

C++ KURS İÇERİĞİ

- 2023.07.24
 - 1. C in C++
- 2023.07.26
 - 1. Const
 - top level const
 - low level const
 - 2. C in C++
 - 3. Initialization
- 2023.07.28
 - 1. Reference Semantics
 - L value reference
 - 2. Value Category
- 2023.07.31
 - 1. Reference Semantics
 - R value reference
 - 2. Type Deduction
 - Auto type deduction
 - 3. Reference Collapsing
- 2023.08.02
 - 1. Reference Collapsing
 - 2. Universal Reference
 - 3. Type Deduction
 - Decltype specifier
 - 4. Unevaluated Context
 - 5. Function Default Argument
 - 6. Constexpr
- 2023.08.04
 - 1. Constexpr
 - 2. Constexpr functions
 - 3. ODR (One Definition Rule)
 - 4. Inline Expansion
 - 5. Enumeration Types
- 2023.08.07
 - 1. Scoped Enum
 - 2. Type-Cast Operators
 - static_cast
 - const_cast
 - reinterpret_cast
 - dynamic_cast
- 2023.08.09
 - 1. Function Overloading

- 2023.08.11
 - 1. Classes
- 2023.08.14
 - 1. Classes
 - This keyword
 - Mutable
- 2023.08.16
 - 1. Classes
 - Constructor
 - Destructor
 - Special Member Functions
 - Global Class Objects
 - Constructor Initializer List
- 2023.08.18
 - 1. Classes
 - Delete bildirimi
 - Class Special Functions
 - Aggregate Class
 - Copy Constructor
- 2023.08.21
 - 1. Classes
 - Copy Constructor
 - Copy Assignment
 - Move Constructor
- 2023.08.23
 - 1. Class special member functions
 - 2. temporary objects C++17
 - 3. explicit ctor
 - 4. conversion ctor
 - 5. moved-from state (taşınmış nesne durumu)
- 2023.08.25
 - 1. copy elision
 - 2. C++17 (mandatory copy elision)
 - 3. Return Value Optimization RVO
 - 4. NRVO (Named Return Value Optimization)
 - 5. new ifadeleri
 - 6. delete ifadeleri
 - 7. operator new fonksiyonları
 - 8. operator delete fonksiyonları
 - 9. sınıfların static veri elemanları
- 2023.08.28
 - 1. Sınıfın static data member
 - Incomplete type
 - Header-Only Library
 - 2. Sınıfı static member functions
 - Named Ctor
 - Singleton Pattern
 - Mayer's Singleton

- 2023.09.01
 1. Delegating Ctor
 2. Friend Bildirimleri
 3. Operator Overloading
- 2023.09.04
 1. Global Operator Function vs Member Operator Function
 2. Operator Overloading'te Function Overloading
 3. Neden member operator function ve global operator function ayrı ayrı var?
 4. Const Correctness & Operator Functions
 5. Operator Overloading ile Value Category İlişkisi
 6. Arrow Operator Overloading Fonksiyon
- 2023.09.06
 1. Operator overload ++ ve -
 2. Operator overload + ve -
 3. Operator []
- 2023.09.08
 1. Arrow operator overloading
 2. Overloading function call operator ()
 3. Tür dönüşüm operatörleri
 4. Conversion Ctor vs Operator Conversion
 5. Operator bool
- 2023.09.11
 1. extern "C" bildirimi
 2. Composition (has a relation-ship)
 3. Copy Ctor && Move ctor && Copy Assignment && Move Assignment
 4. Reference Qualifiers
- 2023.09.13
 1. Namespace (isim alanları)
 2. Using declaration
 3. using namespace directive
 4. ADL (argument dependent lookup) argümana bağlı isim arama
- 2023.09.15
 1. Operator Overloading Enum
 2. Nested classes
 3. Design Pattern (pimpl idiom)
 4. Inheritance (Kalıtım)
- 2023.09.18
 1. Inheritance (Kalıtım)
 2. Kalıtım ve İsim Arama
 3. Kalıtımda Using Bildirimi
 4. Kalıtımda special fonksiyonların durumları
- 2023.09.20
 1. dynamic binding or late binding
 2. Non virtual Interface (NVI)
 3. Override Keyword
 4. Virtual Functions -Multi Level Inheritance
 5. Default Arguman in Virtual Functions
 6. Overloading Functions – Inheritance
 7. Virtual Ctor C++ (Clone idiom)

- 2023.09.22
 - 1. Virtual DTOR
 - 2. Virtual Function Table Pointer (VPTR)
 - 3. Inherited Ctor
 - 4. Covariance or Variant Return Type
 - 5. NVI (non-virtual interface)
 - 6. Final Contextual Keyword
 - Final Class
 - Final Override
- 2023.09.25
 - 1. Private Inheritance
 - 2. Restricted Polymorphism
 - 3. Protected Inheritance (multi-level inheritance daha sık kullanılır)
 - 4. Multiple Inheritance
 - 5. Diamond Formation
 - 6. RTTI (Run Time Type Information)
- 2023.09.27
 - 1. RTTI (Run Time Type Information)
 - Dynamic Cast
 - typeid
 - Typeinfo
 - 2. Template Method
- 2023.09.29
 - 1. std::initializer_list
 - 2. std::string
 - 3. std::string CTOR
 - 4. Small String Optimizasyon (SSO)
- 2023.10.02
 - 1. std::string elemanlara erişme
 - 2. Range-Based For Loop
 - 3. std::string atama işlemleri işlemleri
 - 4. std::string elemanları silme işlemleri
 - 5. std::string yazı ekleme işlemleri
 - 6. std::string arama işlemleri
 - 7. std::string yazıyı değiştirme işlemleri
 - 8. std::string karşılaştırma işlemleri
- 2023.10.11
 - 1. std::string_view
 - 2. std::string tür dönüşümleri
 - 3. Exception Handling
- 2023.10.13
 - 1. Exception Handling
 - 2. Stack Unwinding
 - 3. Exception Rethrow
 - 4. Exception Handling in CTOR
 - 5. Exception Guarantee(s)
- 2023.10.16
 - 1. noexcept
 - noexcept specifier
 - noexcept operator
 - 2. Generic Programlama (Templates)

- 2023.10.18
 - 1. Template Argument Deduction
 - 2. Universal Reference (Forwarding Reference)
 - 3. Trailing Return Type
 - 4. Auto Return Type
 - 5. Overloading in Template Functions
- 2023.10.20
 - 1. Template Specialization
 - 2. Template Fonksiyon tanımlama
 - 3. Non-Type Parameter Template
 - 4. Default Template Arguman
 - 5. Explicit Specialization
- 2023.10.23
 - 1. Explicit Specialization Template
 - 2. Partial Specialization Template
 - 3. Alias Template
 - 4. Variadic Template
 - 5. Recursive Instantiation
 - 6. Fold Expressions (katlama ifadeleri)
- 2023.10.25
 - 1. Template Value Type
 - 2. Static if
 - 3. Standard Template Library
 - Iterators
 - Iterator Category
- 2023.10.27
 - 1. Algorithms
 - Std::copy
 - Std::copy_if
 - Std::count_if
 - Std::find
 - 2. Iterators
 - Const iterator (const_iterator)
 - Reverse_iterator
- 2023.10.30
 - 1. Reverse Iterator
 - 2. Std::back_inserter
 - 3. Std::front_inserter
 - 4. Std::advance
 - 5. Std::distance
 - 6. Std::next
 - 7. Std::prev
 - 8. Std::iter_swap
 - 9. Algorithms
 - Std::find_if
 - Std::transform

- 2023.11.01
 - 1. Lambda expression
 - Mutable lambda expression
 - Capture all by reference
 - Positive lambda expression
 - Noexcept lambda expression
 - Constexpr lambda expression
- 2023.11.03
 - 1. Algorithms
 - Std::for_each
 - Std::any_of
 - Std::all_of
 - Std::none_of
 - Std::replace
 - Std::reverse_copy
 - Std::remove_copy_if
 - Std::replace_copy
 - Std::replace_copy_if
 - 2. Container
 - Std::vector
- 2023.11.06
 - 1. Container
 - Std::vector
 - Functions of std::vector
- 2023.11.08
 - 1. Container
 - Std::vector
 - Functions of std::vector
 - 2. Algorithms
 - Std::remove
 - Std::remove_if
 - Std::unique
 - Std::erase
 - Std::erase_if
 - Erase-remove idiom
 - Std::sort
 - Std::transform
 - 3. Std::ostream_iterator
- 2023.11.10
 - 1. Container
 - Std::deque
 - 2. Algorithms of sorting
 - Std::sort
 - Std::partial_sort
 - Std::stable_sort
 - Std::nth_element
 - Std::partition

- 2023.11.13
 - 1. Algorithms
 - Std::minmax_element
 - Std::partition_point
 - Std::is_sorted
 - Std::is_sorted_until
 - 2. Heap Data Structer
 - 3. Std::queue
 - 4. Std::list
 - 5. Std::lower_bound
 - 6. Std::upper_bound
 - 7. Std::equal_range
- 2023.11.15
 - 1. Std::forward_list
 - 2. Container Adapters
 - Stack
 - Queue
 - Priority_queue
 - 3. Associative Container
 - Set
- 2023.11.17
 - 1. Associative Container
 - Std::set
 - Std::map
 - Std::unordered_set
- 2023.11.20
 - 1. Associative Container
 - Std::unordered_set
 - 2. Reference Wrapper
 - 3. Algorithms
 - Std::generate
 - 4. Function Adaptor
 - Std::bind
- 2023.11.22
 - 1. Function Adaptor
 - Std::bind
 - Std::function
 - Std::mem_fn
 - Std::not_fn
 - 2. Std::array
- 2023.11.24
 - 1. Std::tuple
 - 2. Structed binding
 - 3. Std::tie
 - 4. Std::apply
 - 5. Std::invoke
 - 6. Member Function Pointers

- **2023.11.29**
 - 1. Member Function Pointers
 - 2. Data Member Pointers
 - 3. Std::bitset
 - 4. Dinamik Ömürlü Nesneler
- **2023.12.01**
 - 1. Dinamik Ömürlü Nesneler
 - 2. Smart Pointer
 - Unique Pointer
- **2023.12.04**
 - 1. Smart Pointer
 - Unique Pointer
 - Shared Pointer
- **2023.12.06**
 - 1. Smart Pointer
 - Weak Pointer
 - 2. Curiously Recurring Template Pattern
- **2023.12.08**
 - 1. Input Output Operations
- **2023.12.11**
 - 1. Format
 - On off flag
 - Area flag
 - 2. Std::istreamstream
 - 3. iosstate
- **2023.12.13**
 - 1. std::istream_iterator
 - 2. file operations
- **2023.12.15**
 - 1. File Operations
 - File positions pointer
 - gcount()
- **2023.12.20**
 - 1. Tie
 - 2. Lambda expression
 - 3. Std::ratio
 - 4. Std::chrono
 - duration
- **2023.12.22**
 - 1. Std::chrono
 - Duration
 - Duration_cast
 - System_clock
 - 2. UDL (user-defined literals)

- 2023.12.25
 - 1. Std::chrono
 - System_clock
 - 2. Vocabulary Type
 - Std::optional
 - Value_or()
 - in_place()
 - std::variant
 - hold_alternative
 - get<>
 - in_place_index and in_place_type
 - monostate
 - get_if
- 2023.12.27
 - 1. Vocabulary Type
 - Std::variant
 - Std::any
 - 2. Std::visit
 - 3. Multi-Lambda
- 2023.12.29
 - 1. Std::any
 - 2. Std::random
- 2024.01.03
 - 1. Std::random
 - 2. Concurrency
 - Std::thread
- 2024.01.05
 - 1. Std::thread
 - Get_id
 - Sleep_for and sleep until
 - Hardware_concurrency
 - Thread exceptions
 - 2. Data Race
- 2024.01.08
 - 1. Std::thread
 - 2. Mutex Class
 - 3. Mutex Wrapper
- 2024.01.10
 - 1. Mutex
 - 2. Std::future and std::promise
 - 3. Std::async

2023 07 24

C in C++

```
/*
    undefined behavior
    unspecified behavior
    implementation defined
*/
```

```
/*
    implicit int
    C'de geri dönüş değeri int kabul edilecek. C++'da syntax hatası
    NOT: C'de de geçerliliğini yetirdi.
    func(int x)
    {
        return x + 5;
    }

*/
```

```
/*
    C -old-style function definitions

    func(a, b, c)
    double a, b, c;
    {
        return a + b + c;
    }

*/
```

```
// implicit function declaration
int main()
{
    func(1, 2, 3); // tanımlanmasa bile C'de geçerli C++ geçersiz
}
```

```
int foo(); // parametre değişkeni hakkında bilgi vermiyoruz
int bar(void); // parametre değişkeni yok

int main()
{
    // C'de
    foo(21, 56, 78); // C'de geçerli C++'da geçersiz
    bar(21, 31, 12); // C ve C++ geçersiz
}
```

```
int foo()
{
    // return olmaması C'de geçerli C++'da geçersiz
    printf("emre");
}
```

```
/*
parametre değişkeleri (formal parametre)
void func(int x, int y)

argument (actual parametre)
func(a, b)

*/
```

```
/*
C de karakter sabitleri int C++'da char türündedir
'A'
'\n'

10 > 5 bu ifadenin türü C'de int C++'da bool

*/

int main()
{
    char c1 = 10;
    char c2 = 20;

    c1 + c2 // int
}
```

2023 07 26

Const

```
int main(void)
{
    // Cpp'da geçersiz init etmemiz gereklidir
    const int x; // C'de geçerlidir
}
```

```
int foo();

int main(void)
{
    const int x = 10;
    int a[x]; // C'de geçersiz C++'da geçerlidir

    switch (foo())
    {
        case x; // C'de geçersiz Cpp'de geçerlidir
    }
}
```

```
// C'de
int x = 10; // external linkage
static y = 10; // internal linkage
int main()
{

}

// Cpp'de
const int x = 10; // internal linkage
extern const int x = 10; // external linkage
```

```
// top level const
int main()
{
    int x = 10;
    // const pointer to int
    // top level const
    // right const
    int * const p = &x; // burda const olan p değişkenin kendisi

    int y = 34;
    /*
    top level const olarak tanımlanmış variable'in
    adresi değişmez
    */
    p = &y; // geçersiz

    *p = 34; // geçerlidir
}
```

const int * p ile int const *p arasında fark yok !!! Const'un yıldızdan önce ile sonra olmasının arasında farkı var

```
// pointer to const int
// Low Level const
int main()
{
    int x = 10;
    int y = 34;

    const int* p = &x;
    p = &y; // geçerli
    *p = 12; // geçersiz
}
```

```
// SORU
typedef int* IPTR; // top Level const
typedef const int* CIPTR; // Low Level const
int main()
{
    int x = 5;
    const IPTR p = &x; // int *const p = &x

    int y = 12;
    p = &y; // syntax hatalı
    *p = 56; // geçerli

    CIPTR p1 = &y; // geçerli
    CIPTR p1 = 32; // geçersiz
}
```

```
// const pointer to const int
int main()
{
    int x = 10;
    // *ptr ve ptr'ye atama yapmak syntax hatalı
    const int* const ptr = &x;
}
```

```
// array decay
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    //int *p = &a; // syntax hatalı

    int *p = a; // array decay
    auto *ptr = &a[0];
}
```

```
int main()
{
    // C'de geçerli Cpp'de geçersiz
    char *p = "emre"; // char[5]
    /*
        Cpp'da "emre" const char* decay olur ve
        const char* dan char* dönüşüm yok
    */
    const char *p = "emre"; // Cpp de geçerli
}
```

str karakter dizisi bir dizi olarak bellekte yer kaplar ve üzerinde değişiklik yapılabılırken, p karakter dizisi işaretçisi, "emre" ifadesini gösteren bir adresi tutar ve bu ifade bir sabit karakter dizisi olduğu için üzerinde değişiklik yapmak uygun değildir.

```
int main()
{
    char str[] = "emre";
    char *p = "emre"; // static ömürlü
}
```

```
struct Data{
    // empty struct C'de geçersiz Cpp'de geçerli
};
```

```
struct Data {
    int a, b, c;
};

int main()
{
    struct Data mydata; // C'de struct ile tanımlamam gerek
}
```

```
int nec = 0;
int main()
{
    struct nec {
        char str[64];
    };
    // C'de çıktı 4 çünkü struct ile tanımlamadık int'i aldı
    // Cpp'de çıktı 64 olur
    printf("sizeof(nec) = %zu\n" sizeof(nec));
}
```

Initialization

```
int main()
{
    int x; // default init
    int y = 10; // copy init

    int z(10); // direct init
    // küme parantezi init'leri genel adı uniform init
    int k{ 10 }; //direct list init
    int v{}; // value init

    int m(); // fonksiyon bildirimidir

    ++m; //syntax hatalı
}
```

```
// neden uniform initialization
/*
1. uniform olması
2. narrowing conversion
    veri kaybına neden olan dönüşümlere denir.
    örnek: (float - int) (int x = 3.4)

3. most vexing parse
    değişken tanımlamasıyla fonksiyon bildirimlerinin karışması

*/
```

```
// narrowing conversion
int main()
{
    //narrowing conversion
    int x1 = 3.4; // Legal
    int x2(3.4); // Legal

    int x3{ 3.4 }; // syntax hatalı
}
```

```
//most vexing parse

class A{

};

class B{
public:
    B(A);
};

int main()
{
    // most vexing parse
    B bx(A()); // fonksiyon bildirimi
    B bx{ A{} }; // değişken bildirimi

    /*
        B bx(A()); böyle yazmak ile B bx(A(*)()); böyle yazmak aynı şey
        decay oluyor
    */
}
```

C ve Cpp'deki fonksiyon bildirimlerinin

```
// ikiside aynı
void func(int a[]);
void func(int *a);

// üçüde aynı
void func(int **p);
void func(int *p[20]);
void func(int *p[]);

// ikiside aynı
void func(int (*p)[20]);
void func(int p[][20]);

// ikiside aynı (function pointer)
void func(int(*)(int));
void func(int(int)); // decay oluyor yani fonksiyondan türünden fonksiyon adresine
dönüşüyor
```

```
struct Data{

};

// parametresi olmayan geri dönüş değeri Data olan fonksiyon
void foo(Data(*)());
void foo(Data());
```

```
// C'de nullptr
int main()
{
    int *p = NULL; // NULL başka başlık dosyalarından gelir
    int *x = 0; // 0 nullptr'a dönüşüyor
}
```

```
// C++ nullptr --keyword (modern cpp ile geldi)
int main()
{
    //nullptr bir sabit, keyword ve türü nullptr_t
    // pointer olmayan değişkene atamak syntax hatası
    int *p = nullptr;
    int x = nullptr; // bu syntax hatalı
    // tür güvenliğini sağlar type-safety
}
```

2023 07 28

Referans Semantiği

- L value reference
- R value reference
- Universal reference (forwarding reference)

L value expression

-Eğer bir expression bir nesneye(storoge var, yeri ve kullanılabilir) karşılık geliyorsa ve bir nesne gösteriyorsa bu ifade L value expressiondır.

R value expression

-Bir nesneye karşılık gelmez ama bir değer ifade ediyorsa yani bir nesneye değer olarak verebiliyorsak R value expression denir.

Bir ifadenin başına address operator (&) verirsek ve hata vermiyorsa L value expressiondır veriyorsa R value expressiondır diyebiliriz.

```
/*
    ++x, --x ifadesi C'de R value expression C++'da L value expression
    x, y ifadesi C'de R value expression C++'da L value expression
    x > 10 ? x : y ifadesi C'de R value expression C++'da L value expression
    x = 5 ifadesi C'de R value expression C++'da L value expression
*/
```

Value Category

Primary value categories

- **L value**
- **PR Value (pure R value)**
- **X Value (eXpiring value)**

combined / unified value category

PR value || X value --> R value

L value || X value --> GL value

```
int main()
{
    // x'in value category'si olmaz
    // x'in türü int
    int x = 0;
    // value category'si var -- L value category
    x
    // NOT: isimlerin oluşturduğu ifadeler her zaman L value categorydedir.
}
```

L value expression yapan operatorlar:

- $++x$
- $--x$
- $*ptr$
- $ptr[35]$
- x, y
- $a ? b : c$

yukarıdaki dışındaki operatorlar PR value expressiondır

PR value expression yapan operatorlar:

- $x++$
- $a + b$
- $+x$
- $-c$
- $!a$
- $a > b$
- $a == b$
- $\&a$

```

// Bir ifadenin value category bulan kod
template <typename T>
struct ValCat {
    static constexpr const char* p = "PR Value";
};

template <typename T>
struct ValCat <T&> {
    static constexpr const char* p = "L value";
};

template <typename T>
struct ValCat <T&&> {
    static constexpr const char* p = "X value";
};

#include <iostream>

#define pvcat(expr) std::cout << "value category of expr '" #expr "' is << ValCat<decltype((expr))>::p << "\n"

int foo();
int main()
{
    pvcat(foo); // L value
    pvcat(foo()); // PR value

    int x {0};
    int *p = &x;
    int **ptr = &p;

    pvcat(&p) // PR value

    // Hepsisi L value
    pvcat(x);
    pvcat(p);
    pvcat(*p);
    pvcat(ptr);
    pvcat(*ptr);
    pvcat(**ptr);

    int a[2]{};
    pvcat(a); // L value
    pvcata(a[0]); // L value
    pvcat(&a[0]); // PR value
    pvcat(a + 1); // PR value
}

```

Reference Semantiği

```
int main()
{
    int x = 10;
    int& r = x;

    r = 45; // x = 45
    ++r; // ++x

    r = y // x = y
}
```

```
int main()
{
    int x = 10;
    int *ptr = &x; // bildirimdeki *ptr'deki * declarator'tur
    *ptr = 10; // burdaki (expression) * operator'dur
}
```

```
int main()
{
    int x; // default init
    const int y; // syntax hatalı

    // referans değişkenler default init edilemez
    int &r; // syntax hatalı
}
```

NOT: R value expression'na R value ref, L value expression'a L value ref verebiliriz

```
int main()
{
    // L value ref'a R value expressionla ilk değer vermeyiz.
    int &r = 10; // syntax hatalı

    int x = 10;
    int &r = x; // r referansı x'e bind edilmiş
}
```

```
int foo();
int main{}
{
    int x = 345;

    int &r1 = x; // geçerli
    int &r2 = +x; // geçersiz
    int &r3 = x++; // geçersiz
    int &r4 = ++x; // geçerli
    int &r5 = &x; // geçersiz

    int &r6 = foo(); // geçersiz
}
```

```
// references are not rebindable
int main()
{
    int x{};
    int &r = x;

    int y{};
    &r = y; // syntax hatalı
}
```

```
// const L value reference
int main()
{
    const int x{};

    // ikisi de aynı anlamda
    const int& r1 = x;
    int const& r2 = x;

    int y{};
    const int& r3 = y;

    y = 32;
    r3 = 32; // geçersiz

    const int z{};
    int &r4 = z; // geçersiz
}
```

```
int g = 24;
void foo(int *& r)
{
    r = &g;
}
int main()
{
    int * p = nullptr;
    foo(p); // p'ye adres verdik
}
```

```
int g = 35;
int& foo(void)
{
    return g;
}
int main()
{
    foo(); // L value expression
    // geçerli
    foo() == 99;
    ++foo();

    int *ptr = &foo();
    ++* ptr;
}
```

```
int &foo()
{
    // tanımsız davranış otomatik ömürlü nesnesin adresi döndürüyor
    int x = 45;
    return x;
}
```

2023 07 31

Reference Semantığı

```
int main()
{
    int x = 10;
    const int *p = &x; // Legal

    const int y = 10;
    int *p1 = &y; // Legal değil
}
```

```
// YAPILMAMASI GEREK
int *func(void)
{
    // dangling pointer olur
    int x = 10; // otomatik ömürlü
    return &x;
}
```

```
int *func(void)
{
    // Legal
    static int x = 10;
    return &x;
}
```

```
//trailing return type
auto func(void) -> int (*)(int)
```

```
struct Data {
    int x, y, z;
}

Data& bar(Data &r)
{
    // aldığı nesneyi döndürür
    return r;
}

int main()
{
    Data mydata{ 1, 4, 2};
    Data &dr = bar(mydata);
}
```

```
// Eğer referans const L value referans ise R value expression ile ilk değer
// verebiliriz
int main()
{
    int x = 10;
    double &r = x; // illegal -- tür uyumsuzluğu

    const double &r1 = x // Legal
    // aslında derleyicinin yaptığı bu
    double temp_obj = ival;
    const double& fr = temp_obj;

    const int& r2 = 10; // Legal

    int& r3 = 10; // illegal
    // aslında derleyicinin yaptığı bu
    int temp_object = 10;
    const int &r = temp_object;
}
```

	<i>L value expr</i>	<i>const L value expr</i>	<i>R value expr</i>
<i>T &</i>	<i>bağlanabilir</i>	<i>bağlanmaz</i>	<i>bağlanmaz</i>
<i>const T&</i>	<i>bağlanabilir</i>	<i>bağlanabilir</i>	<i>bağlanabilir</i>

<i>Pointer vs Referans</i>			
	<i>pointer semantiği</i>	<i>referans semantiği</i>	
1)	<i>default init edilir</i>	<i>default init edilemez</i>	
2)	<i>pointer to pointer olur</i>	<i>referans to referans olmaz</i>	
3)	<i>nullpointer var</i>	<i>null reference yok</i>	

	<i>L value reference</i>	<i>R value reference</i>	<i>Universal reference</i>

R value reference

```
int foo();
int main()
{
    int x = 20;
    // R value refler R value expresion bağlanabilir
    int && r = 10; // geçerli
    int && r1 = x; // geçersiz

    int &&r = foo(); // geçerli
}
```

Type Deduction (tür çıkarımı)

- auto
- decltype
- decltype(auto)

Auto Type Deduction

```
int main()
{
    // tür çıkarımı x için değil auto için yapılır
    auto x = 10;
    auto y; // default init syntax hatalı
}
```

```
int main()
{
    const int x = 10;
    // const'luk düşer
    auto y = x; // y'nun türü int
}
```

```
int main()
{
    int x = 10;
    int &r = x;
    auto y = r; // y -> int
}
```

```
int main()
{
    int x = 10;
    const int &r = x;
    auto t = r; // t -> int
}
```

```
int main()
{
    int a[10]{};
    auto b = a; // b -> int *

    const x[10]{};
    auto y = x; // y -> const int *
}
```

```
int main()
{
    auto ps = "emre"; // auto -> const char*
}
```

```
int func(int); // func'un türü int(int)
// &func ifadenin türü int(*)(int)

int main()
{
    auto x = func; // int(*x)(int)
    auto y = &func; // int(*y)(int)
}
```

```
int main()
{
    int x = 10;
    int *p = &x;
    // top level const
    const auto y = p // y --> int *const y
}
```

```
int main()
{
    char c1 = 10;
    char c2 = 20;

    auto x = c1; // x -> char
    auto y = +c1; // y -> int
    auto z = c1 + c2; // z -> int
}
```

```
int main()
{
    const int x = 10;
    auto& y = x; // auto -> const int
    // y -> const int&
}
```

```
int main()
{
    int a[5]{};
    auto &b = a; // auto -> int[5]
    // b -> int(&)[5]
}
```

```
int main()
{
    auto &x = "eren"; // auto -> const char[5]
    // x -> const char(&)[5]
}
```

```
int foo(int);
int main()
{
    auto &f = foo; // auto -> int(int)
    // f -> int(&)(int) function reference
}
```

```
// using bildirimi

using Word = int;
using ciptr = const int*
using inta20 = int[20];
using FCMP = int(*)(const char*, const char*);
```

Reference Collapsing

```
/*
    T&      &      ==> T&
    T&      &&      ==> T&
    T&&     &      ==> T&
    T&&     &&      ==> T&&
*/
```

2023 08 02

Reference Collapsing

```
/*
Reference Collapsing
T&      &      ===> T&
T&      &&      ===> T&
T&&     &      ===> T&
T&&     &&      ===> T&&

L value ref --> int &x
R value ref --> int &&x
Universal ref  --> auto &&x
*/
```

Universal Reference

Bir universal reference her değer kategorisindeki ifadeye bağlanabilir.

- L value ya da R value (PR || X)
- const ya da non const

auto &&r x;

Eğer ilk değer veren ifadenin (x) değer kategorisi L value ise o zaman auto için yapılan çıkarım L taraf referans türü olur ve r değişkenin türü "reference collapsing" ile T &

Eğer ilk değer veren ifadenin (x) değer kategorisi R value ise o zaman auto için yapılan çıkarım referans olmayan türü olur ve r değişkenin türü ise T && olur.

```
int main()
{
    int x = 5;

    auto &&r1 = x;
    auto &&r2 = 10;

    const int y = 67;
    auto &&r3 = y;
}
```

```
int main()
{
    auto &&x = 10; // int && x = 10

    int ival {4};
    auto&& x = ival; // int& && --> int& x = ival
}
```

Decltype Specifier

`decltype(expr) --> bir tür döner
compiler time ile ilgili`

`decltype specifier ile yapılan tür çıkarımınınında iki ayrı
kural seti vardır.`

- 1) aldığı üyenin isim olması
 - `decltype(x)`
 - `decltype(ptr->x)`
 - `decltype(a.b)`
- 2) aldığı üyenin isim olmaması
 - `decltype(10)`
 - `decltype(x + 5)`
 - `decltype((x))`

```
// decltype 1.kural
int main()
{
    int x = 120;
    decltype(x); // int

    const int y = 21;
    decltype(y); // const int

    decltype(y) z = 10;
}
```

```
int main()
{
    int x = 5;
    int& r{ x };

    decltype(r) y = x;
}
```

```
int main()
{
    int&& r = 10;
    // && & --> &x = 56 hata
    decltype(r)& x = 56; // syntax hatalı
    // && && -> y = 32;
    decltype(r)&& y = 32; // geçerli
}
```

```
int main()
{
    int a[5]{};
    // int[5] --> int b[5] = a
    decltype(a)b = a; // hatalı
    decltype(a)b = { 1, 2, 3}; // hata yok
}
```

```
int main()
{
    int x = 56;
    decltype(x)* p = &x; // int *p = &x
}
```

```
int main()
{
    int a[20] {};
    // int (*p)[20] = &a;
    decltype(a)* p = &a;
}
```

```
// decltype 2.Kural
/*
    Diyelim ki T bir tür olmak üzere expression ifadesinin türü
    T olsun.
    Eğer expression ifadesinin primary value kategorisi:
        PR value ise elde edilen türü T
        L value ise elde edilen türü T&
        X value ise elde edilen türü T&&
*/

```

```
int main()
{
    int x = 10;
    // x + 5 --> PR Value
    decltype(x + 5); // --> int
}
```

```
int main()
{
    int a[5]{};
    // a[2] --> L value
    decltype(a[2]); // --> int&
}
```

```
int main()
{
    int x{ 435 };
    int *p{ &x };
    // *p --> L value
    decltype(*p); // --> int&
}
```

```

int foo(); // foo() --> PR value
int& bar(); // bar() --> L value
int&& baz(); /// baz() --> X value

int main()
{
    decltype(foo()); // --> int
    decltype(bar()); // --> int&
    decltype(baz()); // --> int&&
}

```

```

int main()
{
    int x = 10;
    int y = 20;
    // 1. Kural seti
    decltype(x) a = y; // int a = y;
    // 2. Kural seti
    decltype((x)) b = y; // int& b = y;
}

```

```

int main()
{
    const char* p[] = { "eren", "furkan", "meliike" };
    using nectype = decltype(p); // const char *[3];

    nectype x;
}

```

Unevaluated Context (İşlem kodu üretilmeyen bağlam)

- sizeof
- decltype
- typeid
- noexcept

```

int main()
{
    int x = 12;
    auto val = sizeof(++x); // val = 12
}

```

```

int main()
{
    // decltype -> unevaluated context
    int x = 10;
    // ++x --> L value
    decltype(++x) y = x; // int& y = x

    cout << "x = " << x << "\n"; // x = 10
    ++y;
    cout << "x = " << x << "\n"; // x = 11
}

```

```
// variadic fonksiyon
void foo(int, ...);

int main()
{
    // ... olan yere istediğimiz kadar arguman gönderebiliriz.
    //foo(3, x, y z);
}
```

Default Arguman

```
// default arguman
int func(int = 1; int = 2; int 3);
int main()
{
    func(50, 60, 70);
    func(50, 60);
    func(50);
    func();
}
```

```
int y = 10;
void func(int = ++y)
{
}
int main()
{
    func();
    func();
    func();
    cout << "y = " << y << "\n"; // y = 13
}
```

```
void foo(const char *p = "emre"); // geçerli
void foo(const char*= "emre"); // geçersiz
void foo(const char* = "emre"); // geçerli
```

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

    int a = 10;
    int b = 40;
    // ilk token en uzun olacak şekilde
    int c = a+++b; // a++ b;

    cout << "a = " << a << "\n"; // 11
    cout << "b = " << b << "\n"; // 10
    cout << "c = " << c << "\n"; // 50
}
```

```
//
void foo(int x, int y = x); // syntax hatalı
```

constexpr (C++11)

```
int main()
{
    const int y = 45;
    const int x = y;

    int b[y] = {0}; // geçerli
    int a[x] = {0}; // geçersiz
}
```

```
int main()
{
    // bir sabit ifadesi
    constexpr int x = 10; // x'in türü const int

    int y = 43;
    constexpr int z = y; // illegal
}
```

2023 08 04

Constexpr

```
constexpr int x = 5; // x'in türü int
```

```
int g{};
// ilk ikisinin arasında hiçbir fark yok
constexpr int* p = &g;
constexpr int const* p = &g;
// yukarıdakilerden farklı
constexpr const int* p = &g;
```

```
int foo();
const int x1 = foo(); // geçerli
constexpr int x2 = foo(); // syntax hatalı
```

```
//dizi olarak tanımlanabilir
constexpr int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
constexpr int x = primes[4];
```

Constexpr Fonksiyonlar

1. *Constexpr fonksiyon olması için belirli koşulları sağlaması gereklidir:*
 - static ömürlü yerel değişkene sahip olmayacak
2. *Eğer tüm parametrelerine sabit ifadeleri ile çağrı yapılrsa fonksiyonun geri dönüş değeri derleme zamanında elde ediliyor.*
eğer foo'ya verilen argumanların hepsi constexpr ise foo'nun geri dönüş değeri derleme zamanında oluşur.
`constexpr int x = foo(exp1, exp2, exp3)`
Eğer argumanlar constexpr değilse run-time'da elde edilecek

```
constexpr int ndigit(int x)
{
    if (x == 0)
        return 1;
    int digit_count{};
    while (x){
        ++digit_count;
        x /= 10;
    }
    return digit_count;
}

int main()
{
    int a[ndigit(83123)]{}; // compiler-time'da hesaplandı
    const int x = 3234;
    const int y = 23234;
    // fonksiyonun parametresi sabit ifadesi
    constexpr auto val = ndigit(x * y - 19); // compiler-time'da hesaplandı
    // eğer syntax hatalı verseydi artık sabit ifadesi bu diyemezdik
}
```

```

constexpr bool isprime(int val)
{
    if (val < 2);
        return false;
    for (int i{ 7 }; i * i; i <= val; i+=2)
    {
        if (val % i == 0)
            return false;
    }

    return true;
}

int main()
{
    const int x = 12312;
    const int y = 12312;
    constexpr auto a = isprime(x + y - 19); // compiler-time

    int z = 123;
    constexpr auto b = isprime(x / z - 12); // run-time
}

```

NOT: constexpr fonksiyonlar headerda tanımlanır

ODR (One Definition Rule)

Değişkenler, sınıflar ve fonksiyonlar gibi yazılımsal bazı varlıkların bildirimleri program içinde birden fazla kez bulanabilir. Ancak tanımları tek olmak zorundadır.

Inline Expansion

Derleyici yazdığımız kodu fonksiyonun çağrıldığı yerde açar yani tüm kodu çağrıldığı noktaya eklenir. Bunu yapabilmesi derleyicinin fonksiyon kodunu görmesi gereklidir.

Bunun avantajları:

- Linkleme maliyetini azaltır.
- Derleyicinin gördüğü kod statement artar ve optimazyonu artar

statement 1;

statement 2;

x = func(a, b); burda func kodu buraya eklenmiş gibi gözükmektedir

statement 4;

statement 3;

derleyici inline expansion sayesinde func ve bütün statementları aynı anda görür ve optimazyonu ona göre yapar.

Inline Anahtar Sözcüğü

```
inline int foo(int x, int y) {  
    /*  
     * inline olarak tanımlansa bile derleyici inline expansion etmek zorunda  
     * değil. Aynı zamanda inline olarak tanımlanmasa bile derleyici bunu inline expansion  
     * edebilir.  
     */  
    return x * 12 * y;  
}
```

```
ahmet.cpp  
inline int foo(int x, int y)  
{  
    return x * y - 5;  
}  
  
tunahan.cpp  
inline int foo(int x, int y)  
{  
    return x * y - 5;  
}  
  
fonksiyonları inline olarak tanımlamasak ODR'a aykırı olurdu  
*/
```

```
// static ömürlü global ve class member  
inline static int g{12}; // static ömürülü böyle tanımlayabiliriz.
```

Neleri başlık dosyasına koyarsam ODR ihlal etmemiş olurum.

- 1) inline fonksiyon tanımları
- 2) inline değişken tanımları (C++17)
- 3) user-defined type (class)
- 4) constexpr fonksiyonlar implicitly inline

C++ dilinde enumeration types

```
// modern cpp öncesi  
enum Color {Blue, Black, White, Purple, Red};  
int main()  
{  
    Color mycolor;  
    // C'de geçerli  
    mycolor = 3; // syntax hatalı Cpp'da  
}
```

```
/*
 Modern olmayan C++ dilinde enum türlerinin istenmeyen
 özellikleri :
 1) underlying type derleyiciye bağlı olduğu için enum türleri başlık
 dosyalarında
 incomplete type olarak kullanılamaz

 enum Color;
 struct Data {
     Color mc; // syntax hatası olur çünkü Color sizeni bilmiyoruz
 };

=====
 2) Enum türlerinden tamsayılar türüne örtülü dönüşüm olması

 enum Color {Blue, Black, White, Purple, Red};
 int main()
 {
     Color mycolor = Black;
     int ival;
     ival = mycolor;
 }

=====
 3)
 // traffic_ligth.h
 enum TrafficLigth { Red, Yellow, Green};
 // screen.h
 enum ScreenColor{ Magenta, White, Black, Red};

 isim çakışması bu header'Lar syntax hatası yaratır.
 */
```

2023 08 27

Scoped enum

```
// scoped enum
enum class Pos : unsigned char {On, Off, Hold, Standby};
```

```
int main()
{
    int printf = 10;
    printf("emre"); // syntax hatalı
    ::printf("emre"); // geçerli
}
```

```
enum class Color {Blue, Red, Purple, White, Black};
enum class TrafficColor {Red, Yellow, Green};
int main()
{
    // syntax hatalı yok
    Color::Blue;
    TrafficColor::Red;

    Color mycolor{ Color::Red};
    int ival = mycolor; // syntax hatalı
}
```

```
enum class Color {Blue, Red, Purple, White, Black};
void func()
{
    using enum Color; // C++20
    Color c1; = Red;
    c1++; // geçersiz
}
```

Type-Cast Operatorler

```
/*
  C dilinde
  (target type)expr
  (int)dval
  (double)ival

type-cast operatorleri:
  static_cast
  const_cast
  reinterpret_cast
  dynamic_cast

  xxx_cast<type>(expr)
*/
```

```
// static_cast
int main()
{
    int x = 10;
    int y = 30;
    double dval = static_cast<double>(x) / y;
}

// const_cast
int main()
{
    const int x{ 56 };
    const int *xp = &x;
    int *ptr = const_cast<int *>(xp);
}
```

2023 08 09

Function Overloading

```
/*
    function overloading
    - derleme zamanında gerçekleşir.

    a) compile-time
        static binding
        early binding
    b) run time
        dynamic binding
        late binding
    Bir fonksiyonun imzası (signature) fonksiyonun parametre değişkenlerinin türü
    ve sayısıdır
*/
```

```
//redeclaration, overloading değil
int foo(int,int);
int foo(int,int);
```

```
//redeclaration, overloading değil
int foo(int);
int foo(const int);
```

```
// overloading
void func(int *ptr);
void func(const int *ptr);
```

```
//redeclaration
void func(int* ptr);
void func(int* const ptr);
```

```
// syntax hatalı
int foo(int,int);
double foo(int,int);
```

```
// ikisi de pointer (redeclaration)
void func(int *);
void func(int []);
```

```
// array decay (redeclaration)
void func(int(int));
void func(int*(int));
```

```
// overloading dizilerin boyutları farklı
void func(int(*)[10]);
void func(int(*)[12]);
```

```
// redeclaration
void func(int(*)[10])
void func(int[ ][10]);
```

```
// redeclaration
typedef int Nec;
void func(int);
void func(Nec);
```

```
// 3 tane overload var
void func(char);
void func(signed char);
void func(unsigned char);
```

```
// overloading var
void func(int);
void func(int &);
```

```
// 4 tane overload var
void func(int &)
void func(int &&)
void func(const int &)
void func(const int &&)
```

```
// overloading
void foo(int *)
void foo(int &)
```

```
// overloading
void foo(int *)
void foo(std::nullptr_t)
```

```
void func(long double);
void func(char);
int main()
{
    // syntax hatası çünkü iki fonksiyonda geçerli hangisi çağırılacak bilemeyez.
    func(12.5); // ambiguity
}
```

Overloading Süreci:

- 1) aday fonksiyonların saptanması
- 2) legal olarak çağrılabilecek fonksiyonlar tespiti

```
void func(void *);  
void func(int *);  
void func(int, int);  
// viable(uygun) fonksiyon yok ama 3 tane overload var  
int main(){  
    func(56); // amb  
}
```

user defined conversion
standart conversion:
1) exact match (tam uyum)
2) promotion (terfi - yükseltme)
3) conversion (dönüşüm)

```
//exact match spec:  
void func(const int *);  
void func(int *)  
  
int main()  
{  
    const int x = 5;  
    func(&x); // const int *  
  
    int y = 1;  
    /*  
        eğer const ve const olmayan iki fonksiyonda tanımlıysa const olmayan  
        fonksiyon çağrırlacak ama const olmayan tanımlı değilse bu sefer  
        const int * çağrırlacak  
    */  
    func(&y);  
}
```

```
void func(int (*)(int));  
int foo(int);  
int main(){  
    int x = 10;  
    func(&x); //exact match  
  
    int a[5]{};  
    func(a); //array decay (exact match)  
    func(&foo); // int(int)  
}
```

```
void func(int *)  
void func(double *)  
void func(std::nullptr_t)  
  
int main(){  
    func(nullptr) // std::nullptr gelecek  
}
```

```
class MyClass{};  
void func(const MyClass&); // const l value ref parametrelili  
void func(MyClass &&); // r value ref parametrelili
```

2023 08 11

Classes

```
class members
    data members
        static data members
        non-static data members
    member function
        static member function
        non-static member function
    member type / type member / nested member
        class names
```

```
class Myclass {
    int x; // data member
    void func(); // member function
    class Myclass1; // member type
};
```

```
class Myclass {
    int x, y, z;
};

int main()
{
    // sizeof(Myclass) = 12
    std::cout << "sizeof(Myclass) = " << sizeof(Myclass) << "\n";
}
```

```
member selection operators
    . ve ->
        MyClass m;
        m.x;
        MyClass *p = &m;
        p->x;
```

```
access specifier
    private members
    public members
    protected members

class'ta herhangi bir access specifier kullanmazsaK private,
struct'ta ise public olarak default olur

private, public, protected ayrı scope degildir!
```

```

class Fighter{
    public:      //Fighter*
        void func(          );
}
int main(){
    Fighter f;
    f.func(); // burda aslında func gizli bir arguman alıyor o da f adresi
}

```

```

//member fonksiyonları 2 defa tanımlayamıyoruz
class Nec {
    public:
        //syntax hatalı
        int foo(int);
        int foo(int);
};

```

```

class Nec{

private:
    int foo(int);
public:
    int foo(double);
};

int main(){
    Nec myNec;
    myNec.foo(12);
    //Hata döner çünkü önce hangi fonksiyonun çağrılacağı seçimi yapılıyor sonra
erişim yapılmıyor.
}

```

```

class Nec {
    public:
        void set(int x, int y);
    private:
        int mx, my;
}
// nec.cpp
#include <nec.h>
Nec necgp;
void Nec::set(int x, int y)
{
    mx = x;
    my = x;
    // Class'ın kendi scopeunda private elemanlara erişebiliriz.
    necgp.mx = 123; // geçerli
}
// main.cpp
int main()
{
    Nec mynec;
    mynec.set(23, 12);
}

```

```
// Sınıfların üye fonksiyonları içinde isim arama (name Lookup)
//nec.h
class Nec{
public:
    void foo();
private:
    int x;
}

//nec.cpp
int x = 45;
void Nec::foo()
{
    x = 45; // sınıf elemanı
    int x = 67;
    Nec::x = x + ::x
}
```

```
// member functions inline
// nec.h
class Nec{
public:
    void set(int x, int y)
    {
        // implicitly inline
        mx = x;
        my = y;
    }
    inline int getx() const;
private:
    int mx, my;
}
// header da tanımlayacağımızda inline olarak ya bildirimde ya tanımda kullanmalıyız
inline int Nec::getx() const
{
    return mx;
};
```

2023 08 14

Classes

```
//class'Larda member functions storege kaplamaz ancak data member yer kaplar
class Nec {
    public:
        void f1(int);
        void f2(int);
        void f3(int);
    private:
        int mx, my;
};

int main()
{
    // sizeof Nec = 8
    std::cout << "sizeof(Nec) = " << sizeof(Nec) << "\n";
}
```

```
/*
    isim arama önceliği;
    önce bloklarla(kendi scope ve dışındaki scopler)
    class'Larda
    en son olarak namespace (global)

*/
```

this keyword

```
/*
this keyword
1) this bir keyword
2) this bir pointerdir non static üye fonksiyon içinde
hangi nesne için çağrıldıysa onun adresini döndürür.
3) *this o nesneni kendisini
*/
```

```
class Myclass
{
public:
    void foo()
    {
        std::cout << "this = " << this << "\n";
    }
private:
    int mx{}, my{};
};
int main()
{
    Myclass m;
    // aynı adresleri döner
    std::cout << "&m = " << &m << "\n";
    m.foo();
}
```

```
// this bir PR Value
class Myclass {
public:
    void foo();
}

void Myclass::foo()
{
    Myclass m;
    this = &m; // syntax hatalı
}
```

```
class Tamer {
public:
    Tamer* foo() {return this;}
    Tamer* bar() {return this;}
    Tamer* baz() {return this;}
}

int main()
{
    Tamer* p = new Tamer;
    p->bar()->foo()->baz();
}
```

```
// const member functions
class Nec {
public:
    // Nec*
    void foo(); // non-const member function
    // const Nec*
    void bar() const; // const member function
};
```

```
class Myclass {
public:
    void foo() const
    {
        mx = 56; // syntax hatalı

        Myclass m;
        m.mx = 89; // Legal

        // const T* --> T*
        bar(); // illegal
    }

    void bar()
    {
        // T* --> const T*
        foo(); // Legal
    }
private:
    int mx;
}
```

```

class MyClass
{
public:
    // overloading
    void foo();
    void foo() const;

    void bar();
private:
    int mx;
}

int main()
{
    const MyClass m;
    m.foo(); // legal
    m.bar(); // illegal;
}

```

Mutable

```

// mutable keyword
class Fighter
{
public:
    void foo() const
    {
        ++debug_call_count; // legal
        ++m_age; // illegal
    }
private:
    std::string m_name;
    int m_age;
    int m_power;
    mutable int debug_call_count;
}

```

1. const üye fonks içinde sınıfın non-static veri elemanlarına atama yapamayız
2. const üye fon içinde sınıfın non-const üye fonksiyonlarını çağrıramayız
3. const sınıf nesnelerini içinde sadece sınıf const üye fonks çağırabiliriz
4. const sınıf nesneleri için sınıfın non-const üye fonksiyonları çağrıramayız

2023 08 16

Constructor

```
Constructor
1) sınıf isimiyle aynı olmak zorunda
2) non-static member function olmak zorunda
3) ctor free function olamaz
4) ctor static member function olamaz
5) const member function olamaz
6) geri dönüş değeri kavramına sahip değildir
7) overload edilebilir.
8) public, private ve protected olabilir ama diğerleri gibi erişim sıkıntısı
olabilir
9) nokta ya da ok operatörüyle çağrılamıyor

default ctor: parametresi olmaya ya da tüm parametreleri varsılan arguman alan
ctor

class MyClass{
    MyClass(int x = 6) // default ctor
};
```

Destructor

```
Destructor
Bir sınıf nesnesinin lifespan bitmesini sağlayıp hayatını sonlandıran fonks
1) non-static member function olmak zorunda
2) dtor free function olamaz
3) dtor static member function olamaz
4) const member function olamaz
5) geri dönüş değeri kavramına sahip değildir
6) parametre değişkeni olmalıdır
```

Special Member Functions:

```
special member functions:
default ctor
destructor
copy ctor
move ctor (C++11)
copy assignment
move assignment (C++11)

special member denmesinin nedeni bu fonksiyonların kodları (belirli koşullar
altında)
derleyici tarafından bizim için yazılabilmesi
```

Global Sınıf Nesneleri

```
class Myclass;
// aynı kaynak dosyasında olduğu için ctor çağrılmama sırasına
göre
Myclass g;
Nec g_nec;
// main fonksiyonu çağrılmadan g nesnesinin ctor çağrıılır ve main bittiğinden sonra
dtor çağrıılır
int main()
{
}
```

Aynı programın farklı kaynak dosyalarında tanımlanan global sınıf nesnelerinin tanımlanan global sınıf nesnelerin ctor'larının çağrılmama sırası dil tarafından belirlenmiş değildir.

