2023 12 27

```
//vocabulary type
/*
    std::optional
    std::variant
    std::any
*/
```

std::variant

```
// variant boş olma durumu
#include<variant>
struct Data {
    Data(int x) : mx{x} {}
    int mx;
}:
int main()
{
    using namespace std;
    // syntax hatası çünkü Data default ctor yok
    variant<Data, int, double> vx;

    variant<int, Data, double> mx; // legal
    variant<monostate, long, Data> mx1; // bos variant

    cout  << "mx1 indeks " << mx1.index() << "\n"; // bossa sıfır

    if (holds_alternative<monostate>(mx1))
        std::cout << "bos\n";
    else
        std::cout << "bos değil\n"

    if (get_if<monostate>(&mx)) // monostate değilse bos döndürür
        std::cout << "bos\n";
    else
        std::cout << "bos değil\n"

    if (get_if<0>(&mx)) // monostate değilse bos döndürür
        std::cout << "bos\n";
    else
        std::cout << "bos değil\n"

    // default variant boş olur çünkü ilk alternatif monostate
    mx1 = {};
}
```

```cpp
// variant karşılaştırma
int main()
{
    // farklı türden variantlar karşılaştıralamaz
    using namespace std;

    variant<int, double> v1(29), v2(1.2), v3(11);
    // türler farklıysa indekse göre karşılatırma olur
    cout << (v1 < v2) << "\n";  // 0 < 1  true
    // türler aynıysa tuttuğu değere göre karşılatırma olur
    cout << (v1 < v3) << "\n";  // 29 < 11  false
}
```

```cpp
// type alias --variant
int main()
{
    using namespace std;

    using age_t = int;
    using gender_t = char;
    using name_t = string;

    enum : size_t {idx_age, idx_gender, idx_name};
    variant<age_t, gender_t, name_t> p;

    get<age_t>(p);
}
```

```cpp
struct Nec
{
    operator int()const
    {
        throw std::runtime_error{"error error"};
        return 1;
    }
};

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

    try{
        vx.emplace<0>(Nec{});
    }
    catch (const std::exception& ex){
        // bu duruma valueless by exception denir.
        std::cout << "hata yakalandi : " << vx.what() << "\n";
        // garabe value döner çünkü hayata gelemeden exception throw ettik.
        std::cout << vx.index() << "\n";

        cout << vx.valueless_by_exception() << "\n"; // true döner
        cout << (vx.index() == variant_npos) << "\n"; // true döner
    }
}
```

```cpp
// in_place_index ve in_place_type
int main()
{
    using namespace std;
    // perfect_forwarding yaptık böylece
    variant<int, Date, string> v1(in_place_index<1>, 3, 5, 1998);
    variant<int, Date, string> v2(in_place_type<Date>, 3, 5, 1998);
    variant<int, Date, string> v3 (in_place_index<2>, 20, 'm');
}
```

```cpp
int main()
{
    using namespace std;
    variant<int, Date, string, double> vx{4.5};
    if (vx.index() == 0)
    {
        cout << "alternative int code\n";
        cout << get<0>(vx) << "\n";
    }
    else if (vx.index() == 1)
    {
        cout << "alternative Date code\n";
        cout << get<1>(vx) << "\n";
    }
    else if (vx.index() == 2)
    {
        cout << "alternative string code\n";
        cout << get<2>(vx) << "\n";
    }
    else if( vx.index() == 3)
    {
        cout << "alternative double code\n";
        cout << get<3>(vx) << "\n";
    }
}
```

```cpp
//std::visit
struct Visitor
{
    void operator()(int) const
    {
        std::cout << "int alternative code\n";
    }
    void operator()(Date) const
    {
        std::cout << "Date alternative code\n";
    }
    void operator()(const std::string &) const
    {
        std::cout << "string alternative code\n";
    }
    void operator()(double) const
    {
        std::cout << "double alternative code\n";
    }
}
```

```cpp
struct Visitor1
{
    void operator()(auto x)const
    {
        std::cout << x << "\n";
    }
};
int main()
{
    using namespace std;
    variant<int, Date, string, double> vx{Date{3, 6, 1998}};
    visit<Visitor{}, vx>;
}
```

```cpp
struct Visitor{
    void operator()(int) {};
    void operator()(double){};
    void operator()(auto){}; // int ve double dışındakileri bunu çağrır
};
struct Visitor
{
    void operator()(auto x)
    {
        if constexpr (std::is_same_v<decltype(x), int>
        {

        }
        else if constexpr (std::is_same_v<decltype(x), double>)
        {

        }
    }
};
```

