5.Hafta

İçindekiler

> 08_22_07_2023

---

std::optional

- It is recommended to use `optional<T>` sin situations where theere is exactly one, clear (to all parties) reason for having no value of type T, and where the lack of a value is as natural as having any regular value of type T.

Nesnesini nasıl oluştururuz?

- Default initialize edilmiş bir `optional` nesnesi, `std::nullopt` durumunda. Default construct edilmiş nesnesi default contsrukt edilmiş bir `T` türünden nesnesine sahip değil.

```
int main()
{
    using namespace std;
    std::optional<int> opt; // default initialize edilmiş bir nesne
    boolalpha(cout);
    cout << boolalpha;
    cout << op.has_value() << "\n"; // false

    if(op) { cout << "bir deger var\n"; }
    else { cout << "deger yok\n"; }
    cout << ( op == nullopt)<< "\n";
}
```

Erişmenin yolları

- Tipik bir sarmalıyıcı olduğu için doğrudan değiştirilecek bir nesne de kullanabiliriz.
- - Eğer boş bir optional nesnesine erişmeye çalışırsak, undefined behavior oluşur.

```
int main()
{
    using namespace std;
    std::optional<string> op{"deneme"};
    cout << *op<< "\n"; // deneme
    cout << op->size();
    //cout << *op; //Eğer boş olsaydı ub olurdu
}
```

- `op.value()` fonksiyonu ile de değere erişebiliriz ve bu fonksiyon çağırıldığında exception throw ediyor.

```
int main()
{
    using namespace std;
    std::optional<string> op;
    try {
        cout << op.value() << "\n"; // deneme
        cout << op.value().size();
    }
    catch(const std::bad_optional_access& e) {
        std::cout << e.what() << '\n';
    }
}
```

- `op.value_or()` fonksiyonu eğer optional nesnesi bir değer tutuyorsa bu değeri döndürür, eğer boş ise, parametre olarak verilen değeri döndürür.

```
int main()
{
    using namespace std;
    std::optional<string> op;
    cout << op.value_or("bos") << "\n"; // bos
    std::optional<string> op2{"deger"};
    cout << op2.value_or("bos") << "\n"; // deger
}
```

- `value_or` fonksyionu `value` dan farklı olarak sol taraf referans döndürmüyor.

- std::optional nesnesi bir referans tutamıyor fakat reference wrapper ile bunu yapabiliriz. Fabrika fonksiyonlarını kullanabiliyoruz

```
int main()
{
    using namespace std;
    string name {"deneme"};
    std::optional<reference_wrapper<string>> op(ref(name));
    op->get() +=  "test";
    //CTAD
    optional x = 12;
}
```

In_place ve make_optional

Yardımcı bir `in_place_t` nesnesi var ve bu türden bir değişkenimiz var `in_place`.

- Default construct edilmiş bir optional nesnesini oluşturmamız mümkün değil default olarak nesneyi construct etmiyor.
- Eğer inplace objesi ile optional nesnesini oluştururşak, bu durumda nesne doğrudan oluşturuluyor.
- in_place ile aldığı argümanları constructor'a parametre olarak geçiyor ve bu rargümanlar perfectforward ediliyor.

```
class Myclass {
public:
    Myclass() {cout << "default ctor\n";}
    ~Myclass() {cout << "dtor\n";}
    Myclass( const Myclass&) {cout << "copy ctor\n";}
    Myclass( Myclass&&) {cout << "move ctor\n";}
};

int main()
{
```

```
    using namespace std;
    optional<Myclass> op; // burada nesneyi construct etmiyor
    optional<Myclass> op1{Myclass{}}; // eğer move ctor varsaa o yoksa copy
ctor çağırılıyro.
    optional<Myclass> op(in_place);  // doğrudan oluşturuldu
    // optional<Date> x {3,5,1987};
    optional<Date> x {in_place, 3,5,1987};
    auto x2 = make_optional<Date>(3,5,1987);
    cout << *x2 ;
    auto op = make_optional<Myclass>();
    auto op = std::optional<Myclass>(in_place);
}
```

Kullanıldığı Temalar

Değerinin olmasının kadar olmamasının da doğal olduğu durumlar:

- Aramada aranan değerin olması kadar değerin olmaması gibi. Aranan değer bulunursa dolu bir optional nesnesi döndürülür, bulunmazsa boş bir optional nesnesi döndürülür.
- Bir kişinin uygulamada nick'i olabilir/olmayabilir.
- Fonksiyonun parametre değişkeni optional olabilir.
- Sınıfın veri elemanı optional olabilir.
- Karşılaştırma operatörleri ile kullanılabilir. Değerinin nullopt olup olmadığını kontrol edebiliriz.

Üye Fonksiyonları

- `reset` : optional nesnesini boşaltır. Tuttuğu nesnenin destructorını çağırıyor.

```
int main()
{
    using namespace std;
    optional<string> op;
    cout << op.has_value() << "\n"; // false
    op = "neco";
    cout << op.has_value() << "\n"; // true
    op.reset();
    cout << op.has_value() << "\n"; // false
    op ="deneme";
    cout << op.has_value() << "\n"; // true
    op = nullopt;
    cout << op.has_value() << "\n"; // false
    op ="deneme";
    //idiomatik
    op = {};
    cout << op.has_value() << "\n"; // false
}
```

- `swap` : iki optional nesnesinin değerlerini değiştirir.

