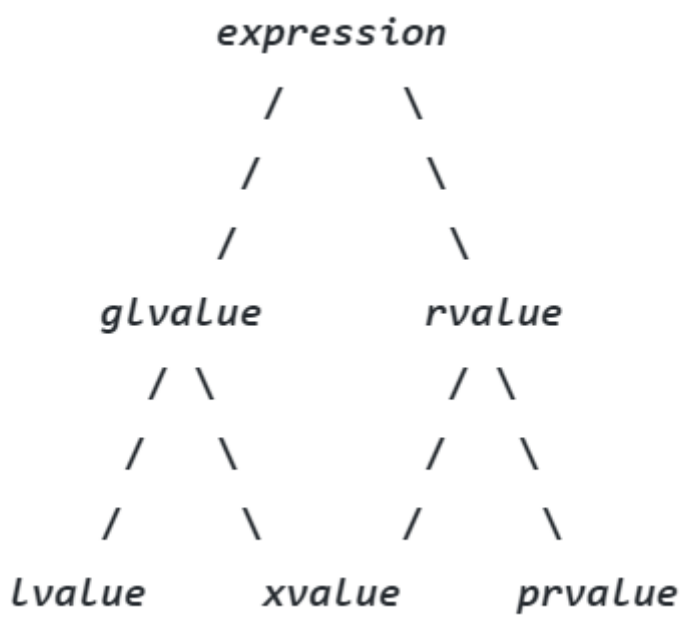
İçerik

- Copy elision
- Boolean context
- Complete / Incomplete Type
    - Incomplete Type
    - Complete Type
- Sınıfın Statik Veri Elemanları Ve Üye Fonksiyonları
- if with initialization
- Inline Variable
- Nested Type
- Piml Idiom(Pointer Implementation)
- Composition
- Aggreagation
- Delegating Constructor
- Raw String Literal
- Namespaces

Reference



Value Category

- RValue

- LValue

Type Category

- RValue reference
- LValue reference

Rvalue: Taşınabilir fakat bellekte herhangi bir adresi yok.

Lvalue: İsim formundaki yani bellekte yer tutan nesnelerin değer kategorisidir.

> Bir sol taraf referansı bir "sağ taraf değeri" ifadesine bağlanamaz.

> Bir sağ taraf referansı bir "sol taraf değeri" ifadesine bağlanamaz. Fakat const sol taraf referansı "sağ taraf değeri" ifadesine bağlanabilir.

> Sol taraf referansları ilk değer ile başlatılmak zorundadır.

```
int x = 10;
int& r = x;
```

> Referanslar rebindable değildir. "T* const ptr" gibidirler.

```
int x = 10;
int y = 5;
int& r = x;
r = y; //Error refrence not rebindable
```

> Bir sol taraf referansı bir "sağ taraf değeri" ifadesine bağlanamaz.

```
int& r = 10; //Error Lvalue reference can not bind rvalue
```

> Bir const sol taraf referansı bir "sağ taraf değeri" ifadesine bağlanabilir.

> İsim formundaki tüm ifadelerin değer kategorisi Lvalue expersion'dır.

```
int  foo();
int& func();
int main() {
    int& a = func(); // valid
    int b = foo();   // valid
    int& c = func(); // invalid(Lvalue ref. can't bind r value)
    const int& d = foo(); //valid. [*]
}
```

`const int& d = foo()` şeklinde belirtilen kodun geçerli olmasının sebebi compiler `const int temp = foo();` şeklinde bir kod üretir ve daha sonra `const int& d = temp` olarak oluşturulur böylece lvalue referans sol değer ifadesine bağlanır.

> Bir ifadenin data type başka value category'si başkadır.

```
int&& ref = 10;
//ref ---> isim formu olduğu için L value expr.
//data type ise sağ taraf referansıdır.(RValue reference)
```

> const lvalue reference tipinde bir değişkene sağ taraf değeri ile ilk değer vermenin en önemli kullanımı move schematic'tir. Sınıf yapılarında bunu tekrar inceleyeceğiz.

```
double dval{2.456};
const int& x = dval; //valid
const int& y(dval); //valid
const int& z{dval}; //invalid. Narrowing conservion
```

- `int x(10);` → direct initialization
- `int x{};` → value, uniform, braced initialization
- `int x{10};` → direct list initialization

Uniform initialization neden eklendi?

1. Neye ilk değer verirsen ver her zaman kullanılabilir.
2. Narrowing conversion durumunu engellemek için.
3. Most vexing parse(Scott Meyers tarafından dile ekledi) durumunu engelemek için.

Most vexing parse

```
#include <iostream>

struct A {

};

struct B {
    B(A) {
        std::cout << "B constructor\n";
    }
};

int main() {
    B b(A()); //function decleration
}
```

Yukarıdaki örnekte derleyici nesne bildirimi yerine önceliği fonksiyon bildirimine verir. Tipik bir `most verxing parse` örneğidir. Fakat `B b{A()}`şeklinde olsaydı bu bir nesne bildirimi olacak ve standart outputa ***B constructor*** yazacaktı.

Auto Type Deduction

Acronym: Kelimelerin baş harfleriyle oluşturulur.

- AAA --> Almost Always Auto
- VIP --> Very Important Person

Auto aşağıdaki şekillerdeki gibi kullanılabilir.

```
auto x = expr;
auto& x = expr;
auto&& x = expr;
```

Aşağıdaki şekilde kullanılması durumda eğer reference ve const varsa her ikiside düşer.

```
const int x = 10;
auto y = x; // int y = x;

int& r = x;
auto y2 = r; // int y2 = r;

const int& z = x;
auto y3 = z; // int y3 = z;
```

> **İstisna:** `auto p = "enes";` ilk değer verme kullanılması durumunda derleyici `const char* temp = "enes";` şeklinde bir kod üretir. Daha sonra auto ifadesi yerine `const char* p = temp` gelecektir.

Reference ile kullanılması durumda

```
int x = 10;
int& r = x;
auto& y = r; // int& y = r;

const int& r2 = x;
auto& y2 = r2; // const int& y2 = x;
```

Son olarak `auto&& x = expr` için aşağıdaki durumların olması durumu sırası ile inceleyelim.

