* Optional - Erase()

```
eray
eray goksu
eray gok

C:\nec\a.exe (process 28480) exited with code 0.
Press any key to close this window . . .
```

```cpp
using namespace std;

optional<string> x{ "eray" };
cout << x.value() << '\n';

*x += " goksu";

cout << x.value() << '\n';

x.value().erase(_Off: 8);

cout << x.value() << '\n';
}
```

* Value_or():

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

    optional<string> x { "eray" };

    cout << x.value_or(_Right: "noname") << "\n";

    x = nullopt;

    cout << x.value_or(_Right: "noname") << "\n";

    x.value_or() = |
}
```

- `x = nullopt;` → null yöntemi
- `cout << x.value_or(...)` → value olmadığında yerine yazılanı dönderir.
- `x.value_or() =` → L value olmadığı için, "=" operatörü sentax hatası!

```cpp
#include <optional>
#include <string>
#include <iostream>

void display_e_mail(const std::optional<std::string>& op)
{
    std::cout << "e posta adresi : " << op.value_or("belirtilmemis") << "\n";
}

int main()
{
    std::optional<std::string> e_mail_address{ "necati@gmail.com" };

    display_e_mail(e_mail_address);
    e_mail_address = std::nullopt;
    e_mail_address = {};
    display_e_mail(e_mail_address);
    //...
}
```

```cpp
#include <optional>
#include <string>
#include <iostream>

// aldigi string'i int degere donusturecek - inte donusturemez ise deger dondurmeyecek

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

std::optional<int> to_int2(const std::string& s)
{
    std::optional<int> ret; // deger tutmuyor
    try {
        ret = std::stoi(s);
    }
    catch (...) {}

    return ret;
}
```

Her biri ayrı ayrı null döndürür
Farklı null döndürme yöntemleri

```cpp
int main()
{
    for (auto s : { "42", " 077", "necati", "0x33" }) {
        // ogeleri int'e donusturup cikis akimina yazdiriyoruz:
        std::optional<int> op = to_int(s);
        if (op) {
            std::cout << s << " yazisi int'e donusturuldu... deger : " << *op << "\n";
        }
        else {
            std::cout << "(" << s << ") yazisi int'e donusturulemiyor\n";
        }
    }
}
```

initializer list

* Value: → Nedenini yazdığı medium sayfasından!

value işlevi

Tutulan nesneye erişmenin bir başka yolu da sınıfın *value* isimli üye işlevini çağırmak. *operator ** işlevi gibi *value* işlevi de tutulan nesneye referans döndürüyor. Boş bir *optional* nesnesi için *value* işlevinin çağrılması durumunda, *std::exception* sınıfından kalıtım yoluyla elde edilen *std::bad_optional_access* türünden bir hata nesnesi gönderiliyor:

```cpp
#include <optional>
#include <string>
#include <iostream>

int main()
{
    std::optional<std::string> op{ "oguz karan" };

    std::cout << op.value() << "\n";
    op.value().assign(5, 'A');
    std::cout << op.value() << "\n";
    op = std::nullopt;

    try {
        std::cout << op.value() << "\n";
    }
    catch (const std::bad_optional_access &ex) {
        std::cout << "hata yakalandi : " << ex.what();
    }
}
```

*Emplace ve Reset:

```cpp
#include <optional>
#include <iostream>
#include <string>

class UserName
{
public:
        explicit UserName(const std::string& str) : m_name(str)
        {
                std::cout << "constructor : " << m_name << "\n";
        }

        ~UserName()
        {
                std::cout << "destructor : " << m_name << "\n";
        }

private:
        std::string m_name;
};

int main()
{
        std::optional<UserName> op_name;

        op_name.emplace("Necati Ergin");

        op_name.emplace("Ali Serce"); //destructor cagrilacak
        op_name.reset(); // destructor cagrilacak
        op_name.emplace("Kaan Aslan"); //destructor cagrilacak
        op_name = std::nullopt;
        op_name.emplace("Oguz Karan");
        op_name = UserName("Nuri Yilmaz");
}
```

*Karşılaştırma İşlemleri:

```cpp
using namespace std;

int main()
{
    optional<bool> oe{ nullopt };
    optional<bool> ox{ false };
    optional<bool> oy{ true };


    cout.setf(ios::boolalpha);

    cout << (oe == ox) << '\n';  //false
    cout << (oe == oy) << '\n';  //false
    cout << (oe < ox) << '\n';  //true
    cout << (oe < oy) << '\n';  //true
    cout << (oe == true) << '\n';  //false
    cout << (oe == false) << '\n';  //false
    cout << (ox == oy) << '\n';  //false
    cout << (ox < oy) << '\n';  //true
}
```

* Find' Find_if