- `emplace` : optional nesnesini doğrudan oluşturur. Eğer nesne dolu ise, mevcut nesneyi destroy eder ve yeni nesneyi oluşturur.

```
int main()
{
    usgin namespace std;
    optional<Myclass>op;
    op.emplace();
    cout << "Main devam ediyor";
    op.emplace();
    cout << "Main devam ediyor";
}
```

- `has_value` : optional nesnesinin değerinin olup olmadığını kontrol eder.
- `operator bool` : optional nesnesinin değerinin olup olmadığını kontrol eder.
- `operator*` : optional nesnesinin değerine erişir.
- `operator->` : optional nesnesinin değerine erişir.
- `value` : optional nesnesinin değerine erişir. Eğer değer yoksa exception throw eder.
- `value_or` : optional nesnesinin değerine erişir. Eğer değer yoksa parametre olarak verilen değeri döndürür.
- Boş optional nesnesi karşılaştırıldığı zaman
- nullopt nesnesi en küçük.

```
int main()
{
    using namespace std;
    optional x = "deneme"s;
    optional<string> y;

    cout << (x == y) << "\n"; // false
    cout << (y < x) << "\n"; // true
    x.reset();
    cout << (x == y) << "\n"; // true
}
```

- bool açılımı.

```
int main()
{
    using namespace std;
    optional<bool> x;
    optional<bool> y{true};
```

```
    optional<bool> z{false};

    cout << boolalpha<< (x == y) << "\n"; // false
    cout << (y < z) << "\n"; // false
    cout <<  (x == z) << "\n"; // false
}
```

- std::optional nesnesi return eden fonksiyonlar için

```
std::optional<string> get_name(int id)
{
    std::optional<string> name;
    if( has_nic(id))
        name = "deneme";
    else
        //return {};
        //return std::optional<string>{};
    //return name;
}

int main()
{
    using namespace std;

    if(optinal<string> name = get_name(12))
        cout << *name << "\n";
    else
        cout << "isim yok\n";

    if(auto op = get_name(12); op)
        cout << *op << "\n";
    else
        cout << "isim yok\n";

    if(auto op = get_name(12))
        cout << *op << "\n";
    else
        cout << "isim yok\n";

//alternatif olarak böyle de yazılabilir.
    if(auto op = get_name(13); op->size() > 10)
        cout << *op << "\n";
    else
        cout << "isim yok\n";
}
```

- Aşağıdaki kod içerisinde, verilen container'da bulunan ilk öğenin konumunu döndüren bir fonksiyon.
- Burada stl'den farklı olarak range almak yerine container ve predict yapıyor.

```
template <typename Con, typename Pred>
auto find_if(Con&& con, Pred&& pred)
{
    using std::begin, std::end;
    auto beg_iter = begin(c), end_iter = end(c);
    auto result = std::find_if(beg_iter, end_iter, pred);
    using iterator = decltype(result);
    if(result == end_iter)
        return std::optional<iterator>();
    return std::optional<iterator>(result);
}

template<typename Con, typename T>
auto find(Con&& c, const T& value)
{
    return find_if(std::forward<Con>(c), [&ival](auto&& x) { return x ==
value; });
}
```

- aldıpi string'i int değere dönüştürecek ve inte dönüştürülemez ise değer döndürülmeyecek.

```
std::optional<int> to_int(const std::string& s)
{
    try{
        return std::stoi(s);
    }
    catch(...) {
        return std::nullopt;
    }
}

std::optional<int> to_int(const std::string& s)
{
    std::optional<int> ret;
    try{
        ret = std::atoi(s);
    }
    catch(...){}

    return ret;
}

int main()
{
    for (auto s : {"42","077","necati","0x33"}) // initializer list sınıfı
    {
        std::optional<int> op = to_int(s);
        if(op)
            std::cout << *op << "\n";
        else
```

```
            std::cout << "gecersiz\n";
    }
}
```

Taşınma semantiği

```
int main()
{
    std::optional<std::string> op1{"deneme"};
    auto op2 = op1; //copy-semantics
    cout << op1->length() << "\n"; // 6
    auto op3 = std::move(op1); //move-semantics
    cout << op1->length() << "\n"; // op1 moved from state'de
}
```

- Allignment ile alakalı problem oluşturabiliyor. Allign storage kullanıldığı için

```
int main()
{
    cout <<  sizeof(double) << "\n"; // 8
    cout << sizeof(std::optional<double>) << "\n"; // 16
}
```

std::variant

Optional ya bir değer tutuyor ya da tutmuyor. Variant ise önceden saptanmış türlerden birini tutuyor. C'deki `union` türüne benzer olarak düşünülebilir. Value semantics'e de uygun oluyor.

- Template argümanları olan türlerden birinden değer tutuyor. Bu türlere alternatif deniyor.
- Başlık dosyası `<variant>`
- `std::monostate` ile nullable type (boş) bir tür oluşturulabilir.

```
temaplate<typename... Args>
class Variant
{};

itn main()
{
    Variant<int,double,std::string>
}
```

> union'ları low-level kodlar dışında çok fazla kullanmamaya çalış. Birlik nesnesi hangi alternatifi tuttuğunu bilemez.

- Burada union yerine variatn türünü kullanırsak hangi türü tuttuğunu bilecek.

- Kaynak sızıntısına bir neden yaratmıyor.

> Bu tür sınıflara **tagged/discriminated union** da deniyor.