- İlk değer veren ifade pr value expression ise,
- İlk değer veren ifade r value expression ise,

- İlk değer veren ifade x value expression ise

expr	auto	çıkarım(x)
T&	&	T&
T&	&&	T&
T&&	&	T&
T&&	&&	T&&

NOT: Basit olarak çıkarımı şu şekilde yapabiliriz. Eğer `expr` sol taraf ifadesi ise `x` sol taraf referansı, eğer `expr` sağ taraf ifadesi ise `x` sağ taraf referansı olacaltır.

```
int y = 10;
auto&& r = 10; // sağ taraf ifadesi bu yüzden int&& r = 10 olacaktır.
auto&& r2 = y; // sol taraf ifadesi bu yüzden int& r2 = y; olacaktır.
```

Eğer ki `auto&&` bir sol taraf ifadesine bağlanılması durumunda `T& &&` çıkarımı yapılır. Burada referans bozulması(reference collapsing) olur. Böylece reference düşer `T&`şeklini alır.

---

decltype

Eğer decltype operatörünün operandı olan ifade bir isim formunda değil ise ifadenin value kategorisine bakılacak.

- İfade Lvalue ise elde edilen tür **T &**
- İfade PRvalue ise elde edilen tür **T**
- İfade Xvalue ise elde edilen tür **T &&**

---

constexpr

En önemli kullanımı fonksiyonların geri dönüş değeri olarak kullanımıdır.

```
constexpr int sum_square(int a, int b)
{
    return a*a + b*b;
}
int main() {
int x = 5, y = 4;
sum_square(x,y); // Geçerli
constexpr int z = sum_square(x,y); //Geçersiz
}
```

Sum_square fonksiyonunun operandına sabit olmayan değişkenler girildiğinde normal bir fonksiyon gibi davranır. Yani run time'da çalışır. Bu yüzden "constexpr int z" sadece sabit bir ifade ile başlayacağından geçersiz olacaktır.

> constexpr fonksiyonlar başlık dosyalarında implicit inline olarak tanımlanır.

Function Overloading

Aynı isimli işlevler aynı kapsamda(scope) olmalıdır. Aynı kapsamdaki aynı isimli işlevlerin imzaları farkı olacak.

Function Redeclaretion

```
void func(int a, int b);
void func(int, int);
```

Syntax Error

```
void func(int a, int b);
int func(int, int);
```

Function Overloading

```
void func(int a, int b);
void func(int);
```

1. Variadic conversion

İstisna: C'de variadic fonksiyonlar sadece *elipsis(...)* kullanılarak tanımlanamaz. C++ bu gerçerlidir.

```
//variadic func.
void func(...); // 1
void func(int x, ...); // 2
void func(int x, int y, ...); // 3

int main()
{
    funcR(1,2,3,4,5); // 3 çağrılır.
}
```

2. User-defined Conversion

   Programlayıcı tarafından tanımlanan dönüşümlere denir.

```
struct Data  {
    Data() = default;
    Data(int);
}
int main() {
    Data mydata;
    mydata = 10; //user-defined type
}
```

3. Standard Convertion
    - int → double
    - double → int
    - double → char
    - enum → int
    - int* → void*

    gibi bazı dönüşümler derleyicinin implicitly dönüşümlerine örnektir.

```
void func(int);    // 1
void func(double); // 2
void func(char);   // 3
int main() {
    func(12.f); // 2
}
```

> **func(double)** çağrılacaktır. Çünkü float'tan double'a promotion vardır.

1. exact match(tam uyum)
2. promotion(terfi-yükseltme)
3. conversion

Yukarıdaki 3 durum içerisinde fonksiyon sırası ile uygun olan fonksiyonu derleyici arar.(exact match > promotion > convertion)

Eğer ki exact match yok ise promortion'a bakılır o da yok ise conversiona bakılır. Eğer ki birden fazla aynı işlevde conversion olacak fonksiyon varsa ambiguity olur.

Exact-Match

- LValue to rvalue conversion
- T* to const T* conversion

- Function to pointer conversion

exact match'tir.

Örneğin;

```
void func(int a);
int main() {
    int x = 10;
    func(x); //Lvalue to rvalue exact-match olacktır.
}
```

Promotion

- Integral promotion

  int'ten küçük türlerden int türüne yapılan dönüşeme denir.

  - char → int
  - signed char → int
  - unsigned char → int
  - bool → int
  - signed short → int

float → double

C ve C++ dillerinde

1. Fonksiyonların parametre değişkenleri dizi (array) olmaz.
2. Fonksiyonların geri dönüş değer türleri dizi(array) olmaz.

```
//function redecleration
void func(int);
void func(const int);
```

const overloading

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

Numaralandırma Türleri

Enum:

> C'den farklı olarak underlying type vardır.

Enum'larda underlying(base) type sabit değildir. Böylece enumların temel tipi bir `implementation defined integral type`'dır.

```
enum Color : char {
WHITE,
GREEN,
RED,
BLACK
};
```

Enum classes:

> Underlying type vardır.

> Forward decleration vardır.

```
enum class Color : char;
```

Enum class 'ların avantajları;

- Underlying type
- Implicit cast yok
- Scope resolution var

```
enum class Color {
    White,
    Black,
    Green,
    //...
};
int main() {
    Color mycolor = White; //Geçersiz çünkü white, enum Color kapsama alanında
    Color mycolor = Color::White; // Geçerli
}
```

Tür Dönüşüm Operatörleri:

- static_cast
- const_cast
- reinterpret_cast
- dynamic_cast

C-style casting

Tüm cast işlemlerinde `()` parantez operandı içerisinde hedef tip belirtilerek yapılır(`(type target)expr`).

```
//C style casting
const int x = 10;
int* ptr = (int*)&x;
*ptr = 48; //Valid but undefined behaviour
```

static_cast

- int*'dan void*'a implicit type conversion vardır.
- int*'dan void*'a veya void*'dan int*'a hem static_cast hem de reinterpret_cast kullanılabilir.

```
int x = 10;
void* sptr = static_cast<void*>(&x);
void* rptr = reinterpret_cast<void*>(&x);
```

const_cast

```
const int x = 10;
int* ptr = const_cast<int*>(&x);
*ptr = 48; //Valid but undefined behaviour
```

