);
}

//ofs.setf(ios::left, ios::adjustfield);
ofs << left;

vector<Person> pvec;

pvec.reserve(10'000u);

for (int i = 0; i < 10'000; ++i) {
    pvec.emplace_back(Irand( 0, 10000 )(), rname() + ' ' + rfname(), rtown());
}

//cout << "pvec.size() = " << pvec.size() << '\n';

sort(pvec.begin(), pvec.end());

for (const auto& [id, name, town] : pvec) {
    ofs << setw(12) << id << '\t' << setw(32) << name << '\t' << town << '\n';
}
}
```

Unutma: size = 10'000 değil şu an size = 0

id: Bunlar yazmadan önce get<0>

name: get<1>

town: get<2>

diyerek alıyordu!

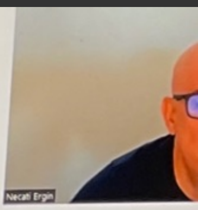
get, referans semantiği ile erişir!

- referans semantiği ile tuple'ın bir öğesine erişip değiştirebiliriz!

```cpp
int main()
{
    std::tuple mytuple(456, std::string("necati"));
    auto& [id, name] = mytuple;

    name = "ergin";

    std::cout << id << "," << name;
}
```

*** std::tuple için structured binding yapıldığında, compiler şu adımları takip ederek kod üretir:**

```cpp
int main()
{
    std::tuple mytuple(456, std::string("necati"));
    auto& [id, name] = mytuple;

    name = "ergin";

    std::cout << id << "," << name;
}

std::tuple mytuple(456, std::string("necati"));


Derleyici, önce bizim doğrudan görmediğimiz böyle bir referans değişken tanımlıyor,

auto& a_hidden_variable = mytuple;

std::tuple_size meta-fonksiyonu ile bu tuple türünün öğe sayısını derleme zamanında kontrol ediyor.;

std::tuple_size_v<std::remove_reference_t<decltype(a_hidden_variable)>>

Eğer bu sayı 2 değil ise sentaks hatası verecek, bir sonraki aşamaya geçilmeyecek.

Burada şimdi derleyicinin get<> fonksiyonlarına çağrı yaparak a_hidden_variable öğelerine erişmesi gerekecek.
Bunun için std::get fonksiyonuna ya da sınıfın üye fonksiyonu get'e çağrı yapması gerekecek.

Ama bu elemanlar birden fazla kez kullanılıyor ise aynı işlemler tekrar yapılmasın diye
yine gizli değişkenler oluşturacak.
```



```cpp
Burada oluşturacağı değişkenlerin türü
tuple_element_t

std::tuple mytuple(456, std::string("necati"));
auto& a_hidden_variable = mytuple;

std::tuple_element_t<0, std::remove_reference_t<decltype(a_hidden_variable)>>& hidden_id = std::get<0>(mytuple);
//                   ^ The type of the member with index 0                                ^ anonymous          ^ Gets the value of member 0
std::tuple_element_t<1, std::remove_reference_t<decltype(a_hidden_variable)>>& hidden_name = std::get<1>(mytuple);



Böylece kod şu hale gelecek:

std::tuple mytuple(456, std::string("necati"));
auto& a_hidden_variable= mytuple;
int& hidden_id = std::get<0>(mytuple);
std::string& hidden_name = std::get<1>(mytuple);
    int ve string tuple elemanları adı verildi.
Buradaki değişkenler L value reference: Böylece gereksiz kopyalamadan kaçınılıyor.

get fonksiyonu L value reference döndürmeseydi ne olurdu?
Derleyici bu durumda R value reference oluşturup geçici nesnelerin hayatlarını uzatırdı.
Bundan sonra derleyci referans değişkenlere "special identifiers" olan id ve name isimlerini veriyor.
Bu "special identifiers" hidden_id ve hidden_name isimlerine bağlanıyorlar ama gerçekte değişken değiller.
```

* Custom Typelar için Structured Binding:

```cpp
class Person {

public:
    Person(int id, std::string name, double wage) :
        m_id{id}, m_name{std::move(name)}, m_wage(wage) {}

private:
    int m_id;
    std::string m_name;
    double m_wage;
};


int main()
{
    Person per{ 348'975, "ali akyar", 245.87 };

    //auto [id, name, wage] = per;
}
```

→ bu şekilde kullanmak için bizim
- bir tuple
- bir tuple size
- her eleman için get'e

ihtiyacımız var

Adım 1: std namespace altında tuple size ve tuple element için explicit specialization:

```cpp
namespace std {
    template <>
    struct tuple_size<Person> : std::integral_constant<size_t, 3u> {

    };

    template <> struct tuple_element<0, Person> {   using type = int;};
    template <> struct tuple_element<1, Person> {   using type = string;};
    template <> struct tuple_element<2, Person> {   using type = double;};

}
```

explicit specialization!

bunun yerine constexpr static size_t value = 3 de yazılabilirdi.

Adım 2: get implementasyonu:

```cpp
template <std::size_t N>
auto get(const Person& p)
{
    if constexpr (N == 0)
        return p.get_id();
    else if constexpr (N == 1)
        return p.get_name();
    else
        return p.get_wage();
}
```

* Hatırlatma:
- return değeri auto olursa, reference return edemez!
→ böyle bir durumda, decltype(auto)

* Space ship Operator (Threeway Operator) : Cpp 20