- Variant türü kendi içerisinde tutuyor dynamic memory allocation tutmuyor ve bu da en az en büyük boyuta sahip türe sahip olan türün boyutu kadar bir yer tutması anlamına geliyor.

```
int main()
{
    using namespace std;
    variant<int,double,string> v;
}
```

Constructor

- Default edilmiş bir variant nesnesi ilk alternatifi tutuyor. O de value initialize edildiği için bu variant default initialize edilebilir değilse bir sentaks hatası oluşmasına sebep olabilir.
- Alternatiflerden birine dönüştürülecek türden bir argümanla construct edilebilir.
-
    - Burada bir ambiguity oluşabilir. Eğer birden fazla alternatiften biri seçilebilir olduğunda ambiguity oluşabilir. Function overload resolution kurallarına göre seçim yapılabilir.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12}; // int
    variant<int,double,string> v{3.4f}; // double
}
```

Hangi alternatifin tutulduğunu öğrenme

- *const* üye fonksiyonu olan indexi çağırarak hangi alternatifin tutulduğunu öğrenebiliriz.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12}; // int
    cout << v.index() << "\n"; // 0
    v = 3.4f;
    cout << v.index() << "\n"; // 1
}
```

- holds_alternative fonksiyon şablonunu çağırmak bu fonksiyon bize bool değer döndürüyor.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12}; // int
    cout << boolalpha <<holds_alternative<int>(v) << "\n"; // true
    cout << holds_alternative<double>(v) << "\n"; // false
    cout << holds_alternative<string>(v) << "\n"; // false
    v = 3.4f;
    cout << holds_alternative<int>(v) << "\n"; // false
    cout << holds_alternative<double>(v) << "\n"; // true
    cout << holds_alternative<string>(v) << "\n"; // false
}
```

- Bazı kontroller compile time içerisinde yapılıyor.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12}; // int
    if(holds_alternative<char>(v)) // compile time hatası
        cout << "char\n";
}
```

Yardımcı sınıflar

in_place_index_t ve in_place_type_t

- bunlar constexpr variable'lar. Bunlarda variant nesnesinin kullanıldığı alternatifi seçmek için kullanılıyor.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{in_place_index<0>, 12}; // int
    variant<int,double,string> v{in_place_index<1>, 3.4f}; // double
    variant<int,double,string> v{in_place_index<2>, 10, 'b'}; // string
    variant<int,double,string> v{in_place_type<double>, 3.4f}; // double
    variant<int,double,string> v{in_place_type<string>, 10,A}; // string
}
```

- Böylece verilen değerler perfect forward ediliyor.
- Eğer default construct etmek istiyorsak kullanılabilir.
- Eğer variantın tuttuğu türlerden birinin constructor'u birden fazla argüman alıyorsa, bunu çağırmak için kullanılabilir.
- Eğer bir ambiguitiy oluşmasını engellmek istiyorsak kullanılabilir.
- Eğer birden fazla tür birbiri ile aynı ise. Bu durumda bunu belirtmek için kullanılabiliriz.

Sizeof

- Variant türü, alternatiflerden en büyük boyuta sahip olan türün boyutu kadar yer tutar.
- Small buffer optimization yapmıyor.

```
int main()
{
    using namespace std;
    cout << " int"<<sizeof(int) << "\n"; // 4
    cout <<" double"<< sizeof(double) << "\n"; // 8
    cout << " string"<<sizeof(string) << "\n"; // 32
    cout << sizeof(variant<int,double,string>) << "\n"; // 24
}
```

Varianta erişme

Get Fonksiyonu

- `get` fonksiyonu ile variant nesnesinin tuttuğu türün değerine erişebiliriz. Eğer tutmadığı bir değere erişmeye çalışırsak, bir exception throw ediyor. Tanımsız bir davranış yok.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12.3}; // int
try {
    cout << get<0>(v) << "\n"; // 12
}
catch(const std::bad_variant_access& e) {
    std::cout << e.what() << '\n';
}
}
```

- Burada verilen index eğer yok ise compile time'da sentaks hatası veriyor.
- Index ile erişmek dışında tür ile erişmek için get fonksiyonu kullanılabilir.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12.3}; // int
    cout << get<int>(v) << "\n"; // 12
    cout << get<double>(v) << "\n"; // 12.3
    cout << get<string>(v) << "\n"; // exception
}
```

Get-if fonksiyonu

- Exception throw etmiyor ve pointer semantiği ile kullanılıyor. Eğer doğru olmayan alternatif seçilmişse nullptr döndürüyor.

```
int main()
{
    using namespace std;
    variant<int,double,string> v{12.3}; // int
    if(auto p = get_if<int>(&v)) // p nin türü int *
        cout << *p << "\n"; // 12
    else
        cout << "int değil\n";
}
```

- Bu şekilde aşağıdaki gibi bir kolaylık sağlıyor.

```
int main()
{
    using namespace std;
    variant<int, double, string> vx("mustafa");
    if (vx.index() == 2)
    {
        std::cout « "alternative string : " « get<2>(vx) < '\n';
    }
    if (holds_alternative<string>(vx))
    {
        std::cout « "alternative string : " « get<2>(vx) « '\n';
    }
    if (auto sptr = get_if<string>)
    {
        std::cout « "alternative string : " « *sptr « '\n';
    }
}
```

- if-else merdiveni kullanarak buradan variantın elde ettiği değer elde edilebilir.