C'de kullanılan `strchr` fonksiyonu buna en güzel örnek olabilir.

```
char* Strchr(const char* p, int c) {
    while(*p++) {
        if(*p == c) {
            return const_cast<char*>(p);
        }
    }
    if(c == '\0')
        return const_cast<char*>(p);
    return nullptr;
}
```

reinterpret_cast

```
int x = 145981;
char* c = reinterpret_cast<char*>(&x);
for(int i = 0; i < sizeof(x); ++i) {
```

```
    std::cout << c[i] << "\n";
}
int* y = reinterpret_cast<int*>(c);
```

Extern "C" Bildirimi

C'de derlenmiş kütüphaneleri C++'da kullanabilmek için belirtilir.

```
extern "C" void f1();
extern "C" void f2();
```

```
extern "C" {
void f1();
void f2();
}
```

Aşağıdaki şekilde ön tanımlı sembolik sabit(`predefining symboling constant`) makrosı ile sarmalanır.

```
#ifdef _cplusplus
extern "C" {
#endif
    void f1();
    void f2();
    void f3();
    void f4();
#ifdef _cplusplus
}
#endif
```

Classes

```
class Myclass {
    // class members
    // data members(veri elemanları/öğeleri)
    // member function
    // type member

    int mx;           // data member
    typedef int Word; // type member
    void func(int);   // member function
}
```

C++ scope'lar aşağıdaki gibidir.

- Namescape scope
- Block scope
- Function prototype scope
- Function scope
- Class scope

> Sınıfın üye fonksiyonları(member func.) sınıf içerisinde yer kaplamazlar.

Sınıf veri elemanları(data members)

- non-static member
- static member → global

olabilir.

Access specifier,

- public
- private
- protected

olmak üzere 3 tanedir.

```
// Class decleration
// Forward decleration
// Incomplete type
class Myclass;
```

const üye fonksiyonlar

```
void func(T*);       // setter, mutator
void func(const T*); // getter, accessor
```

```
class Myclass {
public:
    void func();      // func(Myclass*)
    void foo()const;  // foo(const Myclass*) const member func.
}
```

Sınıfın const üye işlevleri const olmayan üye işlevlerini çağırmamlı.

```
void Myclass::func()
{
    foo(); // Geçersiz T* --> const T* dönüşüm vardır.
```

```
}

void Myclass::foo() const
{
    func(); // Geçersiz const T* --> T* dönüşüm yoktur.
    // func(Myclass*)
    // foo(const Myclass*)
}
```

NOT: const bir sınıf nesnesi ile sadece const üye işlevleri çağırabilir.

```
int main() {
    const Myclass m;
    //&m = const Myclass*
    m.func(); // Geçersiz sentaks hatası const T* --> T*
}
```

Mutable

Sınıfın const bir üye fonksiyonunun sınıfın static olmayan veri elemanlarını değiştirebilmesi için veri elemanının `mutable` olması gerekir.

```
class Date {
public:
    int day_of_year()const;
private:
    int md,mm,my;
    mutable debug_count{};
}
int Date::day_of_year() const {
    ++debug_count; // Geçerli  debug_count mutable
    // ...
    return md;
}
```

One Defination Rule

Bir proje içerisinde aynı varlığın birden fazla tanımı olmaz. Eğer bu varlığın tanımı aynı kaynak dosyası içerisinde olursa sentaks hatası olur. Farklı kaynaklarda olursa sentaks hatası değil fakat ill-formed olur.

> C++ dilinde yazılımsal öyle varlıklar var ki bu varlıkların projeyi oluşturan farklı kaynak dosyalarda birden fazla kez tanımlanması(token by token aynı olması) durumunda ill-formed değildir.

```
// a.cpp
class A {
    int x,y;
    void func();
```

```
}

// b.cpp
class A {
    int x,y;
    void func();
}
```

Farklı kaynak dosyalarında tanımlandıkları halde token-by-token aynı oldukları için ill-formed değil well-formed olurlar. Nitekim başlık dosyalarında oluşturduğumuz class tanımını a.hpp yi a.cpp de include ettiğimiz takdirde bu geçerli olmasaydı ill-formed olurdu.

ODR'a uyanlar;

- class definitions
- inline functions definitions
- inline variable definitions
- class template definitions
- ...

inline fonksiyonlar

compiler optimizasyonu,

1. Derleyici compile time'da kod seçerek
2. Kod optimizasyonu yaparak

gerçekleştirebilir.

```
inline int func(int x) { return x*x+5; }
// fonksiyona giriş kodu
a = func(5); // a = x*x+5 yapabilir.
// fonksiyondan çıkış kodları
```

C++ inline ile C'deki inline kurallarının farklılıkları vardır.

inline fonksiyonlar;

- tanımını (sağlıklı biçimde) başlık dosyasına koyduk böylece derleyiciye inline expansion olağanı verdik.
- kodu expose eder.

```
free function = standalone function = global function
```

Class içerisinde fonksiyonu belirtirsek implicit inline olur.

Hangi fonksiyonlar inline olarak tanımlanır?

1. Sınıfın non-static üye işlevleri
2. Sınıfın static üye işlevleri

3. Sınıfın friendlik verdiği işlevler

Special member functions

Sınıfların özel üye fonksiyonları:

- default constructor X();
- destructor ~X();
- copy constructor X(X const&);
- move constructor(C++11) X(X&&);
- copy assignment X& operator=(X const&);
- move assignment(C++11) X& operator=(X&&)

Constructor(Kurucu fonksiyon)

Statik ömürlü global nesneler için constructor main'den önce çağrılır.

- static initialization fiasco
- static initialization problem

Destructor(Yıkıcı fonksiyon)

Bir sınıfın sadece bir tane destructor'u vardır ve hiçbir parametre almaz.

```
class Myclass {
    ~Myclass();
}
```

Bir fonksiyonun sonuna `delete` anahtar kelimesi eklenerek delete edilebilir.

```
void func() = delete;
```

En önemlisi delete edilen fonksiyonlar, function overloading resolution sürecine katılır.

```
void func();
void func(int);
void func(double) = delete;

func();    // Geçerli
func(12);  // Geçerli
func(2.4); // Geçersiz function deleted.
```

Constructor initialization list

```
class Myclass {
public:
    Myclass() : u(expr), t(expr)
```

```
    {
    }
private:
    T t;
    U u;
}
```

Class içerisindeki üye elemanlarının sırası ile kurucu ilklendirme listesi(constructor initialization list) aynı sırada olması iyi bir alışkanlıktır. Fakat yukarıdaki sınıfda sıraya uyulmuş olunmasada derleyici ilk olarak t'yi daha sonra u'yi ilklendirir.