```
class Myclass;
void foo()
{
    static Myclass m; //static storage class
}
int main()
{
    foo(); foo(); foo(); // foo function 3 defa çağrılmamasına rağmen m objesi
    //bi kere oluşturulacak ve main tamlandıktan sonra yokolacak.
    //1
    {
        //2
        Myclass m; // önce 1 olacak sonra 2 sonra m objesinin ctor oluşturulacak
        //sonra 3 ve m objesinin dtor olacak ve 4 olacak
        //3
    }
    //4
}
```

Ctor Initializer List

```
//önce data memberler( tx,ux, mx) meydanda gelir daha sonra ctor çağrırlır.
class Myclass{
public:
    Myclass() : my(10), mx(my /3) // burda unbehaviour oluyor çünkü
                //mx ilk init edilir my bu sırada garabed value oluyor
    {
    }
private:
    T tx;
    U ux;
    W mx;

    int mx,my;
};
```

```

class MyClass{
public:
    MyClass(int &r) : mx{20}, mr(r)
    {

    }
private:
    // referans ve const'lar default init edilemez
    const int mx; // eğer value init yapmazsak yukarıdaki gibi

    int &mr; // syntax hata dönerdi
}

```

```

class Person {
public:
    Person(const char* p)
    {
        /*
            böyle yaparsak önce default ctor çağrırlıır
            daha sonra copy assignment çağrırlıır
        */
        m_address = p;
    }
private:
    std::string m_address;
}

```

```

class myclass{
myclass() : mx(46) {};
private:
    int mx = 10;
    int my = 20;

/*
    in class initializer cpp11
    default member initializer 10 ve 20'yi derleyici default init edicek
    biz ne yapması gereki̇gi söylüyoruz init eden derleyici
*/
};

```

2023 08 18

Classes

```
class Myclass {  
    public:  
        Myclass(int); // conversion constructor  
};
```

delete bildirimi

```
// delete bildirimi  
void func(int) = delete; // bildirilmiş ama delete edilmiş
```

```
// sadece int parametreli func çağrılabilecek  
template <typename T>  
void func(T) = delete;  
  
void func(int);  
  
int main()  
{  
    func(2.3); // syntax hatalı  
}
```

Class Special Functions

- default ctor
- destructor
- copy ctor
- move ctor
- copy assignment
- move assignment

Classların özel fonksiyonlarının 3 durumu vardır:

1. not declared
2. user declared
 - a. default
 - b. delete
3. implicitly declared
 - a. derleyici tarafından default
 - b. derleyici tarafından delete

Derleyici default ctor default etmek zorunda ve derleyicinin yazdığı defult ctor class'ın data memberlarını default etmek zorunda. Aşağıdaki durumda derleyici int &r ve const int x default edemeyeceği için Myclass sınıfının default ctor'u delete edilmiş durumda

```
class Myclass
{
    public:
    private:
        int &r;
        const int x;
};

int main()
{
    // default ctor deleted
    Myclass m;
}
```

```
class Member {
public:
    Member(int);
};

class Tamer {
    // Tamer sınıfının default ctor durumu: implicitly declared deleted
private:
    Member mx;
};
int main()
{
    Tamer tx;
}
```

Aggregate Class

Bir aggregate class, yalnızca public veri üyelerini içeren ve özel üye fonksiyonları veya üye değişkenleri olmayan bir sınıfır. Bu, C++'ta bir struct veya bir class tanımı içinde yer alabilir.

```
class Myclass {
public:
    int mx{};
    int my{};
};

static_assert(std::is_aggregate_v<Myclass>);

int main()
{
    Myclass m = {1, 2};
}
```

Copy Constructor

Eğer bir sınıf nesnesi hayatı değerini aynı türden; bir başka sınıf nesnesinden alarak geliyor ise copy ctor kullanılır.

```
class Nec
{
public:
    Nec()
    {
        std::cout << "Default ctor this : " << this << "\n";
    }

    Nec(const Nec&)
    {
        std::cout << "Copy ctor this = " << this << "\n";
    }
    ~Nec()
    {
        std::cout << "Destructor this : " << this << "\n";
    }
};

void foo(Nec)
{
}

int main()
{
    Nec mynec;
    std::cout << "&mynec = " << &mynec << "\n";

    foo(mynec);
    std::cout << "main devam ediyor\n";
}
```

```
class Nec {

};

int main()
{
    Nec x; // default ctor
    // copy ctor
    Nec n1 = x;
    Nec n2(x);
    Nec n3{x};
}
```

```
class A{};
class B{};
class C{};

class MyClass {
public:
    MyClass() : ax(), bx(), cx()
    {

    }
    MyClass(const MyClass &other) : ax(other.ax), bx(other.bx),
cx(other.cx)
    {

    }
private:
    A ax;
    B bx;
    C cx;
};

rule of zero
    sınıfı özel fonksiyonları derleyici tarafından yazılmamasına denir.
```

2023 08 21

```
class Address
{
public:
    Address(const char* p) : mlen(std::strlen(p)),
    mp(static_cast<char*>(std::malloc(mlen + 1))){
        if (!mp) {
            throw std::runtime_error{ "not enough memory" };
        }
        std::cout << static_cast<void*>(mp) << "adresindeki bellek blogu
edinildi\n";
        std::strcpy(mp, p);
    }

    Address(const Address& other) : mlen{other.mlen},
    mp(static_cast<char*>(std::malloc(mlen + 1))){
        if (!mp) {
            throw std::runtime_error{ "not enough memory" };
        }
        std::strcpy(mp, other.mp);
    }
    Address& operator=(const Address& other) {
        if (this == &other) {
            // avoid self assignment
            return *this;
        }
        std::free(mp);

        mlen = other.mlen;
        mp = static_cast<char*>(std::malloc(mlen + 12));
        if (!mp) {
            throw std::runtime_error{ " not enough memory "};
        }

        std::strcpy(mp, other.mp);
        return *this;
    }
    ~Address(){
        std::cout << static_cast<void*>(mp) << " adresindeki bellek blogu geri
verildi\n";
        std::free(mp);
    }
    void print()const {
        std::cout << mp << "\n";
    }
    std::size_t length()const {
        return mlen;
    }
private:
    std::size_t mlen;
    char* mp;
};

void process_address(Address x) {
    // copy ctor cagrilar
    std::cout << "process_address fonksiyonu cagrildi\n";
    x.print();
}
```

```

int main() {
    using namespace std;

    Address adx{" sultangazi "};
    adx.print();

    cout << "adres uzunlugu" << adx.length() << "\n";
    process_address(adx);

    // kopyalandıktan sonra dtor çağrıldı adx dangling pointer oldu
    std::cout << "main devam ediyor\n";
    adx.print();

    //

    Address adx1{ " gop " };
    if (adx1.length() > 10)
    {
        Address ady1 {" bayrampasa "};
        ady1.print();

        ady = adx; // copy assignment
        ady.print();
    }
    adx.print(); //
}

```

copy ctor oluşturma durumunu çok nadir de kullanılsa da eğer bir pointer gibi parametre tutuyorsak adresi işaret eden bir fonksiyona bu nesneyi arguman olarak verdiğimizde o nesne kopyalanır ve o fonksiyon sonlanınca o nesne dtor olur bu da adresindeki nesnenin dtor olmasına sebep olur. Yani main'e tekrar döndüğümüzde o nesneyi kullanamayız.

```

class Student
{
    // copy ctor yazmaya gerek yok sorun sadece sınıf veri elemanı pointer
    // olduğunda var
    private:
        int m_id;
        std::string m_name;
        std::string m_address;
        std::vector<int> m_grades;
}

```

Copy Assignment

```
class Myclass {
public:
    Myclass& operator=(const Myclass& other)
    {
        ax = other.ax;
        bx = other.bx;
        cx = other.cx;
        return *this;
    }
private:
    A ax;
    B bx;
    C cx;
}
```

```
/*
Eski C++'da

Big Three
Destructor      release resources
Copy Constructor deep copy
Copy Assignment  release resources and deep copy
*/
```

Move Constructor

```
class Myclass {
public:
    Myclass(); // default ctor
    ~Myclass(); // destructor
    Myclass(const Myclass&); // copy ctor
    Myclass& operator=(const Myclass&); // copy assignment

    Myclass(Myclass&&); // move ctor
    Myclass& operator=(Myclass&&); // move assignment
};
```

```

class Myclass{
    Myclass() = default;

    Myclass(const Myclass&)
    {
        std::cout << "copy ctor\n";
    }
    Myclass(Myclass&&)
    {
        std::cout << "move ctor\n";
    }
};

void func(const Myclass&)
{
    std::cout << "const Myclass&\n";
}

void func(Myclass&&)
{
    std::cout << "Myclass&&\n";
}

void foo(Myclass&& r)
{
    func(r);
}

int main()
{
    Myclass m;

    func(m); // const MyClass&
    func(Myclass{}) // MyClass&&

    func(static_cast<Myclass&&>(m)); // MyClass&&
    // taşıma yapmıyor L value'yu r value yapıyor

    // ne copy ne move ctor çağrırlırs
    func(std::move(m)); // MyClass&&

    foo(std::move(m)); // func(const MyClass&) çağrırlır
}

```

```

int main()
{
    Myclass m;
    // hayata gelen bir nesne yok o yüzden ne move ne copy ctor çağrırlır
    Myclass&& r = std::move(m);
}

```

```

class Myclass{
    Myclass() = default;

    Myclass(const Myclass&)
    {
        std::cout << "copy ctor\n";
    }
    Myclass(Myclass&&)
    {
        std::cout << "move ctor\n";
    }
};

// kaynağı çalan m objesi
void foo(const Myclass&other)
{
    // copy ctor
    Myclass m(other);
}

void foo(Myclass&& other)
{
    // move ctor
    Myclass m(std::move(other));
}

int main()
{
    Myclass m;
    foo(std::move(m));
}

```

```

// derleyicinin yazdığı move ctor
class Myclass
{
    // primative türler ve pointerlar için taşıma olmaz
public:
    Myclass(Myclass&& other) : ax(std::move(other.ax)), x(other.x),
ptr(other.ptr)
    {

    }

    Myclass& operator=(Myclass&& other)
    {
        ax = std::move(other.ax);
        x = other.x;
        ptr = other.ptr;
    }
private:
    A ax;
    int x;
    char *ptr;
};

```

2023.08.23

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
hiçbiri	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

```
class Date {
    private:
        int x{} //default init edilecek
        std::string str // string class'ının default ctor çağrılmak böyle default
};
```

move-only type class

```
class Neco{
public:
    Neco();
    Neco(const Neco&) = delete;
    Neco &operator=(const Neco&) = delete;
    Neco(Neco&&);
    Neco& operator=(Neco&&);
}
```

copy ve move ctor yok

```
class Neco{
public:
    Neco();
    Neco(const Neco&) = delete;
    Neco &operator=(const Neco&) = delete;
}

// Asla asla move member'ları delete etme!!! Çünkü move member delete edersem copy
memberleri bloke ederiz
```

```

class Member {
    public:
        Member(int);
        Member(int,int);
        Member() = delete; // durum2
    private:
        Member() // durum 1

        //durum 3 hiç default ctor olmaması
}

class Nec
{
    private:
        Member mx; // burda syntax hatalı yok
        /*
        Nec sınıfının default ctor delete edilir çünkü derleyicinin yazdığı default
        ctor mx'i default init edecek
        ancak mx'in default ctor olmadığı için kendi default ctor'ın delete edicek
        */
}

int main()
{
    Nec nec; // burda nec nesnesi oluşamıçak çünkü default ctor delete edilmiş
    //durumda
}

```

- **temporary objects C++17**
- **explicit ctor**
- **conversion ctor**

Temporary Objects Cpp17

```
class Myclass{
public:
    Myclass();
    Myclass(int x);
}

void func(const Myclass &);
void foo(Myclass);
int main()
{
    Myclass{}; // PR Value expr
    Myclass(12);
    // iki durumda da ctor çağrıılır böyle objelere temporary objects denir

    Myclass m;
    func(m);

    /*
    eğer bu m nesnesine func ya da foo gibi sadece fonksiyon çağrısı için
    ihtiyacımız varsa
    böyle kullanmalıyız çünkü yukarıdaki gibi kullanırsak ( func(m) )
    hemen nesnenin daha sonra kullanacağımızı düşünülebilir
    hem de mandatory copy elision (c++17)
    */

    func(Myclass{12});

    Myclass& r = Myclass{}; // l ref r value'ya bağlanmak istiyor hatalı
    const Myclass& r = Myclass{}; // const l value ref olur hatasız
    Myclass&& r = Myclass{}; // r value ref olur hatasız
}
```

```
void foo(Myclass);
void bar(Myclass &);
void baz(const Myclass &);
void func(Myclass &&)

main()
{
    foo(Myclass{}) // geçerli
    bar(Myclass{}) // geçersiz
    baz(Myclass{}) // geçerli
    func(Myclass{}) // geçerli

}

/*
    normalde bir geçici nesne oluştuğunda, hayatı gelen geçici nesnenin hayatı
    geçici nesneyi içine
    alan ifadenin yürütülmesiyle sona erer
*/
```

```

main()
{
    std::cout << "[1]\n";
    Myclass {}; // ctor ve dtor 1 ve 2 arasında çağrılır
    std::cout << "[2]\n";

    std::cout << "[3]\n";
    const Myclass &r = Myclass{}; // ctor 3 'ten sonra ama dtor scope bitince
    çağrılar buna life extension denir
    std::cout << "[4]\n";
}

```

moved-from state (taşınmış nesne durumu)

```

int main()
{
    using namespace std;
    string s;
    s = foo(); // move assignment
    /*
        foo() r value expr o yüzden move assignment çağrılır ancak
        l value olsaydı copy assignment çağrılırdı

    */
    s = string(100100, 'A') // move assignment
    string str(200000, 'Z');
    string sx = str; // copy assignment
    string sxx = std::move(str); // move assignment

    //mesela
    string sx;

    {
        string str(2000, 'Z');
        sx = std::move(str);
        // str 'in dtor bu scope sonunda çağrılır
        // str is in moved_from state
        // str yani kaynağı çalınmış nesne geçerlidir ancak değeri bilinmez
        s.empty() --> true döner
    }

    // sx burda kaynağı gittiği için kullanılamaz.
}
/*
    Tipik olarak (böyle bir zorunluluk yok) move member'lar kaynağı çalınmış diğer
    nesneyi
    default ctor edilmiş state'te bırakırlar.

    Kaynağı çalınmış bir nesne
    a) geçerli bir durumda (in valid state)
    b) bilinmeyen değerde (its value unknown)

    standart kütüphanede böyle tasarılmıştır.

*/

```

```

std::string foo();

int main()
{
    string sx(20000, 'a');
    string str = std::move(sx); // sx'i gözden kaçırıldığımız düşünülebilir
}

```

Örnek Kod (Moved-from state)

```

class Vector {

public:
    push_back(const string &); // l value buraya copy ederek
    push_back(string &&); // r value buraya move ederek

};

int main()
{

    ifstream ifs {"notlar.txt"};
    string sline;
    vector <string> svec;
    // satır satır dosyadan okuyoruz
    while(getline(ifs, sline))
    {
        cout << sline << '\n';
        svec.push_back(sline);
        svec.push_back(std::move(sline)); // böyle yaparak daha verimli olur

        /*
            burada sline string'i std::move kullanarak svec taşınıyor.
            sline taşınıyor ama tekrar kullanabilir hale de tekrar atama
            yapılabiliyor
                ama bu garantiye veren standard kütüphane başka bir kütüphane sline'in
            tekrar kullanabileceğini
                garantisi vermez.

        */
    }
}

```

Conversion Constructor (Dönüştüren kurucu işlev)

```
class Myclass {
public:
    Myclass() = default;
    Myclass(int);
    ~Myclass();
}

main()
{
    Myclass mx;
    std::cout << "main [1]\n";
    mx = 5; // eğer Myclass(int) diye ctor yoksa hata verir
    /*
        Myclass(int) ctor'u çağrılar ve mx'in dtor çağrılar
        çünkü mx temp object olarak oluşturuldu

        mx = 5 move assignment ile çağrılar ama biz copy assignment yazarsak
    class'in içinde
        copy assignment çağrılar.
    */
    std::cout << "main [2]\n";
    // ilk tanımladığım mx'in dtor çağrılar
}
```

Function overload resolution

- variadic conversion
- user defined conversion

```
class Myclass {
public:
    Myclass();
    Myclass(int);
    Myclass(bool);
}
void func(Myclass);

int main()
{
    func(12); // bu geçerli çünkü burada user defined conversion gerçekleşti

    Myclass m;
    m = 4.9073; // double olmasına rağmen Legal

    int x = 10;
    int* ptr = nullptr;

    m = &x; // Legal
    m = ptr; // Legal

    // ptr türünden bool türüne dönüşüm var
}
```

Eğer bir dönüşüm aşağıdaki dönüşüm sekanslarından biriyle gerçekleştirilebiliyor ise derleyici bu dönüşümü örtülü olarak yapmak zorunda

user-defined conversion + stardart conversion

standart conversion + user-defined conversion

```
Myclass m ;  
  
double dval = 32.5;  
m = dval; // Önce standart dönüşüm (double -int) sonra user-defined dönüşüm (int - Myclass)
```

Explicit Ctor

```
class Myclass {  
public:  
    Myclass();  
    explicit Myclass(int); // anahtar sözcükelerle dönüşüm ile olur  
(static_cast gibi)  
}  
  
main()  
{  
    Myclass m;  
    m = 23; // hata döner  
    m = static_cast<Myclass>(23); // hata dönmez böylece yanlışlıkla dönüşüm  
yapmayı engeller  
  
    Myclass m1(19); // geçerli direct init  
    Myclass m2{35}; // geçerli direct list init  
    Myclass m3 = 21; // geçersi copy init  
  
/*  
    ctor explicit ise copy init syntax hatası oluşturur ama  
    direct init ve direct list init hata oluşturmaz  
*/  
}
```

Cpp Core Guidelines

Bir sınıfı (özellikle) tek parametreli ctor'larını (aksi yönde karar almanız gerektirecek bir neden olmadığı sürece) explicit yapınız.

Explicit Ctor Örnek:

```
main()
{
    unique_ptr<int> p1{new int};
    unique_ptr<int> p2{new int};
    unique_ptr<int> p3 = new int // geçersiz çünkü unique_ptr explicit ctor sahip
}
```

```
class Myclass {
public:
    Myclass(int);
    explicit Myclass(double);
}

main()
{
    Myclass m = 45.98; // hata olmaz çünkü explicit ctor overload sette hiç
girmiyor
}
```

```
void func(Myclass)
Myclass foo()
{
    //return MyClass();
    //return MyClass {};
    return {} // ctor explicit olursa geçersiz

    // üçüde geçerli
}

main()
{
    func(Myclass());
    func(Myclass{});
    func({}); // ctor explicit olursa geçersiz

    // üçüde geçerli
}
```

25.08.2023

- copy elision
- C++17 (mandatory copy elision)
- Return Value Optimization RVO
- NRVO

- dinamik ömürü nesneler
- new ifadeleri
- delete ifadeleri
- operator new fonksiyonları
- operator delete fonksiyonları

PR Value bir nesne olmuyor C++17' ye göre

mesela

```
Myclass{}; //temp objects
```

Temporary Materialization

```
class Myclass
{
    Myclass(const Myclass&) = delete;
}

void foo(Myclass x)

int main()
{
    Myclass{}; //PR Value

    Myclass x = Myclass{}; //temporary materialization
    const Myclass &x = Myclass{}; //temporary materialization

    foo(Myclass{})
    foo(Myclass{46})
    /*
    Myclass{} ve MyClass{46} bir nesne olmadığı için ctor ve dtor çağrılmacak
    copy ctor delete olmasına rağmen
    */
}
```

Return Value Optimization (RVO)

```
// Return Value Optimization (RVO)

class Myclass {
public:
    Myclass()
    {
        std::cout << "default ctor\n";
    }
    ~Myclass()
    {
        std::cout << "destructor\n";
    }
    Myclass(int)
    {
        std::cout << "Myclass(int)\n";
    }

    Myclass(const Myclass&) = delete;
};

Myclass foo(int n)
{
    //code
    return Myclass{n};
}
int main()
{
    Myclass m = foo(13); // int parametreli ctor çağrılmır bir kez
}
```

Named Return Value Optimization (NRVO)

```
std::string foo()
{
    std::string str(1000, 'A');
    str += "NNNNNNN";

    return str;
}

main()
{
    std::string s = foo();
}
```

Named Return Value Optimization (NRVO) ve Return Value Optimization (RVO), C++ programlamasında dönüş değerini optimize etme amacıyla kullanılan iki benzer tekniktir,

RVO VS NRVO

```
MyClass createObjectWithNRVO() {
    MyClass obj;
    return obj; // NRVO etkin olduğunda, bu nesne doğrudan dönüş değerini olara
    kullanılır
}

MyClass createObjectWithRVO() {
    return MyClass(); // RVO etkin olduğunda, bu nesne doğrudan dönüş değerini
    olara
    kullanılır
}
```

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

    Myclass& operator=(const Myclass&)
    {
        std::cout << "copy assignment\n";
        return *this;
    }

    Myclass& operator=((Myclass&&)
    {
        std::cout << "move assignment\n";
        return *this;
    }

    void foo() {};
    void bar() {};
    void baz() {};
};

Myclass foo()
{
    MyClass m{365};
    std::cout << "&m = " << &m << "\n";
}
```

```

    m.foo();
    m.baz();
    m.bar();

    return m;
}

int main()
{
    Myclass nec = func(); // copy elision deniyor
    std::cout << "&nec = " << &nec << "\n";
/*
    move ya da copy ctor çağrılmadı sadece MyClass(int) ctor çağrıldı.
    m{365} ile nec aynı nesnelerde aynı adreste tutuluyorlar
*/
}

```

```

class Myclass {

public:
    std::string m_str; // 10000;
    std::vector<int> m_vec; // 2000;
};

Myclass foo();

main()
{
    Myclass m;

    m = foo(); // move assignment çağrılacak

    Myclass m1 = foo();
    // copy ve move ctor çağrılmaz
    // mandatory copy elision olur
    // copy elision olması için bir nesneyi hayata getirmemiz gerekiyor

    /*
        mandatory copy elision > move ctor > copy ctor
    */
}

```

copy elision olması için bir nesneyi hayata getirmemiz gerekiyor.

copy elision:

- 1) temporary object passing (mandatory)
 - 2) returning a temporary object (mandatory)
 - 3) returnning an obejct of automatic storage class (optimization)
-
-

static storage class

- global nesneler
- static yerel nesneler
- sınıfların static veri elemanlar (static data members)

automatic storage class

- parameters
- local variableler

dynamic storage class

- new
- delete

thread-local storage class

Dynamic Storage Class

new Fighter --> bu ifadenin türü Fighter*

void* operator new(std::size_t) standart'ta yapılan işlem

static_cast<Fighter *>(operator new(sizeof(Fighter)))->Fighter();

memory leak --> new operator ile alınan alanın geri verilmemesi

resource leak --> dtor ile yapılan işlemlerin yapılamaması (database bağlantısı kesilmemesi gibi)

```

class Myclass {
public:
    Myclass ()
    {
        std::cout << "default ctor this = " << this << '\n';
    }

    ~Myclass()
    {
        std::cout << "destructor this = " << this << '\n';
    }

    void foo() {};
    void bar() {};
};

int main()
{
    Myclass *p = new Myclass;
    Myclass *p1 = new Myclass();
    Myclass *p2 = new Myclass{};

    free(p) // undefined behavior
}

```

delete p ifadesi aşağıdaki işlemleri yapar:

1. p->~Myclass();
2. operator delete(p);

Array New

```

int main ()
{
    std::cout << "kac tam sayi:" ;
    std::size_t n;

    std::cin >> n;

    int *p = new int[n];

    for (std::size_t i{}; i < n; ++i)
    {
        p[i] = i;
    }

    for (std::size_t i{}; i < n; ++i)
    {
        std::cout << p[i] << " ";
    }

    delete [] p; // array delete
}

```

```

int main()
{
    std::cout << "sizeof(Myclass) = " << sizeof(Myclass) << "\n";
    Myclass *p = new Myclass[10]; // 10 tane Myclass nesnesi oluşur 10 ctor oluşur
    // delete p undefined behavior
    delete [] p; // 10 dtor olur
}

```

- sınıfların static veri elemanları
- sınıfların static üye fonksiyonları
- operator overloading
- namespace

Sınıfların Static Veri Elemanları

```

// sınıfların static veri elemanları

class Nec {
    static int mx;
    /*
        sınıfların static veri elemanları static anatır sözcüğü ile bildiriliyor
        bu (tanımlama olmayan) (non-definind declaration)
    */
}

//global variable'Lardan assembly açısından pek farkı yok

main()
{
    Nec::mx;
    // mx oluşturululan tüm nesneler için aynı mx aynı adresde yazılır
    Nec n1,n2;
    n1.mx = 10 // n2.mx = 10 aynı şeydir
}

```

28.08.2023

Sınıfların Static Veri Elemanları

```
// sınıfların static veri elemanları

//nec.h
class Nec {
    public:
        static int x; // derleyici x için bir yer ayırmaz. Sadece bildirimdir
};

//nec.cpp

int Nec::x{}; // static veri elemanları cpp dosyasında yapılır

// forward declaration
class Myclass;

int main()
{
    Myclass m; // incomplete type
}
```

incomplete type'lar ile neler yapabiliriz?

- fonksiyon bildirimlerinde kullanabiliriz
 - Neco foo(Neco);
 - Neco& bar(Neco&);
- type alias declaration
 - typedef Neco* NecoPtr;
 - typedef Neco& NecoRef;
- pointer ya da referans değişkenler tanımlayabiliriz.
 - Myclass *p = nullptr;
- extern bildirim yapabiliriz.
 - extern Myclass ge;
 - extern Myclass ga[];

```
class Nec;

class Myclass {
    static Nec senc; // incomplete type
}
```

```

class Data {

    int mx;
    // Data data; hatalıdır çünkü derleyici Data'nın size bilmek zorunda
    static Data data // hatalı değil size bilmesine gerek yok derleyicinin
}

```

void is a type and void is a incomplete type

Header-Only Library

```

class Myclass {

    inline static std::vector<int> x {1,23,4,50};

    static int sx; // ODR'i ihlal eder eğer inline kullanmazsam

    /*
    sınıfların static, const ve integral type veri elemanlarına sınıf içinde
    ilk değer verilebilir
    */

    static const int ck = 10; // ilk değer verilebilir
    static constexpr double x = 5.4; // implicitly inline
    static int k; // ilk değer veremeyiz
}

```

```

class Nec {
    public:
        Nec(int i): x(i) {};
        // legal değil çünkü ctor x'e ilk değer veriyor
        // x'e ilk değer veremeyiz
        void foo() const
        {
            x = 6;
            // legal ama static olmasa legal olmazdı
        }
        void func()
        {
            x = 5; // Legal
            Nec::x = 5; // Legal
            this->x = 5; // Legal
        }
        int y;
    private:
        static int x;
}

int y = 5;
int Nec::x = y;
//hatalı olur çünkü burda y class scope aranıyor global scope'ta değil
// ortada bir x değişkeni olmadığı içinde syntax hatası verir.

```

Sınıfların Static Üye fonksiyonları(Static Member Functions)

Static üye fonksiyonları

a) class scope'ta bulunurlar (global'den fark olarak)

b) sınıfın private elemanlarına erişebilirler.

-Sınıfların static üye fonksiyonları sınıfların non-static veri

elemanlarını kullanamaz

```
class Nec {
public:
    void foo(); // non-static gizli bir parametre değişkene sahiptir
    static void foo();
    void func();

    //static void foo() const syntax hatasıdır.

    static void bar()
    {
        // burada this anahtarını kullanamayız. Çünkü
        // this'e sahip değildir.

        // mx = 5; hatalıdır çünkü this pointeri yok
        // func(); // legal değil this pointeri yok

        Nec myNec;
        myNec.mx; // Legal
        myNec.func(); // Legal
    }
private:
    int mx;
}
```

```

class Nec {
public:
    static double foo()
    {
        return 3.9;
    }

    static int ival;
};

int foo()
{
    return 2;
}

int Nec::ival = foo();
int Nec::ival = ::foo(); // global'daki foo() çağrıları ival = 2
int main()
{
    std::cout << Nec::ival << "\n"; // ival = 3

    /*
        static veriyi init eden isimler önce class scope'ta aranır
    */
}

```

Named Constructor

```

class Myclass
{

public:
    static Myclass createObject();

};

int main()
{
    auto m1 = Myclass::createObject();
}

```

Overloading Constructor'a farklı bir bakış

```
class Complex
{
public:
    static Complex create_polar(double a, double d)
    {
        return Complex(a, d, 0);
    }
    static Complex create_cartesian(double r, double i)
    {
        return Complex(r, i);
    }

private:
    Complex(double r, double i);
    Complex(double a, double d);
}

int main()
{
    // mandatory copy elision
    auto c1 = Complex::create_cartesian(3.5,1.2);
    auto c2 = Complex::create_cartesian(.2352 , 4.5767);
}
```

Bir class'tan sadece dinamik nesne oluşturulmasını istiyorsak

```
class DynamicOnly
{

public:
    DynamicOnly(const DynamicOnly&) = delete;
    DynamicOnly& operator=const(DynamicOnly&) = delete;

    DynamicOnly* create_object()
    {
        return new DynamicOnly{};
    }

private:
    DynamicOnly();
}
```

Singleton Pattern (Tek Nesne Örütüsü): Bir sınıf türünden tek bir nesne oluşturabiliyor

```
class Singleton
{
    public:
        Singleton(const Singleton&) = delete;
        static Singleton* get_instance()
        {
            if (!mp)
            {
                mp = new Singleton();
            }
            return mp;
        }

        void foo();
        void bar();

    private:
        inline static Singleton *mp{};
}

int main()
{
    // Hep aynı nesneyi kullanıyoruz ve bu nesne programın sonuna kadar
    // dtor olmayacağı
    auto p = Singleton::get_instance();
    Singleton::get_instance()->bar();
    Singleton::get_instance()->foo();
}
```

Meyers' signleton

- -lazy initialization
- thread-safe

```
// Meyers' signleton

class Singleton
{
    public:
        static Singleton& instance()
        {
            static Singleton object;
            return object;
        }
    private:
        Singleton();
}
```

Hayatta olan nesnelerin sayılması

```
// Hayatta olan nesnelerin sayılması

class Myclass
{
    Myclass()
    {
        ++live_object_count;
        ++lived_object_count
    }
    ~Myclass()
    {
        --live_object_count;
    }

    static int get_live_count()
    {
        return live_object_count;
    }

private:
    inline static int live_object_count{};
    inline static int lived_object_count{};
}
```

Hayatta kalan diğer dövüşülerden yardım isticek

```
class Fighter
{
public:
    void call_fighters_for_help()
    {

    };
};

int main()
{
    Fighter f1{"Emre"};
    Fighter f2{"Mehmet"};
    Fighter f3{"Necati"};

    f3.call_fighters_for_help();
}
```

01.09.2023

Bir Fighter hayatta kalan diğer dövüşülerden yardım isticek

```
class Fighter
{
public:
    Fighter()
    {
        fvec_.push_back(this);
    }
    Fighter(std::string name, int age); name_(std::move(name)), age_(age)
    {
        fvec_.push_back(this);
    }

    ~Fighter()
    {
        //std::erase(fvec_, this);
        if (auto iter = std::find(fvec_.begin(), fvec_.end(), this); iter != fvec_.end())
            fvec_.erase(iter);
        else
            std::cerr << "this pointer cannot be found in the container\n";
    }
    Fighter(const Fighter&) = delete;
    Fighter& operator=(const Fighter&) = delete;
    void ask_for_help()
    {
        std::cout << "yetisin basım belada... \n";
        for(auto p: fvec_)
        {
            if ( p!= this && p->get_age > 15)
            {
                std::cout << p->get_name() << " ";
            }
        }
        std::string get_name() const {return name_;}
        int get_age() const {return age_;}
    }
private:
    std::string name_;
    int age_{0};
    inline static std::vector<Fighter* > fvec_;
};

int main()
{
    Fighter f1{"Emre", 50};
    Fighter f2{"Mehmet", 25};
    Fighter f3{"Necati", 19};
    auto p1 {new Fighter("ganoş", 23)};
    f3.ask_for_help();
    delete p1;

    f1.ask_for_help();
    f3.call_fighters_for_help();
}
```

Delegating Ctor (modern cpp)

Çoğu zaman sınıfların ctor'ları overload ediliyor ancak overload edilen ctor'ların bazen ortak bir kodu oluyor. Unutmayınız ki kod tekrarı her türlü belayı beraberinde getirir eski Cpp'de ctor'ların ortak bir kodu bir fonksiyon şekline tanımlanıyor ve ctor'lar bu fonksiyon çağrıyordu

```
class Myclass
{
public:
    Myclass(int) : Myclass(x, x, x)
    {
        //common_code();
    };

    Myclass(int, int) : Myclass(x, x, 0)
    {

    }
    Myclass(int, int, int) : mx(x), my(y), mz(z)
    {
        //common_code();

        //belki ortak kod
    };
    Myclass(const char *p) :Myclass(std::atuo(p))
    {
        //common_code();
    };

private:
    common_code();
    int mx, my, mz;
}
```

Friend Bildirimleri

friend bildirimleri (hemen her zaman sınıfın kendi kodlarına veriliyor)

Sınıfın:

- a) global fonksiyonları (free functions)
- b) yardımcı türler

Örnek1)

```
class Myclass
{
    public:
        // friend bildirim private veya public yapılabılır. Bir fark yok
        friend void ff(Myclass);
    private:
        int mx{};
        void foo(Myclass);
};

void ff(Myclass m)
{
    m.foo();
    Myclass myc;
    myc.mx = 5;
    // private olmasına rağmen eriştiğ
```

Örnek2)

```
class Myclass
{
    public:
        // hidden friend
        friend void bar(Myclass, int x )
        {
            return x * x; // bu fonksiyon class'ın memberi değil
        }
    private:
        int mx{};
        void foo(Myclass);
};
```

Örnek3)

```
class Erg
{
    public:
        Erg(int);
        void foo(int);
};

class Nec
{
    private:
        friend void Erg::foo(int);
        friend Erg::Erg(int); // ctor friend'Lik verebiliriz.
        int mx;
}

void Erg::foo(int x)
{
    Nec necx;

    necx.mx = x; // private eriştil
}
```

Bir sınıf bir başka sınıfa "frined"lik verebilir. Bu durumda friend bildirime konu sınıf incomplete type olabilir.

```
class Nec
{
    private:
        friend class Erg; // Erg bildirimi olmaması rağmen friend'Lik verdik
}
```

- 1) A sınıf'ı B sınıfına friend'Lik verirse A sınıfı B'nin private bölümüne erişemez.
- 2) A B'ye friend'luk vermiş olsun
B C'ye friend'Luk vermiş olsun
C A'nın private bölümüne erişemez. Yani friendlilik geçişken değildir.
- 3) Base sınıfından Der sınıfı kalıtım yoluyla elde edilmiş olsun. Base sınıfı global foo işlevine friend'luk vermiş olsun. foo işlevi Der sınıfının private bölümüne erişemez.
- 4) Bazen ihtiyaç olsa da bir sınıf kendi seçilmiş öğeleri için bir başka kodda friend'luk veremez. Böyle durumarda bazı idioms/ techniques kullanabilir. (attorney)

Operator Overloading

- 1) Run time maliyeti yok tamamen derleme zamanında çalışır.
- 2) Keyfi isimlendirme yok. isim operator anahtar sözcüğünü içerecek
 - a. operator+
 - b. operator!
 - c. operator++
 - d. operator==
- 3) Olmayan bir operator overload edilemez
 - a. a @ b // böyle bir operator olmadığı için @ ile yapamayız
- 4) Operandlar en az birinin user defined olmalı
- 5) Global operator function ya da member operator function olmalı
- 6) Her operator overload edilemiyor. Dilin kuralları bazı opearatörlerin overload edilmesini yasaklıyor.
 - a. . operatorü overload edilemiyor
 - b. ?: (ternarny operator overload edilemiyor)
 - c. sizeof operator overlaod edilemiyor
 - d. .* operatörü (C'de olmayan Cpp dilinde olan bir operator)
 - e. typeid operatoru
- 7) Bazı operatorler yalnızca member operator function overload edilir.
 - a. function call operator (fonksiyon çağrı operator)
 - b. subscript operator a[b]
 - c. assignment operators
 - d. tür dönüştürme operatorleri
 - e. -> arrow operatör
- 8) Bir istisna hariç operator fonksiyonları "default arguman" alamıyor (function call operatör)
- 9) Bu mekanizmada operatör öncelik seviyesi ve operator öncelik yönü değiştirilemiyor
 - a. $a * b + c > 10$ denklemi böyle yazılıyor $((a * b) + c) > 10$ ve bu öncelik değiştirilemiyor.
- 10) Bütün operatör fonksiyonları isimleriyle çağrılabiliyor.

```
a) a + b ya da operator+(a + b)

b) class MyClass
{
    ;
    main()
    {
        MyClass a, b, c;
        a = b; // a.operator(b)
    }
}
```

Bu mekanizmada ile operatörlerin "arity" sini değiştiremeyiz

arity ==> operator unary ya da binary olmasına

```
a > b // binary çünkü 2 variable aldı
!a // unary tek variable aldı

bool operator >(MyClass); // hatalı tek parametre var
bool operator >(MyClass, MyClass); // hata yok
bool operator >(MyClass, MyClass, MyClass); // hatalı 3 parametre var
```

Eğer member operator function ise

```
a > b MyClass::operator>;
a.operator>(b); // yani sol operator this'i alır

class MyClass {

public:
    bool operator>(MyClass) const;
    //burda hata olmaz çünkü sol operand this'tir
    bool operator>(MyClass, MyClass) const; // hatalıdır

    bool operator!() const; // this operand olarak kullanılır sadece
    bool operator!(!MyClass) const; // hatalıdır
}
```

04.09.2023

Global Operator Function vs Member Operator Function

```
/*
global operator function
    a>b - operator>(a, b)
    !x operator!(x)

*/
/*
member operator functions
non-static member function

a>b      myClass::operator>
          a.operator>(b);

x /y      x.operator/(y);

!x       x.operator!();

a*b      operator*(a, b);

~x       operator~(x);

*/
```

```
class Nec
{
    //Member operator function

    Nec operator*(const Nec&) const;
    Nec operator/(const Nec&) const;
    Nec operator+(const Nec&) const;
    Nec operator>(const Nec&) const;

};

//Global operator function
Nec operator*(const Nec&, const Nec&);
Nec operator/(const Nec&, const Nec&);
Nec operator+(const Nec&, const Nec&);
Nec operator>(const Nec&, const Nec&);

main()
{
    Nec n1, n2, n3, n4;
    auto b = n1 * n2 + n3 / n4 > n5;

    //Global operator function (derleyicinin çevirdiği)
    // operator>(operator+(operator*(n1,n2), operator/(n3,n4)) , n5);

    //Member operator function (derleyicinin çevirdiği)
    // n1.operator*(n2).operator+(n3.operator/(n4)).operator>(n5);
}
```

Operator Overloading'te Function Overloading

```
class Nec {  
public:  
    Nec operator+()const; // burada "+" işaret operator'u  
    Nec operator+(const Nec&)const; // burada "+" toplama operator'u  
    // function overloading var  
}
```

Neden member operator function ve global operator function ayrı ayrı var?

```
class Nec {  
  
public:  
    Nec operator+(int) const;  
  
};  
int main()  
{  
    Nec myNec;  
  
    auto x = myNec + x; // bunda hata yok çünkü myNec sol tarafta  
    x = 5 + myNec // hatalı  
}
```

```
class Matrix  
{  
  
};  
  
std::ostream& operator<<(std::ostream& os, const Matrix&);  
  
int main()  
{  
  
    int ival = 10;  
  
    cout << ival;  
    cout.operator<<(ival);  
  
    Matrix m;  
    cout << m; // normalde hatalı ama operator<< ile mümkün kıldık  
  
    operator<<(cout, m).operator<<(ival);  
}
```

Const Correctness & Operator Functions

```
class Matrix
{
public:
    Matrix operator*(const Matrix&) const;
    // m1 ve m2 böyle const oldu

};

Matrix operator*(Matrix&, Matrix&); // uygun değil
Matrix operator*(const Matrix&, const Matrix&);
// m1 ve m2 değişmiyor bu yüzden const olmalı
main()
{
    Matrix m1, m2;

    m1 * m2;
}
```

Operator Overloading ile Value Category İlişkisi

```
Bigint& operator+(const Bigint&, const Bigint&) // hatalı
{
    Bigint result; // otomatik ömürlü nesneyi ref döndürüyoruz
    static Bigint result; // static ömürlü olunca hep aynı nesneyi döner
    // (x + y) * (a + b) x + y ile a + b aynı nesne

    Bigint* result = new Bigint;
    // delete edilemez

    // Code makes result

    return result;
}
```

```
Bigint operator+(const Bigint&, const Bigint* );
/*
    performansı etkilemez çünkü copy elision ya da
    taşıma semantiği olacak
*/

/*
    a = b;
    x+=y;
    ++x;

    L value expr olduğu için L value ref dönmesi gerekiyor
*/
```

```
class Bigint {
public:
    Bigint operator+(const Bigint&) const; // R value
    Bigint& operator=(const Bigint&); // L value

    bool operator==(const Bigint&) const;
};

main()
{
    Bigint b1, b2;

    auto flag = b1 != b2; // Cpp 20 ile geçerli oldu
}
```

Özel bazı durumlar söz konusu değilse

- binary simetrik operatorler

- global operator fonksiyonu olarak
- -(a < b)
- -(a + b)

bazen global operator fonksiyonu class'a friend olarak yaparak sınıfın private elemanlarını erişme imkanımız olabilir.

```
int main{

    vector vec = {1, 2, 3};

    auto val = vec.front()++; // hata olmaz

    const vector v1 = {1, 2, ,3 };

    val = v1.front()++; // hatalı olur
    /*
        Çünkü front fonksiyonun geri dönüş değeri const oldu
        const overloading deniyor
    */
}
```

```
/*
yıldız dereferencing / indirection
. dot operator
-> arrow operator

func fonksiyonun parametre değişkenin türü int ref ref
fonksiyonun içindeki x ifadesinin türü int
(bir ifadenin türü refarasın türü olamaz)
x ifadesinin value category'si ise l value

*/
void func(int&& x)
{
    x
}
```

```

void bar(int &)
{
    std::cout << "1"; // ikinci olarak buraya gelir
    /*
        x ifadesinin value category'si l value olduğu için
        bu fonksiyon çalışır
    */
}

void bar(int&&)
{
    std::cout << "2";
}

void foo(int&& x)
{
    bar(x); // birinci olarak buraya gelir
    // x ifadesinin value category'si l value
}

main() {
    foo(5);
}

```

Arrow Operator Overloading Fonksiyon

```

class Myclass
{
    public:
        void foo();
        void bar(int);
};

class PointerLike
{
    public:
        PointerLike(Myclass*);
        Myclass* operator->();
};

main()
{
    PointerLike p = new Myclass;

    p->foo();
    p.operator->()->foo();

    p->bar(12);
    p.operator->()->bar(12);
}

```

2023 09 06

```
// operatr ++ ve -- overload edilmesi
/*
    increment
    decrement

    ++x      prefix increment
    x++      postfix increment

    --x      prefix decrement
    x--      postfix decrement

    int y = 10;
    auto b = y++;  b = 10

    y = 10
    auto z = ++y;  z == 11

*/
```

```
class Counter {
public:
    Counter& operator++(); // prefix
    Counter operator++(int); // postfix

    Counter& operator--(); // prefix
    Counter operator--(int); // postfix
};

int main()
{
    int a[] = { 1, 2, 3, 4, 5 };
    int *p = a;

    std::cout << *p++ << "\n"; // 1
    std::cout << *p << "\n"; // 2
    std::cout << *++p << "\n"; // 3
}
```

operator + ve – overload

```
// operator + ve - overload
class Nint
{
public:
    Nint operator+()const
    {
        return *this;
    }
    Nint operator-()const
    {
        return Nint(-mx);
    }
};
```

operator[]

```
class String {
public:
    String(const char *p) : mp{ new char[strlen(p) + 1] }
    {
        std::strcpy(mp, p);
    }
    std::size_t length() const
    {
        return std::strlen(mp);
    }
    friend std::ostream& operator<<(std::ostream& os, const String& s)
    {
        return os << " " << s.mp << "\n";
    }
    char& operator[](std::size_t idx)
    {
    }
private:
    char *mp;
};

int main()
{
    String str{"mustafa demirhan"};
    std::cout << str[1] << "\n";

    str[0] = '!';

    for (size_t i{}; i < str.length(); ++i)
    {
        std::cout << str[i] << ' ';
    }
}
```

08.09.2023

Arrow operator overloading

```
class Myclass
{
public:
    void foo()
    {
        std::cout << "Myclass::foo()\n";
    }
    void bar(int)
    {
        std::cout << "Myclass::bar(int)\n";
    }
}

class MyPtr {

public:
    Myclass* operator->();
}

int main()
{
    Myclass m;
    Myclass* p {&m};
    MyPtr myPtr;

    p->foo();
    p->bar(12);

    myPtr->foo();
    //myPtr.operator->()->foo();
    myPtr->bar(12);
    //myPtr.operator->()->bar(12);
}
```

Örnek:

Bir DatePtr nesnesi new ifadesi ile oluşturulmuş dinamik ömürlü bir Date nesnesini gösterir ancak DatePtr nesnesinin hayatı bittiğinde gösterdiği Date nesnesini delete eder.

```

class Date {

public:
    Date(int d, int m, int y) : md{d}, mm{m}, my{y}
    {
        std::cout << "*this" << "değerindeki" << this << " adresinde"
    }
~Date();
friend std::ostream& operator<<(std::ostream& os, const Date& dt);
int get_month_day() const;
int get_month() const;
int set_year(int);
int set_month();
private:
    int md{1}, mm{1}, my{1900};
};

/*
bir DatePtr nesnesi new ifadesi ile oluşturulmuş
dinamik ömürlü bir Date nesnesini gösterir
ancak DatePtr nesnesinin hayatı bittiğinde gösterdiği Date nesnesini
delete eder
*/
class DatePtr {
public:
    DatePtr = default;
    explicit DatePtr(Date *p);
    ~DatePtr() {
        if (mp)
            delete mp;
    }

    DatePtr(const DatePtr&) = delete;
    DatePtr& operator=(const DatePtr&) = delete;

    Date& operator*()
    {
        return *mp;
    };
    Date* operator->(){
        return mp;
    };
private:
    Date* mp {nullptr};
};

int main()
{
    std::cout << "main başladı\n";
    if (true)
    {
        DatePtr p {new Date( 3, 5, 1987)};
        std::cout << *p << "\n";
        std::cout << "ay = " << p->get_month() << "\n";
        p->set_year(2022);
    }
    // Date nesnesi hayatı veda eder
    std::cout << "main sona eriyor\n";
}

```

Overloading function call operator ()

C'de foo(3, 5) üç ayrı varlık kategorisinden birine ait olması gerekiyor

Bunlar:

- fonksiyon ismi (function designator)
- fonksiyon pointer
- function-like macro

Kurallar:

- fonksiyon ismi operator() olmalı
- üye fonksiyon olmak zorunda
- varsayılan arguman alabilir diğer operator overloadingler alamaz
- ismiyle çağrılabılır
- overload edilebilir

```
// Function object
class Myclass
{
public:
    void operator()()
    {
        std::cout << "Myclass::operator()() this:" << this << "\n";
    }
    void operator()(int x = 5 );
}

int main()
{
    Myclass m;
    std::cout << "&m = " << &m << "\n";

    m();
    //m.operator()()
```

Örnek:

```
class Random {
public:
    Random(int low, int high) : mlow{low}, mhigh{high} {}
    int operator()()
    {
        return std::rand() % (mhigh - mlow + 1) + mlow;
    }
private:
    int mlow, mhigh;
};

int main()
{
    Random rand1 { 45, 72 };

    for (int i = 0; i < 10; i++)
    {
        std::cout << rand1() << "\n";
    }
}
```

Tür dönüşüm operatörleri

Kurallar:

- const fonkisyon olmalı
- tür dönüşüm değeri yazılmaz ama o türü dönüşür
- her türlü dönüşüm türüne uygun (class, pointer, double vb.)

```
class Nec
{
public:
    operator int() const;
};

int main()
{
    Nec myNec;
    int ival {0};
    ival = myNec;

    ival = myNec.operator int();
}
```

User Define Conversion:

- conversion ctor
- operator T

Standart Conversion -> User Define Conversion

User Define Conversion -> Standart Conversion

User Define Conversion -> User Define Conversion // bu olmaz

Conversion Ctor

```
class A {  
};  
class B  
{  
    public:  
        B();  
        B(A);  
}  
  
class C  
{  
    public:  
        C();  
        C(B);  
}  
  
int main()  
{  
    B bx;  
    A ax;  
    bx = ax; // user define conversion  
  
    cx = ax; // hatalı çünkü User Define Conversion -> User Define Conversion  
}
```

Operator Conversion

```
class Nec
{
public:
    explicit operator int() const;
    /*
    explicit kullanabiliriz böyle tür dönüşüm operator
    kullanamk zorunda kalırız.
    */
};

int main()
{
    double dval{};

    Nec mynec;

    dval = mynec // explicit olduğu için hatalı
    /*
        önce mynec -> int olacak user define conversion
        sonra int -> double olacak standart conversion
    */
    // geçerli
    ival = static_cast<int> mynec;
    ival = (int)mynec;
    ival = int(mynec)
}
```

Example:

```
class Counter
{
public:
    Counter() = default;
    Counter(int val) : mc{val} {}
    Counter& operator++()
    {
        ++mc;
        return *this;
    }
    Counter operator++(int)
    {
        Counter temp(*this)
        ++*this;
        return temp;
    }

    operator int()const
    {
        return mc;
    }

private:
    int mc{};
}
void bar(int)
{
}

int foo()
{
    Counter c {632}
    return c;
}

int main()
{
    Counter c { 4 };

    for (int i = 0; i < 10; ++i)
    {
        ++c;
    }

    int ival = c;
    //int ival = c.operator int()

    foo(); // hatasız çalışır
    bar(c) // hatasız çalışır
}
```

Operator bool

```
class MyClass
{
public:
    operator bool() const
    {
        return true;
    }
};

int main()
{
    MyClass m1, m2;

    auto int x = m1 + m2;
    //auto int x = m1.operator bool() + m2.operator bool();

    cout << "x = " << x << "\n"; // x = 2
}
```

Sınıfların operator bool fonksiyonları hem her zaman explicit olmak zorundadır.

```
int main()
{
    unique_ptr<int> uptr { new int{35} }
    // operator bool fonksiyon'a sahiptir unique_ptr
    if(uptr)
    {
        //
    }
    else {
        //
    }
}
```

2023.09.13

Namespace (İsim alanları)

```
void x();
int main()
{
    ::x() // :: scope resolution operator
}
```

Namespace nasıl oluşturabiliriz:

- Bir namespace içinde olmalıyız
- Bir fonksiyonun içinde olmaz
- Bir sınıf tanımı içinde olmaz
- Sonuna noktalı virgül koymamamız gereklidir

```
// namespace scope
namespace nec
{
    int x = 10; // namespace içinde tanımlanmış global bir değişken

    void f(int);
    void f(double);
    // function overloading
}
```

```
namespace a
{
    int x;
}

namespace b
{
    int x;
}

int main()
{
    a::x = 5;
    b::x = 4;
}
```

```
// ikisi de aynı namespace kümülatif olarak birikirler
namespace emre
{
    int a;
    int b;
}
namespace emre
{
    int c;
}
```

```
// unnamed namespace
namespace
{
    int a, b, c;
}
```

Bir isim alanı içindeki bir ismin nitelenmeden bulunması için:

- using declaration
- using (namespace) directive
- ADL (argument dependent lookup) argümana bağlı isim arama

Using declaration:

```
using std::cout, std::cin;
```

- 1) Using bildiriminin bir kapsamı vardır o kapsam içinde etkin olur.
- 2) Bildirime konu isim bildiriminin yapıldığı isim alanına enjekte edilir ve o kapsamda tanımlanmış gibi etki eder.

```
namespace emre
{
    int x = 5;
}

int main()
{
    using emre::x;
    int y = x;
}
```

```
namespace ali
{
    int x;
}

namespace veli
{
    using ali::x;
}

int main()
{
    using veli::x;
}
```

Using namespace directive

using namespace directive bir bildirimdir. Tüm bildirimlerde olduğu gibi bu bildirimin de bir kapsamı vardır ve bu bildirim bildirimin kapsamında etkindir.

```
namespace ali
{
    int x, y, z;
}

int main()
{
    using namespace ali;

    x = 5;
    y = 5;
    z = 456;
    // main scope içinde böyle kullanabiliriz
}
```

```
namespace ali
{
    int x = 99, y, z;
}

int main()
{
    using namespace ali;
    std::cout << x << "\n"; // 99 yazar
    int x = 10;
    std::cout << x << "\n"; // 10 yazar
}
```

```
namespace ali
{
    int x = 99;
}

namespace veli
{
    int x = 99;
}

int main()
{
    using namespace ali;
    using namespace veli;

    x = 4; // hatalıdır ambiguous olur
}
```

```
namespace ali
{
    void foo(int);
}
namespace veli
{
    void foo(int, int);
}

int main()
{
    using namespace ali;
    using namespace veli;

    foo(4);
    foo(4, 4);

    // function overloading olur
}
```

```

int g = 10;

namespace ali
{
    using ::g; // global scope'taki g yi tanımlamadık
}
namespace veli
{
    using ali::g;
}

int main()
{
    ++g; // 11
    ++ali::g; // 12
    ++veli::g; // 13

    std::cout < "g = " << g << "\n"; // 13
}

```

ADL (argument dependent lookup) argümana bağlı isim arama

Bir fonksiyona nitelenmemiş bir isimle çağrı yapıldığında eğer fonksiyona gönderilen argümanlardan biri bir namespace içinde tanımlanan türé ilişkin ise söz konusu fonksiyon ismi o namespace içinde de aranır.

```

namespace Nec
{
    class Erg
    {

    };
    void foo();
    void bar(int);
    void baz(Erg);
}

int main()
{
    nec::Erg e;
    baz(e) // hata yok

    foo(); // hatalıdır
    bar(12); // hatalıdır

    endl(std::cout)
    // endl std namespace içinde de aranır
}

```

```

namespace neco
{
    class Nec
    {
        public:
            void foo(int);
    }

    void func(Nec);
}

#include "nec.h"

int main()
{
    neco::Nec x;
    x.foo(12);
    func(x);
}

```

```

namespace ali::veli::can
{
    int x = 5;
}

namespace ali
{
    int a;
}

namespace ali::veli
{
    int v;
}

// inline with namespace
namespace ali
{
    inline namespace old_version
    {
        class Myclass {};
    }

    namespace new_version
    {
        class Myclass{};
    }
}

int main()
{
    ali::Myclass // old_version namespce'deki MyClass kullanılır
}

```

2023.09.15

Operator Overloading Enum

```
enum class Weekday
{
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
Weekday& operator++(Weekday& wd)
{
    using enum Weekday; // Cpp20
    return wd = wd == Saturday ? Sunday :
        static_cast<Weekday>(static_cast<int>(wd) + 1);
}
Weekday operator++(Weekday& wd, int)
{
    Weekday temp {wd};
    ++wd;

    return temp;
}

std::ostream& operator<<(std::ostream&, const Weekday&)
{
    static const char* const pwdays[] = {"Sunday", "Monday",
                                         "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};

    return os << pwdays[static_cast<int>(wd)];
}

int main()
{
    Weekday wd { Weekday::Friday};

    // ++wd; operator overloading ile olur
}
```

Nested classes

```
class Myclass
{
    class Nec; // Nested classes

    //Nested type or type member
    typedef int value_type;
    using value_type = int;
    enum Color {Blue, Red, Green};
}
```

- Nested type or type memberları class'ın içinde tanımladığımız da class scope'ta oluyor namespace scope yerine
- Class scope sayesinde erişim kontrollu kazanmış oluyoruz.
- Logic açından bulunduğu class ile ilişkisel olduğunu belirtmiş olduk.

```
//nec.h
class Nec
{
public:
    class Erg;
private:
    void foo(Erg);
    Erg bar();
}

void Nec::foo(Erg e)
{
    // Hata yok
}
Erg Nec::bar()
{
    /*
    Hatalı çünkü Erg class scope'ta aranmaz ama
    Nec::Erg Nec::bar() diye tanımlasak hata almayız
    */
}
```

```
struct ValueType{};

class Myclass
{
    void foo()
    {
        ValueType = x; // int türünde olur class scope'ta aranır
    }
    ValueType mx; // struct ValueType türünde int türünde değil
    typedef int ValueType;

    class Nec; // incomplete type
}

class Nec {} // Class'ın içinde tanımlanan Nec ile bu Nec farklı
```

```

class Myclass
{
    static void mbar();
    int mx;
    class Erg
    {
        public:
            void bar();
        private:
            void foo()
            {
                // Modern cpp ile geldi
                Myclass::mbar(); // mbar private olmasına rağmen erişebildik
                auto sz = sizeof(mx)
            }
    };
    void func()
    {
        Erg e;
        e.foo(); // kendi nested class'ımızın private bölümüne erişemez
    }
};


```

```

class Myclass
{
    class Nested
    {

    };

    public:
        static Nested foo();
};

int main()
{
/*
    MyClass::Nested access kontrolden kaynaklı hatalı oluyor
    ama auto x hata yok çünkü adını yazmadık MyClass::Nested
    access kontrol isim aramayla ilgili eğer adını yazmayıp auto kullanırsam
    acces kontrole takılmayız.
*/
    Myclass::Nested x = Myclass::foo(); // hatalı
    auto x = Myclass::foo();
}

```

```
//system.h
class System
{
public:
    class Element
    {
        void foo();
        /*
            foo fonksiyonun tanımı inline olarak
            sınıfın içinde yapabiliriz

            System sınıfın içinde foo fonksiyonun tanımını
            yapamayız

            system.cpp içinde tanımlayabiliriz void System::Element::foo();
        */
        Element& operator=(const Element&);
        Element(const Element&);
    };

    void Element::foo(){}; // hatalı
}

//system.cpp

void System::Element::foo(); // böyle tanımlayabiliriz.
System::Element& System::Element::operator=(const Element& other)()
{
    return *this;
}

System::Element::Element(const Element&);
```

Design Pattern (pimpl idiom)

Düzen isimleri:

- pimpl idiom (pointer to implementation)
- handle-body idiom
- cheshire cat
- compiler firewall
- opaque pointer

Bu pattern sınıfın private bölümünü client kodlardan gizlemeye yönelik

Neden sınıfımızın private bölümünü gizlemek isteriz?