```
int main()
{
    using namespace std;
    variant<int, double, string> vx("mustafa");
    if (auto sptr = get_if<string>(&vx))
    {
        std::cout « "alternative string : " « *sptr « '\n';
    }
    else if (auto dptr = get_if<double>(&vx))
    {
        std::cout « "alternative double : " « *dptr « '\n';
    }
    else if (auto iptr = get_if<int>(&vx))
```

```
    {
        std::cout « "alternative int : " « *iptr « '\n';
    }

    if(holds_alternative<string>(vx))
    {
        std::cout << "alternative string : " << get<string>(vx) << '\n';
    }
    else if(holds_alternative<double>(vx))
    {
        std::cout << "alternative double : " << get<double>(vx) << '\n';
    }
    else if(holds_alternative<int>(vx))
    {
        std::cout << "alternative int : " << get<int>(vx) << '\n';
    }

    if(vx.index() == 2)
    {
        std::cout << "alternative string : " << get<2>(vx) << '\n';
    }
    else if(vx.index() == 1)
    {
        std::cout << "alternative double : " << get<1>(vx) << '\n';
    }
    else if(vx.index() == 0)
    {
        std::cout << "alternative int : " << get<0>(vx) << '\n';
    }
}
```

- Bazı durumlarda C++ yeni standartlar ile backward-compatibility kırılabiliyor. Aşağıdaki kodda C++17'de bool seçilirken, c++20'de string seçiliyor.

```
int main()
{
    using namespace std;
    variant<bool, string> vx("mustafa");
    cout << vx.index() << "\n"; // 1 fakat eğer bu kod C++17 olsaydı bool
}
```

- Kodun daha kolay okunabilmesi için aşağıdaki gibi bir kod yazılabilir.

```
int main()
{
    using namespace std;
    enum idx : size_t {age,wage,name};
    using Age = int;
    using Wage = double;
```

```
    using Name = string;
    variant<Age,Wage,Name> vx("test"); // ayni ya int ya double ya da
string
    cout << get<age>(vx);
    cout << get<Age>(vx);
}
```

Atama Yapma Yolları

Doğrudan variant türüne ambiguity oluşturmadan dönüştürülecek bir değer ile atama yapabiliyoruz.

```
int main()
{
    using namespace std;
    variant<int,double,string> v; // int
    v = 12; // int
    v = 3.4f; // double
    v = "neco"; // string
    v = string(19,'a');
}
```

- Emplace fonskiyonu ile variant nesnesinin tuttuğu türden bir atama yapabiliriz. Eski değeri destroy eder ve yeni değer oluşturuyor.

```
int main()
{
    using namespace std;
    variant<int,double,string> v; // int
    v.emplace<int>(12); // int
    v.emplace<double>(3.4f); // double
    v.emplace<string>(19,'a'); // string
}
```

monostate sınıfı

Variant başlık dosyasında std::monostate isimli bir sınıf var ve bu bir örüntü, buradaki tüm alternatifler tek bir değere sahip. Bize iki avantaj sağlıyor.

- Tek bir state'e sahip olduğu için karşılaştırma işlemlerinde variantın boş olması anlamına geliyor.
- Varaintların kullanıldığı bazı türler için default constructor olmuyor bu durumda monostate kullanılıyor.

Tipik olarak variant ilk alternatif yapılıyor.

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

```
int main()
{
    using namespcae std;
    //variant<B,A> vx; bu durudma sentaks hatası oluyor ve bunun yerine
    variant<monostate, A, B> vx; //
}
```

- Eğer monostate durumuna çekmek istersek:

```
int main()
{
    using namespace std;
    variant<monostate,int,double,string> vx(4.5);
    vx ={};
    vx = monostate{};
    vx.emplace<0>();
    vx.emplace<monostate>{};
}
```

Visitor paterni

- Variantlar için en sık yapılan işlem alternatife erişmek ve alternatif üzerinde işlem yapmak. Bu fonksiyon variadic bir fonksiyon şablonu ve bu fonksiyon bir veya birden fazla variantlar üzerinde işlem yapabilir.

```
template<class Visitor, class Variants>
constexpr visit(Visitor&&vis, Variants)
```

- Visit fonksiyonu variantın seçimini kendisi yapıcak. Burada verilen callable için verilen functor classın tüm variantlar için çağırıalbilir olması gerekiyor.

```
struct PrintVisitor
{
    void operator()(int x)const
    {
        std::cout<< "int : " << x << "\n";
    }
    void operator()(double x)const
    {
        std::cout<< "int : " << x << "\n";
    }
    void operator()(const std::string x)const
    {
        std::cout<< "int : " << x << "\n";
    }
};
```

```
struct IncrementVisitor
{
    void operator()(int& x)const
    {
        x++;
    }
    void operator()(double& x)const
    {
        x++;
    }
    void operator()(std::string& x)const
    {
        x += "x";
    }
};

int main()
{
    using namespace std;
    variant<int,double,string> vx("deneme");
    PrintVisitor pr;
    IncrementVisitor inc;
    //visit(pr, vx);
    visit(PrintVisitor{}, vx);
    visit(inc, vx);
    visit(pr, vx);
}
```

- Visitor sınıfı aşağıdaki gibi de yapılabilirdi

```
struct PrintVisitor
{
    template<typename T>
    void operator()(const T& x)const
    {
        std::cout<< "int : " << x << "\n";
        /*kodu ayrı yapmak için*/
        if constexpr(std::is_same_v<T,int>)
        {
            std::cout<< "int : " << x << "\n";
        }
        else if constexpr(std::is_same_v<T,double>)
        {
            std::cout<< "double : " << x << "\n";
        }
        else if constexpr(std::is_same_v<T,string>)
        {
            std::cout<< "string : " << x << "\n";
        }
    }
    //böyle yazmak yerine
```

```
    void operator()(const auto & x)const
    {
        std::cout<< "int : " << x << "\n";
    }
};
```

- Visiter için birden fazla sınıf yazmamız mümkün.
- Burada kullanılan sınıf functor sınıf olduğu için lambda ifadesi de kullanabiliriz.