Default member initializer

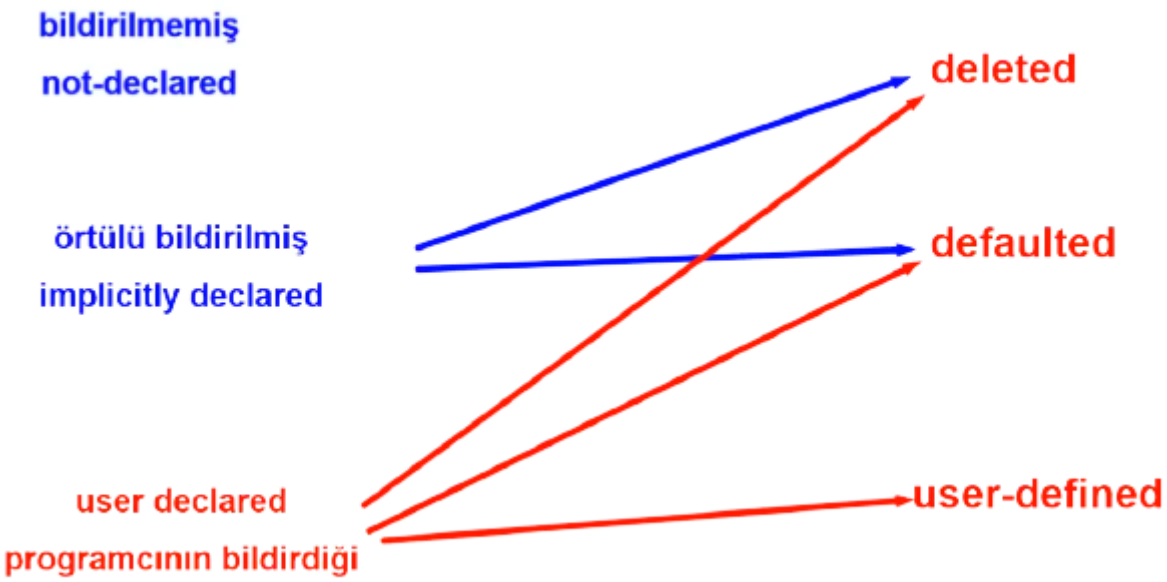
Class içerisinde parantez`()` atomu ile üye elemanları ilklendirme(`default initialize`) yapılamaz.

```
class Myclass {
    int mx(3);      // Geçersiz
    // T t = expr;  // Geçerli
    // T t{ expr }; // Geçerli
    // T t(expr);   // Geçersiz
}
```

Eğer programlayıcı sınıf için hiç bir constructor yazmaz ise derleyici sınıfın default constructor'ını default eder.

Derleyicinin yazdığı(default ettiği) default constructor

1. Sınıfın non-static public inline fonksiyonudur.
2. Bu fonksiyon sınıfın tüm veri elemanları default initialize eder.
3. Ancak eğer bir veri elamanı için default member initialize kullanılmış ise bu durumda derleyici bu init'ı kullanır.



Implicitly declared

```
class Myclass {

};
```

User declared

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

Not declared

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

Eğer derleyici durumdan vazife çıkartarak sınıfın özel bir üye fonksiyonunu default ederse (yani bu özel üye fonk. implicitly declared ise) eğer bu fonksiyonun derleyici tarafından yazımında bir sentaks hatası oluşursa derleyici bu fonksiyonu delete eder.

```
class Myclass {
public:

private:
    const int mx;
};

Myclass m; // Hata ctor deleted
```

```
class Myclass {
public:

private:
    int& r;
};

Myclass m; // Hata ctor deleted
```

Sentaks hatası const bir değişken ve referanslar ilk değer başlatılmak zorunda.

Copy constructor

Bir sınıf nesnesi hayata değerini aynı türden bir sınıf nesnesinden alarak geldiğinde, derleyicinin Myclass sınıfı için yazdığı copy ctor

1. sınıfın non-static public inline fonksiyonudur.
2. parametrik yapısı `Myclass(const Myclass&);`

```
class Myclass {
public:
    Myclass(const Myclass& other) : tx(other.tx), ux(other.ux)
    {
    }
private:
    T tx;
    U ux;
};
```

Hangi durumlarda copy constructor yazmak gerekir?

İdeali derleyicinin bu fonksiyonları kendisinin yazması, buna rule of zero denmektedir.

- Dinamik bir veri elemanı olması durumunda
- Pointer veri elamanı olması durumunda

Kopyalamayı derleyici yapıyorsa shallow copy(sığ kopyalama) yapar. Bu durumda dinamik veri elmanı veya pointer üye elemanımız varsa bu kopyalamayla aynı bellek alanını başka bir nesne ile paylaşmış olacağız böylece bir nesnenin ömrünün sona ermesiyle diğer kopyasını alan nesnenin sona ermesi durumunda free edilen bellek adresi tekrar free edilmeye çalışıldığından run time çalışma hatası olacaktır.