1. Compilation time kısaltır çünkü A, B classları için başlık dosyalarını include etmek zorundayız eğer private gizlersek include etmeyiz ve time kisılır.
2. Eleman eklemek, çıkarmak veya tanımlama sırasını değiştirmek tamer.h include eden kodların hepsi yeniden derlenmesi gereklidir. Eğer elemanları gizlersek tamer.h yapılabilecek değişikler tamer.h kullanan kodları derlenmesine gerek kalmıcaz
3. implementation hakkında bilgi vermemek

Pimpl Idiom örnek

```
// tamer.h
class Tamer
{
public:
    Tamer();
private:
    A ax;
    B bx;
    int ival;
}

// pimpl idiom uygulaması (eski - tip)
// tamer.h
// Maliyeti olan bir işlem
class Tamer
{
public:
    void foo();
    void bar();
private:
    class pimpl;
    pimpl *mp; // smart pointer kullanmak daha mantıklı
}

// tamer.cpp

struct Tamer::pimpl
{
    A ax;
    B bx;
    int ival;
}

Tamer::Tamer() : mp{new pimpl{}};

void Tamer::foo()
{
    mp->ax;
    mp->bx;
}
```

Sınıfın veri elemanlarını başka bir sınıf üzerinden kullanıyoruz yani pimpl sınıfını kullanıyoruz Tamer class'nın data memberlerine erişmek için

Inheritance (Kalıtım)

Kalıtımda kaynak olarak kullanılan sınıfa base (taban). Üretilen sınıfa da derived (türemiş) ismi verilir. (Cpp'da)

3 Ayrı katılım kategorisi var:

- public inheritance
- private inheritance
- protected inheritance

Multiple Inheritance: Bir sınıf tek bir katılım işlemiyle birden fazla taban sınıfının interfacesini alabilir.

Kalıtımı mekanizması ile bir sınıf oluşturmamız için bir complete type a ihtiyacımız var.

```
class Base;
class Der : public Base{}; // public private protected gibi anahtar sözcükleri
                     kunnen gebruikt worden

class Der : Base {}; // private olarak Base classının inheritance etti
struct Der : Base {}; // public olarak Base classının inheritance etti
```

```
class Base
{
public:
    void foo();
    void bar();
    void baz();
};

class Der : public Base
{

};

int main ()
{
    Der myder;
    myder.foo();
    myder.bar();
    myder.baz();

    Der der;

    Base* p = &der;
    Base& br = der;
}

// Bu fonksiyonlara Base sınıfını kalıtım yoluyla elde etmiş sınıflar
// arguman olarak verilebilir
void base_func_ptr(Base* p);
void base_func_ref(Base& p);
```

2023.09.18

Kalıtım Inheritance

```
// Kalıtım

class Base
{
    int x {};
    int y {};
}
class Der : public Base // base object inheritance
{
    int a{};
    int b{};
    Base b // member object containment
}

int main()
{
    std::cout << "sizeof(Base) = " << sizeof(Base) << "\n"; // 8
    std::cout << "sizeof(Der) = " << sizeof(Der) << "\n"; // 16

    // Der classı Baseden inh edildiği için onu size'nida kapsar
    // Her oluşan Der nesnesinin içinde Base object vardır
}
```

```
class Base
{
private:
    int mx;
    void foo();
protected:
    int my;
    int func();
}
class Der : public Base
{
    void bar()
    {
        // syntax hatası var çünkü Base class'ın private bölümünde erişemeyiz
        foo()
        mx = 5

        // protected kısmına erişebiliriz
        my = 6;
        func();
    }
}
```

```

/*
  Dikkat
  Taban sınıfı ve türemiş sınıfı bildirilen aynı isimli fonksiyonlar
  overload değildir
*/
class Base
{
    public:
        void foo(int);
};

class Der : public Base
{
    public:
        void foo(double);
};

```

Kalıtım ve İsim Arama

```

class Base
{
    public:
        void foo();
};

class Der : public Base
{
    public:
        void foo(int);
};

int main()
{
    Der myder;
    myder.foo(); // hatalı
    /*
        İsim arama türemiş sınıf içinde bitti. Türemiş sınıfın içinde
        foo(int) fonksiyonunu derleyici bulur. Hatanın sebebi ise deallocate
        Gerekli int argumanı foo() fonksiyonuna göndermediğimizden kaynaklı
    */
    myder.foo(12); // Der sınıfının foo'su
    myder.Base::foo(); // Base class'ın foo'su
}

```

Türemiş sınıfın bir üye fonksiyonu içinde bir isim nitelenmeden kullanılırsa

- isim önce blokta
 - sonra kapsayan blokta
 - sonra türemiş sınıfın tanımında
 - sonra taban sınıfın tanımında
 - sonra namespace'de aranır

```
int x;
class Base
{
    public:
        int x;
};

class Der : public Base
{
public:
    void foo(){}
    {
        int x;
        auto y = x; // kapsamında içindeki x
        // Der deki x
        y = Der::x;
        y = this->x;

        // Base deki x
        y = Base::x
        //Global x
        y = ::x
    }

    int x;
};
```

```
class Base
{
    public:
        void foo(int);
};

class Der : public Base
{
public:
    void foo(int)
    {
        foo(4); // recursive call
    }
};
```

Kalıtımda Using Bildirimi

```
class Base
{
    public:
        void foo(int);
        void foo(double);
};

class Der : public Base
{
    public:
        using Base::foo; // bu bildirim ile foo fonksiyonları aynı kapsamaa geldi
        void foo(long);
};

int main()
{
    Der myder;
    // using Base::foo bildirimimle bütün foo() fonksiyonları overload olmuş oldu
    myder.foo(1.2); // Base::foo(double)
    myder.foo(1); // Base::foo(int)
    myder.foo(12L); //Der::foo(Long)
}
```

```
// multi level inheritance
class A
{

};

class B : public A
{

};

class C : public B
{

};
// C açısından B 'ye baktığımızda direct Base class (immediate Base class)
// C açısından A ise indirect Base class
```

```

// multi level inheritance
class A
{
};

class B : public A
{
};

class C : public B
{
};

// C açısından B 'ye baktığımızda direct Base class (immediate Base class)
// C açısından A ise indirect Base class

```

Der türemiş bir sınıf olsun

Bir Der nesnesi hayata geldiğinde:

- 1) önce Der içindeki base class object hayata gelecek. (ctor çağırılacak)
- 2) Sonra bildirimdeki sırayla member object'ler hayata gelecek
- 3) sonra Der sınıfının ctor ana bloğuna girecek

Bir Der nesnesinin hayatı bittiğinde:

- 1) Der sınıfın dtor çağırılacak
- 2) En son bildirimden başlayarak member objectlerin dtor çağırılacak
- 3) Base sınıfın dtor çağırılacak

```

class Base
{
    public:
        Base(int); // durum 1
        Base() = delete; // durum 2
    private:
        Base(); // durum 3
};
class Der : public Base
{};
int main()
{
    Der myder; // hatalı
    /*
        Eğer sınıfın special member fonksiyonu derleyici implicitli
        declared edip default ederse ama yazacağı kodda syntax hatası olursa
        default etmesi gereken ctor delete eder.
        Yani Der class'nın ctor delete edilmiş durumda
    */
}

```

```

class Base
{
    Base(int x, int y);
    Base(double dval);
};

class Der : public Base
{
    Der() : Base{10, 20}
    {

    };

    Der (double dval) : Base{dval}
    {
        ;
    };
};

```

```

class Member
{
public:
    Member(int x)
    {
        std::cout << "Member(int x) x = " << x << "\n";
    }
};

class Base
{
public:
    Base (double d)
    {
        std::cout << "Base (double x) d = " << d << "\n";
    };
};

class Der : public Base
{
public:
    Der() : mx(20), Base(3.4) // Buradaki sıranın önemi yok
    {
        std::cout << "Der default ctor\n";
    }
private:
    Member mx;
};

int main()
{
    Der myder;
}

/*
1) "Base (double x) d = " yazısını görücez
2) "Member(int x) x = " yazısını görücez
3) "Der default ctor\n" yazısını görücez
*/

```

```

class Base
{
    public:
        void foo();
};

class Der : public Base
{};

int main ()
{
    Der myder;
    Base *p = &myder;
    Base&r = myder;

    Base mybase = myder; // (object slicing) bu yapılmamalı

    myder.foo();
    /*
        Aslında burda foo()'un içinde Base* argumanı alıyormuş gibi düşünüyor
        derleyici
    */
}

```

Kalıtımda special fonksiyonların durumları:

Copy and Move Ctor

```

class Base
{
    public:
        Base()
        {
            std::cout << "Base default ctor \n";
        }
        Base(const Base&)
        {
            std::cout << "Base copy ctor \n";
        }

        Base(Base &&)
        {
            std::cout << "Base move ctor\n";
        }
};

class Der : public Base
{
    Der()
    {
        std::cout << "Der default ctor \n";
    }
    Der(const Der&)
    {
        std::cout << "Der copy ctor \n";
    }
};

```

```
int main()
{
    // Base default ctor
    // Der default ctor
    // Base default ctor
    // Der copy ctor
    Der ader;
    Der bder(ader);
}
/*
Eğer türemiş bir sınıf için copy ctor yazarsak türemiş sınıfın copy ctor'unda Base sınıfın copy ctor çağrılmasından biz sorumluyuz.

Der(const Der&) : Base() // derleyici Base'in default ctor çağrıları
{
    std::cout << "Der copy ctor \n";
}

Der(const Der&) : Base(other) // Base'in copy ctor çalışması için
{
    std::cout << "Der copy ctor \n";
}

Move ctor içinde:
Der(const Der&) : Base(std::move(other)) // Base'in move ctor çalışması
için
{
    std::cout << "Der copy ctor \n";
}

*/
```

Copy and move assignment

```
class Base
{
public:
    Base& operator=(const Base&)
    {
        std::cout << "Base copy assignment \n";
        return *this;
    }

    Base& operator=(Base &&)
    {
        std::cout << "Base move assignment\n";
        return *this;
    }
};

class Der : public Base
{
public:
    Der& operator=(const Der& other)
    {
        Base::operator=(other);
        std::cout << "Der copy assignment \n";
    }

    Der& operator=(Der&& other)
    {
        Base::operator=(std::move(other));
        std::cout << "Der move assignment \n";
    }
};

int main()
{
    Der d1, d2;

    d1 = d2;
    d1 = std::move(d2);
}
```

2023.09.20

dynamic binding or late binding

Bir base sınıfın fonksiyonları:

- 1) Türemiş sınıflara hem bir interface (arayüz)

hemde bir implementasyon veren fonksiyonlar

- 2) Türemiş sınıflara hem bir interface (arayüz)

hemde bir default implementasyon veren fonksiyonlar

- 3) Türemiş sınıflar yalnızca bir interface veren implementasyon

vermeyen fonksiyonlar

- Bir sınıfın en az 2. maddeki gibi bir fonksiyonu varsa böyle sınıflara polymorphic (çok biçimli) sınıf denir.
- Bir sınıfın en az 3. maddeki gibi bir fonksiyonu varsa böyle sınıflara abstract (çok biçimli) sınıf denir.
- Eğer bir sınıf abstract sınıf ise o sınıfın bir nesne oluşturamayız

```
class Airplane
{
    public:
        void takeoff(); // 1. maddeye örnek
        virtual void fly(); // 2. maddeye örnek
        virtual void land() = 0; // 3. maddeye örnek (pure virtual function)
};

class Airbus : public Airplane
{

};

void flygame(Airplane ); // abstract class'ta geçerli değil
void flygame(Airplane* ); // geçerli
void flygame(Airplane& ); // geçerli
```

override ve final keywordleri contextual keyword (bağlamsal anahtar sözcük)

Eğer bir fonksiyon çağrıSİ taban sınıfından bir nesne ile değil taban sınıfından bir pointer değişken ya da taban sınıfından bir referans değişken ile yapılıyor ise "virtual dispatch" (sanal gönderim)

```

class Base
{
    public:
        virtual int func(int);

};

class Der : public Base
{
    public:
        double func(int);
        // overriding değil overloading değil redefinition var
        int func(int) override; //overriding
}

```

EXAMPLE (RUN TIME POLYMORPHISM)

```

/// Example
// Run Time Polymorphism
class Car
{
public:
    virtual void start()
    {
        std::cout << "Car::start()\n";
    }

    virtual void stop()
    {
        std::cout << "Car::stop()\n";
    }
};

class Audi : public Car
{
public:
    void start()
    {
        std::cout << "Audi::start()\n";
    }
    void stop()
    {
        std::cout << "Audi::stop()\n";
    }
};

class Tesla : public Car
{
public:
    void start()
    {
        std::cout << "Tesla::start()\n";
    }
    void stop()
    {
        std::cout << "Tesla::stop()\n";
    }
};

```

```

Car* create_random_car()
{
    static std::mt19937 eng{std::random_device{}()};
    static std::uniform_int_distribution dist{ 0, 1};

    switch (dist(eng))
    {
        case 0: std::cout << "Tesla\n"; return new Tesla;
        case 1: std::cout << "Audi\n"; return new Tesla;
    }
    return nullptr;
}

int main()
{
    // programın hangi fonksiyonu çağıracağı run-time'da belli olur.
    // buna run time polymorphism denir
    for (int i = 0; i< 1000; ++i)
    {
        Car* p = create_random_car();
        p->start();
        p->stop();

        delete p;
    }
}

```

```

class Base
{
public:
    virtual void foo();
    // NVI (Non virtual Interface)
    void bar()
    {
        foo();
    }
};

class Der : public Base
{
public:
    void foo();
};

void gf1(Base* p)
{
    p->foo();
}

void gf2(Base& p)
{
    p.foo();
}

int main()
{
    Der myder;
    myder.bar(); // Der sınıfının foo() fonksiyonu çağrılır.
}

```

Override Keyword

- Derleyici fonksiyonun virtual olup olmadığını kontrol etmek zorunda oluyor böylece virtual fonksiyonda değişiklik olursa (aldığı parametre değişirse gibi) derleyici hata döner.
- Okuyucuya fonksiyonun override edildiğini belirtiyor

```
class Base
{
public:
    virtual void foo(long);
    virtual void bar(double);
};

class Der : public Base
{
public:
    void foo(long) override;
    void bar(int) override; // hatalı
};
```

Bir taban sınıfının sanal fonksiyonunu override eden türemiş sınıfın fonksiyonu virtual specifier ile nitelenmemiş olsa da yine sanal fonksiyondur.

Virtual Functions -Multi Level Inheritance

```
class Base
{
public:
    virtual void foo(long);
};

class Der : public Base
{
public:
    // virtual ile nitelendirmesek bile virtual'dır
    virtual void foo(long) override;
};

class NDer : public Der
{
public:
    // NDer class'ı foo fonksiyonunu override edebilir.
    void foo(long) override;
}
/*
    Eğer NDer override etmeseydi foo(Long) fonksiyonu Der sınıfın foo(Long)
    fonksiyonu çağrılmıştı. Eğer Der override etmeseydi Base sınıfın foo(Long)
    fonksiyonu çağrılmıştı.
*/
```

```
class Base
{
    private:
        virtual void foo()
    {
        std::cout << "Base::foo()\n";
    }
};

class Der : public Base
{
    public:
        virtual void foo()
    {
        std::cout << "Der::foo()\n";
    }
};

// Base class'taki foo fonksiyonu private olmasına rağmen bu kod Legal

int main()
{
    Der myder;
    myder.foo(); // "Der::foo()" yazısını ekrana yazar
}
```

ÖNEMLİ !

```
class Base
{
public:
    virtual void foo()
    {
        std::cout << "Base::foo()\n";
    }
private:
    virtual void bar();
};

class Der : public Base
{
private:
    virtual void foo()
    {
        std::cout << "Der::foo()\n";
    }
public:
    void baz();
    void bar()override;
};

void gf(Base *p)
{
    p->foo(); // erişim kontrolu devreye girmez
/*
    Kontrol tamamen base göre yapılmıyor. Eğer base class'ın foo()
    fonksiyonu private olsaydı erişim kontrolu devreye girerdi.
    Erişim kontrolu run-time ilişkin (virtual dispatch)
*/
/*
    Derleyici koda baktığında compiler time'da p'nin türü Base ancak
    davranışı belirleyen tür Der sınıfı (run-time). İsim arama static
    türe bağlı olarak yapıldığı için p nesnesi Base sınıfının public foo()
    fonksiyonu olarak ilişkilendirilecek.
*/
    p->baz();
/*
    hatalı olur çünkü isim arama static türe göre olur yani Base sınıfında
    baz fonksiyonu olmadığı için isim arama hatası olur.
*/
    p->bar(); // erişim kontrolu hatası olur
}

int main()
{
    Der myder;
    gf(&myder);

    myder.foo(); // erişim kontrolu olur
    // erişim kontrolu compiler-time ilişkin
}
```

```

/*
 C++ dilini yeni öğrenenler tipik olarak statik tür kavramı (static type)
 dinamik tür kavramını (dynamic type) karıştırırlar.

 Statik tür derleyicinin koda bakarak gördüğü tür demek iken
 Dinamik tür ise run-time'daki davranışını belirleyen tür demek
*/

```

Varsayılan Arguman – Virtual Functions

```

class Base
{
public:
    virtual void foo(int x = 9)
    {
        std::cout << "Base::foo(int x) x = " << x << "\n";
    }
};

class Der : public Base
{
public:
    virtual void foo(int x = 99)
    {
        std::cout << "Der::foo(int x) x = " << x << "\n";
    }
};

void gf(base &r)
{
    r.foo();
    /*
        Varsayılan arguman static türle ilgili olduğu için Base sınıfının
        varsayılan argumani kullanılacak yani x = 9
        Ama r nesnesi runtime da belirlendiği için Der'in foo() fonksiyonu
        çağrırlır

        Çıktı:
            Der::foo(int x) x = 9
    */
}

int main()
{
    Der myder;
    gf(myder);

    myder.foo(); // Der::foo(int x) x = 99
}

```

Overloading Functions – Inheritance

```
class Base
{
public:
    //overLoading fonksiyonlardır
    virtual void foo(int);
    virtual void foo(double);
    // Sadece non-static member functions virtual olabilir
    virtual static void bar(); // syntax hatası
    // Ctor virtual olarak bildirilemez.
    virtual Base(int); // syntax hatalı
}

class Der : public Base
{
public:
    // overLoading fonksiyonlar
    void foo(int) override;
    void foo(double) override;
}
```

Virtual Ctor C++ (Clone idiom)

C++ dilinde sınıfların ctor'ları virtual anahtar sözcüğü ile bildirilemez. Ancak virtual ctor özellikle OOP paradigmında bir ihtiyaç olabilir. Virtual ctor yerine C++ dilinde: virtual ctor idiom ya da clone idiom

```
class Car
{
public:
    virtual Car* clone() = 0;
};

class Volvo : public Car
{
Volvo();
Car* clone()override
{
    new Volvo(*this);
}
};

void car_game(Car *p)
{
/*
    Buraya (run time'da) hangi türden araba gelirse oyunun senaryosu
    gereği run-time'da aynı türden bir araba oluşturacak
}
```

```

    */
    Car *p = p->clone();
}

int main()
{
    Volvo v;
    car_game(&v);
}

```

Dikkat!

- Base sınıfın ctor'i içinde yapılan virtual sınıf fonksiyon çağrıları için; virtual dispatch uygulanmaz. Çünkü henüz hayatı gelmemiş Der nesnesi için sanki o hayattaymış gibi üye fonksiyonu çağrırdık.
- Dtor içinde virtual dispatch uygulanmaz. Çünkü en son base sınıfının dtor çağrıılır. Der sınıf çoktan yokolmuş olacak. Bu yüzden virtual dispatch uygulanmaz

```

class Base
{
public:
    Base()
    {
        vfunc();
    }
    ~Base()
    {
        vfunc();
    }
    virtual void vfunc()
    {
        std::cout << "Base::vfunc()\n";
    }
    void foo()
    {
        Base::foo();
        // Nitelenmiş isimlerde virtual dispatch devreye girmez.
    }
};

class Der : public Base
{
public:
    virtual void vfunc()override
    {
        std::cout << "Der::vfunc()\n";
    }
};

int main()
{
    // Base::vfunc()\n yazar
    Der myder;
}

```

Virtual dispatch devreye girmediği yerler:

- 1) Virtual fonksiyon çağrıları Base sınıfının nesnesi ile yapılıyorsa
- 2) Virtual fonksiyon çağrıları Base sınıfının ctor içinde yapılıyorsa
- 3) Virtual fonksiyon çağrıları Base sınıfının dtor içinde yapılıyorsa
- 4) Virtual fonksiyon çağrıları nitelendirilmiş isim olarak yapılırsa

2023.09.22

Virtual DTOR

```
class Base
{
public:
    virtual ~Base()
    {
        std::cout << "Base Desturctor\n";
    }
};

class Der : public Base
{
public:
    ~Der()
    {
        std::cout << "Der destructor\n";
    }
};

int main()
{
    Base *p = new Der;
    delete p;

    /*
        Eğer Base sınıfın dtor virtual olarak tanımlanmazsa p nesnesi delete
        edildiğin de Base sınıfının dtor çağrırlır bu da tanımsız davranıştır.

        Ancak virtual olarak tanımlanırsa önce Der sınıfının dtor sonra
        Base sınıfının dtor'u çağrırlır.
    */
}
```

Polimorfik classların dtorları ya public virtual ya da protected non-virtual olmalı

```
class Base
{
protected:
    ~Base()
    {
        std::cout << "Base Desturctor\n";
    }
};

class Der : public Base
{
public:
    ~Der()
    {
        std::cout << "Der destructor\n";
    }
};

int main()
{
    Base *p = new Der;
    delete p; // hatalı olur. Çünkü base dtor protected ve Der classı bunu
    // çağrıramaz
}
```

Virtual Dispatch'e alternatifler:

1. type erasure
2. std::variant
3. CRTP (curiously recurring template pattern)

```
class Base
{
    int x{}; // 4 byte
    int y{}; // 4 byte
    void f1(); // 0 byte

    // 4 byte
    virtual ~Base();
    virtual void f2();
    virtual void f3();
};

class Der : public Base
{

};

int main()
{
    std::cout << "sizeof(Base) = " << sizeof(Base) << "\n";
}
```

```
class Base
{
    int mx;
    virtual void foo();
}

base address
-----
mx
vptr (virtual function table pointer)
```

Virtual Function Table Pointer:

Bir veri yapısıdır (data structure) ve her nesne için değil her class için vardır.

```
virtual function table for class Audi
-----
0
-----
1      &Audi::start
2      &Audi::run
3      &Audi::stop
```

Example:

```
void car_game(Car *p)
{
    p->run(); ----> p->vptr[2]();
}
```

dereferencing : adresındaki nesneye erişmek

*Her sanal fonksiyon çağrıları için virtual dispatch uygulanamayabilir.
Compiler optimazition yapılabilir.*

Devirtualization

```

class Base
{
    public:
        void func(int);
};

class Der : public Base
{
    public:
        // Overloading olur böylece
        using Base::func;
        void func(double);
}

int main()
{
    Der myder;
    myder.func(12); // func(int) çağrırlır
}

```

Inherited Ctor

```

class Base
{
    public:
        Base(int);
        Base(int, int);
        Base(const char*);
};

class Der : public Base
{
    public:
        using Base::Base; // 1 option --> inherited ctor
        Der(int x, int y) : Base(x, y) {} //2 option
        void bar();
}

int main()
{
    Der myder(12, 56)
}

```

Covariance or Variant Return Type

```
class B{};
class D : public B{};

class Base
{
public:
    virtual B* foo();
    virtual B& bar();
    virtual B baz();
}

class Der : public Base
{
public:
    D *foo()override;
    D &bar()override;
    D baz()override; // hatalı pointer veya referans olmak zorunda
                      // foo ve bar fonksiyonları covariant oldu
}

int main()
{
    Der myder;
    D* dp = myder.foo();
    auto dp2 = myder.foo();
```

NVI (non-virtual interface)

```
class Base
{
public:
    void foo(int x)
    {
        vfoo(x);
    }
private:
    virtual void vfoo(int);
};

class Der : public Base
{
public:
    void vfoo(int) override;
}
```

Final Contextual Keyword

- final class
- final override

Final Class

```
// Der sınıfından kalıtım yapmak syntax hatası olur final kullanırsak
class Base
{
};

class Der final : public Base
{
};

class NDer : public Der
{
    // syntax hatası -- final keyword kaynaklı
};
```

Final Override

```
class Base
{
public:
    virtual void func();
};

class Der : public Base
{
    void func() override final;
};

class NDer : public Der
{
    void func()override; // syntax hatalı
};
```

2023.09.25

Private Inheritance

```
class Base
{
public:
    void bar();
    void baz();
private:
    void foo();
protected:
    void pro_func();

};

// private inheritance
class Der : Base // private Base
{
    friend void gf();
    void func()
    {
        foo(); // hatalı olur
        /*
            public private ya da protected' da olsa Base sınıfının
            private bölümüne erişemeyiz.
        */
        // private kalıtım olduğu için pro_func'a erişemeyiz
        pro_func() // hatalı
    }

    void der_bar()
    {
        // hatalı değil erişebiliriz
        Der myder;
        Base *o = &myder;
    }
};

void gf()
{
    // hatalı değil erişebiliriz
    Der myder;
    Base *o = &myder;
}

int main()
{
    Der myder;
    /*
        Base sınıfının public interfaces'i Der sınıfının artık private
        interfacesindedir.
    */
    myder.bar(); myder.baz(); // hatalı olur çünkü private

    // private kalıtım olduğu için hatalı
    Base* p = &myder;
    Base& r = myder;
}
```

```

class Base
{
    public:
        virtual void vfunc();
}

class Der : private Base
{
    public:
        void vfunc()override; // geçerli public'te
}

```

Private inheritance containment için (composition) bir alternatif oluşturur.

```

// Y1 Sınıfın içinde X1 Nesnesi var
class X1 // containment
{
    public:
        foo();
};

class Y1
{
    public:
        void bar()
        {
            mx.foo(); // yapabilir
        }
    private:
        X1 mx; // member object
};

// Y2 Nesnesinin içinde X2 nesnesi var
class X2 // private inheritance
{
    public:
        void foo();
};

class Y2 : private X2 // base class object
{
    void bar()
    {
        foo();
    }
};

```

Private inheritance hangi zamanlarda tercih ederim:

- 1) Elemanınım sınıfını sanal fonksiyonunu doğrulan override edemem.

Ancak private taban sınıfın sanal fonksiyonlarını override edebilirim.

- 2) Elemanınım protected bölümüne erişemem ama private taban sınıfım protected bölümüne erişebilirim.
- 3) Türemiş sınıf nesnesi olarak ben elemanınım türünden değilim Ancak türemiş sınıfın üye fonksiyonları için ve türemiş sınıfın friendleri için is relation ship var

Yani :

Der myder;

Base *o = &myder;

- 4) Eğer sınıfımızın bir elemanı bir empty class türünden ise bu sınıfın 1 byte olan storage ihtiyacı alignment gereği sınıf nesnesinin için de padding bytes oluşturabilir.

Ancak empty class private kalıtımı yaparsak derleyeciler EBO denilen optimazasyonu yaparak o 1 byte dahil etmezler.

```
class Empty
{
    void foo();
    void bar();
};

class Nec
{
    //8 byte çünkü alignment var
    Empty member;
    int max;
}

class Nec1 : private Empty
{
    //4 byte
    int max;
}

int main()
{
    std::cout << "sizeof(Empty) = " << sizeof(Empty) << "\n"; // 1 byte
    std::cout << "sizeof(Nec) = " << sizeof(Nec) << "\n"; // 8 byte
    std::cout << "sizeof(Nec1) = " << sizeof(Nec1) << "\n"; // 8 byte
}
```

Restricted Polymorphism

```
class Base
{
    public:
        virtual void vfunc();
};

void foo(Base& baseref)
{
    baseref.vfunc();
}

class Der : private Base
{
    friend void f1();
};

void f1()
{
    Der myder;
    foo(myder);
}

void f2()
{
    Der myder;
    foo(myder);
}
```

Protected Inheritance (multi-level inheritance daha sık kullanılır)

```
class Base
{
    public:
        void foo();
};

class Der : protected Base
{
};

class Nec : public Der
{
    // private Base -- Der olsaydı hata olurdu ancak protected olduğu için hata yok
    void necbar()
    {
        foo();
    };
}
```

Multiple Inheritance

Bir sınıfın birden fazla taban sınıfından tek bir türütme ile oluşturulması.

```
class Base1
{
public:
    Base1(int)
    {
        std::cout << "Base1 ctor\n";
    }
    void foo();
};

class Base2
{
public:
    Base2(int)
    {
        std::cout << "Base2 ctor\n";
    }
    void bar();
};

class Der : public Base1, public Base2
{
    // Yazılma sırasına göre hayata gelirler ama ctor'daki init sırasına göre
    // değil.
    // Kalıtımındaki belirtiği sıraya göre
    Der() : Base2(1), Base1(2) {};
}

int main()
{
    Der myder;
    //geçerli
    myder.foo();
    myder.bar();
    //geçerli
    Base1* p1 = &myder;
    Base2* p2 = &myder;
}
```

```

class Base1
{
public:
    Base1()
    {
        std::cout << "Base1 ctor\n";
    }
    int foo{};
};

class Base2
{
public:
    Base2()
    {
        std::cout << "Base2 ctor\n";
    }
    void foo(int);
};

class Der : public Base1, public Base2
{
void bar()
{
    //ambiguity olur. İ sim arama aşamasında bir öncelik yok
    foo(12.3);
    foo = 10;
    // ancak böyle şekilde hata vermez
    Base1::foo(12.3);
    Base2::foo(12.3);
}
};

// function overloading
void gfoo(Base1&);
void gfoo(Base2&);

void gbar(Base2&);

int main()
{
    Der myder;

    gbar(myder); // geçerli
    gfoo(myder); // ambiguity var

    gfoo(static_cast<Base1&>(myder));

    myder.foo() // ambiguity isim aramadan kaynaklı
    myder.Base1::foo() // hata yok
}

```

Diamond Formation

```
/*
     Stream -> InputStream & OutputStream -> InOutStream
*/
class Stream {

};

class InputStream : public Stream
{
    public:
        void read(int&);

};

class OutputStream : public Stream
{
    public:
        void write(int);
};

class InOutStream : public InputStream, public OutputStream
{};

int main()
{
    InOutStream stream;

    int x{};
    stream.read(x);
    stream.write(12);
}
```

```
class Base
{
    public:
        void foo();
};

class Derx : public Base {};
class Dery: public Base {};
class MDer : public Derx, public Dery {};

int main
{
    MDer md;

    Base* p = &md; // ambiguous hatalı
    Base p1 = static_cast<Derx*>(&md);
    Base p2 = static_cast<Dery*>(&md);

    md.Derx::foo();
    md.Dery::foo();
}
```

```

class EDevice
{
public:
    bool is_open()const;
    {
        return m_flag;
    }
    void turn_on()
    {
        m_flag = true;
    }

    void turn_off()
    {
        m_flag = false;
    }
private:
    bool m_flag{};
}
class Printer : virtual public EDevice
{
public:
    void print()
    {
        if (is_open())
            std::cout << "printer is printing..\n";
        else
            std::cout << "printer cannot print device is off...\n";
    }
};
class Scanner : virtual public EDevice
{
public:
    void scan()
    {
        if (is_open())
            std::cout << "scanner is scanning..\n";
        else
            std::cout << "scanner cannot scan device is off...\n";
    }
};

class Combo : public Scanner, public Printer {};
int main()
{
    Combo mydevice;
    // Eğer virtual public EDevice olarak tanımlamazsak böyle:
    mydevice.turn_on(); // ambiguity var
    mydevice.Printer::turn_on();
    mydevice.scan(); // return scanner cannot scan device is off
    mydevice.print(); // return printer is printing

    //Virtual public EDevice (sVirtual inheritance)
    // Tek bir EDevice nesnesi oluşuyor virtual inheritance ile
    mydevice.turn_on(); // ambiguity yok
    mydevice.scan(); // return scanner is scanning..
    mydevice.print(); // return printer is printing
    mydevice.turn_off();
}

```

```

class Base
{
    public:
        Base (const char *p)
        {
            std::cout << "Base(const char *p) p = " << p << "\n";
        }
};

class Der1 : virtual public Base
{
    public:
        Der1(int) : Base("Der\n") {};
};

class Der2 : virtual public Base
{
    public:
        Der2(int) : Base("Der\n") {};
};

class MulDer : public Der1, public Der2
{
    public:
        MulDer() : Base("Mulder"), Der1(1), Der2(2)
        {
            // Base nesnesi init etmek zorundayız
        };
};

```

RTTI (Run Time Type Information)

- `dynamic_cast`
- `typeid`
 - `type_info`

2023.09.27

RTTI (Run Time Type Information)

- dynamic_cast
- typeid
 - type_info

upcasting : türemiş sınıfından taban sınıfı yapılmasına denir.

downcasting : taban sınıfından türemiş sınıfı yapılmasına denir.

Dynamic_cast

dynamic_cast<target type>(variable)

Mercedes *p = dynamic_cast<Mercedes*>(car_ptr)

Mercedes &p = dynamic_cast<Mercedes&>(car_ref)

Eğer cast başarılı olmazsa p nullptr olur

```
class Base
{
public:
    virtual ~Base() = default;
};

class Der : public Base
{
};

void foo(Base* baseptr)
{
    /*
        Dynamic cast olabilmesi için base sınıfı polymorfik olması gereklidir
        yani en az bir tane virtual fonksiyonu olması gerekiyor.
    */
    Der* derptr = dynamic_cast<Der*>(baseptr);
}
```

Dynamic Cast Usage:

```
void car_game(Car* carptr)
{
    Tesla *tp = dynamic_cast<Tesla*>(carptr)

    if (tp)
    {
        tp->autopilot();
    }
    // böyle yapmak daha havalı
    if (Tesla *tp = dynamic_cast<Tesla*>(carptr))
    {
        tp->autopilot();
    }
}

int main()
{
    for (int i = 0; i < 1000; ++i)
    {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}
```

```
// Reference semantığı ile (Not Null Ref yok)
void car_game(Car* carptr)
{
    try
    {
        Tesla &tp = dynamic_cast<Tesla&>(*carptr); // down_casting
        tr.autopilot();
    }

    catch (const std::exception&ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
/*
    Dynamic_cast kullanımında hedef tür referans türü ise
    down-casting güvenli bir şekilde yapılamıyor ise standart
    kütüphanemizin std::bad_cast sınıfı türünden exception throw edilir.

*/
}

int main()
{
    for (int i = 0; i < 1000; ++i)
    {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}
```

Typeid

```
/*
    typeid bir operatordür

    typeid(*ptr);
    typeid(x);
    typeid(10);
    type(Tesla);
    type(int);

    type info sınıf türünden const bir nesneye referans

    her ayrı (distinct) tür için bir type_info nesnesi var
*/
```

```
#include <typeinfo>
int main()
{
    int x = 10;

    typeid(x); // bu ifadenin türü typeinfo
    auto y = typeid(x); // syntax hatası çünkü typeid copy ctor delete edilmiş.

    //Legal
    const auto &r = typeid(x);
    auto &r2 = typeid(x);
}
```

Unevaluated Context

```
int main()
{
    int x = 10;
    auto sz = sizeof(x++);
    decltype(x++) y = x; // unevaluated context
    std::cout << "x = " << x << "\n";
    // x = 10 çünkü sizeof unevaluated context

    int a[10]{};
    auto s = a[30] // tanımsız davranış
    auto sz = sizeof(a[30]); // tanımsız davranış değil

    int *ptr{};
    auto x = *ptr; // tanımsız davranış
    auto szz = sizeof(*ptr); // tanımsız davranış değil

    int x = 12;
    const auto&r = typeid(++x); // unevaluated context
    std::cout << "x = " << x << "\n"; // x = 12

    int *ptr{};
    const auto&p = typeid(*ptr); // tanımsız davranış değil
}
```

```

class Nec
{
};

int main()
{
    Nec mynec;
    // derleyiciden derleyiciye çıktı değişir o yüzden karşılaştırma yapmamak
    // lazımdır
    std::cout << typeid(Nec).name() << "\n";

    typeid(mynec).operator==(typeid(int)) // false
    typeid(mynec).operator==(typeid(Nec)) // true
}

```

```

class Base
{
    virtual ~Base(){};
};

class Der : public Base {};

int main()
{
    Der myder;
    Base *ptr = &myder;

    /*
    Eğer Base polymorfik bir class ise 1 döner ama değilse 0 döner
    Burada Base polymorfik o yüzden 1 döner
    */
    std::cout << (typeid(*ptr) == typeid(Der)) << "\n";
}

```

```

void car_game(Car *ptr)
{
    if (typeid(*ptr) == typeid(Audi))
    {
        std::cout << "This a Audi\n";
    }

    if (typeid(*ptr) == typeid(Tesla)) // run-time'da yapılır
    {
        static_cast<Tesla *>(ptr)->autopilot();
        // dynamic_cast'deki gibi Teslanın alt sınıfları bu if'e girmez
    }
}

int main()
{
    for (int i = 0; i < 1000; ++i)
    {
        Car* p = create_random_car();
        car_game(p);
        delete p;
    }
}

```

static_cast in class

```
class Base{};  
class Der : public Base {};  
class Nec {};  
  
int main()  
{  
    Nec mynec;  
  
    //Legal değil  
    Der *derptr = static_cast<Der*>(&mynec);  
  
    // Legal  
    Base mybase;  
    Der *derptr = static_cast<Der*>(&mybase);  
}
```

Note: Virtual Function Table'da typeid için yer ayrılıyor

1) RTTI hiç kullanılmazsa ilave bir maliyet var mı?

Var çünkü run-time'da her türlü typeid nesneleri oluşturuluyor ama derleyiciye bağlı olarak biz run-time type nesnesini kullanmacağım dersek RTTI disable edersek bize maliyet oluşturmadan kapatırız

2) Büyük bir hiyerarşi var ve typeid ya da dynamic_cast kullancaksın ikiside

işimizi görüyor. Hangisinin maliyeti düşüktür?

Typeid'nin maliyeti daha düşük çünkü typeid classların türemiş sınıflarını kontrol etmez ancak dynamic_cast türemiş sınıfları kontrol eder.

```
typeid(*ptr)
```

polimorfik tür söz konusunda olduğunda eğer typeid operatörünün operandı olan ifade dereference edilmiş pointer ifadesi ise pointer değerinin nullptr olması durumunda:

std::bad_typeid sınıfı türünden exception throw edilir. bad_typeid sınıfı std::exception sınıfından kalıtım yoluyle elde edilir.

```
void car_game(Car *ptr)
{
    try
    {
        if (typeid(*ptr) == typeid(Tesla))
        {
            static_cast<Tesla *>(ptr)->autopilot();
        }
    }
    catch (const std::bad_typeid& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
        // nullptr
    }
}

int main()
{
    Car* ptr{};
    car_game(ptr);
    // ptr null olduğu için exception verir
}
```

Taban sınıfların dtor'ları ya public virtual ya da protected non virtual olmalı. Ancak NVI kullanarakta tanımlayabiliriz.