```
int main()
{
    using namespace std;
    variant<int,double,string> vx("deneme");
    visit([](const auto& x) { std::cout << x << "\n"; }, vx);
}
```

09_23_07_2023

- Birden fazla variant nesnesi üzerinde işlem yapmak için vereceğimiz callable'un 2 argümanı olabilir.

```
struct DenVis
{
    template<typename T,typename U >
    void operator()(const T& x, const U& y)const
    {
        //std::cout << x << " " << y << "\n";
        std::cout << typeid(T).name() << " " << x << " " <<
typeid(U).name() << " "
        << y << "\n";
    }
    //bunun yerine C++20 ile gelen abbreviated function template
kullanılabilir.
    void operator()(const auto& x, const auto& y)const
    {
        std::cout << typeid(T).name() << " " << x << " " <<
typeid(U).name() << " "
        << y << "\n";
    }
};

struct Ayri
{
    void operator()(double,const std::string&)const
    {
        std::cout << "double string\n";
    }
    void operator(std::string, int)const
```

```
    {
        std::cout << "string int\n";
    }
    void operator()(double, int)const
    {
        std::cout << "double int\n";
    }
    void operator()(const auto&, const auto&)const
    {
        std::cout << "other -types\n";
    }
};

int main()
{
    using namespace std;
    variant<int,double,string> vx(3.4);
    variant<int,double,string> vy(12);
    visit(DenVis{}, vx, vy);
    auto fn = [](const auto& x, const auto& y) { std::cout << x << " " << y
<< "\n"; };
    visit(fn, vx, vy);//bu da olabilir.
    variant<double,string> vz(3.4);
    variant<string,int> vt("neco");
    visit(Ayri{}, vz, vt);
}
```

- Variantı kalıtımda base class olarak kullanabiliyoruz.

```
class Der :  public std::variant<int,double>
{}

int main()
{
    using namespace std;
    Derived d = 12;
    cout << d.index() << "\n"; // 0
    cout << get<0>(d) << "\n"; // 12
}
```

Overloader Idiom'u

- İlk önce inharitance konusunun tekrarı

```
struct A{void foo(int);};
struct B{void foo(double);};
//struct B : A{}; //struct B : public A{}; public yazılmasa da public
oluyor.
//class B : A{}; //class B : private A{}; private yazılmasa da public
```

```
oluyor.
struct Der : A, B{};
int main(){
    Der d;
    //d.foo(12);   Sentaks hatası ambiguity olur
}
```

Bunun sentaks hatası olmamasını istiyorsak sınıf içerisinde using bildirimi yapabiliriz.

```
struct A{void foo(int);};
struct B{void foo(double);};
//struct B : A{}; //struct B : public A{}; public yazılmasa da public
oluyor.
//class B : A{}; //class B : private A{}; private yazılmasa da public
oluyor.
struct Der : A, B{
    using A::foo;
    using B::foo;
};
int main(){
    Der d;
    d.foo(12); // A'nın foosu int
    d.foo(3.4) // B'nin foosu dobule çağırılıyor.
}
```

> Multiple inharitance ve diamond inheritance, virtual base class, konularının incelenmesi gerekiyor.

```
template<typename... Args> //birden fazla template argümanı kullanılabilir.
struct Overloader : Args... // pack expansion, bir ya da birden fazla taban
sınıftan elde eedebiliyoruz.
{
   using Args::operator()...;
};

struct A{

};
struct B{

};
struct C{

};

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

```
}
```

- Der kullanımında hepsi birbirinden farklı sınıflar,

```
template<typename... Args> //birden fazla template argümanı kullanılabilir.
struct Overloader : Args... // pack expansion, bir ya da birden fazla taban
sınıftan elde eedebiliyoruz.
{
   using Args::operator()...;
};

struct A{void foo(int);    void fa()};
struct B{void foo(double);    void fb()};
struct C{void foo(long);    void fc()};

int main()
{
   Der<A,B,C> y;
   Der<A,B>   x;
   Der<A>     z;
   y.fa();
   y.fb();
   y.fc();
   //y.foo(12); ambiguity oluşuyor,
}
```

- Eğer D'sınıfı içerisinde taban sınıfın ismi görünür kıllınsaydı bunu yapabiliyoruz ve artık modern c++ ile variadic olarak using bildirimi yapabiliyoruz.
- - `using Args::foo...;` kodunu sınıf içerisine ekleyerek kullanabiliyoruz.

```
struct A{ A(int);};
struct B{ B(double);};
struct C{ C(int)};
struct D : A,B,C
{};

int main()
{
    D dx = {35,2.3,31};
}
```

- Aşapuda decltype(fn) kullanımı C++20'ye kadar geçersizdi çünkü lambda class'ının default ctor'u yoktu ve bunun için lambda ifadesinin stateless lambda olması gerekiyor.

```
int main()
{
```

```
    auto fn = [](int x){return x+5;};
    //decltype(fn) x;
}
```

- Closure type bir class ve bunu base class olarak kullanabiliyoruz.

```
auto fn = [](int x){return x*4;};
auto fn1 = [](int x){return x*5;};
auto fn2 = [](int x){return x*6;};
auto fn3 = [](int x){return x*7;};
struct A : decltype(fn)
{
};

struct B : decltype(fn1), decltype(fn2), decltype(fn3)
{
};
struct C : delctype([](int x){return x+4;}), decltype(fn2)
{};
```