- Bir nesnenin kendine atanmasına self assigment denilir. Bu durumda tanımsız davranış oluşur.
- Copy Constructor'u siz yazacaksanız sınıfın tüm öğelerinden siz sorumlusunuz. Sadece pointer için yazıp, diğer primitif türler için yazmazsak o öğeler çöp değerler ile başlar.

```
class A {
public:
private:
    T *mp;
    U x, y, z; // Bu öğeler içinde copy ctr içerisinde atama yapman gerekiyor.
};
```

```
class Name
{
private:
    char *mp;
    size_t mlen;
public:
    Name(const char *p) : mlen{std::strlen(p) }
    {
            mp = static_cast<char*>(std::malloc(mlen + 1));
            if (!mp) {
                std::cerr << "bellek yetersiz !\n";
                std::exit(EXIT_FAILURE);
            }
            std::strcpy(mp, p);
    }

    Name(const Name &other) : mlen(other.mlen)
    {
            mp = static_cast<char*>(std::malloc(mlen + 1));
            if (!mp) {
                std::cerr << "bellek yetersiz !\n";
                std::exit(EXIT_FAILURE);
            }
            std::strcpy(mp, p);
    }

    Name &operator=(const Name &r)
    {
        if (this == &r)  // self assignment kontrol ediliyor
                return *this;

        mlen = r.mlen;
        free(mp);
```

```
        mp = static_cast<char*>(std::malloc(mlen + 1));
        if (!mp) {
                std::cerr << "bellek yetersiz !\n";
                std::exit(EXIT_FAILURE);
        }
        std::strcpy(mp, p);
    }

    void print()const
    {
        std::cout << "(" << mp << ")\n";
    }

    size_t length() const
    {
        return mlen;
    }

    ~Name()
    {
        free(mp);
    }
};
```

Move Constructor

Hayatı bitecek bir nesne ile başka bir nesneyi hayata getireceksek, kaynakları kopyalamak yerine hayatı bitecek o nesnenin kaynaklarını alabiliriz. Modern C++ ile dile eklenen bu sağ taraf referanslarının gücü ile bunu yapabiliriz. Sınıfımıza move semantiğini ekleyeceğiz. Tipik move ctor önce gidip diğer nesnenin kaynağını çalıyor, sonra fonksiyona gelen nesneyi destruct edilebilir ama kaynağı olmayan durumda bırakıyor. Eğer bunu derleyicinin yazımına bırakırsak şöyle olmak zorunda.

```
class Myclass {
    T x;
    U y;
public:
    Myclass(Myclass &&r) : x(std::move(r.x)), y(std::move(r.y))
    {

    }
//std::move fonksiyonu, sol taraf değeri türünü sağ taraf değerine dönüştürür.
};
```

```
class Name
{
private:
    char *mp;
```

```
    size_t mlen;
public:
    Name() : mlen(0), mp(nullptr) {}
    ~Name() {
        // free edilen kaynağı tekrar free etme!
        if(mp)
            free(mp);
    }
    Name(const char *p) : mlen{std::strlen(p) } {
        mp = static_cast<char*>(std::malloc(mlen + 1));
        if (!mp) {
            std::cerr << "bellek yetersiz!\n";
            std::exit(EXIT_FAILURE);
        }
        std::strcpy(mp, p);
    }

    Name(const Name &other) : mlen(other.mlen) {
        mp = static_cast<char*>(std::malloc(mlen + 1));
        if (!mp) {
            std::cerr << "bellek yetersiz!\n";
            std::exit(EXIT_FAILURE);
        }
        std::strcpy(mp, other.mp);
    }

    Name(Name &&r) : mlen{r.mlen}, mp {r.mp} {
        r.mp = nullptr;
    }

    Name &operator=(Name &&r) {
        if (this == &r) //self-assignment kontrolü
            return *this;

        free(mp);
        mp = r.mp;
        mlen = r.mlen;
        r.mp = nullptr;
        return *this;
    }

    Name &operator=(const Name &r) {
        if (this == &r)
            return *this;

        mlen = r.mlen;
        free(mp);

        mp = static_cast<char*>(std::malloc(mlen + 1));
        if (!mp) {
            std::cerr << "bellek yetersiz !\n";
            std::exit(EXIT_FAILURE);
        }
        std::strcpy(mp, r.mp);
```

```
    }

    void print()const {
        std::cout << "(" << mp << ")\n";
    }

    size_t length() const { return mlen; }
};
```

Temporary Object (Geçici Nesne)

Geçici nesne oluşturma ifadeleri **sağ taraf değeri** ifadesidir.

```
Name name;
name = "Enes Alp";
// Derleyici sırasıyla aşağıdaki kodu üretir.
// 1. Name temp("Enes Alp");
// 2. name = temp;
// temp bir temporary object yani rvalue olduğu için move assignment
çağrılacaktır.
```

std::move:

Bir lvalue ifadesini rvalue ifadesine, rvalue ifadesini ise yine rvalue ifadesine çeviren yardımcı bir fonksiyondur.

`move == static_cast<T &&>(y)` gibi bir dönüşüm gerçekleşiyor diyebiliriz.

Ne zaman kullanılmalı?

Bir nesne bir daha kullanılmayacaksa o zaman taşıma işlemi yapılmalıdır.

Örneğin;

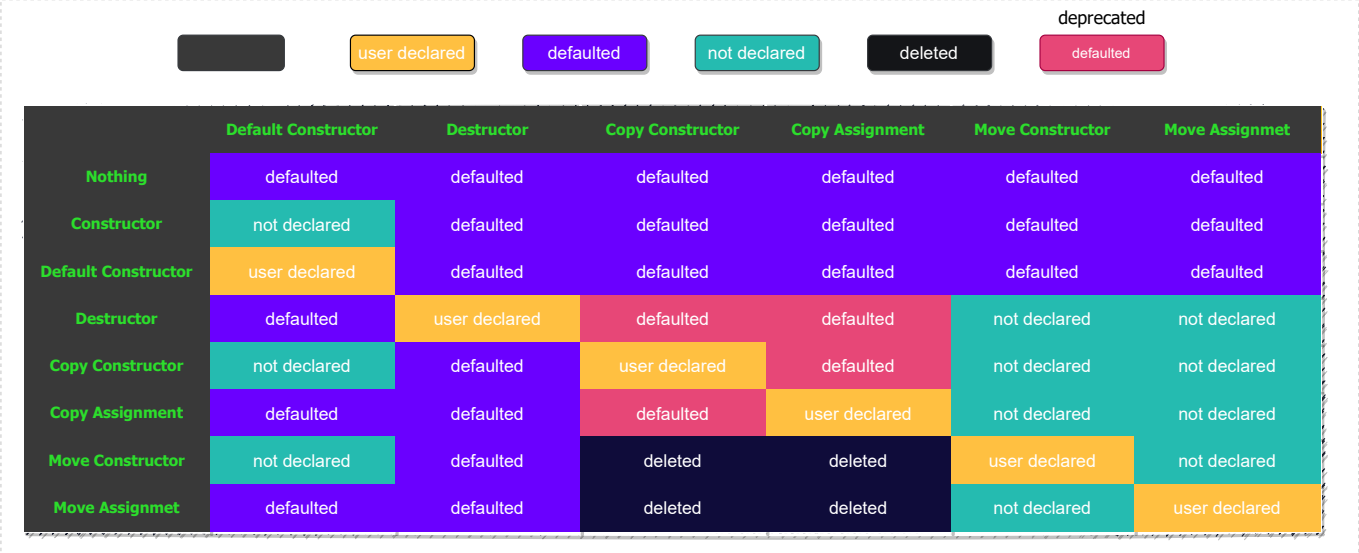
```
swap(T& a, T& b) {
    T tmp(a);   // Şuan a'nın iki kopyasına sahipiz
    a = b;      // Şuan b'nın iki kopyasına sahipiz(+ a'nın bir kopyasını attık)
    b = tmp;    // Şuan tmp'in iki tane kopyasına sahipiz. (+ b'nin bir kopyasını
attık)
}
```