Template Method

```
// template method
class Figther
{
    virtual void attack();
public:
    void fight()
    {
        attack(); // savaşçıyı göre customize edilecek
    }
};
```

Non Virtual Interface ile:

- Taban sınıf kendi kontrollerini dayatabiliyor
- Ortak kodları bir yere toplayabiliyoruz
- İmplementasyon ile interfacesi ayırmış oluyoruz

```
class Figther
{
public:
    void attack()
    {
        //invariant'lar
        attack_impl();
        //invariant'lar
    };
private:
    virtual void attack_impl() = 0;
}
```

2023.09.29

initializer_list

```
int main()
{
    std:: initializer_list<int> x{ 1, 2, 3, 4, 5};

    /*
        const int ar[] = { 1, 2, 3, 4, 5};
        class initializer_list
        {
            const int *ps; başlangıç adresi
            const int *pe; son adresi
        }

    */

    auto x = { 1, 2, 3, 4, 5}; // x'in türü initializer_list<int>
    auto x { 1, 2, 3, 4, 5}; // syntax hatası
    auto x = {1}; // x'in türü initializer_list<int>
    auto x{1}; // x'in türü int
}
```

```
class MyClass
{
public:
    MyClass(std::initializer_list<int>)
    {
        std::cout << "MyClass(initializer_list)\n";
    }
    MyClass(int)
    {
        std::cout << "MyClass(int)\n";
    }
    MyClass(int, int)
    {
        std::cout << "MyClass(int, int)\n";
    }
};
int main()
{
    MyClass m1(10, 20); // return MyClass(int, int)
    MyClass m2{10, 20}; // return MyClass(initializer_list)
}
```

```
//vector<>(size_t)
//vector<>(initializer_list)

int main()
{
    std::vector<int> vec1(100); // size_t - 100 tane 0 sıfır değeri ile başlatacak
    std::vector<int> vec2{100}; // initializer_list - 1 tane 100 değeri ile
    başlatacak
}
```

```
int main()
{
    using namespace std;
    string s1(50, 'A'); // 50 tane A karakteriyle başlatacak
    string s1{50, 'A'}; // 50'in ascii değeri ve A ile başlatacak
}
```

String Class

```
/*
    std::string::size_type //size_t türüne eş
    std::string::npos // türü size_type

    size_type:
        yazı uzunluğu türü
        kapasite türü
        bazı string fonksiyonlarının istediği indeks değeri

    string sınıfındaki fonksiyonların aldığı arguments:

    const string&
    const string&, size_type idx : idx indeksinden sonraki string boyunca
    işlem yapacak
        const string&, size_type idx, size_type len : idx'ten başlayarak len kadar
    işlem yapacak
        const char* (cstring) : null karaterden biz sorumluyuz
        const char*, size_type len
        char
        size_type, char c : n tane c karakteri (fill)
        iterator beg, iterator end : range parametre
        std::initializer_list<char>

*/
```

String Sınıfı Ctor'lar ve Fonksiyonları

```
//string sınıfı ctor'lar ve fonksiyonları
void ps(const std::string& s)
{
    std::cout << " | " << s << " |\n";
}

int main()
{
    using namespace std;

    string s1; //Default ctor
    ps(s1); // boş yazı çıkar

    // s1.size() ve s1.length() return type: size_type
    cout << "s1.size() = " << s1.size() << "\n"; // common container interface
    cout << "s1.length() = " << s1.length() << "\n";

    s1.empty() // is empty()

    // cstring ctor
    string str("emre bahtiyar"); // cstring ctor
    char ar[] = { 'A', 'B', 'C'};
    string str(ar); // tanımsız davranış
    ar[] = "murathan";
    string str(ar + 5 ); // return han --cstring ctor

    //const char*, size_type len ctor
    string str(ar ,3); // return mur (const char*, size_type len )
    string str(ar+3 , 2); // return at
    string str(ar, 20); // tanımsız davranış
    string str(ar, 6); // return murat + null karakter

    std::string str('A') // parametresi char olan ctor yok hatalı

    //substring ctor
    string s1 {"cengizhan"};
    string s2 (s1, 3); //substring ctor : return gizhan
    string s2 (s1, 3, 50); // geriye kalanların hepsini alır - hata yok
}

/*
String CTOR:

    string s1("a") cstring ctor
    string s2(13, 'a') fill ctor
    string s3{'A'} initializer_list ctor
    string s4(s2, 3) substring ctor

*/
```

```

int main()
{
    using namespace std;

    ifstream ifs {"main.c"};

    vector<string> svec;
    string sline;

    while (getline(ifs, sline))
    {
        //svec.push_back(sline);
        svec.push_back(std::move(sline)); // sline geri kullanılabilir
    }
}

```

```

int main()
{
    using namespace std;
    char str[] = { "gokhan girgin" };

    string s1 {str}; //cstr ctor
    string s2 { str , 3}; // array ctor
    string s3 { str , str + 4}; // range ctor
}

```

capacity()const : return size_type

```

int main()
{
    using namespace std;

    string str(153, 'A');
    cout << "str.size() = " << str.size() << "\n"; // return 153
    cout << "str.capacity() = " << str.capacity() << "\n"; // return 159

    /*
        size 153 eğer 6 tane daha karakter eklersem capacity dolacak ve
        reallocation olacak.

        String ve vector sınıflarında kapasitenin artış katsayısı derleyiciye
        bağlıdır.

    */
}

```

```
int main()
{
    /*
        kötü kod çünkü sürekli reallocation yapmış oluyuruz. Baştan size
        belirtseydik reallocation olmıcaktı.

        str.reserve(1200); // böyle kapasiteyi önceden belirlemiş olduk
    */
    string str;

    char c = 'S';

    for (int i = 0; i < 1000; ++i)
    {
        str.push_back(c)
    }
}
```

Small String Optimazsayonu

Derleyeciler belli bir boyutta kadar allocation yapmaz ve kendi içinde oluşturur

2023.10.02

String sınıfı

String Elemanlarına Erişme:

```
#include <string>

int main()
{
    using namespace std;

    string str{ "burak kose" };

    for (size_t i{}; i < str.length(); ++i)
    {
        cout.put(str[i]);
    }

    str[2] = 's';
    // str[12] tanımsız davranış

    try
    {
        auto c = str.at(345); // exception verir at fonksiyonunda
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught: " << ex.what() << '\n';
    }

    str.front() = 'd'; // ilk
    str.back() = 'i'; // son
}
```

Range-Based For Loop:

```
// range-based for Loop

int main()
{
    using namespace std;

    string str {"bugun gunlerden pazartesi kirmizi pazartesi"};

    // range-based for Loop

    for (auto iter = str.begin(); iter != str.end(); ++iter)
    {
        cout << *iter << " ";
    }
    // derleyici yukarıdaki gibi kod yazar
    for (auto i : str)
    {
        cout << i << " ";
    }
}
```

```
/*
for (auto i : a) // kopyalama semantiği
for (auto &i : a) // referans semantiği ( değiştirmeye yapacaksak)
for (const auto i : a) // kopyalama semantiği
for (const auto &i : a) okuma yapacaksak

for (auto iter = begin(con); iter != end(con); ++iter)
{
    auto i = *iter;
    auto &i = *iter;
    const auto i = iter;
    const auto &i = iter;
}

*/
```

```
int main()
{
    string str { "kirmizi pazartesi" };
    for (auto c : str)
    {
        c = "!"
    }

    cout << str << "\n"; // yazı değişmez

    for (auto &c : str)
    {
        c = "!"
    }

    cout << str << "\n"; // yazı değişir
}
```

Dikkat !

std::string bir STL kabıdır ve iterator arayüzüne sahiptir

String Sınıfı Atama İşlemleri

```
std::string get_name()
{
    return "kerim denizoglu";
}

int main()
{
    using namespace std;
    string name = "tayyip erguder";
    string str {" mustafa "};

    cout << "|" << str << "|\\n";

    str = name; // copy assignment
    cout << "|" << str << "|\\n";

    // move assignment çünkü ='in sağ tarafı r value expr
    str = get_name();
    cout << "|" << str << "|\\n";

    str = "alican"; // cstring atama operator fonksiyonu
    cout << "|" << str << "|\\n";

    str = 'X'; // char parametrelî atama operator fonksiyonu
    cout << '|' << str << "|\\n";

    str = {'a', 'l', 'i'}; // init list parametrelî atama operator fonksiyonu
    cout << '|' << str << "|\\n";

    name.resize(20); // boyutu büyük null karakter koyarlar
    name.resize(20, '.')
    name.resize(2); // silme argümanı olarak kullanabiliriz

    name.pushback(".")

}
```

String Sınıfı Container Silme

```
int main()
{
    string name = "tayyip erguder";
    // container silme
    name.clear();
    name.resize(0);
    name = "";
    name = {};
    name = string{}; // default ctor
}
```

String Sınıfı Yazı Ekleme

```
int main()
{
    string str {"emre bahtiyar"};

    str.push_back(".");
    str += "gano";
    str += 's';
    str += {"1", "9"};

    string str;
    string s = "neslihan";

    str.assing(s, 3); // Lihan
    str.assing(s, 1, 3); //esl
}
```

String Sınıfı Arama Fonksiyonları

static constexpr string::size_type npos = -1

```
int main()
{
    //size_t' in en büyük değeri
    std::cout << "npos = " << std::string::npos;

    constexpr auto x = std::string::npos;
    std::string::npos = 57687u // Legal değil npos const

    /*
        string sınıfın arama fonksiyonları indeks döndürürler
        bu değer index bulunamaz ise
        std::string::npos olur
    */

    /*
        Arama fonksiyonları
        find
        rfind
        find_first_of
        find_last_of
        find_first_not_of
        find_last_not_of

        hepsi std::string::size_type döndürür
        eğer aranan bulunamazsa hepsi std::string::npos döndürür
    */

    string str = "emre bahtiyar";
    char c = 'a';

    // kodu böyle yazmak scope leak neden olur
    string::size_type idx = str.find(c);
```

```

if (idx != string::npos)
{
    //code
}
// cpp 17 ile böyle
if (string::size_type idx = str.find(c); idx != string::npos)
{
    //code
}

// cpp 17'den eskilerde
{
    string::size_type idx = str.find(c);
    if (idx != string::npos)
    {
        //code
    }
}

/*
Cpp 20 ile:
yazı aradağım varlıkla mı başlıyor
yazı aradağım varlıkla mı bitiyor

starts_with
end_withs
*/
// Cpp 20
string str = "emre_bahtiyar.bin";
if (str.start_with("emre")) // return bool
{
}

if (str.end_withs("bin"))
{
}

// Cpp 23 bu substring içeriyor mu?
if (str.contains("bahtiyar")) // return bool
{
}

str.rfind('a'); //aramayı sondan yapıyor
str.find_first_of("ptik"); // bu karakterlerden birini bulur ilk geleni
str.find_first_not_of("resim"); //bu karakterlerde olmayan ilkini bulur
str.find_last_of("resim"); // bu karakterlerden sondan olanı
str.find_last_not_of("resim"); // bu karakterlerden olmayan sondan
}

```

String Yazısı Değiştiren Fonksiyonlar

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

    string s1 = "omer faruk";
    string s2 = "selma deniz";
    // append
    s1.append(s2, 5); // 5. indeksten başlayarak
    s1.append(s2, 1, string::npos); // 1. indeksten geriye kalanları almak için

    /*
        STL Container interfaçer gelen
        iterator interface sahiplerdir:
        insert ve erase fonksiyonları

        con.insert(iter, value)
    */

    string s{"mehmet bal"};

    // iterator interface ile yazının başına ! karakteri ekleyin.

    s.insert(s.begin(), '!');
    cout << s;

    while (!s.empty())
    {
        cout << s;
        s.erase(s.begin());
        // son silme
        s.erase(s.pushback());
        s.erase(s.end());
    }

    s.erase(s.begin(), s.begin() + 3); // ilk 3 karakter silincek
    s.erase(s.begin() + 1, s.end() - 1);

    string s1{"emre"};
    string name {"bahtiyar"};

    s.insert(4, name, 2, 3); // 4. indekse name'in 2. indeksinden 3. indeksine
                           // kadar

}
```

String Sınıfı Karşılaştırma İşlemleri

```
int main()
{
    string name{"ayhan"};
    string word{"ekrem"};

    if (name == word)
    if (name < word)
    if (name > word)

}
```

Shrink

```
int main()
{
    string str(100'000, 'A');

    cout << "str.size()" << str.size() << "\n";
    cout << "str.capacity()" << str.capacity() << "\n";

    str.erase(10) // ilk 10 dışındakileri sil

    cout << "str.size()" << str.size() << "\n"; // size 10
    cout << "str.capacity()" << str.capacity() << "\n"; // capacity değişmedi

    str.shrink_to_fit(); // capacity size uygun bi değere getirir
}
```

2023.10.11

String View

```
/*
    class StringView
    {
        public:
            // string sınıfı fonksiyonları

        private:
            // kopyalama yapmadan bu iki pointer ile o str'yi kontrol edebilirim
            const char ps*;
            const char pe*
    }

#endif

#include <string>
#include <string_view>

int main()
{
    std::string_view sv { "necati ergin" };
    // okuma fonksiyonlarına erişebiliriz ve gözlemevi olabiliyoruz
}
```

String Sınıfı Toplama Operatorü

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

    string name { "mert" };
    string surname { "kaptan" };

    auto s = name + ' ' + surname;
    // operator+(operator+(name, ' '), surname);
}
```

String Tür Dönüşümleri

```
int main()
{
    int ival{};
    double dval{};

    auto istr = to_string(ival);
    auto dstr = to_string(dval);

    string str;
    int x = 56;
    str = x; // operator=(char) tür dönüşümü olmaz

    //string to int, Long, double vb
    /*
        stoi
        stoul
        stol
        stod
    */
    string s {"23243"};

    auto ival = stoi(s); // int ival = 23243

    string s1 {"3213emre"};
    size_t idx{};
    auto ival1 = stoi(s1, &idx); // int ival1 = 32133
    // idx kullanılmayan verinin size döndürür

    string s2 {"eee3213emre"};
    auto ival2 = stoi(s1); // exception verir
}
```

Exception Handling

```
/*
    assertions (doğrulama) (C)
        dynamic assertions (runtime)
        static assertions (compiler time)

    assert( x > 0)

    void func(int *ptr)
    {
        assert(ptr != NULL);

        assert(b != 0);
        auto x = a / b;
    }

exception : (run-time error)

try
{
    f1(); // f1 ve içindeki her fonksiyon bu bloğa dahil
}
catch(const std::bad_alloc&)
{
}

catch(Long)
{
}

bir hata nesnesi gönderildi ve bu hata nesnesi yakalamağı bu durumda ne olur?

Bu duruma uncaught exception denir
    std::terminate
    abort

terminate fonksiyonu uncaught exception durumunda abort fonksiyonu çağrırlar
ancak bizim istediğim fonksinu çağırması için set_terminate fonks kullanılır.

void terminate(void) {}

set_terminate(void (*) (void))

*/
```

```
class ExBase
{
public:
    virtual const char* what()const
    {
        return "exbase";
    }
}

class MathException : public Exbase
{
public:
    const char* what()const override
    {
        return "exbase";
    }
}
class DivideByZero : public MathException
{
public:
    const char* what()const override
    {
        return "exbase";
    }
}

void myabort()
{
    std::cout << "myabort cagrildi\n";
    exit(EXIT_FAILURE);
}

void f4()
{
    std::cout << "f4 cagrildi\n";
//throw 1 // exception atiyoruz
    throw DivideByZero;
    std::cout << "f4 sona erdi\n";
}

void f3()
{
    std::cout << "f3 cagrildi\n";
    f4();
    std::cout << "f3 sona erdi\n";
}

void f2()
{
    std::cout << "f2 cagrildi\n";
    f3();
    std::cout << "f2 sona erdi\n";
}

void f1()
{
    std::cout << "f1 cagrildi\n";
    f2();
    std::cout << "f1 sona erdi\n";
}
```

```

int main()
{
    set_terminate(&myabort);
    std::cout << "main basladi\n";
    try
    {
        f1();
    }
    catch (int)
    {
        std::cout << "hata yakalandi catch(int)";
    }

    catch (unsigned int)
    {
        std::cout << "hata yakalandi catch(unsigned int)";
    }

    catch (double)
    {
        std::cout << "hata yakalandi catch(double)";
    }

    catch (const Exbase& ex) // her zaman referans olacak
    {
        std::cout << "hata yakalandi";
    }
    std::cout << "main sonaerdi\n";
}

```

catch her zaman referans olacak çünkü:

- copy ctor'dan korunmak için
- virtual dispacth faydalananmak için

```

/*
    throw expr;

    int x = 5;
    throw x;

derleyici şöyle bir kod oluşturur.
int exception_object{x};

ya da

    throw 12;
    int exception_object{12};

*/

```

```
void f4()
{
    std::cout << "f4 cagrildi\n";
    std::string s {"emre bahtiyar"};
    // out_of_range --> logic_error --> exception
    auto c = str.at(565); // exception

    std::cout << "f4 sona erdi\n";
}
void f3()
{
    std::cout << "f3 cagrildi\n";
    f4();
    std::cout << "f3 sona erdi\n";
}
void f2()
{
    std::cout << "f2 cagrildi\n";
    f3();
    std::cout << "f2 sona erdi\n";
}
void f1()
{
    std::cout << "f1 cagrildi\n";
    f2();
    std::cout << "f1 sona erdi\n";
}

int main()
{
    set_terminate(&myabort);
    std::cout << "main basladi\n";
    try
    {
        f1();
    }
    catch(const std::exception& ex)
    {
        //exception ile std'in tüm hataları yaklayabiliriz
        // logic_error ile logic hataları yakalarız.
        // out_of_range ile sadece bunla ilgili hataları
    }

    catch(...)
    {
        // bütün hataları yakalar
    }
}
```

2023.10.13

```
class Nec {
    unsigned char* buf[1024*1024]{};
};

int main()
{
    std::vector<Nec*> nvec;
    int i{};

    try
    {
        for (i = 0; , i < 1'000'000; ++i)
        {
            nvec.push_back(new Nec());
        }
    }
    //
    catch (const std::bad_alloc& ex)
    {
        std::cout << "exception caught" << ex.what() << "\n";
        std::cout << "i = " << i << "\n";
    }
    catch (const exception& ex)
    {
        std::cout << "exception caught" << ex.what() << "\n";
        std::cout << "i = " << i << "\n";
    }
}
```

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

    string str(10, 'A');

    try
    {
        auto c = str[463]; // undefined behavior
        auto c = str.at(463); // exception -> out_of_range
    }
    // catch blokları özenle genele doğru sıralanmalı
    catch (const std::out_of_range& ex)
    {
        std::cout << "out of range caught" << ex.what() << "\n";
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught" << ex.what() << "\n";
    }
}
```

exception yakalayınca hangi senaryolar devreye girer:

- 1) resumption (kaldığı yerden devam etme)
 - i. resource leak olmıcak (database bağlantısı kopacak vb)
- 2) terminate (sonlandırma)
- 3) kısmi müdahale gerçekleştirip rethrow ediyoruz
- 4) exception translate etmek

```
int foo();

int main()
{
    using namespace std;
    try
    {
        int x = foo();
    }
    catch (const std::exception& ex)
    {
        // hatalı olur x kullanamayız çünkü try scope'ta
        std::cout << "hata yakalandı x = " << x << "\n";
    }
}
```

```
/*
    a) gönderilen exception'lar yakalanıyor mu
    b) hataların yakalanması durumunda kaynak sızıntısı oluyor mu?
*/

class Myclas
{
    //...
};

void foo();
void bar();

void func()
{
    Myclass *p = new Myclas;

    /*
    exception verirse bu fonksiyonlar program akışını
    func() fonksiyonundan çıkar ve p nesnesi delete
    edilemez. Bu yüzden böyle kodlardan uzak durmalıyız

    */
    foo();
    bar();

    delete p;
}
```

```

/*
    RAII

    stack unwinding ( yığının geri sarımı)

        f1 --> f2 --> f3 --> f4 --> f5
*/

class Myclas
{
public:
    ~Myclass()
    {
        std::cout << this << "kaynaklar geri verildi\n";
    }

private:
    int ar[100]{};
}

void f4()
{
    Myclass m4;
    throw std::exception{};
}
void f3()
{
    Myclass m3;
    f4();
}
void f2()
{
    Myclass m2;
    f3();
}
void f1()
{
    Myclass m1;
    f2();
}

int main()
{
    f1(); // MyClass nesneleri böyle yaparsak destroy edilmesi

    try
    {
        // Otomatik ömürlü oldukları için
        f1(); // MyClass nesneleri destroy edilir.
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught" << ex.what() << "\n";
    }
}

```

```

class File
{
    public:
        File(const char*);
        ~File();
};

void foo();
void bar();

void func()
{
    File myfile("necati.txt");
    /*
    exception verse bile aşağıdaki fonksiyonlar
    myfile otomatik ömürlü olduğu için yine stack
    unwinding aşamasında destroy edilir.
    */
    foo();
    bar();
}

```

Aksi biçimde davranışınıza gerekecek olağan dışı bir durum yok ise dinamik ömürlü nesneleri ya da genel olarak kaynakları ham pointer'lar ile değil smart pointer nesneleri ile kullanın.

Exception Rethrow Edilmesi

```

try
{
    foo();
}
// eğer ex adlı parametreyi kullanmıcaksak isim verme
catch(const std::MathError& ex)
{
    /* eğer böyle yaparsak
     1) copy ctor ile yeni ex oluşturulur
     sonra eski ex destroy edilir.
     2) ex divided by zero olsa bile throw ex
     yaparsak MathError exception döner yani
     object slicing olur
    */
    throw ex;

    // rethrow böyle yapılır
    throw;
}

// rethrow
try
{
    foo();
}
catch(const std::MathError& ex)
{
    throw; // yakalanan nesne ile gönderilen nesne aynı
}

```

```

void func()
{
    try
    {
        throw std::out_of_range {"range hatalı"};
    }
    catch (const std::exception& ex)
    {
        std::cout << "func içinde hata." << ex.what() << "\n";
        throw ex; // std::exception bloğuna girer (object slicing)
        throw; // out_of_range bloğuna girer.
    }
}

int main()
{
    try
    {
        func();
    }
    catch (const std::out_of_range&)
    {
        std::cout << "out_of_range";
    }
    catch (const std::exception&)
    {
        std::cout << "hata yakalandı";
    }
}

```

```

void foo()
{
    if (1)
    {
        // hatalı çünkü default ctor yok
        throw std::out_of_range{};
    }
}

```

```

void bar()
{
    throw;
}

void foo()
{
    try
    {
        if (1)
            throw std::out_of_range("out of range error");
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
        bar(); // rethrow eder
    }
}

```

```
void bar()
{
    throw;
}
int main()
{
    bar(); // terminate fonksiyonu çağrırlır.
}

/*
    Eğer bir rethrow statement yürütüldüğünde yakalanan bir hata nesnesi yok ise std::terminate çağrırlır.
*/
```

```
void bar()
{
    throw;
}

int main()
{
    try
    {
        bar(); // terminate edilir catch'e girmez
    }
    catch (...)
    {
        std::cout << "exception caught: ";
    }
}
```

Exception in CTOR

```
class Myclass
{
public:
    Myclass(int x) : np(new int[100])
    {
        if (x < 0)
            throw std::invalid_argument{"gecersiz arguman"};
    }
    ~Myclass()
    {
        std::cout << "Myclas destrurctor\n";
        delete[] np;
    }
private:
    int *mp;
}
int main()
{
    try
    {
        Myclas m{-12}; // dtor çağrılmaz
        /*
        dtor çağrılmaz çünkü bir sınıf nesnesinin
        hayata gelmesi için ctor kodunun tamamı tamamlanması gerekiyor
        */
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
```

```
class A
{
public:
    A(int)
    {
        std::cout << "A(int) ctor\n";
    }
    ~A()
    {
        std::cout << "A dtor\n";
    }
};
class B
{
public:
    B(int)
    {
        std::cout << "B(int) ctor\n";
    }
    ~B()
    {
        std::cout << "B dtor\n";
    }
};
```

```
class Myclass
{
public:
    Myclas() : ax(0), bx(1)
    {
        if(1)
        {
            throw std::bad_alloc{};
        }
    }
    ~Myclass()
    {
        std::cout "Myclass dtor\n";
    }

private:
    A ax;
    B bx;
}
void foo()
{
    Myclas m;
}

int main()
{
    try
    {
        foo();
        /*
            ax ve bx nesnelerinin dtor çağrılr çünkü
            hayata gelmiş durumundalar ancak myclass nesnesi
            hayata gelmediği için onun dtor çağrılmaz
        */
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception: " << ex.what() << "\n";
    }
}
```

```
class Member
{
    Member()
    {
        std::cout << "Member ctor\n";
    }
    ~Member()
    {
        std::cout << "Member dtor\n";
    }
};

class Nec
{
public:
    Nec() : mp{new Member}
    {
        throw 1;
    }
    ~Nec() {
        delete mp;
    }
private:
    Member* mp;
    //std::unique_ptr<Member> mp;
};

int main()
{
    try
    {
        Nec mynec;
    }
    catch(int )
    {
        // Member dtor çağrılmaz çünkü heap'te oluşuyor
        std::cout << "exception caught\n";
    }
}
```

```
class Member
{
public:
    Member(int x)
    {
        std::cout << "Member ctor\n";
        if (x < 0)
        {
            throw std::runtime_error{"exception from Member"};
        }
    }
    ~Member()
    {
        std::cout << "Member dtor\n";
    }
};

class Nec
{
    Nec(int val) : mx(val)
    {
        // bu exception yakalanamaz
        try
        {

        }
        catch (...)
        {
            std::cout << "hata yakalandi\n";
        }
    }
private:
    Member mx;
};

int main()
{
    Nec mynec(-23);
```

Function Try Block

```
void  func(int x)
{
    try
    {
        // all function code
    }
    catch (const std::exception&)
    {

    }
}

// yukarıdaki yerine

int  func(int x)
try
{
    int y = 5;
    // all code
    if (x < 0)
        throw std::runtime_error{ "gecersiz deger" };
    return 3;
}
catch (const std::exception&)
{
    /// y = 5; hatalı
    return x * 6;
},
```

Function Try Blok in CTOR

```
class Member
{
public:
    Member(int x)
    {
        std::cout << "Member ctor\n";
        if (x < 0)
        {
            throw std::runtime_error{ "exception from Member" };
        }
    }
    ~Member()
    {
        std::cout << "Member dtor\n";
    }
};

class Nec
{
    Nec(int val) try : mx(val)
    {
        // function try block eski cpp'de var
    }
    catch (const std::exception& ex)
    {
        // bu sefer Nec sınıfındaki exception yakalanır
        std::cout << "exception caught: " << ex.what() << "\n";
        // yine de kod patlar çünkü derleyici patlatır Nec nesnesi oluşamadığı
        // için
    }

private:
    Member mx;
};

int main()
{
    Nec mynec(-23);
}
```

```

class A
{
public:
    A() = default;
    A(const A&)
    {
        std::cout << "A copy ctor\n";
        throw::std::runtime_error{"error from copy ctr of class A\n"};
    }
};

class Nec
{
public:
    Nec(A) try
    {

    }
    catch (const std::exception& ex) {
        std::cout << "exception caught..." << ex.what() << "\n";
    }
};

int main()
{
    // copy ctor'dan gelen exception yakalayamaz
    A ax;
    Nec mynec(ax);
}

```

Exception Guarantee(s)

Basic guarantee: exception gönderildiğinde bir kaynak sizıntısı olmıcak bu garanti altına alınacak ve nesne veya nesnelerin state'i geçerli olacak.

Strong guarantee: Basic guarantee özelliklerini veriyor ve state sadece geçerli değil ayrıca state aynı değişmiyor (commit or rollback) ya işini yap ya da işe başlamadan önceki state'i koru

```

try
{
    std::vector<int> x {1,2,3,4};
}
catch
{
    burda exception yakalandı ve x'in size değişti 10 dan 12'ye çıktı
    basic guarantee ama size değişmedi strong guarantee
}

```

Nothrow guarantee: exception kesinlikle gönderilmicek derleyici buna göre yaptığı optimazsayonu değiştiriyor

2023.10.16

```
/*
noexcept:
    noexcept specifier
    noexcept operator

noexcept specifier:

void foo(int)noexcept;
exception kesinlikte göndermemicem diyor bu fonksiyon bunun avantajı:
    kodu yazan buna göre kod yazabilir.
    derleyici daha uygun bir kod seçebilir

eğer bu kod run-time'da exception gönderirse terminate fonksiyonu
çağrılır.

*/
```

```
void func()
{
    if (1)
    {
        throw std::exception{};
    }
}

void foo()noexcept
{
    func();
}
void myterminite()
{
    std::cout << "my terminate called\n";
    abort();
}
int main()
{
    set_terminate(myterminite);
// exception yakalanamaz ve terminate fonkisyonu çağrılır
    try
    {
        foo();
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
```

```
// daha çok generic programlama ile alakalı
void foo()noexcept(true);
void foo()noexcept;

void bar()noexcept(false);
void bar();

void foo()noexcept(sizeof(int) > 2);

template<typename T>
void func(T)noexcept(std::is_nothrow_copy_constructible_v<T>);
```

noexcept operator

constexpr bool b = noexcept(expr); compiler time oluşur true ya da false döner

```
void foo(int);

int main()
{
    int a[5][];
    constexpr bool b = noexcept(a[3]); // exception göndermez true

    int x = 423;
    constexpr bool b = noexcept(++x); // exception göndermez true

    // exception gönderir çünkü foo(int) noexcept değil
    constexpr bool b = noexcept(foo(int));
}
```

```
class Myclass
{
public:
    void foo(int, int)noexcept;
};

int main()
{
    // true döner
    constexpr bool b = noexcept(Myclass{}.foo(1, 5));
}
```

```
class Myclass
{
public:
    void foo(int, int)noexcept;
};

int main()
{
    // true döner
    constexpr bool b = noexcept(Myclass{}.foo(1, 5));
}
```

```
class MyClass{};  
  
    //specifier // operator  
void func()noexcept(noexcept(MyClass{}));  
/*  
operator olan MyClass türünden bir nesneyi default olarak init ettiğimizde  
exception throw etmiyorsa true olacak  
func fonksiyonun no throw garantisi ise de operator olan noexcept fonksiyonun  
true olup olmamasına bağlı  
*/
```

```
class MyClass  
{  
public:  
    MyClass& operator++();  
};  
  
void func(MyClass m)noexcept(noexcept(++m)); // no except değil
```

```
template <typename T>  
void func(T x)noexcept(noexcept(++x) && noexcept(--x));  
class MyClass  
{  
public:  
    MyClass& operator++()noexcept;  
    MyClass& operator--()noexcept;  
}  
int main()  
{  
    constexpr auto b = noexcept(func(12));  
    /*  
        true döner func int türünden parametre almış gibi olacak  
        ++x ve --x except göndermez.  
    */  
    MyClass mx;  
    constexpr auto b = noexcept(func(mx)); // true  
}
```

```
class Base  
{  
public:  
    virtual void func()noexcept;  
};  
  
class Der : public Base  
{  
public:  
    void func()override; // syntax hatalı noexcept garantisi vermemiz lazımdır  
};
```

```

void foo(int);
void bar(int)noexcept;

int main()
{
    // geri dönüş değeri int olan bir fonksiyonun adresini tutar
    int (*fp)(int)noexcept;

    auto ival = fp(43); // exception vermicek

    fp = foo; // syntax hatalı verir çünkü foo noexcept değil

    int (*br)(int);
    br = bar; // syntax hatalı vermez
}

```

```

int main()
{
    int x = 35;

    constexpr bool b = noexcept(x++);
    std::cout << "x = " << x "\n"; // x = 35 yazar unevaluated context
}

```

```

class Myclass
{
public:
    ~Myclass();
    // dtor exception throw edemez eğer ederse terminate olur
};

int main()
{
    constexpr bool b = noexcept(Myclass{}.~Myclass());
    /*
        true döner
        dilin kurallarına göre dtor'lar noexcept olmalı
    */
}

```

Bu fonksiyonlar noexcept garantisi vermelii

-move ctor

-swap

-memory deallocation

```

class Nec
{
    public:
        Nec()noexcept;
};

class Myclass
{
    Nec nec;
    // derleyici default ettiği special fonkisyonların kendi noexcept yapma
    kararını verir
};

int main()
{
    /*
        Eğer Nec'in ctor noexcept ise true değilse false
    */
    constexpr bool b = noexcept(Myclass{});
}

```

GENERİC PROGRAMLAMA IN CPP (TEMPLATES)

C'de türden bağımsız kod

```

void gswap(void *vp1, void* vp2, size_t sz)
{
    char* p1 = (char*)vp1;
    char* p2 = (char*)vp2;

    while (sz--)
    {
        char temp = *p1;
        *p1++ = *p2;
        *p2++ = temp;
    }
}
void greverse(void *vpa, size_t size, size_t sz)
{
    char* ps = (char*)vpa;
    char* pe = ps + (size - 1) * sz;

    while (ps < pe)
    {
        gswap(ps, pe, sz);
        ps+= sz;
        pe -= sz;
    }
}
#define swap_fn(t) swap_##t(t* p1, t *p2) ?
{
    t temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

swap_fn(int)

```

Template Kategorileri

- function template
- class template
- variable template
- alias template

```
// Genel Syntax
/*
template paramters <> açısal parantez içinde template parametre'leri olur.

template parametres:
    type parameters
    non-type parametre
    template parametre

    // ikisiide aynı anlama gelir T yerine herhangi bir tür gelebilir (int, string
vb)
    template <typename T>
    template <class T>

    template <typename T, typename U>
    template <typename>
*/

```

```
template <typename T>
class MyClass
{
public:
    T foo(T *);
```

```
template <typename T, typename U>
class MyClass
{
public:
    T foo(U);
    // U ve T farklı ya da aynı tür olabilir
};
```

```
template <int N>
class MyClass
{
    int a[N];
};
```

```
template<typename T, int N>
void func(T(&)[N]) // array, string
{
```

```
template<typename T, std::size_t N>
struct Array
{
    T a[N];
};

// aslında

int main()
{
    std::array<double, 20> x;
```

```
///////////
// T type parametre x type parametre türünden non type parametre
template <typename T, T x>

class Myclass
{

};

// MyClass<long, 20L>

/*
Derleyici türü nasıl anlar

1. deduction
fonksiyon / sınıf (Cpp17)

2. explicit template argument

3. varsayılan arguman

*/
```

2023.10.18

Template

template argumanları derleyici hangi yöntemlerle anlar:

- deduction
- CTAD (Class Template Argument Deduction)

```
std::vector vec{1,2,3,4}; // vector template olmasına rağmen CTAD ile böyle tanımladık.
```

Template Argument Deduction

```
template<typename T>
void foo(T)
{
}

int bar(int);

int main()
{
    // T için yapılan çıkışım:
    foo(19); // int
    foo(1.9) // double

    const int ival = 4;
    foo(ival) // int (const'Luk düşer)

    int y{};
    int& r = y;
    foo(r); // int (ref'Luk düşer)

    int a[5]{};
    foo(5); // int*

    const int a[5]{};
    foo(a); // const int*

    foo(bar); //int (*)(int) // function pointer
}
```

```

// Derleyicinin yaptığı çıkarımı görmek için kullanılabilir
template <typename T>
class TypeTeller;

template <typename T>
void foo(T)
{
    TypeTeller<T> x; // hatalı çünkü incomplete type
}

int main()
{
    foo("bar"); // const char*
}

```

```

template<typename T>
void foo(T, T);

int main()
{
    foo(12, 5); // ikiside int çıkarımı yapılır
    foo(12, 5.8); // ambiguity T için tek bir çıkarım var olabilir

    foo("naci", "cemal"); // string
}

```

```

template <typename T>
void foo(T&)
{
    TypeTeller<T> x;
}

int main()
{
    const int x = 10;
    foo(x); // const int çıkarımı yapılır ref semantигinden dolayı

    int a[4]{};
    foo(a);

    const int a[4]{};
    foo(a);
}

```

```

// mülakat sorusu:

template<typename T>
void foo(T&, T&);

int main()
{
    foo("can", "eda"); // const char[4]
    foo("naci", "cemal"); // const char[5] ve const char[6] ambiguity var
}

```

```

template<typename T>
void foo(T&, T);

int main()
{
    int a[5]{};
    foo(a, a);
    // illegal çünkü ilki için a[5] ikincisi için *a çıkarımı yapılacak
}

```

```

template<typename T>
void foo(T&);

int bar(int);

int main()
{
    foo(bar); //int (int)
}

```

Forwarding Reference ya da Universal Reference

```

template<typename T>
void func(T&&) {}

int main()
{
    int x = 10;
    const double dval = 4.5;

    // bu fonksiyona ne gönderirsek gönderilem legal olacak
    func(x); // L value
    func(45) // R value
}

```

1. ihtimal

arguman L value

2. ihtimal

arguman R value

Reference Collapsing:

T& &

T& &&

T&& &

-Yukarıdakiler için T& çıkarımı yapılır

-T&& && için ise T&& çıkarımı yapılır

```
class Nec{};

using ref = Nec&;
using refref = Nec&&;

int main()
{
    Nec mynec;
    ref& r = Nec{}; // r'in türü Nec& & 'ten Nec&
    ref&& r = Nec{}; // r'in türü Nec& &&'ten Nec&

    refref&& r = Nec{}; // r'in türü Nec&& &&'ten Nec&&

}
```

```
template<typename T>
void func(T&&)
{
    // T için çıkarım Nec& olarak yapılır
    // x'in türü Nec& && olur
    // reference collapsing ile x'in türü Nec&
}

class Nec{};

int main()
{
    Nec mynec;
    // T&&
    func(mynec); // T için çıkarım :Nec&
}
```

```
template <typename T>
void func(T&& x)
{

}

int main()
{
    const int x{};
    func(x); // const int&
}
```

```
template <typename T, int N>
void func(T(&r)[N])
{
}

int main()
{
    int a[20]{};
    double b[10]{};

    func(a); // func<int, 20>(a)
    func(b); // func<double, 10>(b)
    func("melike"); // func<const char, 7>("melike")
}
```

```
template <typename T, typename U>
void func(T (*U) // T 'in çıkışını int(*)(double)
{
}

int foo(double);

int main()
{
    func(foo);
}
```

```
template <typename T>
void func(T)
{
}

int main()
{
    void (*fp1)(int) = func; // func'in int spec
    void (*fp2)(double) = func; // func'in double spec
}
```

```
template <typename T>
void Swap(T& t1, T& t2)
{
    T temp {std::move(t1)};
    t1 = std::move(t2);
    t2 = std::move(temp);
}
```

```
// C++20

template <typename T, typename U>
void func(T x, U y)
{
}

// yukarıdakini kısaltması
void func(auto x, auto y)
{
}
```

Template Functions'larda iki farklı geri dönüş değeri:

- trailing return type
- auto return type

Trailing Return Type

```
auto main() -> int
{
```

```
int bar(int);

int (*foo())(int) // return: int(*)(int)
{
    return bar;
}

// bunun daha anlaşılır kılmak için

auto foo() -> int(*)(int)
{
    return bar;
}
```

trailing return type özellikle:

bir template fonksiyonun geri dönüş değerini fonksiyonun parametreleriyle oluşturalacak değerlerle yapılmış zaman kullanılır.

```
template <typename T, typename U>
auto sum (T x, U y) -> decltype(x + y)
{
}
```

Auto Return Type

```
auto foo(int x)
{
    // derleyici return ifadesinin türünü çıkarımı return ifadesine göre çıkarması
    return x * 1.3;
}
```

```
// ambiguity biri double biri int
auto foo(int x)
{
    if (x > 10)
    {
        return 1;
    }

    return 2.3;
}
```

```
template <typename T>

auto foo(T x)
{
    return x.bar(); // bar fonksiyonun geri dönüş değeri olur
}
```

```

class Myclass
{
    public:
        using value_type = int;
}

template <class T>
void func(T x)
{
    typename T::value_type y{}; // nested type olduğunu belirtmek için
    T::value_type y{}; // bu value_type static bir türmüş gibi ifade ediyor
}

```

```

template <typename R, typename T, typename U>
R sum(const T& x, const U& y)
{
    return x + y;
}

int main()
{
    int x = 45;
    float f = 3.4f;

    auto val = sum <double>(x, f); // val double
    // auto val = sum<double, int, float>(x, f)
}

```

Overloading in Template Functions

```

template<typename T>
void func(T)
{
    std::cout << "function template : " << typeid(T).name() << "\n";
}

void func(int)
{
    std::cout << " func(int) \n";
}/*
    overload olan template değil fonksiyon şablonunundan oluşturulan spec

    void func(int)
    void func<double>(double)
    void func<int>(int)
*/
int main()
{
    func(13.6); // void func<int>(double)
    func(12);   // void func(int) seçilir şablondan üretilen değil
}

```

```
template <typename T>
void func(T) = delete;

void func(int);

int main()
{
    // sadece int arguman olan fonksiyon sytanx hatasi olmıcak
    func(12);
}
```

2023.10.20

Template

```
template <typename T>
class Myclass {};

template <typename T>
class Nec{};

int main()
{
    // MyClass'in int spec Nec'in MyClass<int> spec'inde kullanılır
    Nec<Myclass<int>>;
}
```

```
template <typename T>
class Myclass {};

int main()
{
    // farklı türlerde nesneler
    Myclass<double> dm;
    Myclass<int> im;

    // dm = im; böyle bir dönüşüm yok
}
```

```
template <int>
class Myclass
{};

int main()
{
    // farklı türlerde nesneler
    Myclass<5> dm;
    Myclass<6> im;

    // dm = im; böyle bir dönüşüm yok
}
```

```

template <typename T>
class Myclass
{
public:
    void foo(T x)
    {
        ++x;
    }
};

class Nec{};

int main()
{
    // burda syntax hatalı olmaz çünkü foo'yu daha yazmadı derleyici
    Myclass<Nec> m;
    // burda syntax hatalı olur derleyici foo'yu yazdı ve ++x diye bir şey olmaz
    hata
    Nec mynec;
    m.foo(mynec);
}

```

Template Fonksiyonlarını Tanımlama

Cpp dosyasına yazamayız çünkü derleyici kodu görmeli.

```

// myclass.h
template<typename T>
class Myclass
{
public:
    T bar(T);
    // burada inline olarak tanımlanabilir
    void foo(T x)
    {
        x.f();
        ++x;
        x = 0;
    }
};

```

T bar(T) burada header dosyasında tanımlanabilir.

```

// myclass.hpp
#include "myclass.h"
// burada da tanımlayabiliriz

template <typename T>
T Myclass<T>::bar(T x)
{
    // code
}

```

```
// myclass.h

template<class T, class U>
class Myclass
{
public:
    void foo(int x);
    void bar(T, T)
};

template<typename T, typename U>
void Myclass<T, U>::foo(int x)
{

}

template<typename T>
void Myclass<T,U>::bar(T, T)
{
}
```

```
template <typename T>
class Myclass {};

template <typename T>
void func(const Myclass<T>&); // T'in çıkarımı int oldu

int main()
{
    Myclass<int> x;

    func(x);
}
```

```

template <typename T>
class MyClass{};

bool operator==(const MyClass<int>&, const MyClass<int>&)
{
    // MyClass'in int speclerini karşılaştırabiliriz;
}

template <typename T>
bool operator==(const MyClass<T>&, const MyClass<T>&)
{
    // MyClass bütün spec'lerinin karşılaştırabiliriz
    return true;
}

///////////

template <typename T>
class MyClass
{
public:
    void foo(T&&);
};

int main()
{
    MyClass<int> m;

    m.foo(12); // universal ref değil sağ taraf ref
}

```

```

template <typename T>
class Nec
{
public:
    void myNec()
    {
        // class scope'ta hiçbir farklı yok aşağıdaki ikisinin
        Nec nec
        Nec<T> nec
    }
}

```

```

template<typename T>
class Nec
{
public:
    Nec func(Nec x);

    template<typename U>
    void bar(Nec<U> x)
    {

    }
}

template <typename T>
Nec<T> Nec<T>::func(Nec<T> x)
{
    return x;
}

int main()
{
    Nec<double> x;
    Nec<int> y;

    x.func(y); // error

    x.bar(y) // error yok
}

```

```

template<typename T>
class Nec
{
public:
    template<typename U>
    void func(Nec<U> x)
    {
        std::cout << typeid(*this).name() << "\n"; // Nec<int>
        std::cout << typeid(x).name() << "\n"; // Nec<double>
    }
}
int main()
{
    Nec<int> x;
    Nec<double> y;

    x.func(y);
}

```

```

template<typename T, typename U>

struct Pair
{
    Pair() = default;
    Pair(const T& t, const U& u) : first(t), second(u)
    {

    }

    T first{};
    U second{};
}
int main()
{
    using namespace std;

    pair p1{3, 5.6}; // CTAD
}

```

Untype Parametre

```

template <int n>
class Myclass {};

```

untype parametre:

- tam sayı türleri (int, short)
- enum türleri
- object pointer / reference
- function pointer
- member function pointer

```

// Cpp 17 ve eskileri için bu kod hatalı
template <double n>
class Myclass {};

```

```

template <int (*fp)(int)>
class Myclass {};

int foo(int)

int main()
{
    Myclass<&foo> m;
}

```

```

template <int x>
class Myclass {};

template <long x>
class Myclass {};

int main()
{
    Myclass<5> m1; // syntax hatalı olur
}

// yukarıdaki yerine

template <typename T, T x>
class Myclass {};

int main()
{
    Myclass<int, 5> x;
    Myclass<long, 5> y;
}

```

```

// cpp 17
template <auto n>
class Myclass {};

int main()
{
    // derleyici çıkarır yapar
    Myclass<5> m1;
    Myclass<1u> m2;
    Myclass<'A'> m3;
}

```

Default Template Arguman

```

// default template arguman
template <typename T = int, typename U = long>
class Myclass {};
int main()
{
    Myclass<> m1;
    std::cout << typeid(m1).name(); // Myclass<int, long>
}

```

```

template <int x = 10, int y = 20>
class Myclass {};

int main()
{
    Myclass<3, 5> m1;
    Myclass<3> m2;
    Myclass<> m3;
}

```

```

template <typename T, std::size_t N>
constexpr auto asize(const T(&)[N])
{
    return N;
}

int main()
{
    int a[]{1, 2, 3, 4};

    constexpr auto size = asize(a);
}

```

- a) explicit specialization (full specialization)
- b) partial specialization

Explicit Specialization

```

template <typename T>
class Nec
{
public:
    Nec()
    {
        std::cout << "primary template type T is : "
        << typeid(T).name() << "\n";
    }
};

// explicit specialization
/*
    T 'in int olduğu durumlarda bu kullanılır
*/
template <>
class Nec<int>
{
public:
    Nec()
    {
        std::cout << "explicit template type T is : "
        << typeid(T).name() << "\n";
    }
};

```

```

template <typename T, typename U, int N>
class Myclass{};

template <>
class Myclass<int, double, 20>
{
public:
    Myclass()
    {
        std::cout << "explicit spec. <int, double, 29>\n";
    }
};

```

2023.10.23

Explicit Template

```
template<typename T>
void func(T)
{
    std::cout << 1;
}
//explicit template
template<>
void func(int*)
{
    std::cout << 2;
}

template<typename T>
void func(T*)
{
    std::cout << 3;
}

int main()
{
    int* p= nullptr;
    func(p); // 3 yazar
}
```

```
template<typename T>
void func(T)
{
    std::cout << 1;
}
template<typename T>
void func(T*)
{
    std::cout << 2;
}
//explicit template
template<>
void func(int*)
{
    std::cout << 3;
}

int main()
{
    int* p= nullptr;
    func(p); // 3 yazar ( explicit template)
}

/*
    önce daha spesifik olan template seçilir sonra
    explicit template seçilir
*/
```

Partial Specialization

```
template<typename T>
struct Myclass
{
    public:
        Myclass()
        {
            std::cout << "primary template type T is :" << typeid(T).name() <<
"\n";
        }
};

// partial specialization
template<typename T>
struct <Myclass<T*>>
{
    public:
        Myclass()
        {
            std::cout << "partial spec. T *\n";
        }
};

int main()
{
    Myclass<int> m1; // primary spec
    Myclass<int*> m2; // partial spec
}
```

```
template <typename T, typename U>
struct Mert
{
    Mert()
    {
        std::cout << "primary template\n";
    }
};

template <typename T>
struct Mert<int, T>
{
    Mert()
    {
        std::cout << "partial specialization\n";
    }
};

int main()
{
    Mert<double, int> m1; // primary template
    Mert<int, int> m2; // partial template
}
```

```
template <typename T>
struct Mert
{
    Mert()
    {
        std::cout << "primary template\n"
    }
};

template <typename T, typename U>
struct Mert<std::pair<T, U>>
{
    Mert()
    {
        std::cout << "partial specialization\n"
    }
};
```

```
template <int BASE, int EXP>
struct Power
{
    static const int value = BASE * Power<BASE, EXP - 1>::value;
};

template<int BASE>
struct Power<BASE, 0>
{
    static const int value = 1;
}

int main()
{
    constexpr int val = Power<2, 7>::value;
}
```

```
/*
    alias template (tür es isim şablonu)
    variable template
    variadic templates
    perfect forwarding

*/
```

```

void func(T x)

void foo(T x)
{
    /*
        foo'ya gönderilen arg L value ya da R value olabilir
        arg const ya da non const da olabilir. Eğer func'ta bu durumları
        korursa buna perfect forwarding denir.
    */
    func(x);
}

int main()
{
    foo(arg);
}

```

```

class Myclass {};
void foo(Myclass&)
{
    std::cout << "Myclass&\n";
}

void foo(const Myclass&)
{
    std::cout << "const Myclass&\n";
}

void foo(Myclass&&)
{
    std::cout << "Myclass&&\n";
}
// her foo için farklı call_foo yazabiliriz ama parametre sayısı artıkça işler
zorlaşacak
// bunun yerine

template <typename T>
void call_foo(T&& x)
{
    // std::forward dönüştürüyor
    foo(std::forward<T>(x));
}

int main()
{
    Myclass m;
    const Myclass cm;

    foo(m); // MyClass&
    foo(cm); // const MyClass&
    foo(Myclass{}); // MyClass&&
}

```

Alias Template

```
template <typename T>
using gset = std::set<T, std::greater<T>>;
```

```
int main()
{
    using namespace std;
    set<int, greater<int>> myset;
    gset<int> myset1;
```

Variadic Template

```
template <typename ...TS>
class Myclass {};
```

```
template <typename ...TS>
void func();
```

```
template <int ...VALS>
class Myclass
{
public:
    static constexpr auto x = sizeof...(VALS);
};
```

```
int main()
{
    constexpr auto val = Myclass<1, 3, 7, 6, 8>::x; // 5 olur, parametre sayısını
    verir
}
```

```
template <typename ...Ts> // template parametre pack
void func(Ts ...args) // function parametre pack
```

```
int main()
{
    // void func<int, double, long>
    func(1, 2.3, 45L);
}
```

```
template <typename ...Ts>
void func(Ts&& ...args);

template <typename ...Ts>
void foo(Ts && ...args)
{
    std::forward<Ts>(args)...

    //func(std::forward<int>(p1), std::forward<double>(p2),
std::forward<int>(p3));
}

int main()
{
    int ival{};
    double dval{};

    foo(12, dval, ival);
}
```

```
template <typename ...TS>
void func(TS ...params)
{
    int a[] = {params...};
// int a[] = {p1, p2, p3, p4};
}

int main()
{
    func(1, 5, 7, 9);
}
```

```
class A
{
public:
    void fc();
    void foo();
};

class B
{
public:
    void fc();
    void foo();
};

class C
{
public:
    void fc();
    void foo();
};

template<typename ...TS>
class Myclass : public TS...
{
    using Ts::foo...
};
```

```

int main()
{
    Myclass<A, B, C> m1;

    m1.fa();
    m1.fb();
    m1.fc();
}

```

Recursive Instantiation

```

template <typename T>
void print(const T& t)
{
    std::cout << t << " ";
}

template <typename T, typename ...Ts>
void print(const T&t, const Ts& ...args)
{
    print(args...)
}

int main()
{
    print(1, 2.3, "alican", 4.5f);
}

```

```

template <typename ...Ts>
void print(const Ts& ...args)
{
    using Ar = int[];

    Ar{ ((std::cout << args << '\n'), 0)...};
}

int main()
{
    print(2, 6 ,1.2, "emre", std::string("bahtiyar"), 84234);
}

```

Fold Expressions (katlama ifadeleri)

```

template <typename ...Ts>
auto sum(Ts ...args)
{
    (args + ...) // unary right fold

    // derleyici böyle yapar p1 + (p2 + (p3 + p4))
}

int main()
{
    using namespace std;
    std::cout << sum(1, 3, 6, 5) << "\n";
}

```

Variadic Templates

- recursive function instantiation
- static if
- init list

Fold Expression

- unary right fold
- unary left fold
- binary right fold
- binary left fold

```
class MyClass {};  
template <typename T, typename ...Ts>  
std::unique_ptr<T> MakeUnique(Ts&& ...args)  
{  
    return std::unique_ptr<T>(new T(std::forward<Ts>(args)...));  
}  
  
int main()  
{  
    MakeUnique<MyClass>(2, 3, 16);  
}
```

```
template <typename T>  
constexpr T pi = T(3.143214434212L);  
  
int main()  
{  
    auto x = pi<int>; // x = 3  
}
```

```
template <size_t n>  
constexpr size_t fact = n * fact<n - 1>;  
  
template <>  
constexpr size_t factorial<0> = 1;  
  
int main()  
{  
    auto x = fact<7>;  
}
```

2023.10.25

Template Value Type

```
template <typename T, T v>
struct IntegralConstant
{
    static constexpr T value = v; // sabit
    using value_type = T; // bir tür
    using type = IntegralConstant; // oluşturulan struct kendisi
    constexpr operator value_type()const noexcept {return value;} // tür dönüşüm
operator
    constexpr value_type operator()()const noexcept {return value;}
};

using TrueType = IntegralConstant<bool, true> // true

int main()
{
    IntegralConstant<int, 5>::value; // sabit
    IntegralConstant<int, 5>::value_type; // bir tür
    IntegralConstant<int, 4>::type // IntegralConstant<int, 4>

    constexpr auto val = IntegralConstant<int, 4>{} + IntegralConstant<int, 3>{};
// 7    constexpr auto val = IntegralConstant<int, 5>{}(); // val = 5
}
```

```
template <typename T>
void func(T x)
{
    static_assert(sizeof(T) > 2, "sizeof value must be greater than 2");
}

int main()
{
    func('A');
}
```

```
template<typename T>
struct IsPointer : FalseType {};

template<typename T>
struct IsPointer<T*> : TrueType {};

template <typename T>
void func(T x)
{
    static_assert(IsPointer<T>::value, "only for pointer type");
}

int main()
{
}
```

```
#include <type_traits>

template<typename T>
void func(T)
{
    static_assert(std::is_pointer_v<T>);
    // pointer olmayan türden arguman gönderirsek syntax hatalı verir
}

int main()
{
```

```
template<typename T>
struct RemoveReference
{
    using type = T;
};

template<typename T>
struct RemoveReference<T&>
{
    using type = T;
};

template<typename T>
struct RemoveReference<T&&>
{
    using type = T;
};

template<typename T>
using RemoveReference_t = typename RemoveReference<T>::type;

int main()
{
    RemoveReference<int>::type // int
    RemoveReference<int&>::type // int
    RemoveReference<int&&>::type // int
}
```

Tag-Dispatch

```
// r value gelince ayrı bir kod l value gelince ayrı kod

// tag-dispatch
#include <type_traits>
#include <iostream>

void bar(std::true_type)
{
    std::cout << "implementaion for l values\n";
}

void bar(std::false_type)
{
    std::cout << "implementaion for r values\n";
}

template<typename T>
void func(T&&)
{
    bar(std::is_lvalue_reference<T>{});
}

int main()
{
    func(12); // implementaion for r values

    int x{ 21 };

    func(x); // implementaion for l values
}
```

```
// tam sayı türleri için ayrı implementasyon olmayanlar için ayrı

template<typename T>
void func_impl(T, std::true_type)
{
    std::cout << "tam sayı türleri için\n";
}
template<typename T>
void func_impl(T, std::false_type)
{
    std::cout << "tam sayı olmayan türler için\n";
}
template <typename T>
void func(T x)
{
    func_impl(x, std::is_integral<T>{});
}

int main()
{
    func(12); // tam sayı türleri için
    func(12.2); // tam sayı olmayan türler için
}
```

Static If (cpp 17)

```
template <typename T>
auto get_value(T x)
{
    if constexpr(std::is_pointer_v<T>)
    {
        return *x;
    }
    else
    {
        return x;
    }
}

int main()
{
    int x = 5;
    double dval = 4.971;

    int* ip {&x};
    double* dp{&dval};

    std::cout << get_value(x) << "\n";
    std::cout << get_value(dval) << "\n";
    std::cout << get_value(ip) << "\n";
    std::cout << get_value(dp) << "\n";
}
```

Standart Template Library

Iterators

```
template<typename Iter>
void print_array(Iter beg, Iter end)
{
    while(beg != end)
    {
        std::cout << *beg << ' ';
        ++beg;
    }

    std::cout << "\n"
}

int main()
{
    int a[5] = {1 , 3 , 4, 5};
    print_array(a ,a + 5);

    std::vector<double> dvec{1.2, 4.5, 7.3, 9.3, 2.45};

    print_array(dvec.begin(), dvec.end());

    std::list<std::string> names{"meliike", "tamer", "serhat", "burak"};
    print_array(names.begin(), names.end());
}
```

```

template<typename T, typename A>
class Vector
{
    class iterator
    {
        public:
            T& operator*();
            bool operator!=(const Iterator&) const;
            operator++();
            operator++(int);
    };

    iterator begin();
    iterator end();
}

```

```

int main()
{
    vector<int> ivec(1000);

    auto bg = ivec.begin();
    auto iter = begin(ivec); // global
}

```

Iterator Category

iterator category:

- input_iterator
- output_iterator
- forward_iterator
- bidirectional_iterator
- random_access_iterator

STL Container'ların iterator category'si belirlidir.