multi-lambda

```cpp
// multi-lambda
struct X
{
    void foo(int);
};
struct Y
{
    void foo(double);
};
struct Z
{
    void foo(long);
};
struct A : X, Y, Z
{
    // böyle yaparsak overloading çalışır
    using  X::foo;
    using  Y::foo;
    using  Z::foo;
};
```

```cpp
int main()
{
    A a;
    // syntax hatası çünkü multi inheritance'ta isim arama aynı anda yapılır, 
öncelik yok
    a.foo(2.3);
    a.foo(23);
    a.foo(23L);
}
```

```cpp
template <typename ...Args>
class Myclass {

};

class Nec : public Myclass<int, double, long> // yapabiliriz

int main()
{
    Myclass my<int, doouble, long>

}
```

```cpp
struct A{void foo(); };
struct B{void bar(); };
struct C{void baz(); };

template <typename ...Args>
struct Der : public Args...
{

};

int main()
{
    Der<A, B, C> myder;

    myder.baz();
    myder.foo();
    myder.bar();
}
```

```cpp
struct A{void foo(int); };
struct B{void foo(long); };
struct C{void foo(double); };

template <typename ...Args>
struct Der : public Args...
{
    // ambiguity çözümü
    using Args::foo...;
};

int main()
{
    Der<A, B, C> myder;
    myder.foo(12); // ambiguity

    // CTAD
    Der d1 {A{}};
    Der d2 {A{}, B{}};
    Der d3 {A{}, B{}, C{}};
}
```

```cpp
auto f1 = [](int x){return x * x};
auto f2 = [](int x){return x + x};
auto f3 = [](int x){return x * x - 6};

struct Nec : decltype(f1) , decltype(f2), decltype(f3)// closure type yani bir
class type
{

}

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

```cpp
// multi-lambda idiom
template <typename ...Args>
struct MultiLambda : Args...
{
    // ambiguity kalktı
    using Args::operator()...;
};

int main()
{
    MultiLambda mx{
        [](int x) {return x + 1;},
        [](double x) {return x + 1.3;},
        [](long x) {return x + 5;},

    };

    mx(12); // ambiguity
}
```

```cpp
template <typename ...Args>
struct Overload : Args...
{
    // ambiguity kalktı
    using Args::operator()...;
};

struct Nec{};
struct Erg{};
struct Tmr{};

int main()
{
    variant<int, double, float, Nec, Erg, Tmr> v;

    auto f = Overload{
        [](int) {return "int";},
        [](double) {return "double";},
        [](float) {return "float";},
        [](Nec) {return "nec";},
        [](Erg) {return "erg";},
        [](Tmr) {return "tmr";},
    };

    cout << visit(f, v) << "\n"; // int
}
```

```cpp
struct Visitor {
    void operator()(const string&, int)
    {
        std::cout << "string - int\n";
    }
    void operator()(int, double)
    {
        std::cout << "int - double\n";
    }
    void operator()(int, float)
    {
        std::cout << "int - float\n";
    }
    void operator()(auto x, auto y)
    {
        std::cout << typeid(decltype(x)).name() << " - "
                  <<  typeid(decltype(x)).name() <<  "\n";
    }
};
struct Nec{};
int main()
{
    using namespace std;

    variant<int, double, float, char> v1;
    variant<float, int, char, Nec> v2;

    visit(Visitor{}, v1, v2); // int - float
}
```

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

class Pdf : public Document {
    virtual void print() override  { std::cout << "Pdf::print()\n"; }
};

class Excel : public Document {
    virtual void print() override  { std::cout << "Excel::print()\n"; }
};

class Word : public Document {
    virtual void print() override  { std::cout << "Word::print()\n"; }
};

void process(Document* p)
{
    p->print();
}

int main()
{
    auto pdoc = new Pdf();
    process(pdoc);
}
```

```cpp
class Pdf {};
class Excel {};
class Powerpoint {};
class Word {};

using Document = std::variant<Pdf, Excel, Word, Powerpoint>
struct PrintVisitor {
    void operator()(Pdf x) {

    }
    void operator()(Excel x) {

    }
    void operator()(Word x) {

    }
    void operator()(Powerpoint x) {

    }
};

int main()
{
    Document x{Word{}};

    visit(PrintVisitor{}, x);
}
```

std::any

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

    any x1 { 23 }; // int
    any x2 { 2.3 }; // double
    any x3 { "murathan" }; // const char*
    any x3 { "murathan"s }; // std::string

    if (x.has_value())
        std::cout << "dolu\n";

    cout << x1.type().name() < "\n"; // int
    x = 2.4
    cout << x.type().name() < "\n"; // double
}
```

```
// std::any_cast
int main()
{
    using namespace std;

    any x(10);
    cout << any_cast<int>(x) << "\n";

    x = "necati"s
    cout << any_cast<int>(x) << "\n"; // exception verir bad_any_cast
}
```