

- Variant'ın kollarının birinden bırr de **Kolitüm alternatif olur musa!**
- **Closed Hierarchy:** Bütün ne kollar sınıf türüne göre belir!
- **Kolitüm Alternatifleri:**
 - Type Erasure: Tümler kolların başınumuz, ortak interface oluşturular
 - CRTP: Kolitüm Compile Time da oluşturular
 - Variant: Variant'ın alternatiflerin türünü sınıf bırr is geçer.

④ iki Argumen ile Construct Edilmeli Gerekken Variantlar:

```
int main()
{
    using namespace std;
    variant<complex<double>> v1{ 3.0, 4.0 }; // gecersiz
    //variant<complex<double>> v2{ {3.0, 4.0} }; // ERROR
    variant<complex<double>> v11{ in_place_type<complex<double>>, 3.0, 4.0 };
    variant<complex<double>> v12{ in_place_index<0>, 3.0, 4.0 };
}
```

```
#include <set>

int main()
{
    using namespace std;

    auto pred = [] (int x, int y) {
        return abs(x) < abs(y);
    };

    std::variant<
        set<int>, set ve less argomeni
        set<int, decltype(pred)>> vx{
            std::in_place_index<1> 1 nolu alternatif icin.
            {4, 8, -7, -2, 0, 15},
            pred };
}
```

kullan closer
type'mız

+ Visit Fonksiyonu:

```
int main()
{
    std::variant<char, int, double> var = 'a';
    const auto f = [] (auto x) { std::cout << x << '\n'; };
    //std::visit(f, var);

    var = 123;
    std::visit(f, var);
    var = 4.7;
    std::visit(f, var);
}
```

generalized
lambda

Həsratma:

```
class xyz13 {  
public:  
    template <typename T> operator()(T x)  
    {  
    }  
};  
  
int main()  
{  
    std::variant<char, int, double> var = 'a';  
    const auto f = [](auto x) { std::cout << x << '\n'; };  
    xyz13{}  
}
```

get name
compr auto yu gorince sade bir tempate olustur.

2. Örnek:

```
struct triple {  
    template<typename T> void operator()(T& t) const  
    {  
        t = t + t + t;  
    }  
};  
  
int main()  
{  
    using namespace std;  
  
    variant<int, string, double> vx{ 243 };  
  
    visit(triple{}, vx);  
    cout << get<0>(vx) << "\n";  
    vx = "can";  
    visit(triple{}, vx);  
    cout << get<1>(vx) << "\n";  
    vx = 4.5;  
    visit(triple{}, vx);  
    cout << get<2>(vx) << "\n";  
}
```

predise yerine direkt fonksiyon kullanırı

3. Örnek: → girdiğinde VARIANT / VISIT OZ. trafi

→ buraya dairlik için gen uzaq ama çok iyi bir örnek!

4. Örnek:

```
int main()  
{  
    using namespace std;  
  
    using vtype = variant<int, double, string>;  
    vector<vtype> vec{ 199, 3.14159, "necati ergin", "hakan koc" };  
  
    for (const vtype& val : vec) {  
        std::visit([](const auto& v) {  
            if constexpr (std::is_same_v<decltype(v),  
                           const std::string&>) {  
                std::cout << quoted(v) << ' ';  
            }  
            else {  
                std::cout << v << ' ';  
            }  
        },  
        val);  
    }  
}
```

generalized lambda
turnut içermeyende yazar.