```

template<typename Iter>
void func(Iter beg, Iter end)
{
    // compile time
}

//std::output_iterator_tag
//std::input_iterator_tag
//std::forward_iterator_tag
//std::bidirectional_iterator_tag
//std::random_access_iterator_tag

int main()
{
    vector<string>::iterator::iterator_category // iterator_category söyler
}

```

```
template<typename T, typename U>
struct IsSame : std::false_type {};

template<typename T>
struct IsSame<T, T> : std::true_type {};

template<typename T, typename U>
constexpr bool IsSame_v = IsSame<T, U>::value;

int main()
{
    IsSame_v<int, double> // false
    IsSame_v<int, int> // true
}
```

```
template<typename Iter>
void algo(Iter beg, Iter end)
{
    if constexpr(std::is_same_v<Iter::iterator_category,
std::random_access_iterator_tag>)
    {
        ///
    }
}
```

2023.10.27

Algorithm

```
// copy
// Not : algoritmalar exception throw etmez
template <typename InIter, typename OutIter>
OutIter Copy(InIter beg, InIter end, OutIter destbeg)
{
    /*
        -beg arttıır arttıır ama hiçbir zaman end eşit olmazsa tanımsız davranış
        -destbeg'dan sonra container'da üye yoksa tanımsız davranış
        -eğer bir yazma işlemi yapıyorsak fonksiyonun geri dönüş değeri en son
        yazdığımız konumdan sonraki yerdir.

    */
    // beg en az input_iterator içeriğine sahip olmalı çünkü InIter
    // destbeg en az output_iterator içeriğine sahip olmalı çünkü OutIter

    while (beg != end)
    {
        *destbeg++ = *beg++;
    }
    return destbeg;
}

// farklı container üzerinde kopyalama işlemleri yapabiliriz.
}

int main()
{
    using namespace std;

    vector<int> ivec{ 2, 5, 7, 9, 1, 3};
    list<int> ilist;

    // tanımsız davranış çünkü list'e boş
    Copy(ivec.begin(), ivec.end(), ilist.begin());

    list<int> ilist1(10);
    auto iter = Copy(ivec.begin(), ivec.end(), ilist1.begin());
    ilist.begin(), iter // algoritma'nın写的 öğeler var
}
```

Algoritmaların parametre değişkenleri iterator'dür: Böylece bir algoritma örneğin şunları yapabilir:

- Bir iterator konumundaki nesneye atama yapabilir.
- Aynı range'deki iki nesneyi takas edebilir.
- Range'in ilişkin olduğu olduğu kaba ekleme ve silme yapamaz

```

int main()
{
    const char* const p[] = { "emre" , "bilge", "damla", "tamer", "gokhan" };
    vector<string> svec(6);

    copy(begin(p), end(p), begin(svec));
}

```

Predicate: bool döndüren callable'lar denir. bkz bool isEven(int);

Copy If

```

template<typename Inter, typenameOutIter, typename Upred>
OutIter CopyIf(Inter beg, InIter end, OutIter destbeg, Upred f)
{
    while (beg != end)
    {
        if (f(*beg))
        {
            *destbeg++ = *beg;
        }
        ++beg;
    }

    return destbeg;
}
bool isEven(int x)
{
    return x % 2 == 0;
}

int main()
{
    using namespace std;

    vector<int> ivec{2, 5, 7, 10, 4, 3, 6, 8, 1, 9};
    list<int> ilist(10);

    CopyIf(ivec.begin(), ivec.end(), ilist.begin(), iseven);

    int n;
    cin >> n;

    CopyIf(ivec.begin(), ivec.end(), ilist.begin(),
           [n](int x) {return x % n == 0; } ); // Lambda ifadesi
    /*
        Lambda ifadelerinde derleyici bir class yazar ve
        geçici nesne oluşturur (closure object)
    */
}

```

Count If

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

    vector<string> svec;
    rfill(svec, 100'000, rname);

    cout << count(svec.begin(), svec.end(), "tamer") << "\n";

    char c = 'e';
    count_if(begin(svec), end(svec), [c](const string& s)
    {
        return s.find(c) != std::npos;
    });

    sort(svec.begin(), svec.end()); // küçükten büyüğe vectörü sıralar
}
```

Iter Const'luk

```
int main()
{
    using namespace std;
    vector<string> svec{ "ali", "can", "ece", "tan" };

    const vector<string>::iterator iter = svec.begin();

    ++iter; // syntax hatalı
    *iter = "murathan"; // legal
    iter.operator*() = "murathan" // böyle bir fonksiyon var const olan

    vector<string>::const_iterator iter = svec.begin();
    ++iter; // legal
    *iter = "murathan" // syntax hatalı

    auto iter = svec.begin(); // const değil
    iter = svec.cbegin(); // const

    auto iter = end(svec); // const değil
    iter = cend(svec); // const
}
```

Reverse Iterator

```
int main()
{
    vector<string> svec{ "ali", "can", "ece", "tan" };
    vector<string>::reverse_iterator iter = svec.begin();

    cout << *iter << "\n"; // tan
    cout << *iter++ << "\n" // ece

    auto it = iter.base(); // normal iter döndürür
}
```

Find

```
InIter Find(InIter beg, InIter end, const Key& val)
{
    while (beg != end)
    {
        if (*beg == val)
            return beg;
        ++beg;
    }

    return beg;
}

int main()
{
    using namespace std;

    vector<int> ivec{ 4, 6, 7, 8, 1, 3, 98};

    int ival = 1;

    auto iter = Find(ivec.begin(), ivec.end(), ival);

    if (iter != ivec.end())
    {
        cout << "bulundu " << *iter << "\n";
        cout << "indeks = " << iter - ivec.begin() << "\n";
    }

    auto riter = Find(ivec.rbegin(), ivec.rend(), ival);

    cout << *iter << "\n" // 1 yazar
    cout << *iter.base() << "\n" // 3 yazar
}
```

2023.10.30

Reverse Iter

```
template<typename InIter>
void PrintRange(InIter beg, InIter end)
{
    while (beg != end)
    {
        std::cout << *beg++ << ' ';
    }
}
int main()
{
    using namespace std;

    vector<int>ivec;
    rfill((ivec, 20, Irand{0, 100});

    auto riter_beg = ivec.rbegin();
    auto riter_end = ivec.rend();

    PrintRange(riter_beg, riter_end); // sondan başa doğru yazar
    PrintRange(riter_end.base(), riter_beg.base()); // baştan sona doğru yazar
}
```

Back Insert Iterator

```
template <typename InIter, typename OutIter>
OutIter Copy(InIter beg, InIter end, OutIter destbeg)
{
    while (beg != end)
    {
        *destbeg++ = *beg++;
    }
    return destbeg;
}/*
Yukarıdaki Copy algoritmasına dokunmadan ona çağrı yapırı yapacağımız çağrı dest_vec source_vec'teki öğeleri sondan ekleyecek
*/
template<typename Container>
class BackInsertIterator
{
public:
    BackInsertIterator(Container &c) : rc(c) {}
    BackInsertIterator& operator++() {return *this;}
    BackInsertIterator& operator++(int) {return *this;}
    BackInsertIterator& operator*() {return *this;}
    BackInsertIterator& operator=(const typename Container::value_type& val)
    {
        rc.push_back(val);
        return *this;
    }
}
```

```

int main()
{
    using namespace std;
    vector<int> source_vec {2, 5, 1, 3, 4, 6, 9, 7};
    vector<int> dest_vec; // empty vector

    BackInsertIterator<vector<int>> iter(dest_vec);
    BackInsertIterator iter(dest_vec);

    Copy(source_vec.begin(), source_vec.end(), BackInsertIterator(dest_vec));

    print(dest_vec); // 2, 5, 1, 3, 4, 6, 9, 7
}

```

Yukarıdaki Kodun STL ile yazılmış hali:

```

#include<iterator>
#include<algoritm>

int main()
{
    using namespace std;
    vector<int> source_vec {2, 5, 1, 3, 4, 6, 9, 7};
    vector<int> dest_vec;

    copy(source_vec.begin(), source_vec.end(), back_inserter(dest_vec));
    print(dest_vec);

}

```

Back Inserter

```

#include<iterator>
#include<algoritm>

int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 10'000, [] {return rname() + ' ' + rfname();});

    vector<string> dvec;
    size_t len = 13;

    // uzunluğu 13 olanlar kopyalanır
    copy_if(svec.begin(), svec.end(), back_inserter(dvec),
    [len](const string& s) {return s.length() == len;});
}

```

Front Inserter

```
#include<iterator>
#include<algortihm>

int main()
{
    using namespace std;
    vector<int> source_vec {2, 5, 1, 3, 4, 6, 9, 7};
    list<int> ilist;

    copy(source_vec.begin(), source_vec.end(), front_inserter(ilist));
    print(dest_vec);

}
```

STL'de iteratorleri manipüle eden algoritmalar:

- -advance (bir iter'i npos artırmak için kullanılır) advance(ref)
- -distance (iki iterator arasındaki farkı buluyor) distance(iter1, iter2)
- -next (iter, 5) 5 ilerisini alıyor
- -prev (iter, 5) 5 gerisini veriyor
- -iter_swap (iter_x, iter_y) iter'in x konumuyla y konumu yer değiştirir.

```
using iter = std::vector<int>::iterator

int main()
{
    iter::value_type type; // iter türü yani int
    iter::difference_type diff_type; // iki iter farkından oluşan tür ptrdiff_t
    iter::pointer pointer_type; // int*
    iter::reference ref_type; // int&
    iter::iterator_category; // random_access_iterator_tag
}
```

Iterator Category'sine Göre Fonksiyon Yazma

```
// iterator category'sine göre func_impl etme
// tag dispatch
template <typename Iter>
void func_impl(Iter beg, Iter end, std::random_access_iterator_tag) {}

template <typename Iter>
void func_impl(Iter beg, Iter end, std::bidirectional_iterator_tag) {}

template <typename Iter>
void func_impl(Iter beg, Iter end, std::forward_iterator_tag) {}

template<typename Iter>
void func(Iter beg, Iter end)
{
    func_impl(beg, end, typename Iter::iterator_category{});
}
```

```
// static if

template<typename Iter>
void func(Iter beg, Iter end)
{
    using cat = typename std::iterator_traits<Iter>::iterator_category;

    if constexpr (std::is_same_v<cat, std::bidirectional_iterator_tag>)
    {
        std::cout << " bidirectional_iterator_tag" << "\n";
    }
    else if constexpr (std::is_same_v<cat, std::random_access_iterator_tag>)
    {
        std::cout << "random_access_iterator_tag" << "\n"
    }
}
```

Advance

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

    vector<int> ivec{ 2, 4, 6, 7, 9, 3, 1};
    list<int> ilist{ 2, 4, 6, 7, 9, 3, 1};

    auto vec_iter = ivec.begin();
    auto list_iter = ilist.begin();

    // 3 adım ilerler
    // ikisi farklı fonksiyonlar overloading var complete time'da kod seçimi
    // type dispatch örneği
    advance(vec_iter, 3);
    advance(list_iter, 3);

}
```

```

template<typename Iter>
void Advance(Iter& it, int n, std::random_access_iterator_tag)
{
    it += n; // vector
}
template<typename Iter>
void Advance(Iter& it, int n, std::bidirectional_iterator_tag)
{
    while(n--)
        ++it; // list
}

template<typename Iter>
void Advance(Iter& it, int n)
{
    Advance(it, n typename std::iterator_traits<Iter>::iterator_category{});
}

```

```

int main()
{
    using namespace std;

    vector<int> ivec{ 2, 4, 6, 7, 9, 3, 1};
    auto iter = ivec.end();
    advance(iter, -3); // *iter 9
}

```

Distance

```

int main()
{
    using namespace std;

    list mylist { 2, 5, 8, 9, 3, 1, 8};

    auto iter1 = mylist.begin();
    auto iter2 = mylist.end();

    advance(iter1, 2); // *iter = 8
    advance(iter2, -1); // *iter 1

    auto n = distance(iter1, iter2); // n = 4
}

```

Next and Prev (CPP 11)

```
// next and prev --cpp11

int main()
{
    using namespace std;

    vector<string> svec{ "ali", "can", "ece", "naz", "gul", "eda", "tan"};

    auto iter = next(svec.begin(), 3); // *iter = naz
    iter = next(svec.begin()); // *iter = can
    list<string> slist{ "ali", "can", "ece", "naz", "gul", "eda", "tan"};

    std::cout << *slist.end() << "\n"; // tanımsız davranış
    std::cout << *prev(slist.end()) << "\n"; // tan
}
```

advance: ref parametreli iter'in kendisini artırıyor (call_by_reference)

next: iter değişmez ancak assign etmem gerekiyor (call_by_value)

Swap

```
template <typename Iter1, typename Iter2>
void IterSwap(Iter1 it1, Iter2 it2)
{
    std::swap(*it1, *it2);
}

///////

int main()
{
    using namespace std;
    vector<string> svec{ "ali", "can", "tan", "ata"};
    list<string> slist{ "gul", "eda", "naz", "ela"};

    iter_swap(next(svec.begin()), prev(slist.end()), 2);
}
```

Find If

```
// find_if
template<typename InIter, typename Pred>
InIter FindIf(InIter beg, InIter end, Pred f)
{
    while(beg != end)
    {
        if(f(*beg))
            return beg;
        ++beg;
    }
    return end;
}
int main()
{
    using namespace std;

    list<string> slist;
    rfill(slist, 20, name);
    print(slist);

    {
        char c = '0';

        auto iter = find_if(slist.begin(), slist.end(), [c](const string &s)
        {
            return s.contains(c);
        });

        if (iter != slist.end())
        {
            std::cout << "bulundu... idx = " << distance(slist.begin(), iter) <<
'\n';
        }
        else
        {
            std::cout << "bulunamadi" << "\n";
        }
    }

    // ya da
    char c = '0';
    if (auto iter = find_if(slist.begin(), slist.end(), [c](const string &s)
    {
        return s.contains(c);
    }); iter != slist.end())
    {
        std::cout << "bulundu... idx = " << distance(slist.begin(), iter) <<
'\n';
    }
    else
    {
        std::cout << "bulunamadi" << "\n";
    }
}
```

Transform

```
template<typename InIter, typename OutIter, typename F>
OutIter Transform(InIter beg, InIter end, OutIter destbeg, F f)
{
    while (beg != end)
    {
        *destbeg++ = f(*beg++);
    }
    return destbeg;
}

/////////
auto get_len(const std::string& s)
{
    return s.size();
}
int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 100, rname);
    list<size_t> lenlist;

    transform(svec.begin(), svec.end(),
              back_inserter(lenlist), get_len);

    print(lenlist); // print svec'tekilerin size'ni yazar
}
```

```
std::string revstr(std::string s)
{
    /*
    std::string temp(s);
    std::reverse(temp.begin(), temp.end());

    return temp;
    */

    std::reverse(s.begin(), s.end());
    return s;
}

int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 100, rname);
    list<size_t> lenlist;

    transform(svec.begin(), svec.end(),
              svec.begin(), revstr);
}
```

```
// transform 2. overLoad'u

template<typename InIter1, typename InIter2, typename OutIter, typename F>
OutIter Transform(InIter1 beg, InIter1 end, InIter2 beg2, OutIter destbeg, F f)
{
    while (beg != end)
    {
        *destbeg++ = f(*beg++, **beg2++);
    }

    return destbeg;
}
```

```
int main()
{
    using namespace std;
    vector<int> v{ 1, 4, 7, 2, 9, 9, 3 };
    deque<int> d{ 2, 5, 8, 1, 9, 3, 5};

    list<int> ilist;

    transform(v.begin(), v.end(), d.begin(), back_inserter(ilist)
              [] (int x, int y) {return x * x + y * y});
}

}
```

2023.11.01

Lambda Expression

```
/*
Lambda expression

Lambda ifade kullandığım yerde bir sınıf nesnesi kullanmış oluyorum.
Closure Object oluşturur (PR Value)

Lambda introducer
[Lambda introducer](parametre değişkeni){fonksiyon ana bloğu(code)}

[](){//code}
[]()constexpr{//code}
[]()mutable{//code}
[]()-> type{//code}
[]()noexcept{//code}

eğer parametresi olmıyorsa parametre parantezini kullanmayabiliriz:
[]{}

Ama Cpp 23 ile gelen bir özellikle niteleyicilerden herhangi birini
kullanıyorsak, yine de parametre değişkeni kullanmayabiliriz. Daha
eski Cpp standartlarında kullanmak zorundayız.

[]mutable{} cpp23
[]()mutable{} cpp23'ten eskiler

*/
```

```
int main()
{
    []() {}(); // boş Lambda ifadesi
```

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

```
template<typename F>
void func(F f)
{
    auto val = f(12);
}

int main()
{
    func([] (int a) {return a * a + 34; })
}
```

```

int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 100, rname);
    // Lambda ifadelerin en tipik kullanıldığı yer
    count_if(svec.begin(), svec.end(), [](const string& s) {return s.length() == 7});
}

```

```

int main()
{
    auto val = [] (int x) { return x * x + 5; }(20); // x = 20 verdil
}

```

```

// Lambda function içinde statik ömürlü nesneleri doğrudan isimleriyle
// kullanabiliriz
int g = 5;
int main()
{
    // g'yi burda tanımlarsak syntax hatası
    auto f = [] (int a) {return a * g;};
}

```

Member Mutable Parametre vs Lambda Mutable Expression

```

class Myclass
{
    mutable int x; // const fonksiyonlar bile x'i değiştirebilir.
};

// yukarıdaki ve aşağıdaki mutable kavramları birbirinden tamamen farklı
int main()
{
    int x = 5;
    // illegal çünkü derleyeci const fonksiyon yazar ++x yapamaz
    auto f = [x]() {++x;};
    // const fonksiyon olarak yazmaz eğer mutable kullanırsak
    f = [x]()mutable{++x;};
    string str {"emre"};

    f = [str]() mutable{str[0] = 'a';};

    int k = 3, l = 4, z = 9;

    f = [x, y, z](int a)
    {
        return a * (x + y + z);
    };
    // capture all by copy
    f = [=](int a)
    {
        return a * (x + y + z);
    };
}

```

Capture All by Reference

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

    int a = 67;
    [&a]()
    {
        ++a; // call_by_reference
    };
    int b = 21;
    int c = 1;
    // capture all by reference
    auto f = [&]() {};

    //trailing return type
    f = [](int x)->double
    {
        return x * x; // return double
    };
}
```

```
class xyz_12_nct
{
public:
    template<typename T>
    auto operator()(T x)
    {
        return x * x;
    }
}
int main()
{
    // template Lambda
    auto f = [](auto x) // cpp 14
    {
        return x + x;
    };
}
```

```

int main()
{
    // ikisi farklı türler
    auto f1 = []() {};
    auto f2 = []() {};

    std::is_same_v<decltype(f1), decltype(f2)>; // false

    auto f3 = f1; // bu kod geçerli ( copy ctor'ları var )
/*
    Cpp 20 ve sonrasında bunun geçerli olması için Lambda fonksiyon
    stateless olmalı
    int x = 5;
    auto f1 = [x](); {} // statefull
*/
    decltype(f1) f4; // cpp20 ve sonrası geçerli
}

```

Positive Lambda

```

// positive Lambda

int main()
{
    auto x = [](int a) {return a * a}; // closure type

    std::cout << typeid(x).name() << '\n';

    x = +[](int a) {return a * a}; // function pointer
    std::cout << typeid(x).name() << '\n';
}

```

Noexcept Lambda

```

int main()
{
    auto f = [](int x) {return x * x};
    noexcept((f(12))); // false

    f = [](int x) noexcept{return x * x};
    noexcept((f(12))); // true
}

```

Constexpr Lambda

```

int main()
{
    // constexpr olmasını engellecek bir durum olursa syntax hatalı verecek
    auto f = [](int x) constexpr
    {
        static int x = 10;
        return x * 2;
    } // syntax hatalı
}

```

```

template <typename T>
class Myclass
{
};

int main()
{
    Myclass<decltype>([]{}) m1; // cpp 20
}

```

```

template<auto x = []{}>
struct Myclass
{
    inline static int ival = 5;
};

int main()
{
    Myclass<> m1;

    m1.ival++;
    m1.ival++;
    m1.ival++;

    Myclass<> m2;
    // m1 ve m2 farklı objeler
    std::cout << m1.ival << "\n"; // 8
    std::cout << m2.ival << "\n"; // 5
}

```

Lambda İfadeleriyle:

- 1) isimlendirilmiş değişken halen getirip kullanabiliriz
- 2) bir fonksiyon şablonuna arguman olarak gönderebiliriz

```

vector<int> ivec {1, 2, 3, 4, 5, 6}
sort(ivec.begin(), ivec.end(), [](int a, int b)
{
    return abs(a) < abs(b);
});

```

2023 11 03

std::for_each

```
template <typename Iter, typename F>
F ForEach(Iter beg, Iter end, F f)
{
    while (beg != end)
    {
        f(*beg++);
    }
    return f;
}

///////////

int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 1000, []{return rname() + ' ' + rfname(); });

    for_each(svec.begin(), svec.end(), [](const auto& s)
    {
        cout << s << ' '; // her eleman için dönecek
    });
}
```

```
class Functor
{
public:
    void operator ()(const std::string& s)
    {
        if (s.size() > 12)
            ++m_count;
        std::cout << s << '\n';
    }

    int get_count()const
    {
        return m_count;
    }
private:
    std::size_t m_count{};
};

int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 100, []{return rname() + ' ' + rfname(); });

    auto f = for_each(svec.begin(), svec.end(), Functor{});
    std::cout << f.get_count() << '\n';
}
```

any_of, all_of and none_of

```
avoid raw Loops

vector<Fighter> fighter_vec;

bool flag = false;

for(size_t i{}; i < fighter_vec.size(); ++i)
{
    if (is_wounded(fighter_vec[i]))
    {
        flag = true;
        break;
    }
}

böyle bir kod uygun değil

auto is_wounded=
if (any_of(x.begin(), x.end(), is_wounded)
```

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

    vector<string> svec;
    rfill(svec, 100, rname);

    if (any_of(svec.begin(), svec.end(), [](const string& s)
    {
        return s.contains('x'))
    })
    {
        // eğer bu vector'lerin içinde x geçiyorsa
    }

    /*
        any_of : herhangi biri
        all_of : hepsi
        none_of : hiçbiri
    */

    vector<int> ivec;
    // vector boş
    // true
    bool b = all_of(ivec.begin(), ivec.end(), [](int x) {return x % 2 == 0;});
    // true
    b = none_of(ivec.begin(), ivec.end(), [](int x) {return x % 2 == 0;});
    // false
    b = any_of(ivec.begin(), ivec.end(), [](int x) {return x % 2 == 0;});
}
```

```
// replace
int main()
{
    using namespace std;
    vector<int> ivec;

    rfill(ivec, 100, rand {0, 9})
    // değeri 5 olanları 9999 yapacak
    replace(ivec.begin(), ivec.end(), 5, 9999);
}
```

Bazı Algoritmaların Sonunda:

`reverse_copy`,`replace_copy`,`remove_copy`,`reverse_copy_if`,`replace_copy_if`,`remove_copy_if`

copy bulunur. Yapılan değişikliği başka yere kopyalar.

`std::reverse_copy`

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

    vector<string> svec;
    rfill(svec, 20, rname);

    list<string> slist;
    // svec tersi slist'te kopyalandı
    reverse_copy(svec.begin(), svec.end(), back_inserter(slist))

    vector<int> ivec{ 1, 2, 3, 1, 2, 4, 1, 5, 1, 1, 9};
    vector<int> desvec(10);

    // 1 dışındakileri kopyalar
    auto iter = remove_copy(ivec.begin(), ivec.end(), desvec.begin(), 1);

    std::cout << "toplam" << distance(destvec.begin, iter) << "eleman kopyalandı";
}
```

`std::remove_copy_if`

```
int main()
{
    using namespace std;
    vector<Date> dvec;
    rfill(dvec, 100, Date::random);
    vector<Date> dest_vec;
    // kasım ayı hariç kalanları dest_vec kopyalar
    remove_copy_if(dvec.begin(), dvec.end(), back_inserter(dest_vec),
        [](&const Date& d)
    {
        return d.month() == 11;
    });
}
```

`std::replace_copy` and `std::replace_copy_if`

```
// replace_copy and replace_copy_if
int main()
{
    using namespace std;
    vector<int> ivec;

    rfill(ivec, 100, Irand {0, 9});

    vector<int> dest_vec;
    replace_copy(ivec.begin(), ivec.end(), back_inserter(dest_vec), 5, 3333);

    vector<int> dest_vec_if;

    replace_copy_if(ivec.begin(), ivec.end(), back_inserter(dest_vec_if),
                   [] (int x) {return x % 2 == 0;}, -1);
}
```

Containers

Bir işi 2 farklı bir biçimde yapabiliriz. Biri container sınıfının member fonksiyonunu çağrarak diğer ise algoritmaları kullanarak. Hangisin tercih etmeliyiz.

Eğer container sınıfı üye fonksiyonları işimi görüyorsa onları kullanmamız yeterli.

```
container'ların ortak member fonksiyonları:
.size()
.empty()
.clear()
.erase() // 2 overloading
```

Container tipleri:

Sequence containers

- `std::vector`
- `std::deque`
- `std::list`
- `std::forward_list`
- `std::array`

(array hariç) özel fonksiyonlar:

- `front`
- `back`
- `resize`

Associative Containers

- std::set
- std::multiset
- std::map
- std::multimap

Unordered Associative Containers

- std::unordered_set
- std::unordered_multiset
- std::unordered_map
- std::unordered_multimap

std::vector

```
template<typename T>
class Vector
{
public:
    void push_back(T&&)
    {
        // bir nesnenin move ctor çağrılır
        std::move(r)
    }; // R reference
    void push_back(const T&)
    {
        // copy ctor çağrılır
    }; // L reference

    template<typename U>
    void foo(U&&) // Universel reference
}
```

Container Emplace Fonksiyonları

```
template<typename T>
class Vector
{
public:
    void push_back(T&&)
    {
        // bir nesnenin move ctor çağrılır
        std::move(r)
    }; // R reference
    void push_back(const T&)
    {
        // copy ctor çağrılır
    }; // L reference
    template<typename ...Args>
    void emplace_back(Args&& ...args)
    {
        // perfect forwarding
        T(std::forward<U>(args)...)
    }
}
```

2023 11 06

std::vector

```
class Myclass
{
    public:
        Myclass(int);
};

int main()
{
    using namespace std;

    vector<int> ivec(100);

    auto size = ivec.size();
    auto cap = ivec.capacity();

    bool isEmpty = ivec.empty();

    ivec.reverse(6'000'000); // sürekli reallocation yapmaması için

    // vector ctor
    vector<int> ivec; // default ctor : boş vector
    vector<int> ivec1(20); // value init

    vector<Myclass> mvec(20); // illegal çünkü default ctor'u yok

    vector<int> vec1(12); // 12 tane 0 var
    vector<int> vec2{ 12 }; // 1 tane 12 tane var

    // range ctor
    list<int> ilist{ 2, 5, 6, 1, 9, 1 };
    vector<int> ivec2(ilist.begin(), ilist.end());
    vector<double> dvec(ilist.begin(), ilist.end()); // Legal

    const char const p[] = { "furkan", "emre", "gano" };
    vector<string> svec(begin(p), end(p)); // Legal
}
```

```
// range assign
int main()
{
    using namespace std;
    vector<int> ivec;

    rfill(ivec, 100, Irand{ 0, 40 });
    print(ivec);

    auto myset = set<int>{ ivec.begin(), ivec.end() };

    // range assign
    ivec.assign(myset.begin(), myset.end());
}
```

```

int main()
{
    using namespace std;

    list mylist{ 2, 5, 1, 3, 2, 7, 1, 8};

    // auto -> vector<list<int>::iterator> myvec
    auto myvec= { mylist.begin(), mylist.end() };
}

```

```

using namespace std;
void foo(vector<string>&& x)
{
    auto y = std::move(x) // taşıma var
}

int main()
{
    vector<string> svec(100'000);

    foo(std::move(svec)); // taşıma yok burada
}

```

Vectorlerin Elemanlarına Erişme

```

// Vectorlerin Elemanlarına Erişme
using namespace std;
int main()
{
    vector<string> svec{ "burhan", "ferhan", "nurhan", "alphan" };
    // Eğer vector const olsaydı const overloading fonksiyonlar çağrılacaktı.
    svec[2] = "oğuzhan";
    svec.at(2);
    try
    {
        svec.at(23);
    }
    catch (const std::out_of_range&)
    {
        // exception gönderir
    }

    svec.front() = 'can'; // ilk veri
    svec.back() = 'emre'; // son veri

    for (size_t{}; i < svec.size(); ++i)
    {
        svec[i]; // böyle erişebiliriz
    }
}

```

std::find_if

```
// range base Loop
using namespace std;
int main()
{
    vector<string> svec{};

    rfill(svec, 100, [] {return rname() + ' ' + rfname();} );

    auto iter = find_if(svec.begin(), svec.end(),
        [] (const string &s) {return s.contains('e'); });

    if (iter != svec.end())
    {
        while (iter != svec.end())
        {
            cout << *++iter << "\n";
        }
    }
}
```

```
// range base Loop (reverse iteration)
using namespace std;
int main()
{
    vector<string> svec{};

    rfill(svec, 100, [] {return rname() + ' ' + rfname();} );

    for (auto riter = svec.crbegin(); riter != svec.crend(); ++riter)
        cout << *riter << '\n';
}
```

Vector'e Eleman Ekleme Fonksiyonları

```
// Vector Ekleme Fonksiyonları
template<typename T>
class Vector
{
public:
    void push_back(const T& r)
    {
        new[adress] T(r); // L value argumanda
    }
    void push_back(T&& r)
    {
        new[adress] T(std::move(r)); // r value argumanda
    }

    template<typename ...U>
    void emplace_back(U&& ...args)
    {
        new[adress] T(std::forward<U>(args)...);
    }
};
```

```

int main()
{
    vector<string> svec{ "ali", "kemal", "naz", "derin" };

    svec.push_back("murat"); // sondan ekleme

    svec.insert(svec.begin(), "deniz"); // başa ekler

    svec.insert(next(svec.begin()), "deniz"); // 2'ye ekler

    // init list overload
    svec.insert(next(svec.begin()), { "deniz", "taylan", "emre" }); // 2'ye ekler

    // range overload
    list<string> female_list {"ganoş", "özge", "merve"};
    svec.insert(svec.begin(), female_list.begin(), female_list.end());

    // insert fonksiyonları insert edilmiş üyenin konumu döndürür
    vector<string> svec1 { "ali", "gül", "tan", "eda" };
    auto iter = svec.insert(svec.begin() + 1, "NURULLAH");
    cout << *iter << "\n"; // NURULLAH
}

```

Vector Atama Fonksiyonları

```

// vector atama fonksiyonları
using namespace std;
int main()
{
    vector<int> ivec(100);

    cout << "ivec.size() = " << ivec.size() << "\n";
    ivec = { 2, 3, 5, 1, 7, 11}; // size = 6

    ivec.assign({2, 3, 1, 4, 1}); // init list overload
}

```

2023 11 08

Vector Silme Fonksiyonları

```
/*
    erase (iterator,range)
    pop_back
    clear
*/
```

pop_back

```
// pop_back
using namespace std;
int main()
{
    vector<string> svec;
    rfill(svec, 10, rname);

    while (!svec.empty())
    {
        print(svec);
        svec.pop_back(); // sondan eleman siler
        svec.erase(svec.begin()); // iterator olur ( silinmiş ögeden sonraki ilk
        // öğeyi döndürür)

    }
}
```

Aranan İlk Öğeyi Silmek

```
// aranan ilk öğeyi silmek
using namespace std;
int main()
{
    vector<string> svec;
    rfill(svec, 10, rname);

    string name;
    cout << "silenecek öğe:";

    cin>> name;

    if (const auto iter = find(svec.begin(), svec.end(), name); iter != svec.end())
    {
        svec.erase(iter);
        cout << "bulundu ve silindi\n";
    }
    else
    {
        cout << "bulunamadı\n";
    }
}
```

```
// içinde hangi harf bulunan silinsin
int main()
{
    using namespace std;
    vector<string> svec;
    rfill(svec, 10, rname);

    char c = 'c';

    if (const auto = iter find_if(svec.begin(), svec.end(),
        [c](const string& s) {return s.contains(c);}); iter != svec.end())
    {
        svec.erase(iter);
    }
}
```

```
// sondan ilk elemanı silme
int main()
{
    using namespace std;
    vector<string> svec;
    rfill(svec, 10, Irand(0, 4));

    int ival;
    cout << "silenecek öğe:";
    cin>> ival;

    if (const auto iter = find(ivec.rbegin(), ivec.rend(), ival);
        iter != ivec.rend())
    {
        // iter.base() -> iter.end() sarmalar
        ivec.erase(iter.base() - 1);
    }
}
```

```
//erase range
int main()
{
    vector<string> svec{"nuri", "emre", "gano", "ahmet"};

    svec.erase(iter.begin(), iter.end()); // hepsi silinir
    svec.erase(iter.begin(), next(iter.begin(), 3)); // ilk 3 silinir
    svec.erase(next(iter.begin()), iter.end() - 1)); // başta ve sondakiler kalır
}
```

```
// clear
int main()
{
    vector<string> svec{"nuri", "emre", "gano", "ahmet"};
    svec.empty();
    svec{};
    svec.resize(0);
}
```

shrink to fit

```
#include <format>
using namespace std;
int main()
{
    vector<int> ivec{100'000, 7};
    std::cout << format("size = {1}, capacity = {0}\n",
    ivec.capacity(), ivec.size());

    ivec.erase(ivec.begin() + 5 , ivec.end());
    ivec.shrink_to_fit(); // capacity geri verir
}
```

vector.data()

```
using namespace std;
void array_print(const int* p, size_t size)
{
    while(size--)
        print("%d", p++);
}
int main()
{
    using namespace std;
    vector<int> ivec{ 3, 6, 7, 9, 1, 10};

    array_print(ivec.data(), ivec.size());
    // vector'deki ilk ögenin adresi
    int *p1 = ivec.data();
    int *p2 = &ivec[0];
    int *p3 = &*ivec.begin();

    vector<int> ivec1; // boş container
    auto p = ivec.data();
    cout << (p == nullptr); // true
}
```

```
int main()
{
    vector<string> s1(100'000);
    vector<string> s2(100'000);
    // pointerler swap ediliyor
    s1.swap(s2);
    swap(s1, s2);
    // böyle yapmamalıyız
    auto temp = s1;
    s1 = s2;
    s2 = temp;
}
```

```
int main()
{
    vector<bool> myvec(20);
    auto b = myvec[0]; // b'nin türü std::vector<bool>::reference
}

template<typename T ...>
vector<bool> A<>>
{
    class reference
    {
        operator bool()const noexcept;
        void flip();
        operator=(bool);
    };

    reference operator[](size_t idx);
    reference at(size_t idx);
    reference front();
    reference back();
}

using namespace std;
int main()
{
    vector<bool> myvec(20);

    vector<bool>::reference x = myvec.operator[](5); //myvec[5];
    // Dikkat !
    bool b = myvec[4]; // bool olur
    auto x = myvec[3]; // std::vector<bool>::reference olur
}
```

STL'deki silme algoritmaları

```
/*
    remove
    remove_if
    unique
*/
```

std::remove

```
template<typename Iter, typename T>
Iter Remove(Iter beg, Iter end, const T& val)
{
}

int main()
{
    vector<string> svec { "ali", "can", "eda", "can",
                          "emre", "gano", "can"};
    cout << "size = " << svec.size() << "\n"; // size = 7
    auto logic_end_iter = remove(svec.begin(), svec.end(), "can");
    cout << "size = " << svec.size() << "\n"; // size = 7

    std::cout << "silinmemiş öğe sayısı" <<
        distance(svec.begin(), logic_end_iter); // 4

    std::cout << "silinmiş ögesi sayısı" <<
        distance(logic_end_iter(), svec.end()); // 3

    // Logic olarak silinmiş öğeleri silme işlemi
    svec.erase(logic_end_iter, svec.end());
    cout << "size = " << svec.size(); // size = 4
}
```

Erase Remove İdiom

```
// erase remove idiom
int main()
{
    vector<string> svec { "ali", "can", "eda", "can",
                          "emre", "gano", "can"};
    svec.erase(remove(svec.begin(), svec.end(), "can"), svec.end());
    cout << "size = " << svec.size(); // size = 4

    // cpp20 ile bu idiom yerine erase fonksiyonu var
    auto n = erase(svec, "can"); // n: silinen öğe sayısı
}
```

`std::remove_if` and `std::erase_if`

```
// erase remove idiom
int main()
{
    vector<string> svec { "ali", "can", "eda", "can",
                           "emre", "gano", "can"};
    svec.erase(remove(svec.begin(), svec.end(), "can"), svec.end());
    cout << "size = " << svec.size(); // size = 4

    // cpp20 ile bu idiom yerine erase fonksiyonu var
    auto n = erase(svec, "can"); // n: silinen öğe sayısı
}

// remove_if and erase_if
int main()
{
    vector<string> svec;
    rfill(svec, 2000, rname);

    size_t len = 10; // 10'dan büyük olanlar

    svec.erase(remove_if(svec.begin(), svec.end(), [len](const string& s)
                           {return s.size() > len;}), svec.end());
    // cpp -20
    auto n = erase_if(svec, [len](const string& s)
                           {return s.size() > len}); // n: silinen öğe sayısı
}
```

```
template<typename T>
struct Less
{
    bool operator()(const T &lhs, const T &rhs) const
    {
        return lhs < rhs;
    }
};

template<typename T>
struct Plus
{
    bool operator()(const T &lhs, const T &rhs) const
    {
        return lhs + rhs;
    }
};

int main()
{
    Less<int>{}(12, 5) // 12 < 5 yazmaktan farkı yok
}
```

std::sort

```
int main()
{
    using namespace std;
    vector<int> ivec{ 455, 123, 1221, 545, 12, 1, 12, 43, 513, 123, 53};
    sort(ivec.begin(), ivec.end());
    sort(ivec.begin(), ivec.end(), less<int>{});
    sort(ivec.begin(), ivec.end(), greater<int>{});
}
```

std::transform

```
#include <functional>
int main()
{
    vector<int> x{ 1, 32, 4, 1, 4, 5, 3, 1, 2, 5, 1, 7};
    vector<int> y{ 11, 32, 14, 12, 42, 53, 1, 2, 2, 5, 1, 7};

    // x ve y'yi toplayıp z'ye yazıyoruz
    transform(x.begin(), x.end(), y.begin(), back_inserter(z), plus{});

    vector<int> z;
}
```

```
#include <functional>
int main()
{
    vector<int> x{ 1, 32, 4, 1, 4, 5, 3, 1, 2, 5, 1, 7};
    // x'i negatif yapıyor
    transform(x.begin(), x.end(), x.begin(), negate{});
}
```

std::unique

```
int main()
{
    using namespace std;
    vector<int> x{ 1, 32, 4, 1, 4, 5, 3, 1, 2, 5, 1, 7};
    // ardaşık aynı olanları siler

    // istedigimiz fonksiyonu kullanabiliriz ardaşık öglere bakarken
    x.erase(unique(x.begin(), x.end(), equal_to{}), x.end());
}
```

bir string'teki fazla boşlukları silme

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

    string str = "    emre    bahtiyar    gano    çalışkan    "

    str.erase(unique(str.begin(), str.end(), [](char c1, char c2)
    {
        return isspace(c1) && isspace(c2);
    }), str.end());
}
```

std::ostream_iterator

```
// ostream_iterator
// copy ile standart outputta container yazma

int main()
{
    using namespace std;
    vector<int> ivec {1, 23, 2, 1, 4, 6, 1};
    copy(ivec.begin(), ivec.end(), ostream_iterator<int>(cout, "\n"));
}
```

2023 11 10

Insertion

vector: bir insert işlemi yaptığımızda o insert ettiğimiz konumdan önceki tutan pointerler etkilenmez ama reallocation olmama şartıyla yani kapasite artmadıysa

Erase

vector: silinecek öğeden herhangi birini gösteren noktalar invalid hale gelir ama öncekilere etkilenmez.

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

    vector<string> svec;
    rfill(svec, 40, rtown);

    // uzunluğu 5 olanlar silinsin ve 6 olanlardan bir tane daha ekletsin

    // hatalı
    for (auto iter = svec.begin(); iter != svec.end(); ++iter)
    {
        // iter invalid hale gelir
        if (iter->length() == 5)
        {
            svec.erase(iter);
        }
        else if (iter->length() == 6)
        {
            svec.insert(iter, *iter);
        }
    }

    // hata yok
    auto iter = svec.begin();
    while (iter != svec.end())
    {
        if (iter->length() == 5)
        {
            iter = svec.erase(iter);
        }
        else if (iter->length() == 6)
        {
            iter = next(svec.insert(iter, *iter), 2);
        }
        else
        {
            ++iter;
        }
    }
}
```

std::deque

- ekleme ve silme baştan ya da sondan olacaksa deque kullanabiliriz.
- iterator invalid hale gelmemesini istiyorsak kullanabiliriz.
- reallocation maliyetinden kaçınmak için kullanabiliriz.

```
template <typename T, typename A = std::allocator<T>>
class Deque
{
};

////////

using namespace std;

int main()
{
    deque<int> id;

    srand myrand{ 0, 9999};
    for (int i = 0; i < 100; ++i)
    {
        int val = myrand();
        if (val % 2 == 0)
            id.push_back(val);
        else
            id.push_front(val);
    }
}
```

Insertion

deque: ekleme işlemleri iki uçtan birinde yapılrsa pointer ya da referenceler etkilenmez ama iterator invalid hale gelir.

Erase

deque: silme işlemi iki uçtan birinde yapılrsa silinmiş ögenin iterator ve pointer invalid hale gelir. Diğerlerine bir şey olmaz

```

int main()
{
    using namespace std;

    deque<string> sd;

    rfill(sd, 10, [] {return rname() + " " + rfname();});

    auto& elem = sd[7];
    auto iter = next(sd.begin() + 7);

    cout << elem << "\n";
    cout << *iter << "\n";

    sd.push_back("mustafa aksoy");

    cout << elem << "\n";
    cout << *iter << "\n"; // tanımsız davranış
}

```

Sıralama Algoritmaları:

```

// sıralama algoritmaları
/*
    sort
    partial_sort
    stable_sort
    nth_element
    partition
        stable_partition
*/

```

std::sort

```

int main ()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 1'000'000, rname);

    auto tp_start = chrono::steady_clock::now();
    sort(svec.begin(), svec.end());
    std::ofstream ofs{"out.txt"};
    if(!ofs)
    {
        std::cerr << "out.txt dosyası oluşturulmadı\n";
    }
    copy(svec.begin(), svec.end(), ostream_iterator<string>(ofs, "\n"));
    auto tp_end = chrono::steady_clock::now();

    std::cout << "sıralama bitti: " << chrono::duration<double>(tp_end - tp_start)
    << "\n";
}

```

std::stable_sort

```
//stable_sort

using namespace std;
using sipair = std::pair<string, int>

int main()
{
    vector<sipair> svec;

    rfill(svec, 20'000, []{return make_pair(rname()i Irand{ 20, 70}());});

    // isme göre sıralama

    sort(svec.begin(), svec.end(), [](&const sipair& x, &const sipair& y)
    {
        return x.first << y.first;
    });

    // fonksiyon stable değil bu yüzden isimlerin sıralaması bozuldu
    sort(svec.begin(), svec.end(), [](&const sipair& x, &const sipair& y)
    {
        return x.second << y.second;
    });

    // isimlerde kendi içinde sıralıdır
    stable_sort(svec.begin(), svec.end(), [](&const sipair& x, &const sipair& y)
    {
        return x.second << y.second;
    });
}
```

std::partial_sort and std::partial_sort_copy

nth element algoritması

```
// ortanca öğesini veya belirli bir yüzdelik dilimdeki öğeyi bulmak için  
// kullanılır.  
int main()  
{  
    using namespace std;  
  
    vector<Date> dvec;  
    rfill(dvec, 20'000, Date::random);  
    int n = 1000;  
    // nth_element algoritması, bir dizideki sıralı olmayan öğeler  
    // arasında n'inci öğeyi bulmak için kullanılır.  
    nth_element(dvec.begin(), dvec.begin() + n, dvec.end());  
}
```

```
int get_median(std::vector<int> vec)  
{  
    auto vec_r;  
    std::nth_element(vec.begin(), next(vec.begin(), vec.size() / 2), vec.end());  
  
    return vec[vec.size() / 2];  
}  
  
int main()  
{  
    using namespace std;  
  
    vector ivec { 5, 7, 1, 4, 9, 3, 23, 12, 8, 6, 13, -3, 98};  
    cout << "median is " << get_median(ivec);  
}
```

std::partition

```
int main()  
{  
    using namespace std;  
    vector<string> svec;  
    rfill(svec, 20, rname);  
    char c = a;  
    // içinde a harfi olanlarla olmayanları ayırır  
    auto iter_par_point = partition(svec.begin(), svec.end(), [c](const string& s)  
    {  
        return s.find(c) != string::npos;  
    });  
  
    print(svec, "\n");  
    // iter_par_point: koşulu sağlamayan ilk öğeyi verir  
    std::cout << "partisyon indeksi : " <<  
        iter_par_point - svec.begin() << '\n';  
  
    // koşulu sağlayanlar  
    copy(svec.begin(), iter_par_point, ostream_iterator<string>(cout, "\n"));  
    // koşulu sağlamayanlar  
    copy(iter_par_point, svec.end(), ostream_iterator<string>(cout, "\n"));  
}
```


2023 11 13

std::minmax_element

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

    vector<string> svec;
    rfill(svec, 20, rname);
    print(svec);

    pair<vector<string>::iterator, vector<string>::iterator> ret
        = minmax_element(svec.begin(), svec.end()); // std::pair döndür

    auto ip = minmax_element(svec.begin(), svec.end());

    cout << "min = " << *ip.first << '\n';
    cout << "max = " << *ip.second << '\n';
    // Structured binding
    auto [iter_min, iter_max] = minmax_element(svec.begin(), svec.end());
}
```

std::partition_point

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

    vector ivec{ 2, 4, 6, 8, 2, 20, 4, 3, 7, 9, 101};

    auto iter = partition_point(ivec.begin(), ivec.end(), [](int x )
    {
        return x % 2 == 0; // tekin başladığı yeri döndürür 3'ü
    });

    cout << *iter << '\n';
}
```

std::is_sorted

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

    vector ivec { 2, 4, 6, 8, 20};

    auto b = is_sorted(ivec.begin(), ivec.end()); // b = true
}
```

`std::is_sorted_until`

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

    vectorivec { 22, 14, 6, 18, 1, -1};
    // büyükten küçüğe sıralamanın bozulduğu konumu verir. *iter = 18
    auto iter = is_sorted_until(ivec.begin(), ivec.end(), greater{});
}
```

`heap veri yapısı`

```
// heap veri yapısı
int main()
{
    using namespace std;
    vector<int> ivec;

    rfill(ivec, 20, IRand{ 0, 1000 });

    make_heap(ivec.begin(), ivec.end()); // max heap
    make_heap(ivec.begin(), ivec.end(), greater{}); // min heap

    // heap'in en başındaki öğeyi en sona getirir çıkarabilmek için
    pop_heap(ivec.begin(), ivec.end());
    // heap olma özelliğini korurs
    ivec.pop_back(); // sondaki öğeyi çıkarır

    // heap'e veri ekleme
    ivec.push_back(9021);
    push_heap(ivec.begin(), ivec.end());

    while (!ivec.empty())
    {
        pop_heap(ivec.begin(), ivec.end());
        ivec.pop_back();
    }
}
```

std::queue

```
#include <queue>
int main()
{
    /*
        -kuyruğa farklı sıradaki elemanları ekledik
        -kuyrukta eleman çıkarıyoruz ama çıkan elemanlar en büyük değer

    */
    using namespace std;

    priority_queue<int> x;

    srand myrand{ 0, 19887};

    for (int i = 0; i < 20; ++i)
    {
        int ival = myrand();
        x.push(ival);
        cout << ival << " kuyruga eklendi\n ";
    }

    while (!x.empty())
    {
        cout << x.top();
        x.pop();
    }
}
```

std::list

- Ortalardan ekleme ve silme işlemleride vector'e göre daha avantajlı
- Swap işlemlerinde ve swap işlemlerini yapan algoritmalarla vector'e göre daha avantajlı
- Çok büyük verilerde list vector'e göre daha avantajlı

uyarı: profiling yapmadan list mi vector mu seçilmeli kararı verilmemeli

bidirectional_iterator sahip bu yüzden:

- iki iteratorü büyülüklük küçüklük karşılaştırması yapmak
- iki iteratorün farkını almak
- += -= gibi işlemler yapmak
- [] parantez işlemleri yapmak

bunlar syntax hatasıdır.

std::reverse, std::remove ve std::remove_if

```
int main()
{
    using namespace std;
    // eğer vector olsaydı algoritma kütüphanesini kullanacaktık.
    list<string> mylist;

    rfill(mylist, 10, rname);
    print(mylist);

    mylist.reverse(); // listeye özel

    list<int> ilist;
    rfill(ilist, 100, Irand{0, 5});
    int ival = 3;

    auto n = ilist.remove(ival);

    list<int> ilist1;
    rfill(ilist1, 100, Irand{0, 5});

    auto n = ilist1.remove_if([ival](int x) {return x % ival == 0;});

}
```

std:: merge

```
int main()
{
    using namespace std;
    // karşılaştırma kriteri bilinmeli
    list<string> x;
    list<string> y;

    auto fgen = [] {return rname() + ' ' + rfname();};

    rfill(x, 20, fgen);
    rfill(y, 20, fgen);

    x.sort();
    y.sort();

    x.merge(y);

    cout << x.size() << "\n"; // 40;
    cout << y.size() << "\n"; // 0;
}
```

std::splice (bir listeden çıkartıp başka listeye ekler)

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

    list<string> f{ "esin", "ayse", "fatma", "gizem", "lale", "hulya"};
    list<string> m{ "fuat", "ceyhun", "tarik", "cetin", "kaan"};

    f.splice(f.begin(), m); // en başa m listesini ekledik
}
```

std::lower_bound, std::upper_bound ve std::equal_range

```
/*
4 6 6 7 7 7 9 12 23 34 45 90

Lower bound: verilen bir değer için öyle bir konum ki o değeri sırayı
bozmadan ekleyebileceğimiz ilk konum
(verilen değer 7 ise ekliyebileceğim indeks 3)

upper bound: verilen bir değer için öyle bir konum ki o değeri sırayı
bozmadan ekleyebileceğimiz son konum
(verilen değer 7 ise ekliyebileceğim indeks 6)

equal range: lower_bound ile upper_bound arasındaki range
yani 7 değeri için 3 6 arası