- Struct C de C++17'de sentaks hatası oluyor. C++20 ile lambda ifadelerini unevulated context olarak kullanılıyor. Kullanıldığı yerlerden biri olarak

```
int main()
{
    using namespace std;
    auto fn = [](int x, int y){return abs(x) < abs(y);};
    //set<int, decltype(fn)>myset; C++17'de bu geçersiz fn'yi argüman
olarak göndermemiz gerekiyor çünkü burada arka planda kod default
constructor ediliyor.
    set<int, decltype(fn)>myset(fn); //şeklinde olması gerekiyor.
    //c++20 de
    set<int, decltype(
         [](int x, int y)
         {return abs(x) < abs(y);}
    )> mmmset;
}
```

- Taban sınıf olarak template argümanı olarak lambdaları kullanabiliyoruz

```
template<typename ...Args>
struct Deneme : Args...
{
    using Args::operator()...;
};

int main()
```

```
{
    using namespace std;
    auto f1 = [](){};
    auto f2 = [](){};
    auto f3 = [](){};
    Deneme<decltype(f1),decltype(f2),decltype(f3)> d;
}
```

> Kalıtımda bir taban sınıfı birden fazla kez kullanamayız bu da f1,f2,f3'ün türlerinin aynı tür olmadığını kanıtlıyor.

- Aşağıdaki gibi Overload sınıfını tanımlayabiliriz. C++17'de sentaks hatası oluyor çünkü *deduction guide* gerekiyor. *deduction guide* C++17 ile dile eklendi.

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

};

int main()
{
    Overload x{
        [](int x){std::cout << "int : " << x << "\n";},
        [](int b){std::cout << "int b: " << b << "\n";},
    };
}
```

- Aşağıdaki kodda sentaks hatası var çünkü constructor parametresi referans olduğu T'türünün çıkarımı `int[10]`'a referans oluyor. Problem int türden bir diziyi int türden bir diziyle initialize etmek istiyoruz fakat böyle bir sentaks yok.

```
template<typename T, typename U>
struct Pair
{
    Pair(const T&, const U&) : mfirst(t),msecond(u){}
    //Pair(T,U); olursa burada kopyalanma maliyeti ortaya çıkarıyor
olaiblir.
private:
    T mfirst;
    U msecond;
};

int main()
{
    int a[10]{};
    double b[20]{};
    Pair p1{a,b}; // T : int[10] U : double[20]
}
```

- Fakat burada aşağıdaki gibi tanımlasaydık artık çıkarım int *ve double* olarak yapılıyor.

```
template<typename T, typename U>
struct Pair
{
    Pair( T t, U u) : mfirst(t),msecond(u){}
    //Pair(T,U); olursa burada kopyalanma maliyeti ortaya çıkarıyor
olaiblir.
private:
    T mfirst;
    U msecond;
};
int main()
{
    int a[10]{};
    double b[20]{};
    Pair p1{a,b}; // T : int[10] U : double[20]
}
```

- Bir önceki kodda: Bu sentaksa deduction guide deniyor, Eğer 2 tane argüman geçilirse burada pairin T U açılımı kullanılacak.

Deduction Guide özet

```
template<typename T, typename U>
struct Pair
{
    Pair(const T&, const U&) : mfirst(t),msecond(u){}
    //Pair(T,U); olursa burada kopyalanma maliyeti ortaya çıkarıyor
olaiblir.
private:
    T mfirst;
    U msecond;
};
template<typename T, typename U>
Pair(T,U) -> Pair<T,U>;

int main()
{
    int a[10]{};
    double b[20]{};
    Pair p1{a,b}; // T : int[10] U : double[20]
}
```

- Bu parametrelerin türü `int *` türüne referans ve `double *` türüne referans olarak çıkarım yapılıyor.

- Örneğin aşağıdaki kodda `const char *` için yapılacak çıkarımı `string` çıkarımı şekllinde yapılmasını sağlıyoruz.

```
template<typename T>
class Myclass
{
    Myclass(T);
};
Myclass(const char*) -> Myclass<std::string>;

int main()
{
    Myclass m1("neco");
}
```

```
#include <iostream>

int main()
{
    using namespace std;
    list <int> mylist {3,6,7,9,2,9};
    //vector myvec(mylist.begin(),mylist.end());
    //burada vector'un int açılımı tutuluyor. Burada bir deduction guide
kullanılıyor.
    return 0;
}

template<typename Iter>
//vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_types>
```

- Overload idiomu, Bu kod C++20 ile çalışıyor. C++176 ile çalışabilmesi için *deduction guide* gerekiyor. Overolad

```
#include <iostream>
template<typename ...Args>
struct Overload{
using Args::operator()...;
};
// template<typename ...Args>
//Overload(Args...)->Overload<Args...>
int main()
{
    using namespace std;
    variant<int,double,long,string>vx{"murat"};
    visit(Overload{
    [](int){cout<< "int\n";},
    [](double){cout<< "double\n";},
```

```
    [](long){cout<< "long\n";}
    [](string){cout<< "string\n";}
}, vx);
    return 0;
}
```

- Başka bir örnek

```
#include <iostream>

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

int main()
{
    using namespace std;
    variant<int,string>vx(00);
    visit(overload{
    [](int ival){cout <<"int "<< ival<< "\n";},
    [](const string & sval){cout <<"string "<< sval<< "\n";},
    },vx
);
    auto twice = overload{
    [](std::string &s){s+=s;},
    [](auto &i){i*=2;},
    };7
    visit(twice,vx);
    std::cout << get<0>(vx)<< "\n";
    return 0;
}
```

- Eski alternatif destroy edildikten sonra, yeni alternatif oluşurken eğer constructor exception throw ederse burada variant geçersiz bir duruma geliyor. Standartlarda bunun sınanması için bir fonksiyon var. Variant nesnesinin durumunu sınamadıkça bunu bilmek mümkün değil.