bu kod yerine aşağıdaki kodu tercih etmelisiniz.

```
swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
```

```
    b = std::move(tmp);
}
```

Aşağıdaki görselde bir sınıfın hangi durumlarda tanımlanırsa derleyici hangi özel üye fonksiyonlarını yazacak, silecek veya tanımlamıyacak bunlar gösterilmiştir.



	Default Constructor	Destructor	Copy Constructor	Copy Assignment	Move Constructor	Move Assignmet
Nothing	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 Assignmet	defaulted	defaulted	deleted	deleted	not declared	user declared

Explicit Constructor

Otomatik dönüşümün sentaks hatası vermesi için kullanılır. Ancak tür dönüşüm operatorleri ile kullanılabilir.

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

Mint x(13);  // 1.   Geçerli
Mint y = 13; // 2.   Geçersiz
```

Yukarıdaki durumda derleyici otomatik dönüşüm yapmayacağından yani int türünden Mint sınıf türüne otomatik dönüşüm yapmayacağından 2. ifade geçersiz olur.

Genellikle tek parametreli constructorlar explicit olarak tanımlanır. Bunun en önemli nedeni otomatik dönüşümlerin bulunması zor olan, can sıkıntılı sorunları engelemek içindir.

Bir önceki `Mint` sınıfını expilicit olmadan yeniden tanımlarsak

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

```
};

Mint x = 13;      //Geçerli
Mint y(13);       //Geçerli
Mint z{13};       //Geçerli
Mint f = 13.3;    //Geçerli
Mint g = 13.3f;   //Geçerli
```

Görüldüğü gibi explicit olmadan tanımladığımız tüm ifadeler geçerli durumdadır. Böylelikle yanlış bir ifade girilmesi durumunda logic olarak sentaks hatası beklenilen durumda herhangi bir sentaks hatası yoktur ve logic hatanın bulunmasıda oldukça can sıkıntılı olacaktır.

Derleyici iki şekilde dönüşüm gerçekleştirir.

- UDC(User Defined Conversion)
- SC(Standart Conversion)
    - UDC + SC
    - SC + UDC

Yani derleyici öncelikle SC daha sonra UDC veya tam tersi UDC sonra SC şeklinde kodu üretebilir.

```
Mint x = 13.5; // ilk olarak SC daha sonra USC
//Derleyici Mint x = static_cast<int>(13.5); gibi bir kod üretir.
```

Temporary Object

Öyle ifadeler ki kod içinde isimlendirilmiş bir nesne olmasa da çalışan kodda bir nesnenin varlığı söz konusudur.

Geçici nesnelerin değer kategorisi prvalue expression'dır.

```
Mint(13); //Mint türünden geçici nesne
```

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

int main()
{
    Myclass mx;
    mx = Myclass{13};
    (void)getchar();
```

```
}

// default ctor
// Mclass(int)
// destructor

// destructor
```

Program getchar fonksiyonuna geldiği zaman destructor çağrıldı. Bu bize `Myclass{13}` şeklinde oluşturduğumuz geçici nesnenin derleyici tarafından üretilip daha sonra sonlandığını göstermektedir.

Geçici nesnelerinin hayatlarını uzatabiliriz. Buna `life extension` denilmektedir.

```
const Myclass& r = Myclass{13};
Myclass&& rx = Myclass{13};
```

Yukarıdaki kodda geçici olarak oluşturduğumuz kodu const sol taraf referansı değerine veya sağ taraf referansı değerine bağlayarak life extension yapmış olduk.

Friend Decleration

1. Global bir fonksiyona friend'lik vermek.
2. Bir sınıfın bir üye fonksiyonuna friend'lik vermek.
3. Bir sınıfın tamamına friend'lik vermek.

Yukarıdaki gibi 3 farklı şekilde bir sınıfa friend'lik verilebilir.

```
class Myclass {
private:
    int mx;
    void func());
};

void gf() {
    Myclass my;
    my.func();  // geçersiz
    my.mx = 13; // geçersiz
}
```

Görüldüğü üzere fonksiyon sınıfın özel üye elemanlarına ve işlevlerine erişmesi durumunda sentaks hatası alınacaktır. Fakat bunu sentaks hatası almadan yani access kontrolüne takılmadan erişebilmenin yolu fonksiyona friendlik vererek yapılır.

```
class Myclass {
private:
    int mx;
```

```
    void func();
    friend void gf();
}

void gf() {
    Myclass my;
    my.func();  // geçerli
    my.mx = 13; // geçerli
}
```

`void gf()`fonksiyonuna friendlik vererek sınıfın private kısmına erişmesini sağladık.

Not: Friend bildirimi için access kontrol önemli değildir. İstenilen access modifier içerisinde bildirilebilir.

Operator Overloading

- index`[ ]` operatörü
- `*` ve `->` operatörü
- Fonksiyon çağrı operatörü
- Tür dönüştürme operatörü
- Enum türleri için operatör fonksiyon yazımı

Aşağıdaki operatörler **overload edilemez.**

- Nokta`.` operatörü
- sizeof operatörü
- ternary`?:` operatörü
- çözünürlük`::` operatörü
- .* operatörü
- typeid operatörü

Bazı operatörler **global olamaz.**

- Köşeli parantez`[]` operatörü
- Ok`->` operatörü
- Tür dönüştürme`()` operatörü

Aşağıdaki operatörler **sol taraf değeri** döndürür.