*/
```

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

    vector<int> ivec;
    // karşılaştırma kriteri bilinmeli
    rfill(ivec, 20, Irand{ 0, 5 });
    sort(ivec.begin(), ivec.end(), greater{});

    int key;
    cin >> key;

    auto iterlower = lower_bound(ivec.begin(), ivec.end(), key, greater{});
    cout << "ilk ekleme yapılabilecek yerin indeksi : " << distance(ivec.begin(), iterlower) << '\n';
    auto iterupper = upper_bound(ivec.begin(), ivec.end(), key, greater{});
    cout << "son ekleme yapılabilecek yerin indeksi : " << distance(ivec.begin(), iterupper) << '\n';

    auto [iterlower, iterupper] = equal_range(ivec.begin(), ivec.end(), key);

    cout << key << " degerinden " << distance(iterlower, iterupper) << " tane
var\n";
}
```

Mülakata sorusu: bir vektöre eleman eklicez ama sıralama bozulmıcak

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

    vector<string> svec;

    auto fgen = [] {return rname() + ' ' + rfname(); };

    for (int i = 0; i < 20; ++i)
    {
        auto name = fgen();
        std::cout << name << "ekleniyor\n";
        auto iter = lower_bound(svec.begin(), svec.end(), name);
        svec.insert(iter, std::move(name));
    }
}
```

sorted range algorithms

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

    vector<string> x;
    vector<string> y;

    rfill(x, 30, rtown);
    rfill(y 40, rtown);

    sort(x.begin(), x.end());
    sort(y.begin(), y.end());

    // x ve y kesişimi
    list<string> slist_inter;
    set_intersection(x.begin(), x.end(), y.begin(), y.end(),
back_inserter(slist_inter));
    // x ve y birleşimi
    list<string> slist_union;
    set_union(x.begin(), x.end(), y.begin(), y.end(), back_inserter(slist_union));
    // x fark y
    list<string> slist_diff;
    set_difference(x.begin(), x.end(), y.begin(), y.end(), back_inserter(slist_diff));
}
```

2023 11 15

forwad list (tekli bağlı liste) C++11

```
// forward_iterator kullanılır
int main()
{
    using namespace std;

    // size fonksiyonu yok
    forwad_list<int> mylist{ 2, 3, 1, 2, 5, 12, 123, 11};

    copy(mylist.begin(), mylist.end(), ostream_iterator<int>(cout, " "));
    std::cout << "\n";

    // push_back fonksiyonu yok
    mylist.push_front(123);
    mylist.push_front(999);

    copy(mylist.begin(), mylist.end(), ostream_iterator<int>(cout, " "));
    std::cout << "\n";
    // pop_back fonksiyonu yok
    mylist.pop_front();

    auto iter = next(mylist.begin(), 3);
    cout << *iter << "\n";

    // iter konumundan bir sonraki konuma ekler
    mylist.insert_after(iter, 7777);

    // en başa ekler, before_begin() en baştaki öğeden önceki adresi verir
    mylist.insert_after(mylist.before_begin(), { -7, -8, 1, 3});
    // ilk 3 öğeyi siler
    mylist.erase_after(mylist.before_begin(), next(mylist.begin(), 3));
}
```

Container Adapters

- stack: (LIFO: Last in first out) son girenin ilk çıktı bir veri yapısı
- queue: ilk girenin ilk çıktı bir veri yapısı
- priority_queue: önceliği en yüksek olan ilk çıkar

std::stack

```
template <typename T, typename C = std::deque<T>>
class Stack
{
public:
    void push(const T& val)
    {
        c.push_back(val);
    }
    T& top()
    {
        return c.back();
    }
    void pop()
    {
        c.pop_back();
    }
    bool empty()const
    {
        return c.empty();
    }
    auto size()const
    {
        return c.size();
    }
};
```

```
#include <stack>
// default template argumanı deque
int main()
{
    stack<int, deque<int>> mystack; // stack<int>
    stack<int, vector<int>> mystack1;
}
```

```
// stack class'ını kalıtım yoluyla kullanabiliriz
class NecStack : public std::stack<int>
{
```

```

int main()
{
    using namespace std;

    stack<int> mystack;

    mystack.push(2);
    mystack.push(3);
    mystack.push(5);
    mystack.push(7);
    mystack.push(11);

    cout << "size = " << mystack.size() << "\n";
    cout << "mystack.top() = " << mystack.top() << "\n";

    while (!mystack.empty())
    {
        cout << mystack.top() << "\n";
        mystack.pop(); // 11 7 5 3 2 diye çıkar
    }

    //syntax hatalı
    stack<int> s{3, 13, 23, 12, 4}; // init list ctor ypk
    deque dx{ 3, 5, 7, 7};
    stack<int> mystack { dx }; // geçerli
}

```

std::queue

```

// template argumani deque
#include <queue>
int main()
{
    using namespace std;

    queue<string> names;

    names.push("melike");
    names.push("emre");
    names.push("tamer");
    names.push("furkan");
    names.push("selim");
    names.push("yasar   ");

    std::cout << "kuyrukta " << names.size() << " kisi var\n";
    std::cout << "kuyruk basi " << names.front() << "\n";
    std::cout << "kuyruk sonu " << names.back() << "\n";

    while (!names.empty())
    {
        cout << names.front() << "\n";
        names.pop(); // ilk giren ilk çıkar (melike , emre, tamer ...)
    }
}

```

std::priority_queue

```
// priority_queue
template<typename T, typename C = std::vector<T>, typename Comp =
std::less<typename C::value_type>>
class PrioritQueue{};
// template argumanı vector ve less (karşılaştırma kriteri)
int main()
{
    priority_queue<string> x;

    for (int i = 0; i < 10; ++i)
    {
        auto name = rname();
        x.push(name);
        cout << name << " eklendi\n";
    }

    // en büyükten küçüğe doğru çıkar
    while (!x.empty())
    {
        std::cout << x.top() << " kuyrak cikiyor\n";
        x.pop();
    }
}
```

```
// alias template
template <typename T>
using minpq = std::priority_queue<T, std::vector<T>, std::greater<T>>;
```

Associative Container

- set
- multiset
- map
- multimap

std::set

```
// bidirectional_iterator
#include <set>
int main()
{
    set<int, less<int>, allocator<int>> myset; // set<int>

    set<int> s {3, 1, 4, 2, 5, 7};

    for (auto iter = s.begin(); iter != s.end(); ++iter)
        cout << *iter << '\n';
    for(int ival : s)
        cout << ival << '\n';
}
```

```

int main()
{
    set<string> myset;
    for (int i = 0; i < 1000; ++i)
    {
        myset.insert(rname());
    }
    // size 1000'dan az olabilir çünkü bir değerden sadace 1 tane olur
    cout << "myset.size() = " << myset.size() << "\n";

    multiset<string> mymultiset;
    for (int i = 0; i < 1000; ++i)
    {
        mymultiset.insert(rname());
    }
    // size 1000 olacak. Multisette bir değerden birçok tane olabilir
    cout << "mymultiset.size() = " << mymultiset.size() << "\n";
}

```

Set Karşılaştırma Kriteri

```

class scomp
{
public:
    bool operator()(const std::string& s1, const std::string& s2) const
    {
        return s1.size() < s2.size() || (s1.size() == s2.size() && s1 < s2);
    }
};

int main()
{
    using namespace std;

    set<string, scomp> myset;

    for (int i = 0; i < 100; ++i)
    {
        myset.insert(rname());
    }

    cout << "myset.size() = " << myset.size() << "\n";

    // artik scomp'a göre karşılaştırması olacak
    for (const auto& s : myset)
    {
        std::cout << s << " ";
    }
}

```

```

bool mycomp(const std::string& s1, const std::string& s2)
{
    return s1.size() < s2.size() || (s1.size() == s2.size() && s1 < s2);
}
int main()
{
    using namespace std;

    set<string, decltype(&mycomp)> myset(mycomp);

    for (int i = 0; i < 100; ++i)
    {
        myset.insert(rname());
    }

    cout << "myset.size() = " << myset.size() << "\n";

    // artk scomp'a göre karşılaştırması olacak
    for (const auto& s : myset)
    {
        std::cout << s << " ";
    }
}

```

```

int main()
{
    using namespace std;
    auto fcomp = [] (const std::string& s1, const std::string& s2)
    {
        return s1.size() < s2.size() || (s1.size() == s2.size() && s1 < s2);
    };
    set<string, decltype(fcomp)> myset(fcomp);

    for (int i = 0; i < 100; ++i)
    {
        myset.insert(rname());
    }

    cout << "myset.size() = " << myset.size() << "\n";

    // artk scomp'a göre karşılaştırması olacak
    for (const auto& s : myset)
    {
        std::cout << s << " ";
    }
}

```

Strict Weak Ordering

Karşılaştırma fonksiyonu yazarken:

- $a < b$ true ise $b > a$ false olmalı (antisymmetric)
 - $a < a$ false olmalı (küçük eşittir olmaz) (irreflexive)
 - $a \text{ operator } b$ true ve $b \text{ operator } c$ true ise $a \text{ operator } c$ true olmalı (transitive)
-
- $!(a < b) \&\& !(b < a)$ true ise ve $!(b < c) \&\& !(c < b)$ true ise
 - $!(a < c) \&\& !(c < a)$ true olmalı (transitivity of equivalence)

set.insert()

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

    set<string> myset;
    vector<string> svec { "derya", "ceyhun", "nalan", "tekin" };

    myset.insert("ayse")
    myset.insert({"ali", "zeki", "nuri", "derya"});
    myset.insert(svec.begin(), svec.end());

    cout << "eklenecek isim girin : " ;
    string name{};

    cin >> name;

    //pair<set<string>::iterator, bool> p = myset.insert(name);
    auto p = myset.insert(name);
    // sette varsa bool false döner iterator var olan yeri döner

    if (p.second)
    {
        std::cout << "ekleme yapildi... \n";
        cout << *p.first << "\n";
    }
    else
    {
        std::cout << name << "sette var\n";
        cout << *p.first << "\n";
    }
}
```

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

    set<string> myset;
    rfill(myset, 10, rname);
    print(myset);

    cout << "aranacak isim: "
    string name{};
    cin >> name;
    // Logaritmik karmaşıklıkta
    if (auto iter = myset.find(name); iter != myset.end())
    {
        std::cout << "bulundu..." << *iter << "\n";
    }
    else
    {
        std::cout << "bulunamadı..." << *iter << "\n";
    }
    // iterator kullanmayacaksak böyle kullanabiliriz
    if (myset.count(name))
    {
        std::cout << "bulundu...\n";
    }

    if (myset.contains(name)) // cpp 20 de geldi
    {
    }
}
```

2023 11 17

set.insert() -- hint insert

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

    set<string> myset{ "ayca", "berna", "derya", "meliha", "saliha"};
    // eğer doğru konum verirsek performans verir.
    myset.insert(myset.begin(), "aliye"); // hint insert

    vector<string> svec { "eda", "naz", "tan" };
    copy(svec.begin(), svec.end(), inserter(myset, myset.begin()));
}
```

inserter

```
// inserter
int main()
{
    using namespace std;
    vector<string> svec { "eda", "naz", "tan" };
    set<string> myset{ "ayca", "berna", "derya", "meliha", "saliha"};
    copy(svec.begin(), svec.end(), inserter(myset, myset.begin()));
}
```

set.emplace_hint()

```
// emplace hint
int main()
{
    // perfect forwarding kullanarak set'in sağladığı bellek alanında nesneyi
    // oluşturur
    myset.emplace_hint(myset.begin(), 10, 'A');
}
```

```

int main()
{
    using namespace std;

    set<string> myset;
    rfill(myset, 10, rname);

    cout << "eski ve yeni ismi giriniz:";
    string old_name, new_name;
    cin >> old_name >> new_name;

    // eski cpp için find ile bulup silip daha sonra yeni elemanı eklicez

    if (auto iter = myset.find(old_name); iter != myset.end())
    {
        myset.erase(iter);
        myset.insert(new_name);
        std::cout << "anahtar degistirildi\n";
    }
    else
    {
        cout << "bulanamadi\n";
    }
}

```

set.extract()

```

int main()
{
    using namespace std;

    set<string> myset;
    rfill(myset, 10, rname);

    cout << "eski ve yeni ismi giriniz:";
    string old_name, new_name;
    cin >> old_name >> new_name;

    auto iter = myset.find(old_name);
    auto handle = myset.extract(iter);

    cout << "size = " << myset.size() << "\n"; // size = 9

    handle.value() = new_name;
    myset.insert(std::move(handle));

    cout << "size = " << myset.size() << "\n"; // size = 10
}

```

```

int main()
{
    using namespace std;

    multiset<int> myset;
    rfill(myset, 20, Irand{0, 5});
    print(myset);

    auto [iter_first, iter_last] = myset.equal_range(3);
}

```

Set Lambda Karşılaştırma Kriteri

```

int main()
{
    using namespace std;

    set<int, decltype([](int x, int y)
    {
        return x % 100 < y % 100;
    })> myset;
}

```

std::map

```

int main()
{
    using namespace std;
    // key value
    map<int, string, less{}> mx;

    for (int i = 0; i < 10; ++i)
    {
        mx.insert(pair{ rand(), rname()});
    }
    for (const auto&p : mx)
    {
        cout << p.first << " " << p.second << "\n"; // key'e göre sıralar
    }

    auto iter = mx.begin();
    iter->first = 323; // syntax hatası çünkü key const
    iter->second = "emre"; // hata yok
}

```

std::map'a öğe ekleme

```
int main()
{
    using namespace std;
    map<string, int> mx;

    pair<string, int> p1 {"tayfun", 767};
    mx.insert(p1);

    mx.insert(pair{ "alican", 871});

    mx.insert({"emrecan", 823});

    mx.insert(make_pair("deniz", 761));

    mx.emplace("damla", 732);

    for (auto iter = mx.begin(); iter != mx.end(); ++iter)
    {
        std::cout << iter->first << " " << iter->second << "\n";
    }

    // for-ranged
    for (const auto& p : mx)
    {
        std::cout << p.first << " " << p.second << "\n";
    }

    // structured binding C++17
    for (const auto& [name, grade] : mx)
    {
        std::cout << name << " " << grade;
    }
}
```

[] operator fonksiyonu

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

    map<int, string> mymap;

    for (int i = 0; i < 10; ++i)
    {
        mymap.emplace(irand{0 , 70}(), rname());
    }

    print(mymap, "\n");
    int key;
    cin >> key;
    // ref semantiği ile key'e erişebiliyoruz
    mymap[key] = "sadullah";
    // eğer key yoksa o key'i eklemiştir oluyoruz

    mymap.at(key) = "sadullah"; // anahtar yoksa exception throw eder
}
```

```

// vectordeki isimlerden kaç tane olduğunu map'a yazıyoruz
int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 10000, rname);

    map<string, int> cmap;

    for (const auto& name : svec)
    {
        ++cmap[name]
    }

    vector<pair<string, int>> myvec{cmap.begin(), cmap.end()};

    sort(myvec.begin(), myvec.end(), [](const auto& p1, const auto& p2)
    {
        return p1.second > p2.second; // sıralama yaptık sayıya göre
    });
}

```

```

int main()
{
    using namespace std;

    map<string, int>::value_type // pair<string, int>
    map<string, int>::key_type // string
    map<string, int>::mapped_type // int
}

```

```

int main()
{
    using namespace std;
    map<string, int> mymap;

    for (int i = 0; i < 10; ++i)
    {
        mymap.emplace(rname(), Irand { 0, 70}());
    }

    string old_key, new_key;

    cout << "eski ve yeni anahtarları girin: ";
    cin >> old_key >> new_key;

    auto handle = mymap.extract(old_key);

    if (handle)
    {
        handle.key() = new_key;
        mymap.insert(move(handle));
        cout << "anahtar degistirildi\n";
    }
}

```

Hash Table

- Hash'ta anahtar ile arama constant time complexity
- Sette anahtar ile arama linear time complexity

```
#include <functional>
int main()
{
    using namespace std;

    hash<int> hasher;

    size_t val = hasher(87423);
    cout << val << "\n"; // hash değeri
}
```

```
#include <functional>
int main()
{
    using namespace std;

    hash<string> hasher;

    cout << hasher("deniz") << "\n"; // hash değeri
    cout << hasher("denis") << "\n"; // hash değeri
    cout << hasher("denizcan") << "\n"; // hash değeri
}
```

```
struct DateHasher
{
    std::size_t operator()(const Date& dt) const
    {
        std::hash<int> hasher;

        return hasher(dt.month_day()) + hasher(dt.month()) + hasher(dt.year());
    }
}

int main()
{
    using namespace std;

    Date mydate{ 17, 11, 2023};
    cout << DateHasher{}(mydate) << '\n';
    ++mydate;
    cout << DateHasher{}(mydate) << '\n';
}
```

std::unordered_set

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

    unordered_set<int, hash<int>, equal_to<int>> // unordered_set<int>
}
```

```
// Eğer kendi veri tipimizi kullanıcazsa

class DateHasher
{
    std::size_t operator()(const Date&) const;
};

struct DateEqual
{
    bool operator()(const Date&, const Date&) const;
};

int main()
{
    using namespace std;
    unordered_set<Date, DateHasher> myset;
}
```

lambda kullanarak unordered_set oluşturma

```
// Lambda kullanarak unordered_set oluşturma

int main()
{
    using namespace std;

    auto fhash = [] (const Date&)
    {
        return 12u;
    };

    auto eq = [] (const Date&, const Date&)
    {
        return true;
    };

    unordered_set<Date, decltype(fhash), decltype(eq)>;
}
```

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

    unordered_set<int> myset(400); // bucket sayısını belirleriz

    cout << myset.bucket_count() << "\n";
}
```

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

    unordered_set<string> myset;

    for (int i = 0; i < 10; ++i)
    {
        myset.insert(rname());
    }

    print(myset);

    for (int i = 0; i < 10; ++i)
    {
        string name;
        cout << "isim girin: ";
        cin >> name;

        if (myset.contains(name))
        {

        }
    }
}
```

2023 11 20

std::unordered_Set ---equal_to ve hasher

```
// unordered_set için equal_to
struct Point
{
    Point() = default;
    Point(double, double, double);
    bool operator == (const Point&)const;

    double mx{}, my{}, mz{};
};

//unordered_set için hasher
template<>
struct std::hash<Point>
{
    std::size_t operator()(const Point& pt)const
    {
        std::hash<double> hasher;
        return hasher(pt.mx) + hasher(pt.my) + hasher(pt.mz);
    }
};

class PointHasher
{
    std::size_t operator()(const Point& pt)const
    {
        std::hash<double> hasher;
        return hasher(pt.mx) + hasher(pt.my) + hasher(pt.mz);
    }
};

struct PointEqual
{
public:
    bool operator()(const Point&, const Point&)const;
};

int main()
{
    using namespace std;

    unordered_set<Point> myset; // struct std::hash<Point>
    unordered_set<Point, PointHasher, PointEqual> myset; // class PointHasher

    myset.insert(Point{ 2.3, 5.6, 78.9});
}
```

```

int main()
{
    using namespace std;
    unordered_set<string> myset;

    for (int i = 0; i < 100; ++i)
    {
        myset.insert(rname()); // hiçbir sıra yok
    }

}

```

Load Factor ve Max Load Factor

```

int main()
{
    using namespace std;

    unordered_set<string> myset;

    for (int i = 0; i < 100; ++i)
    {
        myset.insert(rname()); // hiçbir sıra yok
    }

    // Load factor, max load factor geçince rehash yapılır.
    std::cout << "bucket count = " << myset.bucket_count() << '\n';
    std::cout << "size = " << myset.size() << '\n';
    std::cout << "load factor = " << float(myset.size()) / myset.bucket_count()
    << '\n';
    std::cout << "load factor = " << myset.load_factor() << '\n';
    std::cout << "max load factor = " << myset.max_load_factor() << '\n';

    for (size_t i{}; i < myset.bucket_count(); ++i)
    {
        // i. bucketin sizeni yazar
        std::cout << i << "      " << myset.bucket_size(i));

        for (auto iter = myset.begin(i); iter != myset.end(i); ++iter)
        {
            // bucket içindeki verileri yazar
            std::cout << *iter
        }
    }

    string name = "emre";
    if (myset.contains(name))
    {
        // hangi bucket olduğunu yazar
        std::cout << "bucket : " << myset.bucket(name) << "\n";
    }

}

```

Function Adaptor

Bizim callable'mızı (fonksiyon, lambda expression vb.) alıp adapte edip, özellikler verip yeni bir callable döndürür.

callable f ---> adaptor ---> callable ret

function adaptor:

- std::bind
- mem_fn
- not_fn
- std::invoke

reference_wrapper

```
int main()
{
    int x = 10;
    int y = 45;

    int &r = x;

    r = y; // x = y

    int *p[20]; // pointer dizisi
    int &p[20]; // syntax hatalı

    std::vector<int *> myvec; // pointer container
    std::vector<int&> myvec; // syntax hatalı;
}
```

```
template<typename T>
class ReferenceWrapper
{
public:
    ReferenceWrapper(T &t) : mp{&t} {}

    ReferenceWrapper& operator=(const T&)
    {
        mp = &t;
    }

    operator T&()
    {
        return *mp;
    }

    T& get()
    {
        return *mp;
    }
private:
    T *mp;
};
```

```

template<typename T>
void func(T x)
{
}

template<typename T>
ReferenceWrapper<T> Ref(T& x)
{
    return ReferenceWrapper<T>{x};
}

int main()
{
    int x = 12;
    ReferenceWrapper<int> r(x);

    cout << r; // 12    cout << r.operator int& ();

    string str(100'000, 'a');
    func(ReferenceWrapper<string>(str)); // func(string& str); yerine
    kullanabilirz
    func(ReferenceWrapper(str)); // CTAD Cpp 17

    func(Ref(str)); // factory method
}

```

```

template<typename T>
void foo(T x)
{
    x += 100;
}
int main()
{
    using namespace std;

    int ival = 20;
    int& r = ival;

    foo(r);
    // ival = 20 çünkü ref olarak gitmez template arg kurallarına göre
    cout << "ival = " << ival << "\n";

    foo<int &>(r);
    // ival = 120
    cout << "ival = " << ival << "\n";

    foo(reference_wrapper<int>{ival});
    // ival = 120
    cout << "ival = " << ival << "\n";

    foo(ref{ival});
    // ival = 120
    cout << "ival = " << ival << "\n";
}

```

```

struct BigPred
{
    bool operator()(int)
    {
        return true;
    }

    char buf[2 * 4096]{};
};

int main()
{
    using namespace std;

    BigPred pred;

    vector<int> ivec(100'000);
    vector<int> destvec;
    // ref kullanarak pred nesnesini kopyalanmasını engelledik.
    copy_if(ivec.begin(), ivec.end(), back_inserter(destvec), ref(pred));
}

```

```

// generate algoritması
int foo()
{
    return 12;
}
int main()
{
    using namespace std;

    vector<int> ivec(100);

    // fonksiyon ya da Lambda geri dönüş değerini vector'e yazar.
    generate(ivec.begin(), ivec.end(), foo);
    generate(ivec.begin(), ivec.end(), []
    {
        return rand() * 2; //
    });
}

```

```

// reference_wrapper örnek
int main()
{
    using namespace std;

    mt19937 eng;
    vector<unsigned int> uvec(10'000);

    std::cout << "sizeof(eng) = " << sizeof(eng) << "\n"; // sizeof(eng) = 5000
    generate(uvec.begin(), uvec.end(), eng); // verimli değil
    generate(uvec.begin(), uvec.end(), ref(eng));
}

```

```
// CTAD örnek
int main()
{
    using namespace std;

    string name { "necati ergin" };
    reference_wrapper r = name; // CTAD
}
```

```
// reference_wrapper örnek
int main()
{
    using namespace std;

    std::list<string> mylist {"kutay", "tarik", "cemal"};
    // myvec'in elemanları listedeki elemanlarına reference
    vector<reference_wrapper<string>> myvec{mylist.begin(), mylist.end()};

    sort(myvec.begin(), myvec.end(), [](auto r1, auto r2)
    {
        return r1.get() < r2.get();
    });

    for (const auto& s : mylist)
    {
        cout << s.get() << " ";
    }
    std::cout << "\n";
}
```

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

    int x = 321, y = 123;

    pair p(ref(x), ref(y));
    p.first *= 10;
    p.second *= 10;

    cout << "x = " << x << "\n"; 3210
    cout << "y = " << y << "\n"; 1230
}
```

- `reference_wrapper` incomplete type ile kullanabiliriz
- `reference_wrapper` fonksiyonlarda kullanabiliriz.

```
int sumsquare(int x, int y)
{
    return x * x + y * y;
}

int main()
{
    using namespace std;

    reference_wrapper rf = sumsquare;
    auto val = rf (10, 20);

    cout << "val = " << val << "\n";
}
```

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

    int x = 10, y = 45;

    reference_wrapper r{x};

    ++r; // x = 11, y = 45
    r = y; // r = &y
    ++r; // x = 11, y 46

    r.get() = y // x = y
}
```

cref()

```
// cref
int main()
{
    using namespace std;
    // const reference_wrapper
    int x = 356;
    auto r = cref(x) // (reference_wrapper<const int> r = x
}
```

std::bind

```
#include <functional>
int foo(int x, int y, int z)
{
    std::cout << "x = " << x << "y = " << y << "z = " << z;

}

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

    auto f = bind(foo, 10, 20, 30);
    f(); // foo(10, 20, 30); çağırırı aslında

    f = bind(foo, 10, 20, _1);
    f(99); // foo(10, 20, 99);

    f = bind(foo, _1, _1, _1);
    f(99); // foo(99, 99, 99);

    f = bind(foo, _1, 77, _2);
    f(99, 21); // foo(99, 77, 21);

    f = bind(foo, _3, _1, _2);
    f(10, 20, 30); // foo(30, 10, 20);
}
```

2023 12 22

std::bind

```
// std::bind
#include <functional>
int foo(int x, int y, int z)
{
    std::cout << "x = " << x << "y = " << y << "z = " << z;
}

int main()
{
    using namespace std;
    using namespace std::placeholders;
    auto f = bind(foo, _2, _1);
    f(333,999); // foo(20, 999, 333); çağrılar aslında
}
```

```
struct Functor
{
    void operator()(int a, int b)
    {
        std::cout << "a = " << a << " b = " << b << "\n";
    }
};

int main()
{
    using namespace std;

    bind(Functor{}, placeholders::_1, 6512)(90); // function object
    bind([](int a, int b, int c) // Lambda fonksiyon
    {
        return a + b + c;
    }, 99, placeholders::_1, placeholders::_1);

    auto val = fn(35); // val = 169
}
```

Class member fonksiyonlarda std::bind

```
class Nec
{
public:
    void foo()const
    {
        std::cout << "Nec::foo()\n";
    }

    void bar(int a)const
    {
        std::cout << "Nec::bar(int a) a = " << a << "\n";
    }
};

int main()
{
    using namespace std::placeholders;

    Nec mynec;

    auto f1 = std::bind(&Nec::foo, _1);
    f1(mynec);

    auto f2 = std::bind(&Nec::bar, _1, 89);
    f2(mynec);
}
```

```
void func(int& x, int& y, int& z)
{
    x += 10;
    y += 10;
    z += 10;
}

int main()
{
    int a = 72;
    int b = 82;
    int c = 92;

    auto f1 = std::bind(func, a, b, c);
    f1();
    std::cout << a << " " << b << " " << c << "\n"; // 72 82 92

    auto f2 = std::bind(func, ref(a), ref(b), ref(c));
    f2();
    std::cout << a << " " << b << " " << c << "\n"; // 82 92 102
}
```

```

int main()
{
    using namespace std;
    using namespace std::placeholders;

    vector<int> ivec(100);
    generate(ivec.begin(), ivec.end(), [] {return rand() % 1000;});

    int val = 900;

    cout << count_if(ivec.begin(), ivec.end(), [] (auto i){return i > 900;}) <<
"\n";

    cout << count_if(ivec.begin(), ivec.end(), bind(greater{}, _1, val)) << "\n";
}

```

```

// bind parametrelerin kopyasını çıkarır
void increment(int& x)
{
    ++x;
}

int main()
{
    using namespace std;

    int ival{35};

    auto fn1 = bind(increment, ival);
    auto fn2 = bind(increment, ref(ival));

    fn1();
    cout << "ival = " << ival << "\n"; // ival = 35
    fn2();
    cout << "ival = " << ival << "\n"; // ival = 36
}

```

std::function

function<>

template param. olarak bir callable'in çağrılmaya aday fonksiyon türünü kullanmamız gerekiyor.

```
#include <functional>
int foo(int x)
{
    std::cout << "foo(int) cagrildi\n";
    return x * 19;
}

int bar(int x)
{
    std::cout << "bar(int) cagrildi\n";
    return x * x;
}
double baz(double);
int main()
{
    /*
        foo'un türü: int(int)
        &foo'un türü int(*)(int)
    */
    std::function<int(int)> f(foo);

    cout << "ret = " << f(90) << "\n"; // foo(90);    ret = 109

    f = bar;
    cout << "ret = " << f(90) << "\n"; // bar(90);    ret = 90 * 90

    std::function fctad = foo; // CTAD kullanabiliriz
    fctad = baz; // tür uyumusuzluğu olur.
}
```

```

class Nec
{
public:
    Nec(int x) : mx{x} {};
    void print_sum(int a) const
    {
        std::cout << mx << " + " << a << " = " << mx + a << "\n";
    }

    static void print(int a)
    {
        std::cout << mx << " + " << a << " = " << mx + a << "\n";
    }
private:
    int mx;
};

void print_int(int x)
{
    std::cout << "[" << x << "] \n";
}

int main()
{
    using namespace std;

    function<void(int)> f(print_int); // function
    f(23);

    f = [] (int x) // Lambda
    {
        cout << " x = " << x << "\n";
    };

    f(45);

    f = Nec::print // class static function

    function fc = &Nec::print_sum; // syntax hatası
    function<void(const Nec&, int)> fc = &Nec::print_sum; // legal
}

```

std::function --exception

```

// std::function --exception
int main()
{
    using namespace std;
    function<int(int)> f;
    try {
        auto val = f(45); // bad_function_call throw eder
    } catch (const std::bad_function_call& ex){
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}

```

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

    function<int(int)> f;

    if(f) //f.operator bool()
        std::cout << "bos degil\n";
    else
        std::cout << "bos\n";
}
```

```
// std::function amaci
void foo(int (*fp)(int))
{
    /*
        Bu fonksiyona sadece function pointer gönderebilirim. Ama,
        Function object, state lambda gönderemem
    */
    auto val = fp(12);
}

// int(int) olan her türlü callable bar fonksiyonu kabul eder.
void bar(std::function<int(int)>);

int f(int);
struct Nec
{
    static int foo(int);
};

struct Erg
{
    bool operator()(int) const;
};

int func(int, int, int);

int main()
{
    auto fn = [] (int x){return x;};
    // hepsi geçerli
    bar(f);
    bar(Nec::foo);
    bar(Erg{});
    bar(fn);

    auto fb = bind(func, _1, _1, _1);
    bar(fb);
}
```

```
// Kullanım Senaryosu 1:  
class Myclass  
{  
    public:  
        void foo()  
        {  
            auto val = mf(12);  
        }  
    private:  
        std::function<int(int)> mf;  
}
```

```
// Kullanım Senaryosu 2:  
int f1(int);  
int f2(int);  
int f3(int);  
int f4(int);  
struct Functor  
{  
    int operator()(int) const;  
};  
  
using fntype = std::function<int(int)>;  
int main()  
{  
    using namespace std;  
    vector<fntype> myvec{ f1, f2, f3, f4 };  
  
    myvec.push_back(Functor{});  
  
    for (auto& f : vec)  
    {  
        auto val = f(12);  
    }  
}
```

std::mem_fn

```
class Myclass
{
public:
    void func()const
    {
        std::cout < "Myclass::func()\n";
    }
    void foo(int x)const
    {
        std::cout < "Myclass::foo(int x) x = \n" << x << "\n";
    }
}

int main()
{
    using namespace std;
    auto f1 = mem_fn(&Myclass::func);

    Myclass mx;
    f1(mx); // "MyClass::func()"

    auto f2 = mym_fn(&Myclass::foo);
    f2(mx, 768); // "Myclass::foo(int x) x = 768"
    Myclass* ptr = &mx;
    f1(ptr);
}
```

```
//std::mem_fn Kullanım Senaryosu 1
int main()
{
    using namespace std;
    vector<string> svec;

    rfill(svec, 20, [] {return rname() + ' ' + rfname();});

    vector<size_t> lenvec(svec.size());

    transform(svec.begin(), svec.end(), lenvec.begin(), [] (const string& s)
    {
        return s.size();
    });

    // ya da

    transform(svec.begin(), svec.end(), lenvec.begin(), mem_fn(&string::size));
}
```

```

//std::mem_fn Kullanım Senaryosu 2
class Nec
{
public:
    Nec(int val) : mval(val) {}
    void print()const
    {
        std::cout << "[" << mval << "]\n";
    }
private:
    int mval;
};

int main()
{
    using namespace std;
    vector<Nec> nvec;

    for(int i = 0; i < 20; ++i)
    {
        nvec.emplace_back(i);
    }

    for_each(nvec.begin(), nvec.end(), mem_fn(&Nec::print));
}

```

std::not_fn

```

int main()
{
    int x = 11;
    auto b = std::isprime(x); // true

    auto f = not_fn(isprime);
    b = f(x); // false
}

```

```

//std::not_fn Kullanım Senaryosu

int main()
{
    using namespace std;
    vector<int> ivec;
    rfill(ivec, 100, Irand{0 , 100'000});

    copy_if(ivec.begin(), ivec.end(), ostream_iterator<int>(cout, "\n"), [](int x)
    { return !isprime(x)} );
    copy_if(ivec.begin(), ivec.end(), ostream_iterator<int>(cout, "\n"),
    not_fn(isprime));
}

```

std::array (C dizilerini sarmalayan bir wrapper)

```
int main()
{
    std::array<int, 5> ar; // ar'in elemanları garbage durumda
```

Neden std::array kullanım?

- STL uyumlu
- Interface var
- .at exception throw ediyor
- array decay yok (dizinin ilk elemanın bir pointer adresine dönüşmesi)
- Geri dönüş değeri ya da parametresi std::array olan fonksiyonlar yazabiliriz.

```
int main()
{
    int a[0]; // C-style array -syntax hatalı

    std::array<int, 0> ax; // boş array geçerli
    boolalpha(cout);
    cout << "ax.size() = " << ax.size() << "\n";
    std::cout << ax.empty() << "\n";
}
```

```
std::array<int,3> foo(int a, int b, int c)
{
    // hepsi geçerli
    //return std::array<int,3> {a, b, c};
    //return std::array {a, b, c}; //
    return {a, b, c};
}
```

```
template <typename T, std::size_t n>
std::ostream& operator << (std::ostream& os, const std::array<T, n>& ar)
{
    for (std::size_t i{}; ar.size(); ++i)
    {
        os << ar[i] << ", ";
    }

    return os << ar.back() << "]";
}

int main()
{
    using namespace std;

    array<int, 3> ax{2, 5, 7};
    cout << ax << "\n";
}
```

```
int main()
{
    using namespace std;
    array a = {"naci", "cemal", "emre"}; // std::array<const char*>
    // literal operator function
    array b = {"naci"s, "cemal"s, "emre"s}; // std::array<string>
}
```

2023 11 24

array decay

```
// array decay
int main()
{
    using namespace std;

    array a {1, 3, 5, 1, 23};
    int *ptr = a; // syntax hatalı (array decay olmuyor)
    int *p1 = a.data();
    int *p2 = &a[0];
    int *p3 = &*a.begin();

    int a[5]{};
    a[2]; // *(a + 2)
    3[a]++; // *(a + 3) ++
}

array<int, 5> b{};
// 3[a]++; gecersiz
```

std::tuple

Farklı veri tiplerini bir arada tutmak için kullanılır

```
#include<tuple>
int main()
{
    // bir container değil
    std::tuple<> t0;
    std::tuple<int> t1;
    std::tuple<int, double> t2;
    std::tuple<int, double, long> t3;
}
```

```
int main()
{
    // default ctor ya da zero init eder
    std::tuple<int, double, long, Date> tp;

    cout << get<0>(tp) << "\n"; // int& döndürür 0

    cout << get<3>(tp) << "\n"; // 1 Ocak 1900
    cout << ++get<3>(tp) << "\n"; // 2 Ocak 1900

    std::tuple<int, double, long> tp1 = {4, 4.5, 45L};
    std::tuple tp2{4, 4.5, 45L}; // CTAD
}
```

`std::make_tuple`

```
// make_tuple
template <typename ...Args>
auto MakeTuple(Args && ...args)
{
    return std::tuple<Args>(std::forward<Args>(args)...);
}

int main()
{
    int x = 235;
    double dval = 3.4;
    char c = 'A';

    auto tp = make_tuple(x, dval, c);
    cout << get<0>(tp) << "\n";
    cout << get<1>(tp) << "\n";
    cout << get<2>(tp) << "\n";

    cout << get<int>(tp) << "\n"; // eger birden fazla int varsa ambiguity olur
}
```

```
// tuple --enum

int main()
{
    using namespace std;

    enum {AGE, NAME, ID};

    using age = int;
    using name = std::string;
    using id = long;

    tuple<age, name, id> tp{ 41, "korhan", 754334L};

    cout << get<AGE>(tp) << " " << get<NAME>(tp) << " " << get<ID>(tp) << "\n";
}
```

```
using namespace std;
using PersonData = std::tuple<int, std::string, Date>;

PersonData get_person_data()
{
    return { Irand(20, 60)(), rname(), Date::random() }
}

int main()
{
    for (int i = 0 ; i < 10; ++i)
    {
        auto [age_name, emp_date] = get_person_data(); // structured binding
        cout << age_name << " " << emp_date;
    }
}
```

```
// structed binding
int main()
{
    auto tp = make_tuple(235, 5.6, "emre");
    auto [x, y, z] = tp;
}
```

```
// structed binding -- C dizileri
int main()
{
    int a[3]() { 54, 78, 90};

    auto &[x, y, z] = a; // dizideki öğe sayısı ile structed bind öğe sayısı aynı olmalı

    cout << "x = " << x << "\n";
    cout << "y = " << y << "\n";
    cout << "z = " << z << "\n";

    a[1] *= 10;

    cout << "y ? " << y << "\n"; // y = 780
}
```

```
// structed binding --struct
struct Data
{
    int a, b, c;
};

Data foo()
{
    return { 10, 20, 30 };
}

int main()
{
    auto [a, b, _] = foo();
}
```

tuple_size and tuple_element

```
using namespace std;
using ttype = std::tuple<int, char, double, Date>

int main()
{
    consteval auto n = tuple_size<ttype>::value; // n = 4
    consteval auto n1 = tuple_size_v<ttype>; // n1 = 4

    tuple_element<1, ttype>::type // char türü
    tuple_element_t<2, ttype> // double türü
}
```

std::tie --reference tutan tuple

```
// reference tutan tuple
int main()
{
    int a = 23; double dval = 4.02; string name {"emre"};

    tuple<int&, double&, string&> tp{ a, dval, name};
    // ya da
    auto tp1 = make_tuple(ref(a), ref(dval), ref(name));
    // ya da
    auto tp2 = tie(a, dval, name); // reference tuple döndürür

    a *= 100;
    dval +=12.0;
    name += "gano"

    cout << get<0>(tp) << '\n';
    cout << get<1>(tp) << '\n';
    cout << get<2>(tp) << '\n';
}
```

```
tuple<int, double, string> foo()
{
    return { 12, 4.56, "emre" };
}

int main()
{
    int ival;
    double dval;
    string name;

    tie(ival, dval, name) == foo();
}
```

std::tuple karşılaştırma

```
// std::tuple karşılaştırma
using namespace std;
                                // id   town   name   tarih
using person = std::tuple<int, string, string, Date>

int main()
{
    vector<person> pvec;
    pvec.reserve(20'000);

    for (int i = 0; i < 20'000; ++i)
    {
        pvec.emplace_back(irand{0, 100}, rtown(), rname(), Date::random());
    }
    sort(pvec.begin(), pvec.end()); // id -> town -> name -> tarih bu sıraya göre
    // sıralar
}
```

```

class Date
{
public:
    Date(int day, int mon, int year);
    friend bool operator<const Date& d1, const Date& d2)
    {
        if (d1.myyear != d2.myyear)
            return d1.myyear << d2.myyear;
        if (d1.mmon != d2.mmon)
            return d1.myyear << d2.myyear;
        return d1.mday < d2.mday;

        // yukarıdaki yerine

        return std::tuple(d1.myyear, d1.mmon, d1.mday)
            < std::tuple(d2.myyear, d2.mmon, d2.mday);
    }
private:
    int mday;
    int mmon;
    int myear;
};

```

```

// rotating assignment
int main()
{
    int x = 10; int y = 20; int z = 30; int t = 40;
    // x = 40, y = 10, z = 20, t = 10;
    int temp = x;
    x = y;
    y = z;
    z = t;
    t = temp;
    // ya da
    tie(x, y, z, t) = tuple(y, z, t, x); // x = 40, y = 10, z = 20, t = 10;
}

```

std::apply

```

using namespace std;
int sum(int x, int y, int z)
{
    return x + y + z;
}

int main()
{
    tuple tx { 3, 6, 9};

    auto ret = apply(sum, tx);
    cout << "ret = " << ret << "\n";
}

```

std::invoke

```
// std::invoke
#include <functional>
using namespace std;

int sum(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 5, int b = 9;
    int ret = sum(a, b);

    ret = invoke(sum, a, b); // generic kodlarda kullanılır
}
```

```
int foo(int);
int main()
{
    // &foo int (*)(int)
    // foo int(int)

    int (*fp1)(int) == foo;
    int (*fp2)(int) == &foo;
}
```

```
// Member Functions Pointers
class Nec
{
public:
    static int foo(int);
    int bar(int);
};

int main()
{
    int (*fp)(int) = &Nec::foo;
    int (*fp1)(int) = &Nec::bar; // syntax hatalı

    int (Nec::*fp2)(int) = &Nec::foo;
}
```

.* operator

```
class Nec
{
public:
    int foo(int)
    {
        std::cout << " Nec::foo(int x) x = " << x << "\n";
        return x * x;
    }
};

int main()
{
    auto fp = &Nec::foo;

    Nec mynec;
    (mynec.*fp)(20);
}
```

```
class Nec
{
public:
    int foo(int x);
    int bar(int x);
    int baz(int x);
    int func(int x);
};

using necfp = int (Nec::*)(int);

int main()
{
    necfp fptr;
    Nec mynec;

    fptr = &Nec::foo;
    fptr = &Nec::bar;
    fptr = &Nec::baz;

    (mynec.*fptr)(456);

    vector<necfp> myvec{ &Nec::foo, &Nec::bar};

    myvec.push_back(&Nec::baz);
    myvec.push_back(&Nec::func);

    necfp ar[] = { &Nec::foo, &Nec::bar, &Nec::baz, &Nec::func};
}
```

Bir fonksiyonun hangi member fonksiyonu çağıracağını seçicek

```
class Nec
{
public:
    void func()
    {
        (*this->*mfp)(34)
    }

    int foo(int x)
    {
        std::cout << "Nec::foo(int x) x = " << x << "\n";
        return x + 5;
    }

    int bar(int x)
    {
        std::cout << "Nec::bar(int x) x = " << x << "\n";
        return x * x;
    }

    int baz(int x)
    {
        std::cout << "Nec::baz(int x) x = " << x << "\n";
        return x * x * xx;
    }
private:
    int (*Nec::*mfp)(int) = &Nec::foo;
};

void gf(Nec& nec, int (*Nec::*fp)(int))
{
    (nec.*fp)(21);
}

int main()
{
    Nec mynec;
    gf(mynec, &Nec::bar);
}
```

```
int main()
{
    int (*Nec::*fp)(int) = &Nec::bar;

    Nec* pnec = new Nec;
    ((*pnec).*fp)(345);
    // ya da
    (pnec->*fp)(457);

    // invoke
    std::invoke(fp, mynec, 345);
    std::invoke(fp, necptr, 99);
}
```

2023 11 29

Member Functions Pointers

```
class Nec
{
    public:
        void foo(int);
};

int main()
{
    auto fp1 = &Nec::foo;
    void (Nec::*fp2)(int) = &Nec::foo;

    Nec mynec;
    (mynec.*fp)(12);

    Nec* necptr = &mynec;
    (necptr->*fp)(12);
}
```

```
// Kullanım 1:
class Nec
{
    public:
        void f1(int);
        void f2(int);
        void f3(int);
        void f4(int);
};

void foo(int, void (Nec::*mfp(int)))
{
    // hangi üye fonksiyonun işi yapmasını istiyorsak onun adresini göndercez
}

using Mfptr = int (Nec::*)(int);
int main()
{
    Mfptr fpa[] =
    {
        &Nec::f1,
        &Nec::f2,
        &Nec::f3,
        &Nec::f4,
    }

    Nec m;

    for (auto ptr : fpa)
    {
        std::cout << (m.*ptr)(10) << '\n';
        std::invoke(ptr, m, 10) << '\n';
    }
}
```

Data Member Pointer

```
struct Nec
{
    int a {3};
    int b {5};
    int c {7};

};

int main()
{
    Nec mynec;

    auto ptr = &mynec.a; // int*
    auto ptr1 = &Nec::a // int Nec::*;

    std::cout << mynec.*x << "\n"; // 3
    std::invoke(ptr, mynec) = 9999;

    std::cout << mynec.a << '\n';

}
```

```
// Kullanım 1:
struct OHLC
{
    double open;
    double high;
    double low;
    double close;
};

double get_moving_average(const std::vector<OHLC>& candles, double OHLC::*fp)
{
    double sum{};
    for (const auto& candle : candles)
    {
        //sum += candle.*fp;
        sum += std::invoke(fp, candle);
    }

    return sum / candles.size();
}
```

std::bitset

```
#include <bitset>
int main()
{
    using namespace std;
    // bir container değil
    bitset<16> bs1; // 16 bit sağlıyor

    bitset<32> bx{"101010110110101100"};

    bx[4] = true;
    bx[4].flip(); // ters çevirme
    auto n = bx.count(); // kaç tane bit true olduğunu söyler

    auto non = bx.none(); // zero bitse true döner
    auto any = bx.any(); // herhangi bir true ise true döner
    auto all = bx.all(); // tüm bitler true ise true döner

    auto test = bx.test(2); // 2. bit true mu false mu döner

    auto ulong = bx.to_ulong(); // bit'i ulong'a dönüştürür
    auto to_string = bx.to_string(); // bit'i string'e dönüştürür

    bx.set(5, false); // 5. biti false set ettik
    bx.reset(5); // 5. biti resetledik ( false yaptık )

    bx.set(4).reset(3).flip(7);

    bitset<16> x {3124u};
    bitset<16> y {3124u};

    cout << x == y << "\n"; // true
    //cout << x < y; // böyle bir operator yok

    auto fless = [] (const bitset<32>& b1, const bitset<32>& b2)
    {
        return b1.to_ulong() < b2.to_ulong();
    }

    set<bitset<32>, decltype(fless)> bset;

    bitset<16> z{1u};
    cout << (x << 2) << "\n";
    cout << x << 2 << "\n";
}
```

Dinamik Ömürlü Nesneler

global operator new ile neler yapabilirim:

- overload edebilirim
- isimliye çağrılabılırız
- bir sınıf için overload edebiliriz

new T

operator new(sizeof(T))

```
// my operator new overload
void* operator new(std::size_t n)
{
    std::cout << "operator new(size_t) called\n";
    std::cout << "n = " << n << "\n";

    auto vp = std::malloc(n);
    if (!vp)
        throw std::bad_alloc{};

    std::cout << "the address of the allocated block is " << vp << "\n";
}

void operator delete(void *vp)
{
    std::cout << "operator delete called vp = " << vp << "\n";
}

class Neco
{
    unsigned char buffer[2048]{};
};

int main()
{
    std::cout << "sizeof(Neco) = " << sizeof(Neco) << "\n";

    Neco* p = new Neco;

    delete p;
}
```

operator new işlevinin başarısız olması durumunda std::bad_alloc sınıfı türünden bir exception throw eder.

```
#include <new>
void myhandler()
{
    std::cout << "myhandler called!!!";
}

int main()
{
    vector<void*> myvec;

    // exception throw etmek yerine myhandler çağrılacak her döngüde
    set_new_handler(myhandler);

    try
    {
        for (int i = 0; i < 10000; ++i)
        {
            myvec.push_back(operator new(1024 * 1024));
        }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
```

set_new_handler nasıl kullanabilirim:

1. Daha önce allocate ettiğim belliğin geri verilmesi
2. Daha özel exception sınıfını throw edebiliriz
3. Başka bir handler fonksiyonu çağrılabılır.
4. Bir kaç defa deneme yapıp set_new_handler(nullptr) yapabilir.

2023 12 01

Dinamik Ömürülu Nesneler

- new ifadesiyle oluşturduğumuz bellek alanını free edemeyiz.
- malloc ile oluşturduğumuz bellek alanını delete edemeyiz.

```
class Nec
{
public:
    Nec();
    Nec(int);
    Nec(int, int);
    Nec(const char*);
};

int main()
{
    Nec *p1 = new Nec;
    Nec *p2 = new Nec(234);
    Nec *p3 = new Nec(234, 21);
    auto p = new const Nec(243); // const Nec*
    // array new
    Nec *p = new Nec[10]; // 5 tane Nec* olusacak
    delete[] p; // array delete
}
```

placement new

```
//new (x, y, z)MyClass
#include <new>
void* operator new(size_t, void* ptr)
{
    return ptr;
}

class Nec
{
public:
    Nec(){
        std::cout << "CTOR: " << this << "\n" ;
    }
    ~Nec(){
        std::cout << "DTOR: " << this << "\n" ;
    }
private:
    unsigned char buf[256]{};
};

int main()
{
    unsigned char buffer[sizeof(Nec)];
    std::cout << "buffer dizisinin adresi : " << static_cast<void*>
    // buffer ile aynı adreste Nec oluşturuz.
    Nec *p = new(buffer)Nec; // void* operator new(size_t, void* ptr)
    // tanimsız davranış çünkü bellek alanı operator new ile elde edilmedi
    delete p;
    p->~Nec() // delete yerine dtor çağırıcaz
}
```

placement new -- emplace_back

```
// placement new -- emplace_back
template<typename ...Args>
void vector<T>::emplace_back(Args&& ...args)
{
    new(adres)T(std::forward<Args>(args)...);
}
```

```
/*
 void* operator new( std::size_t count, const std::nothrow_t& tag);
new(nothrow)
başarısız olursa exceptions throw etmez nullptr döndürür

*/
class Myclass;
int main()
{
    auto p = new(nothrow)Myclass; // exception throw etmez

    if (!p) // başarısız olduysa nullptr döndürür
    {

    }
}
```

new ifadesini neden doğrudan kullanmamalıyız?