* Overloader Idiom: → Dersin yeknesik 1 saat LS: dörtlükteki anlatı
→ Tam anlamadım. Teşrif et!

```
template <typename ...Ts>
struct Overloader : Ts... {
    using Ts::operator()...;
};

int main()
{
    using namespace std;

    Overloader x{
        [](int x) {std::cout << "int " << x << '\n'; },
        [](double x) {std::cout << "double " << x << '\n'; },
        [](long x) {std::cout << "long " << x << '\n'; },
        [](string x) {std::cout << "string " << x << '\n'; }
    };
    x(2.3);
}
```

→ Variant ve Overloader Idiom:

```
template <typename ...Ts>
struct Overloader : Ts... {
    using Ts::operator()...;
};

int main()
{
    using namespace std;

    variant<int, double, long, string> vx{ 34L };

    visit(Overloader{
        [](int x) {std::cout << "int " << x << '\n'; },
        [](double x) {std::cout << "double " << x << '\n'; },
        [](long x) {std::cout << "long " << x << '\n'; },
        [](string x) {std::cout << "string " << x << '\n'; }
    }, vx);
```

I

argumen olarak vx

* Std:: Any:

→ C dilindeki void *'a benziler. → Tstedigimiz türden bir nesnenin adresi atelir.

→ Vard *'den farklı:

- Vard *'a benzerdir, ancak nesnenin türünü belirlemek için type_info türünden referans alır. Genel seviyedeki türdeki bilgileri kullanır!
- Any'de bu sevgi runtime'da yapabiliyor.

* Any'in bunu yapabilmesinin sebebi, türdeki değer ile birlikte, type_info türünden referans alıyor!

↳ type_id'ının

sondardaki değeri!

```
#include <variant>
#include <any>
#include <bitset>
#include <vector>

int main()
{
    using namespace std;

    any x = 10;
    x = 4.5;
    x = "mustafa";
    x = "necati"s;
    x = bitset<16>(246u);
    x = vector<int>{ 1, 2, 3, 4, 5, 6, 7 };
}
```

Tür ataması
Bir any_cast eklenerek
farklı template kullanılır.

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

    any x = 1.0;           →iger uyarısı varsa
    auto val = any_cast<int>(x); exception
}                                throw edilecek
                                    ↓
→ Bad - Any cast türünden
```

* Any nesnesi herhangi bir değer türümle zarında değil. Reset fonksiyonu neye basarılıdır! → Böylelikle değer tutmaz

```
using namespace std;

any x = 10;

x.reset();      I
x = any{};
x = {};
```

* has_value ne değerin cuih olduğunu söylüyor.

```
#include <variant>
#include <any>
#include <bitset>
#include <vector>

int main()
{
    using namespace std;

    any x = 10;           ↘ an döw
    cout << (x.has_value() ? "dolu" : "bos") << "\n";
}
```

* Any Constructors:

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

    //any a1{ 12 }; //int      I
    //any a2 = 4.5; //double
    any a3{ "necati" }; //const char *
    //any a4{ "necati"s }; //std::string
    //any a5{ std::bitset<16>{} }; //std::bitset<16>
    //any a6{ std::vector<int>{1, 3, 5} }; //std::vector<int>
    any b1; //empty
    any b2{}; //empty
    any b3 = {};//empty
}
```

* make_Any:

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

    auto a1{ make_any<string>(10, 'X') };
    auto a2{ make_any<complex<double>>(1.2, 4.5) };

    cout << any_cast<string>(a1) << '\n';
    cout << any_cast<complex<double>>(a2) << '\n';
}
```

* Any, okşurdan nesneye farklı bir bellek alanı sunarak optimize

```
template <auto n>          → non-type parameter
struct Nec {                ve n dedektör ne bilindik!
    unsigned char buffer[n]{};
};

int main()
{
    using namespace std;
    any x = Nec<48>{};     → buut bellek alanının yararlı
                            olduğu durumda, new operatör
                            kullanır.
}
```

(*) Any.cast returns öndürmez!

```
int main()
{
    using namespace std;    I
    any x = 10;
    any_cast<int>(x) = 67;
}
```

```
int main()
{
    using namespace std;    I
    any x = 10;
    any_cast<int&>(x) = 67;  → after return
                            icin reference
                            cast et!
}
```

* Any nesnesinin hangisi bitince Destruktor çağırılır.