```cpp
template<typename Con, typename Pred>
auto find_if(Con&& c, 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& tval)
{
    return find_if(std::forward<Con>(c),
        [&tval](auto&& x) {return x == tval; });
}
```

<T> Provide sample template arguments for IntelliSense

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

    vector<int> ivec{ 1, 4, 7, 10, 2 };

    cout << **find_if(ivec, [](int x) {return x % 5 == 0; }) << "\n";

}
```

In_place:

```cpp
int main()
{
    std::optional<Date> op1{ std::in_place, 12, 5, 1987};
```

- optional'da argüman direk kullanılan tür, çağırılacak constructor'ı eğer birden fazla argüman alacaksa

* Bu şekilde in_place kullanmak zorunda

```cpp
int main()
{
    using namespace std::string_literals;
    std::optional<std::complex<double>> op1{ std::in_place, 1.2, 5.6 };
```

Alternatif: make_optional

```cpp
using namespace std;

std::optional<Date> op1{ std::in_place, 12, 5, 1987};

auto opd = make_optional<Date>(12, 5, 1987);
```

```cpp
using namespace std;

auto x1 = make_optional(12);
auto x2 = make_optional("ali");
auto x3 = make_optional("ali"s);
auto x4 = make_optional<string>("ali");
```

→ make_optional bir template. O yüzden itiler CTAD, itiler implicit declaration ile tanımlanabilir.

* Optional and move:

```cpp
#include <optional>
#include <iostream>

struct A {
    A() { std::cout << "default ctor()\n"; }
    A(const A&) = delete;
    A& operator=(const A&) = delete;
    A(A&&) { std::cout << "move ctor()\n"; }
    //...
};

int main()
{
    std::optional<A> op1;
    op1.emplace(); //default ctor
    std::optional<A> op2{ std::move(op1) }; //A's move ctor.
    std::cout << std::boolalpha;
    std::cout << op1.has_value() << "\n";
    std::cout << op2.has_value() << "\n";
}
```

```cpp
A ax;

std::optional<A> op1{ std::move(ax) };
std::optional<A> op2{ std::move(*op1) };
```

* Std:: Variant:

- Önceden belirlenen türde veriler tutabilen bir nesne (union)
- * Arka planda union gibi olsa da daha güvenli
- Class template'dir

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

    variant<Date, double, long> x;

}
```

variant alternative

Tuttuğu türü öğrenmek için:

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

    variant<Date, double, long> x;

    cout << x.index() << '\n';
}
```

- Default construct edilmiş tüm variant nesneleri 0. index değerinin türündedir.

```cpp
#include <optional>
#include <iostream>
#include <string>
#include <variant>
#include "date.h"


int main()
{
    using namespace std;

    variant<long, int, string> x;

    try {
        cout << get<1>(x) << "\n";
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
    }
```

- x; → default init ettiğimiz için erişilecek tür long
- get<1> → bad variant access türünden exception throw edecek!
- get<1> → set değer döndürür
- → Aktif olmayana erişim

- → get'in return value'su referans
- → get ile referans init edilebilir.
- → get<type>(x) de olabilir. type alımı da ver.

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

    variant<long, int, string> v{ 12 };

    cout << get<1>(v) << '\n';
```

- → Türün int olduğunu bildi.
- cout << get<1>(v) << '\n'; ⇒ 12

```cpp
    using namespace std;

    variant<long, int, string> v{ 12 };
    cout << "index = " << v.index() << '\n';

    cout << get<1>(v) << '\n';

    get<1>(v) = 99;

    cout << get<1>(v) << '\n';
    ++get<1>(v);
    cout << get<1>(v) << '\n';

    v = 23L;

    cout << "index = " << v.index() << '\n';

}
```

- ++get<1>(v); → Aritmetik işlemler yapılabilir.
- v = 23L; → yeniden atama yapılarak, variant'in türü değiştirilir!

```cpp
class NoDefault {
public:
    NoDefault(int);
};


int main()
{
    using namespace std;

    variant<NoDefault, int, string> v;


}
```

default int edilemeyeği için syntax hatası

Githubdaki kodları inceliyor.

* get_if:

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

    variant<int, string, Date> v{ "necati ergin"s };

    auto p = get_if<string>(&v);


}
```

p'nin türü string*

=> get_if, eğer template argümanı aktif durumdaysa, o nesneye pointer döndürür. değilse nullptr döndürür.

```cpp
#include <variant>
#include <iostream>

class Data {
public:
    Data(int x) : mx{ x } {}
private:
    int mx;
};

int main()
{
    using namespace std;

    variant<monostate, Data, int, double> v2; //valid


    v2 = Data{ 13 };
    cout << "index = " << v2.index() << '\n';
    v2 = 23;
    cout << "index = " << v2.index() << '\n';
    v2 = 4.5;
    cout << "index = " << v2.index() << '\n';
    v2 = std::monostate{};
    cout << "index = " << v2.index() << "\n";
    v2 = {};
    cout << "index = " << v2.index() << "\n";
}
```

variantın boş olduğu hali