- Assignment`=`
- Subscript`[]`

- Class member access `->`
- Pointer to member selection `->*`
- Dereference `*`
- new/delete
- Smaller than `<=`
- Greater than `>=`
- Prefix Increment `++`
- Prefix Decrement `--`

Operatörler hem sınıf içerisinde hemde global olarak tanımlanabilir. Bazı operatörlerde unary, binary veya ternary olabilir.

Unary Operator

Global olarak

```
~x ---> operator~(x)
```

Sınıf içerisinde

```
~x ---> operator~()
```

Binary Operator

Global olarak

```
x<y ---> operator<(x,y)
```

Sınıf içerisinde;

```
x<y ---> x.operator<(y)
```

> Sınıfın nesnesini değiştiren operatörler üye operatörlere simetrik iki operand olan operatörler ise global yazılması tavsiye ediler.

> Operatörlerin dildeki belirlenmiş öncelik seviyesi ve öncelik yönünü associativity değiştirilemez.

Index([ ]) operatorü

- T& opeartor[](size_t idx);
- const T& operator[](size_t idx)const;

Const doğruluğunu korumak amacıyla operatörün const overloading fonksiyonu yazılır. Böylece

```
class Myarray {
public:
    Myarray(int size) : msize(size), mp(new int[size]) {
        memset(mp,0,msize * sizeof(int));
    }
    int& operator[](size_t idx) {
        return mp[idx];
    }
    const int& operator[](size_t idx)const {
        return mp[idx];
    }
private:
    size_t msize;
    int* mp;
}
```

İçerik(*) operatorü

Unary bir operatordür. Global olarak tanımlanamazlar. Sadece sınıf içerisinde tanımlanabilir.

Counter sınıfı üzerinden devam edecek olursak.

```
int& operator*()const {
    return mp;
}
```

Ok(->) operatorü

Unary bir operatördür. Geri dönüş değeri sınıf nesnesinin adresini döndürür.

```
class Counter {
public:
    Counter(int count) : mcount(count) {}
    ~Counter() {
        std::cout << "Counter() destructor\n";
    }
    int get_value()const { return mcount; }
    friend std::ostream& operator<<(std::ostream& out, const Counter& c)
    {
        return out << c.mcount;
    }
private:
```

```
    int mcount;
};
class CounterPtr {
public:
    CounterPtr(Counter* c) : mc(c) {}
    ~CounterPtr() {
        if(mc)
            delete mc;
    }
    Counter* operator->()const {
        return mc;
    }
    Counter& operator*() const {
        return *mc;
    }
private:
    Counter* mc;
};
```

Copy elision

- Derleyicin kullandığı bir optimizasyon tekniğidir.
- C++17 stadartları ile bazı durumlarda `mandatory copy elision` uygulanır.
- Eğer bir fonksiyonun parametresi bir sınıf türündense ve bu fonksiyon bir sağ taraf değeri sınıf nesnesi ile çağrılırsa copy elision uygulanır.

```
//RVO
Myclass func() {
    cout << "func cagrildi\n";
    return Myclass{};
}
int main() {
    Myclass x = func(); // mandatory copy elision uygulanır
}
```

```
//NRVO(Named Return Value Optimization)
//mandory değil debug modda çalışırsa optimizasyon uygulanmaz
Myclass func() {
    cout << "func cagrildi\n";
    Myclass m;
    cout << "func calismasina devam ediyor\n";
    return m;
}
int main() {
    Myclass mx = func();
    // Release modda sadece ctor çağrılır.
```

```
    // Fakat normalde func'ın içerisinde m nesnesi hayata gelir ve fonksiyondan
çıktığında
    // hayati biter geri dönüş değerinde ise move ctor çağrılması gerekirdi.
}
```

Boolean context

Logic operatörlerin operandları

- if parantezindeki ifade
- while parantezindeki ifade
- do while parantezindeki ifade
- for döngü deyiminin iki noktalı virgül arasındaki ifade

Complete \ Incomplete Type

Eğer derleyici bir kaynak kod dosyasında o sınıfın tanımını görüyorsa complete type, sadece bildirimini görüyorsa incomplete type'dır.

- Sınıfın veri elemanı incomplete type olamaz.
- Bir sınıfın kendi elemanından veri elemanı olamaz fakat statik veri elemanları kendi türünden olabilir.
- Sınıfın statik veri elemanları incomplete type olabilir.

Incomplete Type

- İşlev bildirimlerinde parametre veya geri dönüş değeri olarak kullanılabilir.
    - `A func(B,C*);`
- Typedef veya using bildirimlerinde
    - `typedef Myclass mcs;`
    - `typedef Myclass* mcsp;`
- Sınıfların static veri elemanları
    - `class A { static A a; }`
- Pointer değişkenler
    - `class B; class A { B* b; static A* a;}`
    - `class A; A* a;`
- sizeof operatörünün operantı olamaz

Complete Type

- Instantiation yapılacaksa yani bir nesne oluşturulacaksa
    - `class A; A a;`**//Geçersiz**
    - `class A{ }; A a;`

- sizeof operatörünün operandı
    - `class A{ }; sizeof(A)`

Incomplete type olarak tanımlama yapmanın en önemli nedeni başlık dosyalarıdır. Bir başlık dosyasının baka bir başlık dosyasını dahil etmesi durumunda birden fazla başlık dosyası eklenmiş olabilir. Biz istemediğimiz başlık dosyalarınıda include etmiş olabiliriz. Bu bizim compile time süremizi uzatmakla birlikte, asıl önemli olan bağımlılık oluşturmasıdır. Bağımlılığı azaltmak için incomplete type olarak tanımlanması gerekir.

