* Visit Devam:

```cpp
struct NecVis {

    template <typename T, typename U>
    void operator()(const T& t, const U& u)
    {
        std::cout << typeid(T).name() << ", " << typeid(U).name() << '\n';
        cout << "(" << t << ", " << u << ")\n";
    }

};



int main()
{
    using namespace std;

    variant<int, double, string> vx(3.4);
    variant<long, int, Date> vy(56);

    visit(NecVis{}, vx, vy);


}
```

```cpp
using namespace std;

auto fn = [](const auto& t, const auto& u)
{
    std::cout << typeid(t).name() << ", " << typeid(u).name() << '\n';
    std::cout << "(" << t << ", " << u << ")\n";
};


variant<int, double, string> vx(4.5);
variant<long, int, Date> vy(in_place_index<2>, 23, 7, 2023);

visit(fn, vx, vy);
```

Abriviated Template Syntax

Cpp 20

⊕ Variantdan kalıtım yapılamaz X

Variantten kalıtım yapılabilir. Base Class olacak!!

```cpp
class Derived : public std::variant<int, std::string> {};


int main()
{
    using namespace std;

    Derived d = { "hello" };
    cout << d.index() << '\n';
    cout << get<1>(d) << '\n';
    d.emplace<0>(77);
    cout << get<0>(d) << '\n';
}
```

```cpp
template <typename ...Args>
struct Der : Args ... {

};


struct A {  };
struct B {  };
struct C {  };


int main()
{
    Der<A> x;
    Der<A, B> y;
    Der<A, B, C> y;
}
```

variadic template paketi

pack expansion

* Aggregate Initialization: Hatırlatma

```cpp
struct A {
    A(int);
};

struct B {
    B(double);
};

struct C {
    C(int);
};


struct D : A, B, C {

};


int main()
{
    D dx =  {35, 2.3, 34};
}
```

* Lambda Expression: Hatırlatma

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

    decltype(fn) fx;
}
```

Bu bildirim Cpp 20'den sonra geçerli ✓

→ Cpp 20 kadar, closure typelarin default ctor'u deleted!

* Overloader Idiom:

```cpp
template <typename ...Args>
struct Overload : Args... {

};


int main()
{
    Overload x{
        [](int a) {return a * a; },
        [](int a) {return a * 3; },
        [](int a) {return a * 4; },
        [](int a) {return a * 3; },
    };
}
```

burada tarz cikarmanin sagliyor: CTAD

→ Cpp 17'de calismasi, cunku deduction guide gerekli!

Cpp 17'de eklendi!

* Variant ve Overloader Idiom'u:

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


int main()
{
    using namespace std;
    variant<int, string> vx(99);

    visit(overload{
        [](int ival) {cout << "int: " << ival << '\n'; },
        [](const std::string& s) {
            std::cout << "string: " << s << '\n'; },
        },
        vx);

    auto twice = overload{
        [](std::string& s) { s += s; },
        [](auto& i) { i *= 2; },
    };

    visit(twice, vx);
    std::cout << get<0>(vx) << "\n";
}
```

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


int main()
{
    using namespace std;
    variant<int, double, string> vx("ahmet");
    variant<int, long, string> vy("aksoy");

    visit(
        overload{
            [](int, int) { cout << "int, int\n"; },
            [](int, string) { cout << "int, string\n"; },
            [](double, long) { cout << "double, long\n"; },
            [](auto, auto) { cout << "other types\n"; },
        }, vx, vy);


}
```

* Valueless By Exception:

```cpp
struct S {
    operator int()const {
        throw std::runtime_error{ "hata" };
        return 1;
    } // any conversion to int throws
};

int main()
{
    using namespace std;

    variant<double, int> var{ 12.2 };
    try {
        var.emplace<1>(S{});
    }
    catch (const std::exception& ex) {
        cout << "hata yakalandi ... " << ex.what() << "\n";
        cout << boolalpha << var.valueless_by_exception() << "\n";
        cout << "var.index() = " << var.index() << "\n";
        std::cout << (var.index() == variant_npos) << "\n";
    }
}
```

emplace varyantlari alternatifi destroy eder!

return degeri bool

* Exception throw edildigi için, variant gecersiz!

* Eski öğe destroy olup, yeni öğenin oluşması sırasında exception throw edildi!

* Std :: Variant Kullanım Alanları: • N tane alternatif dan herhangi variant kullanılabilir.

→ Error type için kullanılabilir!