```
struct Nec {  
    Nec(int x) : mx{x}  
    {  
        std::cout << "Nec(int)\n";  
    }  
    int mx{};  
    Nec& operator=(const Nec& other)  
    {  
        std::cout << "operator=()\n";  
        mx = other.mx;  
        return *this;  
    }  
};  
int main()  
{  
    using namespace std;  
    any x = Nec{ 35 };
```

→ 1. destruktur NEC iin
2. destruktur ANY x iin

```
int main()  
{
```

```
    using namespace std;  
  
    Nec n1{ 35 };  
    Nec n2{ 47 };
```

any a = n1;)
a = n2; *yeni değer
elemanı da
eski değerden destruktur
çağırılır!*

* Type Info:

```
using namespace std;  
  
any a = 1.2;  
  
cout << a.type().name() << "\n";
```

```
any a;  
  
if (a.type() == typeid(void)) {  
    std::cout << "bos\n";}
```

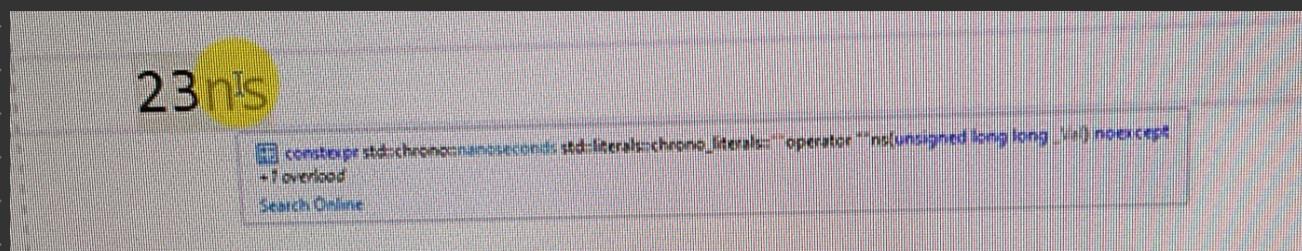
* Emplace:

```
class Nec  
{  
    int mx, my;  
public:  
    Nec(int x, int y) : mx{x}, my{y} {}  
    //...  
};  
  
int main()  
{  
    using namespace std;  
  
    auto fcmp = [](int x, int y) {return abs(x) < abs(y); };  
    any a;  
  
    //a = 12; //int  
    //a = Nec{ 10, 20 }; //Nec  
    a.emplace<Nec>(2, 5); //Nec  
    a.emplace<string>(10, 'A'); //string → ya da user defined literal  
    a.emplace<vector<int>>(100); //vector<int>  
    a.emplace<set<int, decltype(fcmp)>>({ 1, 5, -4, -6, 3 }, fcmp);  
}
```

"met" s

* User Defined Literals:

- "metre" > sınıfında bir string olusturduğumuzda / ya da chrono 12s gibi, arka planda "literal operator function" oluşturuyor.
constexpr



- Cooked ve uncooked örneklerde 2 farklıdır.

```
//cooked  
//uncooked  
  
int main()  
{  
    // 3456_km  
    // operator""_km(3456)  
}
```

Annotations in red:

- eger sagisinde direkt gonderiliyorsa
cooked
- eger sonrakilerde tek tek
Okuyup reverse uncooked

```
constexpr double operator""_km(long double x)  
{  
    return static_cast<double>(x * 1000);  
}  
  
int main()  
{  
    34.5_km  
}
```

```
#include <array>  
  
namespace Nec  
{  
    constexpr size_t operator""_KB(unsigned long long size)  
    {  
        return static_cast<size_t>(size * 1024);  
    }  
  
    constexpr size_t operator""_MB(unsigned long long size)  
    {  
        return static_cast<size_t>(size * 1024 * 1024);  
    }  
}  
  
int main()  
{  
    using namespace Nec;  
    auto size{ 4_KB };           // size_t size = 4096;  
  
    using byte = unsigned char;  
    auto buf1 = std::array<byte, 16_KB>{};  
    auto buf2 = std::array<byte, 1_MB>{};  
}
```