- 1) dinamik ömürlü, otomatik ömürlü yada statik ömürlü mü belirsiz
- 2) nullptr değerinden mi?
- 3) delete etmeli miyim?
- 4) hangi delete ifadesini kullanmalıyım?
- 5) sadece delete yeterli mi?

smart pointer

- unique_ptr : bir kaynağın tek sahibi var
- shared_ptr : bir nesneyi gösteren birden fazla pointer olabilir
 - weak_ptr

std::unique_ptr

```
template <typename T>
struct DefaultDelete
{
    void operator()(T* p)
    {
        delete p;
    }
};

// D ==> deleter parametre
template <typename, typename D = std::default_delete<T>>
class UniquePtr
{
public:
    ~UniquePtr()
    {
        if (mp)
            D{ } (mp);
    }
private:
    T *mp;
}
```

```
#include <memory>
#include "date.h"

int main()
{
    using namespace std;
    unique_ptr<Date> up;

    if (up)
        std::cout << "dolu\n";
    else
        std::cout << "bos\n";

    if (up == nullptr)
    if (up != nullptr)
    if (up.get() == nullptr)

    // eğer unique_ptr boşsa  tanımsız davranış
    cout << *up << "\n";
    cout << up->month_day() << "\n";
}
```

```

class Myclass
{
public:
    Myclass()
    {
        std::cout << "Myclass default ctor this = " << this << "\n";
    }
    ~Myclass()
    {
        std::cout << "Myclass dtor this = " << this << "\n";
    }
};

int main()
{
    std::cout << "main basladi\n";
    {
        // unique_ptr explicit olduğu için hatalı
        // std::unique_ptr<Myclass> uptr = new Myclass;
        std::unique_ptr<Myclass> uptr{new Myclass};
        // scope sonunda uptr'in ömürü biter
    }

    std::cout << "main devam ediyor\n";
}

```

```

template <typename T, typename ...Args>
std::unique_ptr<T> MakeUnique(Args&& ...args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

int main()
{
    using namespace std;

    auto up = make_unique<Date>(3, 5, 1898);
}

```

```

void foo(std::unique_ptr<std::string> up);

int main()
{
    // ctor explicit olduğu için syntax hatası
    foo(new std::string{"necati"}); // syntax hatası
    unique_ptr<Date> x{new Date};
    auto y = x; // syntax hatası çünkü copy ctor deleted
}

```

```
class Tamer {
public:
/*
    unique_ptr'in copy ctor ve assignment delete olduğu için Tamer sınıfının
    copy ctor ve assignment'i delete eder derleyici.
*/
private:
    std::unique_ptr<Date> uptr;
}

int main()
{
    Tamer tx;
    Tamer ty;

    tx = ty; // syntax hatalı
    tx = std::move(ty); // geçerli
}
```

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

    auto upx = make_unique<Date>(2, 5, 1982);
    auto upy = make_unique<Date>(2, 5, 1982);

    // upy'in dtor çağrıılır move yapınca
    upx = move(upy);
}
```

```
std::unique_ptr<int> foo()
{
    auto up = std::make_unique<int>(24);
    return up; // move only class'ları döndürebiliriz
}

///

void foo(std::unique_ptr<int>)
{

}

int main()
{
    using namespace std;

    auto up = make_unique<int>(345);
    // foo(up); // Legal değil --copy ctor deleted
    foo(move(up)); // Legal
    foo(make_unique<int>(7123)); // Legal
    foo(unique_ptr<int>(new int)); // Legal --temp obje

    // R value olanların hepsini foo'ya arguman olarak verebiliriz.
}
```

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

    auto *p = new Date(3, 6, 1923);

    // tanımsız davranış
    unique_ptr<Date> upx(p);
    unique_ptr<Date> upy(p);

    /*
        upx'in hayatı bittiğinde upy'in hayatı bitmemiş olacak
        yani dangling pointer olacak
    */
}

}
```

```
// unique_ptr'in hayatını sonlandırma
int main()
{
    using namespace std;
    auto up = make_unique<Date>(31, 12, 2023);
    // sonlandırma yöntemleri
    up.reset(); //up.reset(nullptr);
    up = nullptr;
    up = unique_ptr<Date>{};
    up = {};
}
```

```
// unique_ptr reset() fonksiyonu
int main()
{
    using namespace std;
    auto up = make_unique<Date>(31, 12, 2023);

    up.reset(new Date(1, 2, 2024));
    up = make_unique<Date>(1, 1, 2024);
}
```

```
// unique_ptr release() fonksiyonu
int main()
{
    using namespace std;
    auto up = make_unique<Date>(31, 12, 2023);

    Date *p = up.release(); // up'a boşça çıkar ama dtor çağrılmaz

    // ikisi aynı şey
    auto x = move(up);
    unique_ptr<Date> x(up.release());
}
```

2023 12 04

std::unique_ptr

```
class Myclass;
int main()
{
    using namespace std;
    unique_ptr<Myclass>; // incomplete type olarak kullanabiliriz
}
```

```
/*
    uptr.release(): sarmaladığı adresi döndürür mülkiyeti bırakır (dtor çağrırmaz)
    uptr.get(): sarmaladığı adresi döndürür
*/
```

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

    auto uptr = make_unique<string>("alican korkmaz");
    // ikiside aynı
    cout << uptr << "\n";
    cout << uptr.get() << "\n";
    // tanımsız davranış
    auto p = uptr.get();
    unique_ptr<string> upx(p);
    // geçerli
    auto p = uptr.release();
    unique_ptr<string> upx(p);
}
```

```
template<typename T>
struct DefaultDelete
{
    void operator()(T* p)
    {
        delete p;
    }
};

template<typename T, typename D = std::default_delete<T>>
class UniquePtr
{
public:
    ~UniquePtr()
    {
        if (mp)
        {
            D{}(mp);
        }
    }

private:
    T* mp;
}
```

```

#include <iomanip>
struct SDeleter
{
    void operator()(std::string* p) const noexcept
    {
        std::cout << std::quoted(*p) << " delete ediliyor\n";
        delete p;
    }
}
void fdeleter(std::string *p)
{
    std::cout << std::quoted(*p) << " delete ediliyor\n";
}
int main()
{
    using namespace std;

    {
        //1
        unique_ptr<string, SDeleter> uptr{new string("tamer dundar")};
        //2
        unique_ptr<string, decltype(&fdeleter)> uptr{new string("tamer dundar")};
        //3
        auto fdel = [] (string *p)
        {
            std::cout << std::quoted(*p) << " delete ediliyor\n";
            delete p;
        };
        unique_ptr<string, decltype(fdel)> uptr{new string("tamer dundar")};

        //4
        decltype([]) string *p
        {
            std::cout << std::quoted(*p) << " delete ediliyor\n";
            delete p;
        };
        unique_ptr<string, decltype([]) string *p>
        {
            std::cout << std::quoted(*p) << " delete ediliyor\n";
            delete p;
        };> uptr{new string("tamer dundar")};

    }

    std::cout << "main devam ediyor\n"
/*
    tamer dundar delete ediliyor\n
    main devam ediyor
*/
}

```

Neden kendi deleter'ımız kullanmalıyız:

- Delete etmek dışında kaynağı başka bir şekilde sonlandırmak istersek.
- Delete ederken yanında başka işlemler yapmak istersek.

```
#include<cstdio>
int main()
{
    auto fdel = [](std::FILE* f)
    {
        std::cout << "file is being closed!\n";
        std::fclose(f);
    };

    std::unique_ptr<std::FILE, decltype(fdel)> uptr{fopen("melike.txt"), "w"  };
    fprintf(uptr.get(), "Emre Bahtiyar");
}
```

```
// C-style Ctor Dtor
// unique_ptr ile herhangi bir kaynağı sarmalayabiliriz.
struct Data
{

Data * createData(void);
void do_something(Data *);
void do_this(Data *);
void do_that(Data *);
void destroyData(Data *);

int main()
{
    auto fdel = [](Data *p)
    {
        destroyData(p);
    };

    unique_ptr<Data, decltype(fdel)> uptr(createData());

    do_something(uptr.get());
    do_this(uptr.get());
    do_that(uptr.get());
    destroyData(uptr.get());
}
```

```

struct Nec
{
    Nec()
    {
        std::cout << "Nec default ctor this : " << this << "\n";
    }

    ~Nec()
    {
        std::cout << "Nec default dtor this : " << this << "\n";
    }

    char buf[256]{};
}

int main()
{
    using namespace std;
    // tanımsız davranış 4 defa ctor çağrıları ama bi kere dtor çağrıları
    unique_ptr<Nec> uptr(new Nec[4]);

    // 4 kez ctor çağrıları 4 kez dtor çağrıları
    auto fd = [](Nec *p) {delete []p;};
    unique_ptr<Nec, decltype(fd)> uptr(new Nec[4]);

    // unique_ptr için partial spec for array pointer
    unique_ptr<Nec[]> uptr(new Nec[4]);
    auto up = make_unique<Nec[]>(5);
}

```

```

int main()
{
    Date *p = new Date{3, 12, 1872};

    {
        unique_ptr<Date> upx(p);
    }
    // tanımsız davranış
    unique_ptr<Date> upy(p); // p dangling pointer oldu
}

```

```

// unique_ptr -- container'da tutma
int main()
{
    using namespace std;

    vector<std::unique_ptr<Date>> dvec;

    dvec.reserve(10);
    for (auto i = 1; i <= 10; ++i)
    {
        dvec.push_back(new Date {i ,i 2000 +i});
    }
}

```

```

int main()
{
    using namespace std;
    vector<unique_ptr<Date>> dvec;

    auto uptr = make_unique<Date>(1, 5, 1986);

    dvec.push_back(move(uptr));
    dvec.push_back(make_unique<Date>(1, 5, 1986));

    dvec.emplace_back(new Date{5, 6, 1965});
}

```

```

void fsink(std::unique_ptr<Date> uptr)
{
    std::cout << *uptr << '\n';
    // Date sınıfı bu scopeta yok olur
}

int main()
{
    using namespace std;
    fsink(make_unique<Date>(3, 6, 1987));
    cout << "main devam ediliyor\n";
}

```

```

// pass-through
std::unique_ptr<Date> pass_through(std::unique_ptr<Date> uptr)
{
    std::cout << *uptr << '\n';
    return uptr;
}

int main()
{
    using namespace std;
    auto up = pass_through(make_unique<Date>(3, 6, 1987));
    // uptr main scope sonunda sonlanır
    cout << "main devam ediyor\n";
}

```

std::shared_ptr

```

// std::shared_ptr -- bir nesnenin birden fazla sahibi olabilir
int main()
{
    using namespace std;
    std::cout << "sizeof(unique_ptr<string>)" = << sizeof(unique_ptr<string>) <<
"\n"; // 4
    std::cout << "sizeof(shared_ptr<string>)" = << sizeof(shared_ptr<string>) <<
"\n"; // 8
}

```

```
template<typename T>
class SharedPtr
{
};

int main()
{
    using namespace std;

    shared_ptr<Date> sp1(new Date{ 1, 2, 1998});
    {
        auto sp2 = sp1;
        // sp2 burda biter ama dtor çağrılmaz
    }
    std::cout << "main devam ediyor\n";
    // sp1 burda sonlanır ve dtor çağrılr
}
```

```
// use_count()
int main()
{
    using namespace std;
    shared_ptr<Date> sp1(new Date{ 1, 2, 1998});
    auto sp2 = sp1;

    //use count = 2
    cout << "use count = " sp1.use_count() << "\n";
    cout << "use count = " sp2.use_count() << "\n";

    {
        auto sp3 = sp2;
        cout << "use count = " sp3.use_count() << "\n"; // use count = 3
    }

    cout << "use count = " sp2.use_count() << "\n"; // use count = 2
}
```

```
void* operator new(std::size_t n)
{
    std::cout << "operator new called n = " << n << "\n";

    void* vp = std::malloc(n);
    if (!vp)
    {
        throw std::bad_alloc{};
    }
    std::cout << "the address of the allocated block is: " << vp << "\n";

    return vp;
}

struct Nec
{
    char buf[512]{};
};

void foo()
{
    std::cout << "foo called\n";
    auto pnc = new Nec;
    std::shared_ptr<Nec> sptr(pnc);
    // burda önce Nec için blok açılıyor
    // sonra shared_ptr'in kontrol bloğu için yer açılıyor
}

void bar()
{
    std::cout << "foo called\n";
    auto pnc = new Nec;
    std::shared_ptr<Nec> sptr(pnc);
    // burda Nec ve kontrol bloğu için ayrılan yer aynı anda açılıyor
    // derleyici optimasyon yapıyor
}

int main()
{
    foo();
    bar();
}
```

```

// type erasure -- shared_ptr
class Myclass
{
public:
    ~Myclass()
    {
        std::cout << "Myclass dtor\n";
    }
};

struct MyclassDelete
{
    void operator()(Myclass* p) const
    {
        std::cout << "the object at the address of " << p << "is being deleted\n";
        delete p;
    }
};

int main()
{
    using namespace std;

    {
        shared_ptr<Myclass> sptr(new Myclass, MyclassDelete{});
    }

    std::cout << "main devam ediyor\n";
}

```

```

// unique_ptr to shared_ptr
int main()
{
    using namespace std;
    auto uptr = make_unique<Date>(1, 1, 2024);
    shared_ptr<Date> sptr(move(uptr));

    /*
        std::shared_ptr kontrol bloğu ne zaman oluşur:
        - eğer default dtor ile oluşturulduysa kontrol bloğu oluşmaz
        - eğer 2. ya da daha fazla shared_ptr oluşturulduysa kontrol bloğu
        oluşmaz.
        - unique_ptr'den shared_ptr dönüştürüyorsak kontrol bloğu oluşur
        - 1. kez shared_ptr oluşturuyorsak kontrol bloğu oluşur.
    */
}

```

```
// shared_ptr fonksiyonları
int main()
{
    using namespace std;
    shared_ptr<Date> sp1(new Date{1, 1, 2024});

    auto sp2 = sp1;
    auto sp3 = sp2;
    // içinde aynı pointer sarmalar yani get fonksiyonları aynı adresi döndürür
    cout << sp1.get() << "\n";
    cout << sp2.get() << "\n";
    cout << sp3.get() << "\n";

    if (sp) // boş mu dolu mu operator bool ile yapılabilir
        sp1.reset(); // mülkiyeti bırakıyor unique_ptr'deki release gibi
}
```

```
void foo(std::unique_ptr<std::string>)
{
    // reference sayacı artıracak sonra fonksiyon sonlanınca azalacak
}

void bar(std::unique_ptr<std::string>& r)
{
    // reference sayacı artıracak
    // eğer shared_ptr değerini değiştirmiceksek, reset yapmıcaksak
    // fonksiyon parametresini reference yapmaya gerek yok
}
```

2023 12 06

std::weak_ptr

```
#include <memory>
#include <string>

using namespace std;

int main()
{
    /*
        shared_ptr ile oluşturduğumuz weak_ptr ile shared_ptr'in kaynağının
        sonlanıp sonlamadığına bakabiliyoruz.
    */

    shared_ptr<string> sptr(mew string{"tamer dundar"});
    cout << "sptr.use_count() = " << sptr.use_count() << "\n";
    weak_ptr wp = sptr;
    // sptr.use_count() sabit kalır
    cout << "sptr.use_count() = " << sptr.use_count() << "\n";

    weak_ptr wp1 = sptr;
    cout << "wp.use_count() = " << wp.use_count() << "\n";
    cout << "wp1.use_count() = " << wp1.use_count() << "\n";

    auto wp3 = wp2;
    cout << "wp3.use_count() = " << wp3.use_count() << "\n";

    // hepsi 1 çıkar.
}
```

```
// weak_ptr --expired and Lock
int main()
{
    shared_ptr<string> sptr(mew string{"tamer dundar"});
    weak_ptr wp = sptr;
    // true dönerse kaynak sonlamış, false verirse kaynak halen hayatta
    wp.expired()

    // kaynak hayattaysa shared_ptr döndürür, hayatta değilse nullptr döndürür
    auto spx = wp.lock()
    // cpp 98s
    if (shared_ptr<string> spx1 = wp.lock(); spx != nullptr)
    {

    }

    shared_ptr sp(wp); // kaynak sonlamışsa bad_weak_ptr throw eder
}
```

```

struct A
{
    std::shared_ptr<B> bptr;
    A()
    {
        std::cout << "A default ctor this = " << this << "\n";
    }
    ~A()
    {
        std::cout << "A default ctor this = " << this << "\n";
    }
};

struct B
{
    std::shared_ptr<A> bptr;
    B()
    {
        std::cout << "B default ctor this = " << this << "\n";
    }
    ~B()
    {
        std::cout << "B default ctor this = " << this << "\n";
    }
};

int main()
{
    using namespace std;

    shared_ptr<A> spa(new A);
    shared_ptr<B> spa(new B);
    // dtor çağrılmaz
    spa->bptr = spb;
    spb->aptr = spa;
}

```

CRTP (Curiously Recurring Template Pattern)

```

// CRTP (Curiously Recurring Template Pattern)
using namespace std;
//CRTP Base
template <typename T>
class Base
{
    // taban sınıf türemiş sınıfa interface sağlar ve taban sınıf bu interfaceye
    // türemiş sınıfın fonksiyonlarını kullanabilir.
    void func()
    {
        static_cast<Der*>(this).foo();
    }
};

class Der : public Base<Der>
{
};

```

```

class Nec{};
int main()
{
    Nec *ptr = new Nec;

    // tanımsız davranış dangling pointer oluşturur
    shared_ptr<Nec> sp1(ptr);
    shared_ptr<Nec> sp2(ptr);

    cout << sp1.use_count() << '\n'; // 1
    cout << sp2.use_count() << '\n'; // 1

    // tanımsız davranış yok
    shared_ptr<Nec> sp3(sp2);
    cout << sp3.use_count() << '\n'; // 2
}

```

Eğer bir sınıfın üye fonksiyonu içinde shared_ptr ile hayatı kontrol edilen *this nesnesini gösteren shared_ptr'nin kopyasını çıkartmak isterseniz sınıfınızı CRTP örüntüsü ile kattığınız yoluyla std::enable_shared_from_this sınıfından elde etmelisiniz.

```

class Nec : public std::enable_shared_from_this<Nec>
{
public:
    void func()
    {
        /*
            reference count 2 olmaz hala 1 olur. tanımsız davranıştır
        */
        shared_ptr<Nec> spx(this);
        cout << spx.use_count() << "\n";
    }

    void bar()
    {
        auto spx = shared_from_this();
        cout << "spx.use_count() = " << spx.use_count() << "\n"; // 2
    }
};

int main()
{
    shared_ptr<Nec> sptr(new Nec);

    cout << "sptr.use_count() = " << sptr.use_count() << "\n"; // 1
    sptr->func();
    cout << "sptr.use_count() = " << sptr.use_count() << "\n"; // 1

    auto p = new Nec;
    try{
        p->bar(); // exception throw eder
    }
    catch(const std::bad_weak_ptr& ex) {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}

// container'da shared_ptr tutma

```

```
#include <list>
using namespace std;
int main()
{
    list<shared_ptr<Date>> mylist;

    mylist.push_back(make_shared<Date>(2, 6, 1999));
    mylist.push_back(make_shared<Date>(2, 6, 1993));
    mylist.push_back(make_shared<Date>(21, 11, 1997));
    mylist.push_back(make_shared<Date>(7, 10, 2004));

    for (auto ptr : mylist)
    {
        cout << *ptr << "\n";
    }

    vector<shared_ptr<Date>> myvec(mylist.begin(), mylist.end());

    // vectordeki shared_ptr değiştirsem Listteki shared_ptr değişir

    sort(myvec.begin(), myvec.end(), [](auto p1, auto p2)
    {
        return *p1 < *p2;
    });

    for (auto sp : myvec)
    {
        cout << *sp << '\n';
    }
}
```

```

using svector = std::vector<std::string>
class NameList
{
    public:
        NameList() = default;
        NameList(std::initializer_list<std::string> list) : sptr(new
svector{list})
        {

    }
    void add(const std::string& name)
    {
        sptr->push_back(name);
    }

    void remove(const std::string& name)
    {
        sptr->erase(std::remove(sptr->begin(), sptr->end(), name), sptr-
>end());
    }

    size_t size()const
    {
        return sptr->size();
    }

    void print()const
    {
        for(const auto& s : *sptr)
        {
            std::cout << s << " ";
        }
        std::cout << "\n";
    }

    void sort()
    {
        std::sort(sptr->begin(), sptr->end());
    }
    private:
        std::shared_ptr<svector> sptr;
}

int main()
{
    // x, y ve z 'nin veri elemani olam shared_ptr ayni vektörü gösterir.
    NameList x{ "ali", "gul", "eda", "naz"};
    NameList y = x;
    NameList z = y;

    x.add("nur");
    y.add("tan");

    std::cout << "listede " << x.size() << "isim var\n";
    z.sort()
    x.print();
    y.remove("gul");
    z.print();
}

```

```
// Bir sınıf için dinamik bellek yönetimini özelleştirme
class Myclass
{
    Myclass()
    {
        std::cout << "default ctor this : " << this << "\n";
    }

    ~Myclass()
    {
        std::cout << "dtor this : " << this << "\n";
    }

    // operator new ve delete static yazmasak bile static üyelerdir
    void* operator new(size_t)
    {
        std::cout << "Myclass::operator new n:" << n << "\n";
        auto p = std::malloc(n);

        if (!p)
        {
            throw std::bad_alloc{};
        }

        std::cout << "address of the allocated block is " << p << "\n";
    }
    void operator delete(void* vp)noexcept
    {
        std::cout << "Myclass::operator delete vp:" << vp << "\n";
        std::free(vp);
    }

    void func()
    {
        std::cout << "Myclass func() this = " << this << "\n";
    }

    private:
        unsigned char buf[512]{};
};

int main()
{
    auto p = new Myclass;
    delete p;
}
```

2023 12 08

input output operations

```
/*
    cout ve cin bir sınıf nesnesidir.

    ios_base

    basic_ios<c, t> ios

    basic_istream      basic_ostream
    istream            ostream

    basic_iostream
*/
```

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

    cout << "mert sirakaya\n";
    operator<<(cout, "mert sirakaya\n");

    cout << 'A'; //--A
    operator<<(cout, 'A'); // char --A
    cout.operator << ('A'); // int --65
}
```

```
class ostream
{
public:
    ostream& operation<<(int);
    ostream& operation<<(double);
    ostream& operation<<(float);

    ostream& operation<<(ostream&(*)(ostream&)); // function pointer
};

// global fonksiyonlar
operator<<(ostream&, const char* );
operator<<(ostream&, char );
```

```
/*
    format state
    on-off flags

    on          off          default
    true-false   1/0          off
    showpos
    uppercase

*/
```

```
// ios::boolalpha
int main()
{
    using namespace std;
    cout << true << false << "\n"; // 10
    cout.setf(ios::boolalpha);
    cout.setf(cout.flags() | ios::boolalpha);
    cout << true << false << "\n"; // truefalse

    if (cout.flags() & ios::boolalpha)
    {
        cout << "true false olarak yazar\n";
    }
    else
    {
        cout < "1 0 olarak yazar\n";
    }

    cout.flags(cout.flags() & ~ios::boolalpha);
    cout.unsetf(ios::boolalpha);

    cout << true << false << "\n"; // 10
}
```

```
/ios::showpoint

int main()
{
    using namespace std;

    double dval = 4.;
    cout << dval << "\n"; // 4
    cout.setf(ios::showpoint);
    cout << dval << "\n"; // 4.0000000000
}
```

```
/*
    Gerçek sayıların gösterimi
    a) fixed           ios::fixed
    b) scientific      ios::scientific
    c) büyülüğüne bağlı

*/

int main()
{
    cout.setf(ios::fixed, ios::floatfield);
    cout << "ios::fixed : " << (cout.flags() & ios::fixed ? "set" :
"unset") << "\n";
    cout << "ios::scientific: " << (cout.flags() & ios::scientific? "set" :
"unset") << "\n";
    cout << 7324.72345 << "\n";
    cout << 8732480245453.872345<< "\n";
}
```

```

// ostream manipulator
class ostream
{
public:
    ostream& operator<<(int);
    ostream& operator<<(double);
    ostream& operator<<(ostream&(*fp)(ostream&))
    {
        return fp(*this);
    }
    // cout << endl end fonksiyonuna cout göndermiş oluyoruz
};

///////

std::ostream& Boolalpha(std::ostream& os)
{
    os.setf(std::ios::boolalpha);
    return os;
}
std::ostream& NoBoolalpha(std::ostream& os)
{
    os.unsetf(std::ios::boolalpha);
    return os;
}

int main()
{
    std::cout << Boolalpha << (10 > 5) << NoBoolalpha << (10 > 5=;

    std::cout << std::boolalpha << (10 > 5) << std::noboolalpha << (10 > 5=;
}

```

```

std::ostream& dline(std::ostream& os)
{
    return os << "\n-----";
}

int main()
{
    cout << 12 << dline << dline << 23.5 << "emre";
}

```

2023 12 11

formatlama

```
/*
formatlama nitelikleri 2 ye ayrılır:
- on-off bayrakları
- alan bayrakları

on-off
.setf(ios::boolalpha)
.unsetf(ios::boolalpha)

ios::showbase
ios::uppercase
ios::skipws
ios::showpoint

alan bayrakları
.setf(ios::hex, ios::basefield);
.setf(ios::left, ios::adjustfield);
.setf(ios::fixed, ios::floatfield);

*/
```

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

    std::cout << "bir tam sayı girin: ";
    int x{};

    cin.setf(ios::hex, ios::basefield); // ab girdik
    cin >> x; // 171 çıktı

    cout << "x = " << x << "\n";

    int k{};
    int l{};
    int m{};

    cin >> hex >> k >> oct >> l >> dec >> m;
}
```

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

    cin.unsetf(ios::dec);
    // hiçbir if'e girmez
    if (cin.flags() & ios::hex) { std::cout "hex set\n";}
    if (cin.flags() & ios::dec) { std::cout "dec set\n";}
    if (cin.flags() & ios::oct) { std::cout "oct set\n";}

    int x{};
    // kendi karar verir hangisi olacağına
    cout << "bir sayı girin: ";
    cin >> x;
    cout << "x = " << x << "\n";
}
```

```
// setbase
#include <iomanip> // parametreli manipülatör için bu lib kullanılır
int main()
{
    using namespace std;
    int x;

    cin >> setbase(0); >> x;
    cin >> hex >> x;
}
```

```
int main()
{
    /* std::unitbuf() kullanmadık. Eğer fonksiyona gönderilen arguman bir
     * namespace ilişkin türden ise fonksiyon o namespace içinde aranır. Buna
     * ADL (argument dependent lookup)
     */
    unitbuf(std::cout);
    endl(std::cout);
    flush(std::cout);
}
```

```
// C memory yazma
#include <cstdio>
int main()
{
    int x = 5;
    double dval = 4.5;

    char name[] = "melike";
    char buffer[256];

    sprintf(buffer, "x = %d dval = %f name = %s", x, dval, name);

    int k, l, m;

    char buf[] = "213 456 790";
    sscanf(buf, "%d%d%d", &k, &l, &m);

    printf("k = %d l = %d m = %d");
}
```

```
// C++ belleğe formatlı yazma
#include <iostream>
int main()
{
    // basic_ostringstream<char> = ostringstream<char> aynı anlama türes işim

    ostringstream os;
    int amount = 200'000;

    os << "benim " << amount << " dolarım var\n";

    auto s = os.str();
    reverse(s.begin(), s.end());
    cout << s << "\n";

    os << " emre";
    auto s = os.str() // üzerine yazar
}
```

```
int main()
{
    int day, mon, year;

    cout << "tarihi giriniz: ";
    cin >> day >> mon >> year;

    ostringstream oss;

    oss << setfill('0') << setw(2) << day << '_' << setw(2) << mon << '_' << year
    << ".txt";
    cout << "[" << oss.str() << "]"; //04_08_1987.txt
}
```

```

// mülakat sorusu
class date
{
public:
    date(int day, int mon, int year) : day_(day), mon_(mon), year_(year){}
    friend std::ostream& operator<<(std::ostream& os, const date& dt)
    {
        //40 karakterlik alana sadece 11 yazar çıktı 11 -12-
2023mustafa
        //return os << dt.day_ << "-" << dt.mon_ << "-" << dt.year_;
        os << dt.day_ << "-" << dt.mon_ << "-" << dt.year_;
        // 11-12-2023mustafa yazar
        return os.str();
    }
private:
    int day_, mon_, year_;
}

int main()
{
    date mydate{ 11, 12, 2023};
    cout << left << setw(40) << mydate << "mustafa\n";
}

```

```

// ostringstream formatlama
int main()
{
    using namespace std;

    ostringstream oss;

    oss << hex << uppercase << showbase;
    oss << 47802 << " " << 54807;
    cout << oss.str() << "\n"; // 0XBABA 0XD617
}

```

```

//istreamstream
using namespace std;
int main()
{
    cout << "toplanacak sayilari girin: ";

    string str;
    getline(cin, str);

    cout << "[" << str << "]";
    istreamstream iss {str};
    int ival;
    int sum{};
    while(iss >> ival){
        sum += ival;
        cout << ival << "\n";
    }
    cout << sum;
}

```

```
int main()
{
    cout << "sayilar giriniz: ";
    string str;
    getline(cin, str);

    istringstream iss{ str };
    int ival;

    vector<int> ivec;

    while(iss >> ival)
    {
        ivec.push_back(ival);
    }

    sort(ivec.begin(), ivec.end());

    for (const auto val : ivec)
    {
        cout << val << "\n";
    }
}
```

```
int main()
{
    cout << "bir cumle girin : ";
    string str;

    getline(cin, str);
    istringstream iss(str);

    string word;
    vector<string> svec;

    while (iss >> word)
    {
        svec.push_back(move(word));
    }

    mt19937 eng;

    for (int i = 0 ; i < 20; ++i)
    {
        shuffle(svec.begin(), svec.end(), eng);
        copy(svec.begin(), svec.end(), ostream_iterator<string>(cout, " "));
    }
}
```

```

// condition state --iostate
/*
    .good() akım(stream) sağlıklı durumdaysa true
    .eof() akım hata durumunda ve hatanın nedeni stream karakter olmadıysa true
döner
    .fail() akımda hata var ama kullanabiliriz
    .bad() // akımda hata var ve akımı kullanamayız
*/
int main()
{
    cout << boolalpha;
    cout << "bir tam sayı giriniz: ";
    int x{};

    cin >> x; // string verirsek fail ve bad true dönür int verirse good true
döner
    cout << "cin.good() : " << cin.good() << "\n"; // true
    cout << "cin.eof() : " << cin.eof() << "\n"; // false
    cout << "cin.fail() : " << cin.fail() << "\n"; // false
    cout << "cin.bad() : " << cin.bad() << "\n"; // false

}

```

```

// iostate
/*
    ios::goodbit
    ios::eofbit
    ios::failbit
    ios::badbit
*/
int main()
{
    cout << ios::goodbit; // 0 yazar
    cout << ios::eosbit; // 1 yazar
    cout << ios::failbit; // 2 yazar
    cout << ios::badbit; // 4 yazar
}

```

```

void print_state(const ios& s)
{
    const auto stream_state = s.rdstate();
    if (stream_state == 0)
    {
        cout << "stream is ok not bit set\n";
    }

    if (stream_state & ios::failbit)
    {
        cout << "failbit set\n";
    }
    if (stream_state & ios::badbit)
    {
        cout << "badbit set\n";
    }

    if (stream_state & ios::eosbit)
    {
        cout << "eosbit set\n";
    }
}

int main()
{
    int x ;
    cin > x;
    print_state(cin); // 12 ok , ali --failbad set, ctrl z eosbit set
    // eğer state hata durumdaysa yazmaya devam etmez
    cin.clear();
    print_state(cin); // hata flaglari sonlandirir ve good hale getirir

    if (cin) // cin.good()
    {
        std::cout << "ok\n";
    }
    if (!cin) // cin.fail()
    {
        std::cout << "not ok\n";
    }
}

```

```

int main()
{
    cout << "bir tam sayi girin : ";

    int x{};

    while (!(cin >> x))
    {
        if (cin.eof())
        {
            std::cout << "ama hic giris yapmadiniz ki\n";
            cin.clear();
            cout << "tekrar deneyin\n";
        }
        else
        {
            cin.clear();
            string sline;

            getline(cin, sline);
            cout << quoted(sline) << "gecerli bir sayi degil... tekrar deneyin\n";
        }
    }
}

```

```

// cin.exceptions()
int main()
{
    auto cstate = cin.exceptions(); // good state döner ios::good

    cin.exceptions(ios::eofbit); // eofbit exception throw eder
    cin.exceptions(ios::failbit | ios::badbit); // failbit ya da badbit exception
throw eder
}

// istream_iterator
using namespace std;
int main()
{
    cout << "sayilar giriniz: ";

    // toplama yapacak girilen degerlerin
    cout << accumulate(istream_iterator<int>{cin}, istream_iterator<int>{}, 0)
<< "\n";
    cout << max_element(istream_iterator<int>{cin}, istream_iterator<int>{}) <<
"\n";

    string str{"2.3 5.6 6.7 4.5 1.2 4.4 3.9"}
    istringstream iss(str);

    vector<double> dvec{istream_iterator<double>{iss},
istream_iterator<double>{}};

}

```

2023 12 13

std::istream_iterator

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

    istream_iterator<int> iter{cin}; // streamdeki öğeleri alıyoruz
    int val{};

    val = *iter;

    cout << "val = " << val << "\n";
    ++iter;
    val = *iter;
    ++iter;
    cout << "val = " << val " \n";
}
```

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

    istringstream iss{"burak kose damla kubat furkan mert melike kaptan"};
    // begin end
    copy(istream_iterator<string>{iss}, istream_iterator<string>{},
        ostream_iterator<string>{cout, "\n"});

    istringstream iss1{"98 123 43 523 123 54 64 12 56 64 23"};
    auto sum = accumulate(istream_iterator<int>{iss1}, {}, 0);
    auto max = *max_element(istream_iterator<int>{iss1}, {}, 0);

}
```

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

    ostream os_hex{ cout.rdbuf() };
    ostream os_oct{ cout.rdbuf() };

    os_hex << hex << uppercase << showbase;
    os_oct << oct << showbase;

    Irand myrand{34523, 123412};

    for (int i = 0; i < 10; ++i)
    {
        auto val = myrand();
        cout << val << "\n"; // decimal
        os_hex << val << "\n"; // hex
        os_oct << val << "\n"; // octal
    }
}
```

Dosya İşlemleri

```
/*
    std::ofstream    yazma işlemleri için
    std::ifstream    okuma işlemleri için
    std::fstream     yazma ve okuma işlemleri için

    #include <fstream>
*/
/*
    ifstream ctor ve open fonksiyonu parametre olarak:
        dosyanın ismi
        açış modulus
            ios::in okuma
            ios::out yazma
            ios::app  append / sona yazma
            ios::trunc truncate
            ios::ate   at end
            ios::binary
*/

```

```
#include <fstream>
int main()
{
    using namespace std;

    ofstream ofs{"emre.txt", ios::trunc | ios::out}; // default ofstream
    ifstream ifs{"emre.txt", ios::in};
}
```

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

    ofstream ofs{"kutay.txt"};

    if (!ofs) // !ofs.good() ya da !ofs.fail()
    {
        cerr << "cannot open file\n";
        return 1;
    }

    std::cout << "file opened successfully";
}
```

```

int main()
{
    using namespace std;

    ifstream ifs;
    boolalpha(cout);
    // is_open streamin state kontrol etmez stream nesnesiyle ilişkilenmiş açık
bir dosya var mı kontrol eder
    cout << "ifs.is_open() = " << ifs.is_open() << "\n"; // false
    ifs.open("main.cpp");
    cout << "ifs.is_open() = " << ifs.is_open() << "\n"; // true
    ifs.close();
    cout << "ifs.is_open() = " << ifs.is_open() << "\n"; // false

}

```

```

int main()
{
    using namespace std;

    ofstream ofs{ "kutay.txt" };

    if (!ofs)
    {
        cerr << "dosya olusturulamadi\n";
        return 1;
    }

    for (int i = 0; i < 100; ++i)
    {
        ofs << i << " ";
    }

    // ofstream dtor dosyayı kapatır.
    // eğer stream good state ise dosyayı kapatıp başka bir dosyada işlem
yapabiliriz
}

```

```

// dosyaya yazma işlemleri
#include <format> // cpp 20
int main()
{
    using namespace std;
    ofstream ofs{ "emre.txt" };
    if (ofs.fail())
    {
        cerr << "dosya olusturulamadi\n";
    }
    Irand myrand {0, 300'000};
    ofs << left;
    for (int i = 0; i < 10'000; ++i)
    {
        ofs << format("{:<12} {:<16} {:<20} {}\n", myrand(), rname(), rfname(),
rtown());
    }
}

```

```

// bir vektörü dosyaya yazma yöntemleri:
int main()
{
    using namespace std;

    vector<string> svec;
    rfill(svec, 10'000, rname);

    ofstream ofs { "emre.txt" };
    if (!ofs)
    {
        cerr << "dosya olusturulamadi\n";
    }
// 1. yöntem
for (const auto& name : svec)
    ofs << name << "\n";
// 2. yöntem
copy(svec.begin(), svec.end(), ostream_iterator<string>(ofs, "\n"));

copy_if(svec.begin(), svec.end(), ostream_iterator<string>(ofs, "\n"), []
(string& s)
{
    return s.length() == 5 && s.front() = 'a';
});

generate_n(ostream_iterator<string>(ofs , "\n"), 1000, []
{
    return rname() + ' ' + rfname();
});
}

```

```

// okuma işlemleri
int main()
{
    using namespace std;
    ifstream ifs{"primes.txt"};

    if (!ifs)
        cerr << "dosya acilamadi\n";
    int v;
// (ifs >> word).operator bool
    while(ifs >> v) // akım fail olursa false döner ve while sonlanır
    {
        cout << v << " ";
    }
}

```

```

void func(std::ifstream);
int main()
{
    ifstream ifs {"emre.txt"};

    func(ifs); // error copy ctor yok
    func(std::move(ifs)); // hata yok
}

```

```
/*
    ostream'deki sınıfları incompling type olarak kullanıcksak
    #include<iostream> dahil etmek yeterli header'a iostream daha light bir include
*/

```

```
std::ifstream open_text_file(const std::string& filename)
{
    std::ifstream ifs {filename};
    if (!ifs)
    {
        throw std::runtime_error{filename + " dosyayisi acilamıyor\n"};
    }
    return ifs;
}

std::ofstream create_text_file(const std::string& filename)
{
    std::ofstream ofs {filename};

    if (!ofs)
        throw std::runtime_error{filename + "dosyayisi olusturulamadi\n"};

    return ofs;
}

int main()
{
    try
    {
        auto ofs = create_text_file("emre.txt");
    }
    catch(const std::exception &ex) {
        std::cout << "exception caught: " << ex.what() << "\n";
    }

    try
    {
        auto ifs = open_text_file("emre.txt");
    }
    catch(const std::exception &ex) {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
```

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

    int ival = 2345;
    string name{"emrebahtiyar"};

    auto filename = (ostringstream{} << name << "_" << ival << ".txt").str();

    ofstream ofs{"out.txt"} << "bugun hava cok soguk";
}
```

```
// byte byte dosya okuma
int main()
{
    ifstream ifs {"main.cpp"};

    if (!ifs)
        cerr << "dosya acilamadi\n";

    int c;
    while ((c = ifs.get()) != EOF)
    {
        cout << static_cast<char>(c); // cout.put((char)c);
    }
}
```

```
// dosyayı bir defa da okuma
int main()
{
    auto ifs = open_text_file("emre.txt");
    cout << ifs.rdbuf();
}
```

```
// getline
int main()
{
    auto ifs = open_text_file("emre.txt");
    string sline;
    getline(ifs, sline); // ifs döndürür

    cout << "[" << sline << "]";

    while (getline(ifs, sline)) // getline (ifs, sline, "\n");
    {
        cout << sline;
    }
}
```

```
// dosyada veri okuyup, vector'e atıp, sıralama
int main()
{
    vector<string> linevec;
    linevec.reserve(10'000);

    auto ifs = open_text_file("kutay.txt");
    string sline;

    while (getline(ifs, sline))
    {
        linevec.push_back(move(sline));
    }

    sort(linevec.begin(), linevec.end());
    copy(linevec.begin(), linevec.end(), ostream_iterator<string>{cout, "\n"});
}
```

```
int main()
{
    auto ifs = open_text_file("kutay.txt");
    // 1
    vector<int> ivec({istream_iterator<int>{ifs}, {}});

    // 2
    vector<int> ivec1;
    ivec.reserve(100'000);
    ivec.assign(istream_iterator<int>{ifs}, {});
}
```

```
// formatsız dosyaya yazma
int main()
{
    using namespace std;

    ofstream ofs{"primes.dat", ios::binary};

    int prime_count = 0 ;
    int x = 2;

    while (prime_count < 1'000'000)
    {
        if (isprime(x))
        {
            ofs.write(reinterpret_cast<char*>(&x), sizeof(int));
            ++prime_count;
        }
        ++x;
    }
}
```

2023 12 15

Dosya İşlemleri

```
std::ifstream open_binary_file(const std::string& filename)
{
    std::ifstream ifs {filename, std::ios::binary};
    if (!ifs)
    {
        throw std::runtime_error{ filename + " dosyayisi acilamiyor\n"};
    }
    return ifs;
}

int main()
{
    using namespace std;
    constexpr size_t n = 10'000u;

    int x{};
    int prime_count{};

    auto ofs = create_binary_file("primes10000.dat");

    while (prime_count < n)
    {
        if (isprime(x))
        {
            ofs.write(reinterpret_cast<char*>(&x), sizeof(int));
            ++prime_count;
        }
        ++x;
    }

    ofs.close();
}
```

```
int main()
{
    auto ifs = open_binary_file("primes10000.dat");

    int x{};

    while (ifs.read(reinterpret_cast<char*>(&x), sizeof(x)))
    {
        cout << x << "\n";
    }

    vector<int> ivec(10'000);

    ifs.read(reinterpret_cast<char*>(ivec.data()), 10'000 * sizeof(int));
}
```

```
// bir dosyayı parçalara bölme
int main(int argc, char* argv[])
{
    using namespace std;

    if (argc != 3)
    {
        cerr << "kullanım: <dbol> <dosya ismi> <byte sayisi>\n";
        return 1;
    }

    char c{};
    int file_count{};
    int byte_count{};

    auto ifs = open_binary_file(argv[1]);

    int chunk = atoi(argv[2]);
    ofstream ofs;
    ostringstream ostr;
    ostr.fill('0');

    while (ifs.get(c))
    {
        if (!ofs.is_open())
        {
            ostr << "parca" << setw(3) << file_count + 1 << ".par";
            ofs.open(ostr.str(), ios::binary);

            if (!ofs)
            {
                cerr << ostr.str() << "dosya olusturulamadi\n";
                return 1;
            }
            ++file_count;
        }
        ofs.put(c);
        ++byte_count;
        if (byte_count % chunk == 0)
        {
            ofs.close();
        }
    }
}
```

```

int main(int argc, char **argv)
{
    using namespace std;

    if (argc != 2)
    {
        cerr << "kullanım: <dbir> <yeni dosya ismi>\n";
        return 1;
    }

    auto ofs = create_binary_file(argv[1]);

    int file_count{};

    for(;;)
    {
        ostringstream ostr;
        ostr.fill('0');
        ostr << "parca" << setw(3) << file_count + 1 << ".par";

        ifstream ifs{ ostr.str(), ios::binary };

        if (!ifs)
            break;

        char c;
        while (ifs.get(c))
        {
            ofs.put(c);
            ++byte_count;
        }
        ++file_count;
        ifs.close();

        if (remove(ostr.str().c_str()))
        {
            cerr << "dosya silinemedi\n";
            return 2;
        }
    }
}

```

```

// dosya konum göstericisi
/*
seekg okuma amaçlı konumlandırma      --ifstream
seekp yazma amaçlı konumlandırma      --ofstream

tellg okuma konumunu döndürür        --ifstream
telli yazma konumunu döndürür        --ofstream

ios::beg konumlandırma dosyanın başından yapılımak için
ios::end konumlandımanın dosyanın sonunda yapılması için
ios::cur dosya konum gösterisinin son konumdan gösterim yapmak için
(mevcut konumu döndürür)
*/

```

```

#include <sstream>
#include <iostream>
#include <iomanip>
int main()
{
    using namespace std;

    istringstream iss{"necati ergin"};

    string str;
    iss >> str;

    cout << quoted(str) << "\n"; // "necati"
    iss.seekg(0); // iss.seekg(0, ios::beg) farkı yok
    iss >> str;
    cout << quoted(str) << "\n"; // "necati"

    iss.seekg(1, ios::cur);
    iss >> str;
    cout << quoted(str) << "\n"; // "ergin"

    iss.seekg(-6, ios::cur);
    iss >> str;
    cout << quoted(str) << "\n"; // "ergin"

    iss.seekg(-6, ios::end);
    iss >> str;
    cout << quoted(str) << "\n"; // "ergin"
}

```

```

int main()
{
    using namespace std;

    ostringstream oss{"furkan mert"};

    cout << "[" << oss.str() << "]\n"; // [furkan mert]

    oss.seekp(3);
    oss.put('!');
    cout << "[" << oss.str() << "]\n"; // [fur!an mert]

    oss.seekp(0);
    oss.put('*');
    cout << "[" << oss.str() << "]\n"; // [*ur!an mert]
}

```

```

int main()
{
    using namespace std;

    auto ifs = open_binary_file("primesmillion.dat");
    int n{};

    std::cout << "kacinci asal sayi : ";
    cin >> n;

    ifs.seekg((n -1) * sizeof(int), ios::beg);

    int x;
    ifs.read(reinterpret_cast<char*>(&x), sizeof(int));

    cout << n << ".asal sayi " << x << "\n";
}

```

```

void print_file(const std::string& filename, int ntimes)
{
    auto ifs = open_text_file(filename);

    while (ntimes--)
    {
        std::cout << ifs.rdbuf();
        (void)getchar();
        ifs.seekg(0);
    }
}

int main()
{
    print_file("main.cpp", 5); // 5 kez dosyayı yazacak
}

```

```

// gcount()
int main()
{
    int *p = new int[20000];
    auto ifs = open_binary_file("primes10000.dat");

    ifs.read(reinterpret_cast<char*>(p), sizeof(int) * 20000);
    // 20klik int değeri dolduramadı o yüzden stream hata durumunda oldu
    if (ifs)
    {
        std::cout << "stream hata durumunda degil\n";
    }
    else
    {
        std::cout << "stream hata durumunda\n";
    }

    // okunan byte sayısı 40'000
    std::cout << "okunan byte sayısı : " << ifs.gcount() << "\n";

    delete p;
}

```

2023 12 20

Tie

```
// tie
#include <iostream>
int main()
{
    using namespace std;
    cin.tie(); // ostream* döner
    if (auto osptr = cin.tie())
    {
        // cin cout nesnesi tie edilmiş yani bağlanmış
        cout << "cin is tied. the address of the stream cin tied to is : " <<
osptr << "\n";
        cout << "&cout = " << &cout << "\n";
    }
}
```

lambda ifadeleri

```
// Lambda init capture
int main()
{
    using namespace std;

    int x = 46;

    // derleyici a'nın değerine x ifadesinin değerine eşitler
    auto fn = [a = x]() // a closure type'taki veri elemanına verdiğim isim
    {
        return a * 5;
    };
    // genel kullanım böyle
    auto fn = [x = x + 5]()
    {
        return x;
    };
}
```

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

    auto uptr = make_unique<string>("necati ergin");

    auto f = [&uptr]()
    {
        cout << *uptr << "\n";
    };

    f();

    cout << (uptr ? "dolu" : "bos") << "\n"; // dolu çünkü adresini gönderdik

    // Lambda ifadesine biz taşıma yapmak istersek eğer:
    auto f = [uptr = move(uptr)]()
    {
        cout << *uptr << "\n";
    };

    cout << (uptr ? "dolu" : "bos") << "\n"; // boş çünkü taşıdıktı.
}
```

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

    int x = 5;

    auto fn = [&x = x] {++x;};
    fn();
    cout << "x = " << x << "\n"; // x = 6

    fn();
    fn();
    cout << "x = " << x << "\n"; // x = 8
}
```

```
int main()
{
    int a[10]{};

    auto fn = [a] {
        std::cout << typeid(a).name() << "\n";
    };
    fn(); // int const [10]

    auto fn1 = [a = a] {
        std::cout << typeid(a).name() << "\n";
    };
    fn(); // int *
}
```

```

// this lambda capture
class Nec
{
public:
    void foo()
    {
        auto fn = [] (int a) {return a * a;};
        auto fn1 = [x] (int a) {return a * x;};

        // copy capture ama bir pointer olduğu için reference etmem ile fark
yok
        auto fn2 = [this] (int a) {return a * (mx + my);};
        auto fn2 = [&] (int a) {return a * (mx + my);};
    }
private:
    int mx, my;
}

```

std::ratio

```

// std::ratio
template <int NUM, int DEN = 1>
class Ratio{
public:
    constexpr static int num = NUM;
    constexpr static int den = DEN;
};

#include <ratio>
int main()
{
    using namespace std;
    // 2 / 5;
    ratio<2, 5>::num; // 2

    ratio<5, 45>::num; // 1
    ratio<5, 45>::den // 9

    ratio<12, -18>::num // -2
    ratio<12, -18>::den // 3

    cout << typeid(ratio<2, 5>::type).name() << "\n"; // struct std::ratio<2,5>
}

```

```
// ratio meta fonksiyonları
int main()
{
    using namespace std;

    using rt = ratio_add<ratio<1, 3>, ratio<2, 5>>::type; // 11 / 15

    rt::num; // 11
    rt::den // 15

    ratio_multiply<ratio<2, 3>, ratio<3, 2>>
    ratio_equal<ratio<1, 3>, ratio<2, 6>>::value // true
    ratio_less<ratio<87123, 98134>, ratio<87981, 99134>>::value // false

    milli::num // 1
    milli::den // 1000
}
```

std::chrono

```
// duration türü (sureyi ifade eden bir tür)
#include <chrono>
int main()
{
    std::chrono::duration

    namespace chr = std::chrono; // namespace alias
    chr::duration;
}
```

```
using namespace std;
using namespace chrono;

using HalfDay = std::chrono::duration<int, std::ratio<12 * 60 * 60>>;

int main()
{
    // duration default açılım
    duration<int, ratio<1, 1>>;
    duration<int, ratio<1>>;
    duration<int>;

    duration<int, ratio<1, 2>> // yarım saniyeyi belirtir (1, 2)
    duration<int, milli> // milli saniye belirtir

    duration<int, ratio<3600>> // saatleri
}
```

```
int main()
{
    duration<int> x = 12; // copy init syntax hatalı

    duration<int> x(12) // direct init
    duration<int> x{12};
}
```

2023 12 22

```
// std::chrono::duration
using namespace std;
using namespace std::chrono;

// count()
int main()
{
    milliseconds ms1 { 321313 };

    auto val = ms1.count(); // 321313
}
```

```
// duration fonksiyonlar
int main()
{
    // nanoseconds
    auto dr = milliseconds{456} + nanoseconds{6'423'123} + seconds { 2 };
    cout << dr.count();

    milliseconds ms {2131};
    ms = 45; // örtülü dönüşüm syntax hatalı
    int ival = ms; // syntax hatalı
}
```

```
int main()
{
    milliseconds ms {3123};
    microseconds us {4'434'543};

    us = ms; // geçerli
    ms = us; // geçersiz (daha ince türden daha kaba türe dönüşüm syntax hatalı
olur)
}
```

```
using halfsec = duration<double, ratio<1, 2>>;
int main()
{
    halfsec hs = ms; // geçerli
}
```

```
int main()
{
    cout << milliseconds{ 345 } << "\n"; // 345ms --C++20 ile geldi
}
```

UDL (user-defined literals)