```
#include <iostream>

struct s
{
    operator int()const
    {
        throw std::runtime_error{"hata"};
        return 1;
    }
```

```
};

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";
        cout << (var.index() == variant_npos) << "\n";
    }
    return 0;
}
```

- Generic programlama için öğeler:

```
#include <iostream>

using vtype = std::variant<int,double,long>;

int main()
{
    using namespace std;
    constexpr auto n = variant_size<vtype>::value; // n'nin değeri 3,
variantın kaç
    variant_alternative<1,type>::type x{}; //x'in türü double
//alias template' da var
    variant_alternative_t<1,vtype>x {};
    return 0;
}
```

Kullanım Senaryoları

- Önceden belirlenmiş alternatiften birini tutcağını bildiğimiz tüm senaryolar için uygun.
- Hatalar için:
- - Fonksyion ya data'yı döndürecek ya da hata oluşursa hata türünü döndürücek.

```
struct Data{};
enum ErrorType{system,archive,log};

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

- Komut satırı argümanları parse ederken o türden bir değer işlenebilir.
- Fakat en önemli senaryolardan biri kalıtıma bir alternatif oluşturması ve run-time polymorphism sağlaması. Virtual-dispatch'in birden fazla maliyeti var. Derleyici run-time polymorphism'de her sınıf nesnesi için vpointer oluşturuyor. Hem run-time da hem de bellek kullanımı açarısında maliyeti var. Hiyerarşideki tüm nesneleri için de vpointer oluyor.
- - Ayrıca run-time başında bu tabloların oluşturulması gerekiyor.
- - Asıl problem virtual-dispatch'te her polymorphic nesne için allocation-deallocation var. Dinamik ömürlü nesneye bağlı olarak yüksek işlem maliyeti oluşturuyor.
- - Ayrıca sınıflar birbirine bağımlı hale geliyor.
- *closed-hierarcy* daha önceden belirlenmiş ise, bu durumda varianttan yaralanabiliriz.

```
class Document{void print() = 0};
class Xls{};
class Pdf{};
class Txt{};
class Word{};

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

```
#include <iostream>
#include <variant>
#include <string>
#include <list>
#include <algorithm>
class Cat{
public:
    Cat(std::string name) : mname(name){}
    void meow()const
    {
        std::cout << mname << " miyavladı\n";
    }
private:
    std::string mname;
};
class Dog{
public:
    Dog(std::string name) : mname(name){}
    void bark()const
    {
        std::cout << mname << " havladı\n";
    }
private:
    std::string mname;
};

class Lamb{
public:
```

```
    Lamb(std::string name) : mname(name){}
    void bleat()const
    {
        std::cout << mname << " mırladı\n";
    }
private:
    std::string mname;
};
using Animal = std::variant<Dog, Cat, Lamb>;

struct AnimalVoice
{
    void operator()(const Dog &d)const
    {
        d.bark();
    }
    void operator()(const Cat &c)const
    {
        c.meow();
    }
    void operator()(const Lamb &l)const
    {
        l.bleat();
    }
};
template<typename T>
bool is_type(const Animal&s)
{
    return std::holds_alternative<T>(s);
}

int main()
{
    using namespace std;
    list<Animal> animals;
    animals.push_back(Dog("Kara"));
    animals.push_back(Cat("Pamuk"));
    animals.push_back(Lamb("Kuzucuk"));
    for(const auto &a : animals)
    {
        visit(AnimalVoice{}, a);
    }

    for(const Animal &a: animals)
    {
        switch(a.index())
        {
            case 0: get<Dog>(a).bark(); break;
            case 1: get<Cat>(a).meow(); break;
            case 2: get<Lamb>(a).bleat(); break;
        }
    }
    cout <<"----------------\n";
    for(const Animal &a: animals)
```

```
    {
        if(auto p = get_if<Dog>(&a))
        {
            p->bark();
        }
        else if(auto p = get_if<Cat>(&a))
        {
            p->meow();
        }
        else if(auto p = get_if<Lamb>(&a))
        {
            p->bleat();
        }
    }
    cout <<"----------------\n";
    for(const Animal &a: animals)
    {
        visit(AnimalVoice{}, a);
    }
    cout <<"----------------\n";
    return 0;
}
```

kodun calışanı

std::any

- Bir sınıf şablonu değil ve any türünden bir nesne herhangi bir türe dönüştürülebiliyor. Başlık dosyası `<any>`.
- Doğrudan belirli bir türlü hayata başlatabiliyoruz.
- Böylece void pointer'a bir alternatif oluşturulmuş oluyor. `Void *` tür bilgisi bilmiyorken, any bu bilgiye sahip.
- any türü storage'i küçük nesneleri kendi içerisindeki buffer'da tutabiliyor fakat burada kendi içinde tutmuyor ve dinamik bir bellek alanı elde edebiliyor.

```
#include <any>
int main()
{
    any x1;
    any x2{};
    boolalpha(cout);
    cout << x1.has_value() << "\n"; // false
    cout << x2.has_value() << "\n"; // false
    any a3 = 12;
    any a4{"deneme"};
    any a5{"deneme"s};
    any a{vector<int>{2,5,7,9,12,56}};
}
```

> type_id operatörünün bilinmesi gerekiyor.

- Çünkü any sınıfı içerisinde ayrıca bir typeinfo sınıfı tutuyor.

typeid ve typeinfo sınıfı

- type_info isimli bir sınıf var ve bu sınıfı kullanmanın tek yolu type_id operatörü.
- Var olan type_id referansı ile erişilen sınıfın üye fonksiyonları var.