```cpp
struct Data {
    //...
};

enum ErrorType {system, archive, log};

std::variant<Data, ErrorType> foo();
```

→ Run time polymorphism'e alternatif • Virtual Dispatch mekanizmasında, compiler bütün hiyerarşideki fonksiyonlara v pointer görür. Sanal fonksiyon tablosunda o fonksiyon adresleri oluşturur. → bellek ve zaman açısından maaliyet

- Polimorfik nesneler dinamik ömürlüdür. Her biri için new / delete operasyonu gerçekleştirilir.
- Sınıflar birbirine bağımlı!

```cpp
class Xls {

};

class Pdf {

};

class Txt {

};

class Word {

};

using Document = std::variant<Xls, Pdf, Txt, Word>;

int main()
{
    using namespace std;

}
```

Artık ortak bir interface'e sahip olmak zorunda değiller!

başta bizim dokümanlarımız belli: Concrete

→ otomatik ömürlü nesneler olusur!

```cpp
class Document {
public:
    virtual void print()const = 0;
    //
};


class Pdf : public Document {
public:
    virtual void print()const override;
};
```

* Std:: Any:

- Bu bir class template değil!
- Type erasure var!

* Optional → default construct edilince değere sahip değil!
* variant → default construct edilince: ilk alternatif! → boş alternatif yok! monostate kullanılır!
* any → default construct edilince değere sahip değil!
- has value ile değer var mı yok mu sorgulanabilir
- Yeniden atama yapılabilir!

```cpp
any a(45);

a = 1.5;
a = "mustafa";
a = string{ "yilmaz denizhan" };
```

* Type Fonksiyonu:

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

    any a(45);

    a = "ali"s;

    if (a.type() == typeid(int)) {
        std::cout << "int\n";
    }
    else if(a.type() == typeid(double)) {
        std::cout << "double\n";
    }
    else if (a.type() == typeid(string)) {
        std::cout << "string\n";
    }
```

typeid tanımdan referans

- Eğer bir tür yoksa, type = void

* any_cast:

```
any a{3.4};
cout << any_cast<int>(a) << "\n";
```

double

Bad any cast tanımdan exception throw edildi!

→ Kullanım amacı, any nesnesine erişmek!

* make_any / reset / emplace:

```
auto a = make_any<string>(20, 'T');
cout << boolalpha << a.has_value() << "\n";
a.reset();
cout << a.has_value() << "\n";
a.emplace<Date>(4, 5, 1987);

cout << a.has_value() << "\n";
cout << any_cast<Date>(a) << "\n";
```

* Std:: Any in Containers:

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

    vector<any> avec{ 12, 1.43, "alican", 34L, "murathan"s };

    for (const auto& a : avec) {
        if (auto pi = any_cast<int>(&a)) {
            std::cout << *pi << "\n";
        }
        else if (auto pd = any_cast<double>(&a)) {
            std::cout << *pd << "\n";

        }
        else if (auto pc = any_cast<const char*>(&a)) {
            std::cout << *pc << "\n";
        }
        else if (auto pl = any_cast<long>(&a)) {
            std::cout << *pl << "\n";
        }

    }

}
```

```cpp
using tv_pair = std::pair<std::string, std::any>;

int main()
{
    using namespace std;

    vector<tv_pair> vec;

    vec.emplace_back("name", "ahmet aksoy"s);
    vec.emplace_back("year", 1998);
    vec.emplace_back("month", 11);
    vec.emplace_back("month day", 22);
    vec.emplace_back("wage", 87.67);
    vec.emplace_back("town", "eskisehir"s);
    vec.emplace_back("gender", "male"s);
    vec.emplace_back("country", "Turkey"s);
    //

    cout << left;

    for (const auto& [property, value] : vec) {
        if (value.type() == typeid(int))
            cout << setw(16) << property << any_cast<int>(value) << '\n';
        else if (value.type() == typeid(double))
            cout << setw(16) << property << any_cast<double>(value) << '\n';
        else if (value.type() == typeid(string))
            cout << setw(16) << property << any_cast<string>(value) << '\n';
    }
}
```