```
// operator""ms(823) --823ms operator overloading
int main()
{
    auto dur = 345ms;
    constexpr auto dur = 345ms;
    constexpr auto dur1 = operator""ms(345);

    auto val = 567ms + 3457us + 65423ns;
}
```

```
int main()
{
    "mustafa"s // string sınıfından
    operator""s("alican", 6); // "alican"s
}
```

```
operator""_x(Long Long double);
operator""_y(unsigned Long long);
operator""_z(char);
operator""_k(const char*, size_t);

cooked 569s 569'u doğrudan operator fonksiyonuna gönderiyorum
uncooked 21312x bir yazı olarak gönderiyorsak null karakter görene kadar
okuyosak
```

```
// UDL oluşturma (namespace almak daha mantıklı)
constexpr double operator""_m(long double val)
{
    return static_cast<double>(val);
}

constexpr double operator""_cm(long double val)
{
    return static_cast<double>(val / 100);
}

constexpr double operator""_mm(long double val)
{
    return static_cast<double>(val / 1000);
}

constexpr double operator""_km(long double val)
{
    return static_cast<double>(val * 1000);
}

int main()
{
    constexpr auto x = 4.5_m // double
    constexpr double y = operator""_m(4.5657);

    constexpr auto distance = 3.5_m + 876.35_cm + 1234.12_mm + 0.0812_km;

    cout << distance << " metre\n";
}
```

```
constexpr std::size_t operator""_KB(unsigned long long size)
{
    return static_cast<std::size_t>(size * 1'024);
}
constexpr std::size_t operator""_MB(unsigned long long size)
{
    return static_cast<std::size_t>(size * 1'024 * 1'024);
}
constexpr std::size_t operator""_GB(unsigned long long size)
{
    return static_cast<std::size_t>(size * 1'024 * 1'024 * 1'024);
}

int main()
{
    std::array<char, 12_KB> buf1{};
    std::array<char, 1_MB> buf2{};
}
```

```
constexpr int operator""_i(char c)
{
    return c;
}
int main()
{
    using namespace std;

    cout << 'A' << "\n"; // A
    cout << 'A'_i << "\n"; // 65
}
```

```
unsigned constexpr int operator""_b(const char* p)
{
    using result{};
    while (*p)
    {
        char digit = *p;

        if (digit != '1' && digit != '0')
        {
            throw std::runtime_error{"invalid ch : "s + digit}i
        }

        result = result * 2 (digit - '0');
        ++p;
    }

    return result;
}
int main()
{
    using namespace std;

    auto val = 1010101_b;
    cout << val << "\n";
}
```

```

class Kilogram
{
public:
    class prevent_usage{};
    Kilogram(prevent_usage, double val) : mweighth{ val } {}
    friend constexpr Kilogram operator+(const Kilogram& x, const Kilogram& y)
    {
        return Kilogram{Kilogram::prevent_usage{}, x.mweighth + y.mweighth};
    }
private:
    double mweighth;
};

constexpr Kilogram operator""_kg(long double x)
{
    return Kilogram{ Kilogram::prevent_usage{}, static_cast<double>(x) };
}

constexpr Kilogram operator""_gr(long double x)
{
    return Kilogram{ Kilogram::prevent_usage{}, static_cast<double>(x / 1000) };
}

constexpr Kilogram operator""_mg(long double x)
{
    return Kilogram{ Kilogram::prevent_usage{}, static_cast<double>(x / 1000 / 1000) };
}

int main()
{
    constexpr Kilogram kg1 = 5.6_kg;
    constexpr auto weight = 4.5_kg + 1234.87_gr + 345'434.55_mg;
}

```

```

// std::chrono (duration_cast)
int main()
{
    using namespace std::chrono;

    auto ms == 75624ms;
    seconds s = ms; // syntax hatası

    seconds s(ms.count() / 1000); // geçerli

    duration_cast<seconds>(ms); // veri kaybı olur
    cout << sec.count() << "\n"; // 75
}

```

```

int main()
{
    long long val;
    std::cout << "milisaniye olarak degeri girin : ";
    cin >> val;

    milliseconds ms{val};

    hours hrs = duration_cast<hours>(ms);
    minutes mins = duration_cast<minutes>(ms % 1h);
    seconds sec = duration_cast<seconds>(ms % 1min);

    if (hrs.count())
        cout << hrs.count() << " saat";
    if (mins.count())
        cout << mins.count() << " dakika";
    if (sec.count())
        cout << sec.count() << " mili saniye"\n";
}

```

```

// yuvarlama işlemleri
int main()
{
    auto ms = 923'879ms;

    cout << duration_cast<seconds>(ms) << "\n"; // 923
    cout << floor<seconds>(ms) << "\n"; // 923
    cout << ceil<seconds>(ms) << "\n"; // 924
    cout << round<seconds>(ms) << "\n"; // 924
}

```

```

template <typename Rep, typename Period>
std::ostream& operator<<(std::ostream& os, const std::chrono::duration<Rep,
Period>& dur)
{
    return os << dur.count() << " * " << Period::num << " / " << Period::den <<
}

using tensc = std::chrono::duration<int, std::ratio<10>>
int main()
{
    using namespace std;
    using namespace std::chrono;

    cout << 3456ms; // 3456 * ( 1 / 1000)
    cout << tensc {6512} << "\n";
}

```

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

    system_clock::time_point // time_point<system_clock, system_clock::duration>

    auto tp1 = system_clock::now();

    auto tp2 = system_clock::now();
    auto diff = tp2 - tp1;
}
```

```
// süre ölçümü
std::vector<size_t> get_sorted_vector(std::size_t n)
{
    std::vector<size_t> vec(n);

    std::generate_n(vec.begin(), n, rand);
    std::sort(vec.begin(), vec.end());

    return vec;
}
```

```
int main()
{
    using namespace std;
    using namespace std::chrono;
    auto tp_start = steady_clock::now();
    auto vec = get_sorted_vector(450'000);

    auto tp_end = steady_clock::now();

    cout << duration_cast<milliseconds>(tp_end - tp_start).count() << " milisaniye\n";
    cout << duration_cast<double>(tp_end - tp_start).count() << " saniye\n";
    cout << duration_cast<double, micro>(tp_end - tp_start).count() << " micro saniye\n";

    cout << "vec.size()" << vec.size() << "\n";
}
```

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

    auto tp = system_clock::now();

    time t ct_time == system_clock::to_time_t(tp);
    cout << ctime(&ctime) << "\n";
}
```

2023 12 25

```
//system_clock

#include <chrono>
#include <ctime>
int main()
{
    using namespace std;
    using namespace std::chrono;

    days oneday{ 1 };

    auto tp_today = system_clock::now();

    auto tp_tomorrow = tp_today + oneday;

    time_t t1 = system_clock::to_time_t(tp_today);
    time_t t2 = system_clock::to_time_t(tp_tomorrow);

    cout << ctime(&t1) << "\n"; // şu an
    cout << ctime(&t2) << "\n"; // bir gün sonrası
}
```

vocabulary type (C++ 17)

```
std::optional --template class
    bir değerin olması ya da olmaması problem domaininde söz konusu ise
    kullanılır

    std::variant --template class
        n tane farklı türden herhangi bir değer tutabilir
    std::any --class
        void* türüne alternatif
```

std::optional

```
// std::optional
#include <optional> // wrapper
#include <string>

int main()
{
    using namespace std;

    optional<string> o1;
    optional<int> o2;
    optional<Date> o3;
```

```

#include <optional> // wrapper
#include <string>

int main()
{
    using namespace std;
    optional<int> x{ 678 };

    if (x) // x.operator bool()
        std::cout << "dolu bir degere sahip\n";
    else
        std::cout << "bos, bir degere sahip degil\n";

    optional<int> x1{ nullopt }; // constexpr bir deгiшken bos halde ayaга gelir

    optional x2 = 10; // CTAD optional<int>
    optional x3 = "damla"; // optional<const char*>
    optional x3 = "damla"s; // optional<std::string>
}

```

```

int main()
{
    optional<int> x1; // bos
    optional<int> x2{}; // bos
    optional<int> x3{nullopt}; // bos
    optional<int> x4(); // fonksiyon bildirimi bu

    if (x1.has_value()) // false
}

```

```

int main()
{
    optional x(4345);

    cout << *x << "\n"; // 4345
    *x = 123;
    cout << *x << "\n"; //123

    optional<string> osx{ "naci cemal" };
    cout << *osx << "\n";
    osx->length();

    *osx+= " dagdelen";
    // егер optional bos ise * -> operatorlarini kullanmak tanimsız davranisitir.
    cout << osx.value() << "\n";
}

```

```
// value_or()
int main()
{
    // eğer boşsa değeri geçer
    optional<string> os{ "furkan mert" };
    cout << os.value_or("noname"); << "\n"; // furkan mert
    os = {};
    cout << os.value_or("noname"); << "\n"; // noname
}
```

```
//std::optional Kullanım
void print_e_mail(const std::optional<std::string>& op)
{
    std::cout << "e-posta adresi: " << op.value_or("yok") << "\n";
}

int main()
{
    print_e_mail("necati ergin");
    print_e_mail(std::nullopt);
}
```

```
class Person
{
private:
    //...
    std::optional<std::string> m_middle_name;
};
```

```
class Record {
public:
    Record(const std::string& name, std::optional<string> nick,
std::optional<int> age) :
        m_name(name), m_nick(nick), m_age(age) {}

private:
    std::string m_name;
    std::optional<std::string> m_nick;
    std::optional<int> m_age;

};

int main()
{
    Record r1{"emre bahtiyar", nullopt, nullopt};
}
```

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

```
class A
{
public:
    A() {};
    A(int) {};
};

int main() {
    using namespace std;
    optional<A> ox;

    ox.emplace(); // default ctor
    // yeni bir emplace'te önce dtor eder sonra yenisini oluşturur
    ox.emplace(2); // int ctor
    // sonrasında sarmaladığı nesneyi sonlandırır

    ox = 5; // Legal implicitly
}
```

```
// in_place
#include "date.h"
int main()
{
    using namespace std;

    //optional<Date> ox(4, 5 1987); // hatalı
    optional<Date> ox(in_place, 4, 5, 1987) // gecerli
}
```

```
int main()
{
    using namespace std;
    optional<complex<double>> ox{in_place, 2.3, 6. 7};
}
```

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

    optional<string> o1(in_place, 20, 'X');
    auto o2 = make_optional<string>(10, 'Y');
}
```

```

// optional karşılaştırma
int main()
{
    using namespace std;

    optional x = 5;
    optional y = 125;

    cout << (x == y) << "\n";
    cout << (x != y) << "\n";
    cout << (x < y) << "\n";

    //bos olursa her zaman küçük
    optional<int> z;
    cout << x < z << "\n"; // true
    cout << x < y << "\n"; // true
}

```

std::variant

```

// std::variant
#include<variant>
// union ile benzer
union Nec {
    int x;
    double dval;
};

int main()
{
    // x'in türü ilk tanımlanan yani int olacak
    variant<int, double, long, char> x; // x'in türü bunlardan biri olacak
    cout << x.index() << "\n"; // 0

    variant<int, double, long, char> x(4.5);
    cout << x.index() << "\n"; // 1 yani double
}

```

```

int main()
{
    variant<int, double, long, char> x;
    cout << x.index() << "\n"; // 0

    x = 3.4;
    cout << x.index() << "\n"; // 1

    x = 'a';
    cout << x.index() << "\n"; // 3
}

```

```
//holds_alternative
int main()
{
    variant<string, double, int> x{57};

    cout << "x.index() = " << x.index() << "\n";

    if (holds_alternative<int>(x))
    {
        cout << "int tutuyor" << "\n";
    }
}
```

```
// get<>
int main()
{
    variant<int, double, long> x;
    // value init edilir yani garabe value değil
    cout << get<0>(x) << "\n"; // 0
    x = 3.7;
    cout << get<double>(x) << "\n"; //

    get<3>(x); // geçersiz indeks syntax hatalı
}
```

```
int main()
{
    variant<int, double, long> x{3.6};

    try
    {
        auto val = get<2>(ex) //
    }
    catch (const std::exception& ex)
    {
        // bad variant acces throw eder
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
```

```
// in_place_index ve in_place_type
int main()
{
    variant<int, Date, string> x{in_place_index<1>, 12, 5, 1987};
    variant<int, Date, string> x1{in_place_type<Date>, 2, 5 1987};

    cout << get<1>(x) << "\n";

    x.emplace<string>(10, 'a');
    cout << get<2>(x) << "\n";
}
```

```
// monostate
int main()
{
    using namespace std;
    // hiçbir değer tutmuyor monostate tutuyor boş gibi
    variant<monostate, int, double, long> x;
}
```

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

    variant<int, double, long> x(2.3);
    // eğer int'se adres döndürür değilse nullptr
    if (auto *ptr = get_if<int>(&x))
    {
        cout << *ptr << "\n";
    }
}
```

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 = {};
}
```

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

    variant<int, double> v1(29), v2(1.2), v3(11);
    // türler farklısa 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
}
```

```
// 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);
}
```

```
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 yakalandı : " << vx.what() << "\n";
        // garabe value döner çünkü hayatı 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
    }
}
```

```
// 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');
}
```

```
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";
    }
}
```

```
//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";
    }
}
```

```

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>;
}

```

```

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

        }
    }
};

```

multi-lambda

```

// 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;
};

```

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

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

};

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

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

```
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();
}
```

```

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{}};
}

```

```

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){}
}

```

```

// 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
}

```

```

template <typename ...Args>
struct Overload : Args...
{
    // ambiguity kalkti
    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
}

```

```

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(y)).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
}

```

```

// 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);
}

```

```

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

2023 12 29

std::any

```
// std::any
int main()
{
    using namespace std;
    any a(45);
    cout << a.type().name() << '\n';

    if (a.type() == typeid(int))
    {
        auto ai = any_cast<int>(a); // bad_any_cast throw eder eğer int değilse
    }

    int &r = any_cast<int&>(a);
    ++r;
    cout << any_cast<int>(a); // 46
}
```

```
// reset()
int main()
{
    using namespace std;
    any a = "emre can suster";
    cout << (a.has_value() ? "dolu" : "bos") << "\n"; // dolu
    a.reset();
    cout << (a.has_value() ? "dolu" : "bos") << "\n"; // bos
}
```

```
void *operator new(std::size_t n)
{
    std::cout << "operator new called... n = " << n << "\n";
    return std::malloc(n);
}
void operator delete(void* vp)
{
    if (vp)
        std::free(vp);
}
struct Nec {
    unsigned char buffer[1024]{};
};
int main()
{
    using namespace std;
    // size derleyiciden derleyiciye değişir
    std::cout << "sizeof(any) = " << sizeof(any) << "\n"; // 40
    std::cout << "sizeof(type_info) = " << sizeof(type_info) << "\n"; // 12
    any ax = 12;
    ax = 4.5;
    ax = bitset<128>{};
    ax = Nec{}; // operator new çağrıılır
}
```

```

int main()
{
    using namespace std;

    int arr[100]{};

    any a = arr;

    std::cout << "sizeof(arr) = " << sizeof(arr) << "\n"; // 400

    if (a.type() == typeid(int*))
        std::cout << "pointer tutuyor\n"; // pointer tutuyor çıkar
    else if(a.type() == typeid(int[100]))
        std::cout << "dizi tutuyor\n";
}

```

```

// emplace ve make_any
int main()
{
    using namespace std;

    any a;
    a = 4567;
    a.emplace<Date>(3, 5, 1987);
    make_any<Date>(2, 5, 1987);
}

```

```

// Kullanım Senaryosu
using tvp = std::pair<std::string, std::any>
int main()
{
    using namespace std;
    vector<tvp> vec;

    vec.emplace_back("name", "tamer dundar");
    vec.emplace_back("birth_year", 1994);
    vec.emplace_back("month", 11);
    vec.emplace_back("price", 59.97);
    vec.emplace_back("town", "mugla");
    vec.emplace_back("gender", "male");

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

```

```
// in_place_type
int main()
{
    using namespace std;
    any ax{in_place_type<Date>, 1, 1, 2024};
}
```

Std::random

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

    default_random_engine // std::mt19937
    cout << (typeid(mt19937) == typeid(default_random_engine)) << "\n";

    mt19937_64 // 64 bitlik üretir

    cout << mt19937::default_seed << "\n";
}
```

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

    mt19937 eng{123};
    for (int i = 0; i < 10000 - 1; ++i)
        (void)eng(); // bütün derleyicilerden aynı olmak zorunda

    cout << "min = " << mt19937::min() << "\n";
    cout << "max = " << mt19937::max() << "\n";
}
```

```
int main()
{
    using namespace std;
    // kopyalama yapmak çok maliyetli
    std::cout << "sizeof(mt19937) = " << sizeof(mt19937) << "\n"; // 5000
}
```

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

    mt19937 eng();
    //// eng böyle kullanırsak kopyalama olur
    generate_n(vec.begin(), 10'000, eng);
    // böyle kullanınmamla
    generate_n(vec.begin(), 10'000, ref(eng));
}
```

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

    mt19937 e1(32);
    mt19937 e2(32);
    mt19937 e3(56);

    cout << (e1 == e2) << "\n"; // true
    cout << (e1 == e3) << "\n"; // false

    // rastgele sayı üretince state değişir.
    auto val = e1();
    ++val;
    cout << (e1 == e2) << "\n"; // false
    val = e2();
    cout << (e1 == e2) << "\n"; // true
}
```

```
int main()
{
    mt19937 eng{123};

    for (int i = 0; i < 10; ++i) {
        (void)eng();
    }

    stringstream ss;
    ss << eng;

    mt19937 eng2;
    ss >> eng2; // state kopyalamış olduk
}
```

```
int main()
{
    using namespace std;
    // 10 - 14 arasında değerler veririr uniform olarak
    uniform_int_distribution<int> dist{ 10, 14 };

    mt19937 eng;

    for (int i = 0; i < 100; ++i){
        cout << dist(eng) << ' ';
    }
}
```

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

    map<int, int> cmap;
    uniform_int_distribution dis{0 , 20};

    for (int i = 0; i < 20'000'000; ++i)
    {
        ++cmap[dist(eng)]; // rastgele sayılar birbirine yakın sayıda üretilir
    }
}
```

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

    mt19937 eng;
    uniform_real_distribution dist{ 5.6, 6.1};

    for (int i = 0; i < 100; ++i)
        cout << dist(eng) << "\n";
}
```

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

    mt19937 eng;

    normal_distribution<double> dist(50, 5.);
}
```

2024 01 03

std::random

```
// std::bernoulli_distribution
#include <random>
#include <iostream>
#include <chrono>
int main()
{
    using namespace std;
    // exe her çalıştırın da farklı bir seed(tohum) çalışacak
    mt19937 eng{std::chrono::steady_clock::now().time_since_epoch()};
    mt19937 eng1{ random_device{}() };
    bernoulli_distribution dist{0.81};

    std::size_t n = 1'000'000u;
    int cnt{};

    for (size_t i{}; i < n; ++i)
    {
        if (dist(eng))
        {
            ++cnt;
        }
    }

    cout << static_cast<double>(cnt) / n << "\n";
}
```

```
// her çağrıldığında aynı engine çağrılacak
std::mt19937& engine()
{
    static std::mt19937 eng{ std::random_device{}() };
    return eng;
}

int main()
{
    using namespace std;
    uniform_int_distribution dist{0 , 99};

    cout << dist(engine()) << "\n";
}
```

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

    uniform_int_distribution dist1{ 0, 99 };

    auto prm = dist1.param();
    //dist1 parametrelerini dist2'ye geçirdik
    uniform_int_distribution dist2{dist1.param()}; //
}
```

```

/// algoritmalarда random
int main()
{
    using namespace std;
    vector<int> ivec(100'000);

    mt19937 eng;
    uniform_int_distribution dist{ 4671, 8812};
    generate(ivec.begin(), ivec.end(), [&eng, &dist]{ return dist(eng);});
}

```

Concurrency

```

#include <thread>
int main()
{
    using namespace std;
    // move only type
    thread th1[] {cout << "emre bahtiyar\n"}; // ctor'u callable alır
    thread th1[] {cout << "emre bahtiyar\n"}; // ctor'u callable alır
    thread th2[] {cout << "emre bahtiyar\n"}; // ctor'u callable alır
    thread th3[] {cout << "emre bahtiyar\n"}; // ctor'u callable alır
}

```

```

void func(int a, int b, int c)
{
}

int main()
{
    using namespace std;

    // eğer join() ya da detach() fonksiyonu çağrılmaz terminate fonksiyonu
    // çağrıılır
    thread tx{ func, 10, 20, 30 };

    // fonksiyon bitene kadar bu nokta beklesin anlamına gelir
    tx.join();
    // fonksiyon ile ana thread ayrıılır tx kendi biter
    tx.detach();
}

```

```

/*
Bir thread nesnesi joinable ve unjoinable durumunda olabilir,
eğer bir thread nesnesi joinable durumduysa ve dtor'u çağrılırsa
doğrudan terminate fonksiyonu çağrılır.
*/

void nec_terminate()
{
    std::cout << "ne terminate cagrildi\n";
    std::abort();
}

void func()
{
    std::cout << "ben func fonksiyonuyum\n";
}

int main()
{
    using namespace std;

    set_terminate(nec_terminate);

    {
        thread tx{func};
        tx.join(); // bunu çağrırmazsaq terminate çağrılacak
    }
}

```

```

// joinable()
int main()
{
    using namespace std;

    thread t;
    cout << t.joinable() << "\n"; // false çünkü iş yükü yok

    thread t1{[] {}};
    cout << t1.joinable() << "\n"; // true çünkü iş yükü var
    t1.join();
    cout << t1.joinable() << "\n"; // false artık join edilmiş
}

```

```

// thread taşıma
int main()
{
    thread t1{[] {}};
    thread t2{move(t1);

    cout << t1.joinable() << "\n"; // false artık taşındı.
    cout << t2.joinable() << "\n"; // true.

    t2.join();

    cout << t1.joinable() << "\n"; // false
    cout << t2.joinable() << "\n"; // false.
}

```

```

void func()
{
    std::cout << "ben func fonksiyonuyum\n";
}

int main()
{
    using namespace std;

    thread tx{ func };
    tx.join();

    tx.join(); // exception throw eder
}

```

```

// jthread --cpp 20

void func()
{
    throw std::runtime_error{ " hataaaaa" };
}

void foo()
{
    //code
}

void bar()
{
    /*
        dtor çağrıldığında jthread eğer sarmaladığı thread'in joinable ise join
        eder
    */
    jthread t{ foo };

    func(); // exception throw eder
    t.join(); // buraya girmez
}

int main()
{
    try
    {
        bar();
    }
    catch (std::exception &ex)
    {

    }
}

```

```

void func(char c)
{
    for (int i = 0; i < 1000; ++i)
    {
        std::cout.put(c);
    }
}

int main()
{
    using namespace std;

    vector<thread> tvec(26);

    for (auto& t : tvec)
    {
        t = thread{func, c++}; // taşınma semantiği
    }

    for (auto& t : tvec)
    {
        t.join();
    }
}

```

thread'e verdığımız callable exception verirse exception'ı yakalamayız

```

int foo(int x, int y)
{
    // code
    return x * y;
}

int main()
{
    using namespace std;

    thread tx{ foo, 10, 56 };
    // foo fonksiyonun geri dönüş değerini thread nesnesi aracılıkla alamayız
    tx.join();
}

```

```
// thread fabrika fonksiyonu
std::thread make_thread()
{
    std::thread tx{[]
    {
        std::cout << "emre\n";
    }};
    // otomatik ömürlü nesneler l value to x value olur
    return tx;
}

std::thread func(std::thread tx)
{
    return tx;
}

int main()
{
    using namespace std;

    auto t1 = make_thread();
    auto t2 = move(t1);
    t1 = func(move(t2));

    t1.join();
}
```

```

// threadlere verilebilecek callable'lar
class Functor
{
public:
    void operator()(int x) const{
        std::cout << x << "\n";
    }
}
void func(int x)
{
    std::cout << x << "\n";
}
struct Nec
{
    static void print(int x)
    {
        std::cout << x;
    }
    void display(int x) const
    {
        std::cout << x;
    }
}
int main()
{
    using namespace std;

    thread t1(func, 1); // fonksiyon
    thread t2([]{int x} {cout << x << "\n";}, 2); // Lambda
    thread t3(Functor{}, 3);
    thread t4(Nec::print, 4);

    Nec mynec;
    thread t5(&Nec::display, mynec, 5);
    // çıktı belli olmaz
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
}

```

```

void func(int& r)
{
    r +=200;
}

int main()
{
    using namespace std;
    int x = 67;

    thread t{ func, x}; // syntax hatalı
    cout << "x = " << x << "\n";

    thread tx{ func, ref(x)}; // syntax hatalı yok;
    cout << "x = " << x << "\n"; // 267
}

```

```
void func(std::string && r)
{
    std::cout << r.size() << "\n";
    auto x = std::move(r);
}

int main()
{
    using namespace std;

    string name{ "emre bahtiyar "};

    thread tx{ func, name};
    tx.join();

    cout << name.size() << "\n";
}
```

2024 01 05

Std::thread

```
/*
    thread t{foo , x}
        x bir dizi ise pointera dönüştürüləcək
        x bir fonksiyon ise fonksiyon adresine dönüştürüləcək.
        x const ise non-const'a dönüştürüləcək
*/
```

```
class MyClass
{
public:
    MyClass() = default;
    MyClass(const MyClass&)
    {
        std::cout << "copy ctor\n";
    }
    MyClass(MyClass&&)
    {
        std::cout << "move ctor\n";
    }
};

void foo(MyClass)
{
    /*
        -- MyClass& olsaydı syntax hatası olur çünkü fonksiyonun
        parametresine L ref değil r expr gönderiyoruz.
        R value expr L ref ile bağlanamaz

        -- MyClass&& olsaydı syntax hatası olmaz
        sadece copy ctor çağrırlır

        -- const MyClass& olsaydı syntax hatası olmaz
        const L value ref R value expr bağlanabilir
        sadece copy ctor çağrırlır
    */
}

int main()
{
    using namespace std;

    MyClass m;

    thread t{ foo, m };
    /*
        decay copy yapılacak --copy ctor çağrılacak
        fonksiyon parametresine r value veriyoruz --move ctor çağrılacak

        copy ctor
        move ctor
    */
}
```

```

t.join();

thread t1{foo, Myclass{}};
/*
    gönderdiğimiz r value olduğu için move ctor ile oluşturulur
    fonksiyona geçerken r value veriyoruz move ctor çağrılır.

    move ctor
    move ctor
*/
t1.join();

const Myclass m1;
/*
    const'luk düşer
    copy ctor
    move ctor
*/
thread t2{foo, m1};
t2.join();

Myclass m2;
thread t3{foo, ref(m)};
/*
    copy veya move ctor çağrılmaz.
*/
t3.join();
}

```

```

// get_id
void func()
{
}

int main()
{
    using namespace std;

    thread t1{ func };
    auto x = t1.get_id(); // auto --> class std::thread_id
    cout << x << "\n";
    t1.join()
}

```

```

/*
    tüm threadlerin id'leri farklı
    eğer thread joinable durumda değilse get_id 0 döner
*/

// this_thread
#include <syncstream>
void func()
{
    osyncstream{cout} << std::this_thread::get_id() << "\n";
}

```

```
}
```

```
int main()
```

```
{
```

```
    using namespace std;
```

```
    thread t1{func};
```

```
    cout << t1.get_id() << "\n";
```

```
    t1.join();
```

```
}
```

```
std::thread::id main_thread_id;
```

```
void func()
```

```
{
```

```
    if (std::this_thread::get_id() == main_thread_id)
```

```
        std::cout << "main thread\n";
```

```
    else
```

```
        std::cout << "ikincil thread\n";
```

```
}
```

```
int main()
```

```
{
```

```
    using namespace std;
```

```
    main_thread_id = std::this_thread::get_id();
```

```
    func(); // main thread
```

```
    thread t{ func };
```

```
    t.join(); // ikincil thread
```

```
}
```

```
/*
```

```
thread kütüphanesinde sonu for ve until ile biten fonksiyonlar bulunur
```

```
xxxfor      duration ister
```

```
xxxuntil    timepoint ister
```

```
*/
```

```
// sleep_for
```

```
void func()
```

```
{
```

```
    std::this_thread::sleep_for(2300ms);
```

```
    std::cout << "emre bahtiyar\n";
```

```
}
```

```
int main()
```

```
{
```

```
    using namespace std;
```

```
    thread t{func};
```

```
    t.join();
```

```
}
```

```
// native_handle()
```

```
int main()
```

```
{  
    using namespace std;  
  
    thread t{ [] {}};  
    // işletim sistemin apilerine geçicek handle verir  
    auto handle = t.native_handle();  
}
```

```
// hardware_concurrency()  
int main()  
{  
    using namespace std;  
    // kaç tane thread açabilirim  
    auto b = thread::hardware_concurrency()  
}
```

```
// thread exceptions  
void func(int x)  
{  
    // exception burda yakalanırsa catch bloğuna girer  
    if (x % 2 == 0)  
    {  
        throw std::runtime_error{ "even number error\n" };  
    }  
}  
  
int main()  
{  
    using namespace std;  
    /*  
       exception vermez doğrudan terminate fonksiyonu çağrılır.  
    */  
    try  
    {  
        thread t{ func, 12};  
        t.join();  
    }  
    catch (const std::exception &ex)  
    {  
  
        std::cout << "exception caught: " << ex.what() << "\n";  
    }  
}
```

```
/*  
 *  
 * std::exception_ptr  
 * exception sarmalar  
 * current_exception()  
 * bir exception yakalandığı zaman exception ptr nesnesi döndürür döndürür  
 * rethrow_exception()  
 * exception ptr'yi rethrow eder.  
 */
```

```

void handle_saved_exception(std::exception_ptr eptr)
{
    try
    {
        if (eptr)
        {
            std::rethrow_exception(eptr);
        }
    }
    catch (const std::out_of_range& ex)
    {
        std::cout << "hata yakalandi : " << ex.what() << "\n";
    }
}

int main()
{
    std::exception_ptr expr;
    try{
        std::string name{"emre"};
        auto c = name.at(45);
    }
    catch (...) {
        eptr = std::current_exception();
    }
}

```

```

std::exception_ptr eptr{ nullptr };
void func(int x)
{
    std::cout << "func cagrildi x = " << x << "\n";
    try {
        if (x % 2 != 0)
        {
            throw std::runtime_error{"hatali arguman " + std::to_string(x)};
        }
    }
    catch(...)
    {
        eptr = std::current_exception();
    }
    std::cout << "func islevi sona eriyor\n";
}

```

```

int main()
{
    thread tx{ func, 12};
    tx.join();
    try {
        if (eptr)
            rethrow_exception(eptr);
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
std::vector<std::exception_ptr> g_epvec;
std::mutex mtx;
void f1() {throw std::runtime_error{ "hata" };}
void f2() {throw std::invalid_argument{ "gecersiz arguman" };}
void f3() {throw std::out_of_range{ "aralik disi deger" };}

void tfunc1()
{
    try {
        f1();
    }
    catch (...) {
        std::lock_guard guard{mtx};
        g_epvec.push_back(std::current_exception());
    }
}
void tfunc2()
{
    try {
        f2();
    }
    catch (...) {
        g_epvec.push_back(std::current_exception());
    }
}
void tfunc3()
{
    try {
        f3();
    }
    catch (...) {
        g_epvec.push_back(std::current_exception());
    }
}

int main()
{
    using namespace std;

    thread t1 {tfunc1};
    thread t2 {tfunc2};
    thread t3 {tfunc3};

    t1.join();
    t2.join();
}

```

```

t3.join();

for (const auto &ex : g_epvec)
{
    try {
        rethrow_exception(ex);
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught : " << ex.what() << "\n";
    }
}
}

```

```

// make_exception_ptr()
int main()
{
    using namespace std;

    exception_ptr eptr;

    try
    {
        throw std::runtime_error{ "hata" };
    }
    catch(...)
    {
        eptr = current_exception();
    }

    // yukarıdaki yerine
    auto eptr = make_exception_ptr(std::runtime_error{ "hata" });
}

```

thread_local storage

```

// thread_local bir keyword

thread_local int x{0};

void foo(const std::string& thread_name)
{
    // her thread'in thread_local değişkeni o thread için tek
    ++x;

    osyncstream{cout} << "thread" << thread_name << " x = " << << "\n";
}

int main()
{
    jthread tx{ foo, "tx" };    // thread tx x = 1
    jthread ty{ foo, "ty" };    // thread tx x = 2
    jthread tm{ foo, "tm" };    // thread tx x = 3
    jthread tz{ foo, "tz" };    // thread tx x = 4
}

```

```

thread_local int ival = 0;

void thread_func(int *p)
{
    *p = 42;
    std::cout << "*p = " << *p << "\n";
    std::cout << "ival = " << ival << "\n";
}

int main()
{
    std::cout << "ival = " << ival << "\n"; // ival = 0

    ival = 9;
    std::cout << "ival = " << ival << "\n"; // ival = 9

    std::thread t{thread_func, &ival};
    t.join() // *p = 42, ival = 0

    std::cout << "ival = " << "\n"; // ival = 42
/*
    -thread_func'ta ival hala 0 thread_local olduğu için
    -main thread ival'nın adresini gönderdik o 9 değerini 42 yaptı
    bundan dolayı ival son olarak 42 kaldı
*/
}

```

```

// ÖRNEK
std::mutex mtx;
void foo(int id)
{
    int x = 0;           // automatic storage
    static int y= 0;     // static storage
    thread_local int z = 0; // thread_local storage

    ++x;
    ++z;
    mtx.lock();
    ++y;

    std::cout << "thread id: " << id << " x(automatic) = " << x << "\n";
    std::cout << "thread id: " << id << " y(static) = " << y << "\n";
    std::cout << "thread id: " << id << " z(thread_local) = " << z << "\n";
    mtx.unlock();
}

```

```

}

int main()
{
    using namespace std;
    jthread t1{ foo, 1};
    jthread t2{ foo, 2};
    jthread t3{ foo, 3};
    jthread t4{ foo, 4};
    /*
        static değişken 4 e kadar çıkar
        automatic ve thread_local 1 çıkar hep
    */

}

// ÖRNEK
thread_local std::string name{ "cemal" };
void func(const std::string& surname)
{
    name += surname;
    // name'Lerin adresleri farklı olur.
    std::osyncstream{ std::cout } << name << "&name = " << &name << "\n";
}

int main()
{
    const char* const pa[] = { "akkan", "toprak", "bahtiyar", "ersoy",
    "canberk" };
    vector<thread> tvec;

    for (auto p : pa)
    {
        tvec.emplace_back(func, p);
    }

    for (auto& t : tvec)
        t.join();
}

```

```

class Myclass {
public:
    Myclass(int x)
    {
        std::osyncstream{ std::cout } << "Myclass ctor x = " << x << " this = "
        << this << "\n";
    }
    ~Myclass()
    {
        std::osyncstream{ std::cout } << "Myclass dtor this = " << this << "\n";
    }
};

void func(int x)
{
    thread_local Myclass m{ x };
}

void foo(int x)
{
    std::osyncstream{ std::cout } << "foo(int) cagrildi x = " << x << "\n";
    func(x);
}

```

```
    std::iosyncstream{ std::cout } << "foo(int) sona eriyor x = " << x << "\n";
}
int main()
{
    using namespace std;
{
    jthread t1{ foo, 1};
    jthread t2{ foo, 1};
    jthread t3{ foo, 1};
    jthread t4{ foo, 1};
}

    std::cout << "main devam ediyor\n";
}
// ÖRNEK

/*
    thread_local değişken yerel değişken de olabilir. Böyle bir değişkenin hayatı
    kapsamı sonununda bitmez. Threadin yürütülmesi sonunda hayatı sona erer.
*/
thread_local std::mt19937 eng{ 4542345u};

void foo()
{
    std::uniform_int_distribution dist{ 10, 99};

    for (int i = 0; i< 10; ++i)
    {
        std::cout << dist(eng) << ' ';
    }
}

int main()
{
    std::thread t1{ foo };

    t1.join();

    std::cout << "\n";
    std::thread t2{ foo };
    t2.join();
}
```

```
//ÖRNEK
// MyClass nesnesi thread_local olduğu için thread'in sonuna kadar yaşar
class MyClass
{
public:
    MyClass()
    {
        std::osyncstream{ std::cout } << "MyClass ctor this : " << this <<
"\n";
    }

    ~MyClass()
    {
        std::osyncstream{ std::cout } << "MyClass dtor this : " << this <<
"\n";
    }
}

void foo()
{
    std::ostringstream{ std::cout } << "foo called\n";
    thread_local MyClass m;
    std::ostringstream{ std::cout } << "foo ends\n";
}

void bar()
{
    using namespace std::chrono_literals;

    std::ostringstream{ std::cout } << "bar called\n";
    foo();
    std::this_thread::sleep_for(3s);
    std::ostringstream{ std::cout } << "bar ends\n";
}

int main()
{
    std::thread t{ bar };
    t.join();
}
```

```
/*
 data race: birden fazla thread paylaşımılı değişkeni kullanacak ve
 en az biri yazma amaçlı kullanacak. data race varsa tanımsız davranıştır

 race condition: paylaşımılı değişkenin kullanılması ama
 herhangi bir tanımsız davranış zorunda değil

*/

// data race
int gcount = 0;
void foo()
{
    for (int i = 0; i < 100'000; ++i)
        ++gcount;
}

void bar()
{
    for (int i = 0; i < 100'000 << ++i)
        --gcount;
}

int main()
{
    /*
        thread kullanmasak önce gcount 100'000 olacak sonra 0 olacak
        data race olmıcak bu durumda.

        thread kullanınca gcount son değeri belirsiz olacak bu data race
        tanımsız davranış
    */
    using namespace std;
    {
        jthread t1{ foo };
        jthread t2{ bar };
    }
    cout << "gcount = " << gcount << "\n";
}
```

```
// mutual exclusion
/*
    acquire
    critical section
    release
*/
// std::mutex
int gcount = 0;

std::mutex mtx;

void func()
{
    mtx.lock(); // kilit boştaysa buraya gelen thread kiliti edenir. Diğer
thread'ler giremez
    ++gcount;
    mtx.unlock(); // kilit serbest bırakılır
}

int main()
{
    thread t1{ func };
    thread t2{ func };
    thread t3{ func };
    thread t4{ func };
}
```

2024 01 08

Std::thread

```
#include <thread>
void foo(int &x)
{
}

using namespace std;

int main()
{
    int x{ 35 };

    jthread t1{ foo , ref(x)};
    jthread t2{ foo , ref(x)};
    jthread t3{ foo , ref(x)};

}
```

```
/*
    Birden fazla thread'in ortaklaşa olarak kullandığı paylaşımı bir değişken
varsayı
tanımsız davranış olmaması için:

a) paylaşımı bir değişken const bir değişken olacak
b) thread'ler okuma amaçlı paylaşımı değişkeni kullanacak

sequenced-before relationship
eğer tek bir thread söz konusuya:
    x = 5;
    y = x;

    kodunda, y = 5 olmasını garantisi var

happens-before relationship
birden fazla thread varsa:
Mesela,
A thread'in oluşturduğu sonuç B Thread'i tarafından görülebilir durumdadır.
*/
```

MUTEX SINIFLARI

```
/*
    MUTEX SINIFLARI
        std::mutex
        std::timed_mutex
        std::recursive_mutex
        std::recursive_timed_mutex
        std::shared_mutex
        std::shared_time_mutex
*/
/*
    std::mutex
    Lock() - mutex'i edinmek için
    unlock() - mutex'i serbest bırakmak için
    try_lock() - mutex'i edinmeye çalışmak için
*/

```

```
#include <mutex>
std::mutex mtx;
void func()
{
    mtx.lock();
    // critical section
    mtx.unlock();

    if (mtx.try_lock()) // bool döndürür
    {
        // edinirse burayı yapacak
    }

    mtx.native_handle(); //başka apillerde kullanmak için
}

//////

std::mutex mtx;
int main()
{
    // kopyalama ve taşıma yok
    auto y = move(mtx);
    auto x = mtx;
}
```

```
class Myclass
{
public:
    void foo() const
    {
        mtx.lock();

        mtx.unlock();
    }
private:
    mutable std::mutex mtx;
}
```

```

// SORU
using namespace std;

std::mutex mtx;
int cnt = 0;

void foo()
{
    for (int i = 0; i < 10'000; ++i)
        mtx.lock();
        ++cnt;
        mtx.unlock();
}

void bar()
{
    for (int i = 0; i < 10'000; ++i)
        mtx.lock();
        --cnt;
        mtx.unlock();
}

int main()
{
    std::thread t1{foo};
    std::thread t2{bar};

    // senkronizasyon gereklidir
    t1.join();
    t2.join();
}

```

```

std::mutex mtx;
void foo()
{
    throw std::runtime_error{ "error from foo "};
}

void func()
{
    mtx.lock();
    // Lock_guard(mtx) // RAI
    try
    {
        foo();
        mtx.unlock(); // unlock olmamak buna dead lock denir
    }
    catch (const std::exception& ex)
    {

    }
}

```

MUTEX SARMALAYAN RAIİ SINIFLARI

```
MUTEX SARMALAYAN RAIİ SINIFLARI
    lock_guard
    unique_lock
    scoped_lock
    shared_lock
*/
```

```
// std::lock_guard
std::mutex mtx;

void foo()
{
    // mutex'i sarmalar ve lock eder
    lock_guard<mutex> lock{ mtx }; // scope sonunda unlock eder
    // lock_guard lock{mtx} // CTAD

    lock_guard lg{mtx}
    auto x = lg; // no copy
    auto y = move(lg); // no move
}
```

```
std::mutex mtx;
void func()
{
    // adopt_lock
    lock_guard lg{ mtx, adopt_lock };
}
```

```
// std::timed_mutex
/*
    try_lock_for
    try_lock_until
*/
void func()
{
    std::timed_mutex mtx;

    if (mtx.try_lock_for(50ms))
    {
        // 50ms boyunca denicek ama kiliti elde edemezse false döncek
    }
}
```

```
// ÖRNEK
using namespace std;

int cnt{};
std::timed_mutex mtx;
void try_increment()
{
    for (int i = 0; i < 100'000; ++i)
    {
        if (mtx.try_lock_for(1ms))
        {
            ++cnt;
            mtx.unlock();
        }
    }
}

int main()
{
    vector<thread> tvec;

    for (int i = 0; i < 10; ++i)
        tvec.emplace_back(try_increment);

    for (auto& t : tvec)
        t.join();

    cout << "cnt = " << cnt << "\n";
}
```

```
// ÖRNEK
std::mutex mtx;

void foo()
{
    std::cout << "foo is trying to lock the mutex\n";
    mtx.lock();
    std::cout << "foo has locked the mutex\n";
    std::this_thread::sleep_for(600ms);
    std::cout << "foo is unlocking the mutex\n";
    mtx.unlock();
}

void bar()
{
    std::this_thread::sleep_for(100ms);

    std::cout << " bar is trying to lock the mutex\n";

    while (!mtx.try_lock())
    {
        std::cout << " bar could not lock the mutex\n";
        std::this_thread::sleep_for(100ms);
    }

    std::cout << "bar has locked the mutex\n";
    mtx.unlock();
}

int main()
{
    std::jthread t1{ foo };
    std::jthread t2{ bar };
}
```

```

class List{

public:
    void push_back(int x)
    {
        mtx.lock();
        mlist.push_back(x);
        mtx.unlock();
    }

    void print()const
    {
        std::lock_guard lg{ mtx};
        for (const auto val : mlist)
        {
            std::cout << val << ' ';
        }
        std::cout << "\n";
    }

private:
    std::mutex mtx;
    std::list<int> mlist;
};

void func(List& list, int x)
{
    for (int i = 0; i < 10; ++i)
        list.push_back(x + i);
}

```

```

//dead Lock: bir threadin ileryememesi
// std::lock --birden fazla mutex veriyoruz dead lock'tan korur
// std::scoped_lock
// Lock_guard maliyet olarak farkı yok ama dead lock'tan korur

```

```
std::mutex m;
timed_mutex tm;
recursive_mutex rm;

void foo()
{
    scoped_lock<std::mutex, std::timed_mutex, std::recursive_mutex> slock{ m, tm,
rm};
}

// std::recursive_mutex
class Myclass
{
    public:
        void foo()
    {
        mtx.lock();
        bar();
        mtx.unlock();
    }

        void bar()
    {
        mtx.lock();

        mtx.unlock();
    }

    private:
        //mutable std::mutex mtx; // mutex'i birden fazla kitlemek tanumsız
davranış
}        mutable std::recursive_mutex mtx; // birden fazla kitlemek legal

int main()
{
    using namespace std;

    Myclass m;

    thread t{ &Myclass::foo, ref(m)};
    t.join();
}
```

2024 01 10

```
#include <shared_mutex>

using namespace std::literals;

int cnt = 0;
std::shared_mutex mtx;

void writer()
{
    for (int i = 0; i < 10; ++i) {
        {
            std::scoped_lock lock(mtx);
            ++cnt;
        }
        std::this_thread::sleep_for(100ms);
    }
}

void reader()
{
    for (int i = 0; i < 100; ++i) {
        int c;
        {
            // yazma amaçlı thread kiliti edinemez ama oku amaçlı edinebilir
            std::shared_lock lock(mtx);
            c = cnt;
        }
        std::osyncstream{ std::cout } << std::this_thread::get_id() << ' ' << c <<
        '\n';
        std::this_thread::sleep_for(10ms);
    }
}

int main()
{
    std::vector<std::jthread> tvec;

    tvec.reserve(16);
    tvec.emplace_back(writer);

    for (int i = 0; i < 16; ++i)
        tvec.emplace_back(reader);

}
```

```

// std::unique_lock
std::mutex mtx;

int main()
{
    unique_lock<mutex> ulock;

    // diğer Lock sınıflarından farklı olarak aşağıdakilere sahip:
    ulock.lock();
    ulock.unlock();
    ulock.try_lock();

    // unique_lock move only type
}

```

```

using namespace std;
std::mutex mtx;

void foo()
{
    unique_lock lock{ mtx }; // kiliti ediniyoruz lock() çağrıılır
}

void bar()
{
    // lock() çağrılmaz
    unique_lock lock{ mtx, adopt_lock }; // kiliti edinmiş durumda alıyoruz
}

void bar()
{
    // lock() çağrılmaz
    unique_lock lock{ mtx, defer_lock }; // kiliti kitlemiyor.
    // burada kilitlemek gerekiyor.
}

void bam()
{
    // kiliti edinemeye çalışıyor edinememezse bloke edilmiyor
    unique_lock lock{ mtx, try_to_lock };

    if (lock.owns_lock())
    {
        // kiliti edinmişse bu bölgeye girer.
        lock.unlock();
    }
}

```

```
int cnt{};

std::mutex mtx;

void func()
{
    for (int i = 0; i < 1'000'000; ++i)
    {
        std::unique_lock ulock{ mtx, std::defer_lock };
        ulock.lock();
        ++cnt;
        ulock.unlock();

        ulock.lock();
        ++cnt;
        ulock.unlock();
    }
}

int main()
{
    {
        std::jthread t1{ func };
        std::jthread t2{ func };
        std::jthread t3{ func };
        std::jthread t4{ func };
        std::jthread t5{ func };
    }

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

```
// std::once_flag and std::call_once

using namespace std;

unique_ptr<string> uptr;
once_flag flag;

void initialize()
{
    osyncstream{ cout } << "initialize " << <this_thread::get_id() << "\n";
    uptr = make_unique<string>("emre bahtiyar");
}

const string& get_value()
{
    // herhangi bir thread sadece bir kez initialize fonksiyonu çağıracak
    call_once(flag, initialize);
    return *uptr;
}

void workload()
{
    const std::string& rs = get_value();
    osyncstream{ cout } << &rs << "\n";
}

int main()
{
    vector<thread> tvec;
    tvec.reverse(20);
    for (int i = 0; i < 16; ++i)
    {
        tvec.emplace_back(workload)
    }
}
```

```
// thread-safe singleton
class Singleton {
public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    static Singleton* get_instance()
    {
        call_once(m_init_flag, Singleton::init);

        return m_instance;
    }

    static void init()
    {
        m_instance = new Singleton();
    }
private:
    static std::once_flag m_init_flag;
    static Singleton* m_instance;
    Singleton() = default;
};

Singleton* Singleton::m_instance{};

std::once_flag Singleton::m_init_flag;

void func()
{
    std::osyncstream{ std::cout } << Singleton::get_instance() << '\n';
}

int main()
{
    std::vector<std::thread> tvec;
    for (int i = 0; i < 100; ++i) {
        tvec.emplace_back(func);
    }

    for (auto& th : tvec)
        th.join();
}
```

```

// thread-safe singleton
class Singleton {
    public:
        static Singleton& get_instance()
        {
            static Singleton s;
            return s;
        }
}

// call_once alternatif
using namespace std;
void func()
{
    osyncstream{cout} << "func cagrildi " << this_thread::get_id() << "\n";
}

void foo()
{
    std::this_thread::sleep_for(50ms);
    static auto f = [] {func(); return 0; }();
}

int main()
{
    vector<jthread> tvec;

    for(int i = 0; i < 10; ++i)
        tvec.emplace_back(foo);
}

```

std::future and std::promise

```

#include <future>
int main()
{
    using namespace std;

    std::promise<int> prom;
    future<int> ft = prom.get_future();

    prom.set_value(12);

    int val = ft.get(); // val 12
}

```

```
void produce(std::promise<double> prm, double val)
{
    prm.set_value(dval * dval);
}

int main()
{
    using namespace std;

    promise<double> prom;
    auto ft = prom.get_future();

    thread t{produce, move(prom), 4.543};

    auto val = ft.get();

    cout << "value = " << val << "\n";

    t.join();
}
```

```
std::string foo(std::string str)
{
    auto temp = str;
    reverse(str.begin(), str.end());

    return temp + str;
}

void bar(std::promise<std::string>&& prom, std::string str)
{
    prom.set_value(foo(str));
}

using namespace std;

int main()
{
    promise<string> prom;

    future<string> ft = prom.get_future();

    thread t{bar, move(prom), "tamer" };

    cout << ft.get() << "\n";

    t.join();
}
```

```
struct Div
{
    void operator()(std::promise<int>&& prom, int a, int b)
    {
        if (b == 0)
        {
            auto str = "divide by zero error " + std::to_string(a) + "\\" +
std::to_string(b);
            prom.set_exception(std::make_exception_ptr(std::runtime_error(str)));
        }
        else
        {
            prom.set_value(a / b);
        }
    }
};

using namespace std;

int main()
{
    promise<int> prom;

    auto ft = prom.get_future();

    thread th{ Div{}, move(prom), 12, 3};

    try
    {
        cout << ft.get() << "\n";
    }
    catch (const std::exception& ex)
    {
        std::cout << "exception caught : " << ex.what() << "\n";
    }

    th.join();
}
```

Std::async

```
// std::async
int foo(int x, int y)
{
    return x * y + 5;
}
void bar(int x, int y)
{
    return x + y + 2;
}
int main()
{
    using namespace std;

    auto ft = async(foo, 10, 30); // std::future<int> döner
    auto ft1 = async(bar, 10, 30); // std::future<int> döner

    auto val = ft.get() + ft1.get();

}
```

```
// std::packaged_task
int foo(int, int);
int main()
{
    using namespace std;

    packaged_task mytask(foo, 2, 5);

    packaged_task <int(int, int)> task{ foo };

    thread th{ task, 3, 6};
}
```