```
equality operators

==
!=

< <= >= <=   relational
```

→ hepsi bool return eder!

→ Custom typelar için

```
a != b  →  !(a == b)

a > b   →  b < a
a >= b  →  !(a < b)
a <= b  →  !(b < a)
```

↳ Custom typelar için sadece "<" ve "==" implement edersek, aynı logic mantığı elde etsek de semantik açıdan doğru olmayan noktalar var!

Örneğin:

```cpp
a.operator<(5)

5 < a
```

- A sınıfının type conversion < overloadu olsun

→ a < 5 yazılır ✓

5 < a yazılamaz ✗

- Boilerplate kodları derleyiciye yaptırabiliriz.
(tüm overloadları)

```cpp
public:
    [[nodiscard]] bool operator<(const Myclass&)const;
```

Nodiscard attribute'u ile, eğer return değer kullanılmamışsa, compiler hata uyarıyor!

→ Spaceship operator kullanırken, compiler bizim yerimize boilerplate kodları, noexcept ve nodiscard ekler!

- Spaceship operator, bize boolean değil, karşılaştırmanın kesin sonucunu döndürür! Fakat, strcmp gibi int return etmez X

```cpp
ret = strcmp(s1, s2);

ret > 0   → s1
ret < 0   → s2
0            s1 = s2
```

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

(eğer overload edilmezse, compiler <=> alarak yazacak!)

+ Cpp 20 ile birlikte:

→ Primary operatörler reverse edilebilir!

→ Secondary operatörler, rewritable, yani primary ifadeler kullanılarak yazılabilir!

```cpp
bool b1 = (m == 5); // ✓
bool b2 = (m != 5); // → Cpp 20 ile bu !(m==5)
bool b3 = (5 == m); // → Cpp 20'den önce syntax hatası!
bool b4 = (5 != m);
}
```

↓ Cpp 20 ile:

```cpp
class Myclass {
public:
    bool operator==(int)const;   // tek overload ile
};

int main()
{
    Myclass m;

    bool b1 = (m == 5);   // Hepsi legal

    bool b2 = (m != 5);
    bool b3 = (5 == m);
    bool b4 = (5 != m);
}
```

```cpp
#include <iostream>

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


int main()
{
    using namespace std;

    boolalpha(cout);

    Myclass m1{ 24 }, m2{ 56 };

    cout << "m1 <  m2  : " << (m1 < m2) << "\n";
    cout << "m1 <= m2  : " << (m1 <= m2) << "\n";
    cout << "m1 > m2   : " << (m1 > m2) << "\n";
    cout << "m1 >= m2  : " << (m1 >= m2) << "\n";
    cout << "m1 == m2  : " << (m1 == m2) << "\n";
    cout << "m1 != m2  : " << (m1 != m2) << "\n";
```

* Operator `==` 'de default edebilir. Eğer o zaman sadece `=!` ve `==` legal olur!

* `<=>` default edersek, otomatik olarak `==` da, default etmiş olucaz!

Spaceship operatörü karşılaştırılan türlere göre → farklı türden değerler elde edilir.

- strong ordering → karşılaştırmanın sonucu kesin belli [x==y ise f(x) = f(y) olmalı]
- weak ordering → eşitlik ile eşdeğerlik arasında fark var [mesela MERT ile MERT eşit ama eşdeğer değil]
- partial ordering → karşılaştırılamayacak değerler var. CNAN, sonsuz ...

→ Bu ordering türlerine göre, elde edilecek değerler,

```cpp
strong_ordering::equal
strong_ordering::equivalent
strong_ordering::less
strong_ordering::greater

// * Daha sıkı tür, daha gevşek tür dönüşebilir!

weak_ordering::equivalent
weak_ordering::less
weak_ordering::greater


partial_ordering::equivalent
partial_ordering::less
partial_ordering::greater
partial_ordering::onordered
```

boolean!

- `a <=> b < 0`

boolean değil

*** Spaceship Operatorünü Default Etmek Yerine Biz Yazsaydık:**

```cpp
public:
    Person(const char *p, int a) : name{p}, age{a} {}
    std::strong_ordering operator <=>(const Person& other)const
    {
        if (auto cmp = name <=> other.name; cmp != 0)
            return cmp;

        return age <=> other.age;
    }
private:
    std::string name;
    int age;
};


int main()
{
    using namespace std;

    Person p1{ "muhittin", 56 };
    Person p2{ "ayse", 40};

    cout << boolalpha << (p1 > p2) << "\n";
}
```

(strong_ordering operator →) weak / partial ordering return etmesi syntax hatası değil!

(return age <=> other.age; →) Çünkü int karşılaştırması

↓ Fakat!

```cpp
public:
    Person(const char* p, int a, double s) : name{ p }, age{ a }, salary{ s }{}
    auto operator <=>(const Person& other)const
    {
        if (auto cmp = name <=> other.name; cmp != 0)
            return cmp;

        if (auto cmp = age <=> other.age; cmp != 0)
            return cmp;

        return salary <=> other.salary;
    }
private:
    std::string name;
    int age;
    double salary;
};
```

(return salary →) Şimdi syntax error! Çünkü deduction yapamaz!!!

→ Double türü, partial ordering

⊛ std::strong_order:

→ partial veya weak türler belirli kurallara tabi tutularak, strong ordering return etmeye zorlayabiliriz.

```cpp
using namespace std;

class Employee {
private:
    std::string mname;
    double mwage;
public:
    std::strong_ordering operator<=> (const Employee& rhs) const
    {
        if (auto cmp = mname <=> rhs.mname; cmp != 0)
            return cmp;

        return std::strong_order(mwage, rhs.mwage);
    }

};
```

→ Böylelikle, syntax hatası olmaz