```
int main()
{
    //auto x = typeid(int); // type-info sınıfının copy ctor'u delete
edilmiş, bu yüzden sentaks hatası oluyor.
    //type_info &r = typeid(std::string); const referans olmadığı için
sentaks hatası oluyor.
    const auto &r = typeid(std::string);
    typeid(std::string) == typeid(x); //r.operator== //ile iki nesneinin
aynı tür olup olmadığını sınayabiliriz.

}
```

any sınıfı devam

```
void *operator new(std::size_t sz)
{
    std::cout << "new called"<< sz<<  "\n";
    if(sz == 0)
    {
        ++sz;
    }
    if(void *p = std::malloc(sz))
    {
        return p;
    }
    throw std::bad_alloc{};
}
struct Den
{
    char buf[100];
};
int main()
{
    std::cout << "sizeof(std::any) = " << sizeof(std::any) << "\n";
    any a = 356; //int içinde buffer da tutabilir
}
```

- Any türünden bir nesnenin farklı türden bir değer tutması sağlanabilir.

```
int main()
{
    using namespace std;
    any a = 12;
    a ="deneme"s;
    a = "char";
}
```

- in_place_type ile tuttuğu bellek alanında tuttuğu türden bir nesne oluşturulabilir. Bunu doğrudan kullanmak istemiyorsak `make_any` fonksiyonu kullanılabilir.

```
int main()
{
    any a{in_place_type<Date>5,5,1955};
    //auto a = make_any<Date>(5,5,1955);
    cout << any_cast<Date>(a) << "\n";
}
```

Tuttuğu Türe erişme

- sınıfın bir type fonksiyonu var ve bu fonksiyon `const type_info &` dönüyor.

```
int main()
{
    using namespace std;
    any a = 12;
    // any a = 3.5;
    // any a = "neco"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";
}
```

- Eğer any nesnesi boş ise type fonksiyonu `void` döndürüyor.

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

- Burada gene decay oluyor.

```
int main()
{
    int ar[10]{};
    any a = ar;
    cout<< boolalpha << (a.type() == typeid(int *)) << "\n";
    cout<< boolalpha << (a.type() == typeid(int [10])) << "\n";
}
```

Tuttuğu değere erişme

- `any_cast<tür>` ile erişiyoruz ve eğer tuttuğu türden başka bir türe erişmeye çalışırsak `std::bad_cast` exception throw ediyor.

```
int main()
{
    using namespace std;
    any a = 12;
    cout << any_cast<int>(a);
    try{
        cout << any_cast<double>(a);
    }
    //catch(const std::bad_any_cast &ex)
    catch(const std::bad_cast &ex)
    {
        cout << "exception caught : " << ex.what() << "\n";
    }
}
```

Any Üye Fonksiyonları

- reset fonksiyonu ile boşaltabiliyoruz.
- emplace fonksiyonu var.

```
int main()
{
    using namespace std;
    any a = make_any<string>(20,'T');
    cout << boolalpha << a.has_value() << "\n";
    a.reset();
    cout << a.has_value() << "\n";
    a.emplace<int>(12);
    cout << any_cast<int>(a) << "\n";
    cout << a.has_value() << "\n";
}
```

- Move semantiğinden faydalanabiliriz.

```
int mian()
{
    any a{"Deneme"s};
    auto &ra = any_cast<string&>(a);
    ra[0] = 'T';
    //ra.at(0) = 'T';
    //ra.front() = 'T';
    std::cout << any_cast<string const&> (a) << "\n";
    auto str = any_cast<string&&>(move(a));
    static_assert(is_same_v<decltype(str), string>);
    //any_cast<const string&>(a).size();
    std::cout <<"a.size() "<< any_cast<string>(&a)->size() << "\n";
    cout << str << "\n";
}
```

- any_cast'i `get_if` gibi kullanabiliriz.

```
int main()
{
    any a = 12;
    if(any_cast<int>(&a))
    {
        std::cout << "int\n";
    }
    else if(auto ptr = any_cast<double>(&a))
    {
        std::cout << "double: " << *ptr << "\n";
    }
    else if(any_cast<string>(&a))
    {
        std::cout << "string\n";
    }
}
```

- Container ile kullanabiliriz.

```
int main()
{
    using namespace std;
    vector<any> vx{1,3,"deneme"s,34L,"alican"};
    for(const auto &a : vx)
    {
        if(auto p = any_cast<int>(&a))
        {
            cout << *p << "\n";
```

```
        }
        else if(auto p = any_cast<string>(&a))
        {
            cout << *p << "\n";
        }
        else if(auto p = any_cast<long>(&a))
        {
            cout << *p << "\n";
        }
    }
}
```

- direkt any olarak tutmak yerine pair-any gibi tutabiliriz.

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

int main()
{
    using namespace std;
    vector<tv_pair> vx;
    vx.push_back({"int"s, 12});
    vx.push_back({"double"s, 3.4});
    vx.push_back({"string"s, "neco"s});
    vx.push_back({"long"s, 12L});
    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";
        }
        else if(value.type() == typeid(long))
        {
            cout << setw(16) << property << " : " << any_cast<long>(value)
<< "\n";
        }
    }
}
```