Sınıfın Statik Veri Elemanları Ve Üye Fonksiyonları

- Sınıfın statik veri elemanları oluşturulacak tüm sınıf instance'ları için kullanılır.
- Static const(integral type) ile sadece sınıfta ilklendirme yapılır.

```
class Myclass {
    static double x = 13.3;     //Sentaks hatası
    static int y = 13;          //Sentaks hatası
    static const int z = 13;    //Geçerli
    static const float f = 13.f //Sentaks hatası
};
```

- C++17 ile inline variable eklendi böylelikle sınıf içerisinde ilklendirme yapılabilir.

```
class Myclass {
    inline static double x = 13.3;     //Geçerli
    inline static int y = 13;          //Geçerli
};
```

- Statik veri elemanları veya üye fonksiyonlarının tanımlarında `static` anahtar kelimesi olmayacaktır olursa sentaks hatası olur.

```
class Myclass {
    static int x;
    static void func(); // static member function "inline olarak tanımlanabilir."
    void foo();  // non-static member function
};
static void Myclass::func(){}// Sentak hatası
void Myclass::func(){}// Geçerli
int Myclass::x = 13; // geçerli
static int Myclass::x = 13 //Sentaks hatası
```

- Statik veri elemanları this pointer'i olmayan doğrudan bir sınıf nesnesi için çağrılmayan sınıf için çağrılan class scope erişim kontrolüne tabi sınıfın private üye değişkenlerine erişebiliyor.

```
class Myclass {
public:
    static void func();
    void foo();
private:
    int mx;
};

void Myclass::func() {
    mx = 13; //Sentaks hatası çünkü this pointer yok
    foo();   //Sentaks hatası çünkü this pointer yok
}

void Myclass::func() {
    Myclass m;
    m.mx = 13; //Geçerli
    m.foo();   //Geçerli
}

//Aşağıdaki ifadelerin tümü geçerlidir.
int main() {
    Myclass::func();
    Myclass mx,my,mz;
    Myclass* p{&mx};
    auto pd{new Myclass};
    Myclass& rm = my;
    mx.func();
    p->func();
    rm.func();
    mz.func();
}
```

Eğer sınıf türünden sadece dinamik ömürlü nesne oluşturmak istersek statik üye fonksiyonu tanımlamamız gerekir. Bunun statik olmasının nedeni doğrudan sınıf nesnesi oluşturmadan kullanmak ki zaten sınıfın kurucu üye fonksiyonu private kısmıma taşıyarak sadece bu statik üye değişkeni üzerinden nesneyi oluşturmak.

```
class Myclass {
public:
    static Myclass* createObject() {
        return new Myclass;
    }
private:
    Myclass();
};

Myclass m; //Sentaks hatası(Sınıfın constructor'ına erişim yok)
Myclass* n = Myclass::createObject(); //Geçerli
```

Singleton(tek nesne örüntüsü) günümüzde pek tercih edilmemektedir hatta bazen anti-patern olarakta görebilmekteyiz bunu sebebi çok fazla kod singleton olarak yazılıp sonradan bu sınıfın singletondan çıkması durumunda yaşanan kod karmaşasıdır.

```
class Myclass {
public:
    static Myclass& get_instance() {
        if(!smp) {
            smp = new Myclass;
        }
        return *smp;
    }
    void func();
    void foo();
    ///
private:
    Myclass();
    inline static Myclass* smp{nullptr};
};
```

Meyer's Singleton Class

```
class Myclass {
public:
    Myclass(const Myclass&) = delete;
    Myclass& operator=(const Myclass&) = delete;
    static Myclass& get_instance() {
        static Myclass singleton;
        return singleton;
    }
    void func();
    void foo();
    ///
private:
    Myclass();
};
```

if with initialization

- C++17 ile dile eklendi.
- Asıl kullanımı scope leakage(kapsam sızıntısı)'ın önüne geçmek içindir.

```
if(int val:func(); val > 10) {
    //code
    ++val;
    ////
}
```

Normalde üsteki fonksiyonu C++17'den önce `int val = func(); if(val > 10){++val}`şeklinde tanımlayabileceğimizi biliyoruz fakat buradaki `val` değişkenini sadece if koşul ifadesi içerisinde kullanacaksak gereksiz yere isimin kapsamını büyütmüş olacağız ki bunada `scope leakage` denilmektedir.

> patern ile idiom arasındaki fark patern genel dillerdeki kullanılabilen kalıplarken, idiom dile özgüdür. Örneğin singleton, C++ singleton, C# singleton, Java singleton... gibi örnekler paterne örnektir. C++ RAII ise bir idiom'dur sadece C++ diline özgüdür, genellik yoktur.

Inline Variable

- C++17 ile dile eklendi.
- ODR'a uyum sağlamak amacıyla sıklıkla kullanılır.
- Genel kullanımı header only dosyaları oluşturmak içinidir.

```
//a.h
inline int ival = 13;
class Myclass {
    inline static int x = 13;
    inline float f = 13.f;
};
```

Bir sınıf içerisinde;

1. Data Members
    - static data members
    - non-static data members
2. Member Function
    - static member function
    - non-static member function
        - const member function
        - non-const member function
3. Member Types

olacak şekilerde ifadeler tanımlanabilir.

Nested Type

```
class Myclass {
    class Nested {
        //...
    };
    Nested func();
    void foo(Nested);
};
```

```
class Encloser {
    static void func();
    int mx;
    class Nested {
        void foo() {
            func(); //Geçerli
            auto n = sizeof mx; //Geçerli
        }
    };
};

int main() {
    Myclass mx;
    Myclass::Nested retval = mx.foo(); //Geçersiz   1.
    auto retval2 = mx.foo();           //Geçerli    2.
}
```

1'in geçersiz 2'nin geçerli olması C++'ın kurallarından kaynaklanıyor. Auto ile belirsek hata değil fakat açık bir şekilde tanımlarsak hata olacakır.

Piml Idiom

Pointer implementation veya private implementation olarakta bilinir. Sınıfın private bölümünü gizlemeye yönellik geliştirilmiş bir idiom'dur. Ancak asıl kullanımı private bölümünü gizlemenin yanında bağımlığı azaltmasıdır. Böylelikle başlık dosyalarını, kaynak(`.cpp`) dosyasına ekleyeriz.

Fakat bu idiomun dezavantajı heap alanında nesne oluşturmak zorunda olduğumuz için maliyeti arttırmış olacağız. Normal statik alanda 1 maliyetle işlem yapmamıza karşın heap'te oluşturduğumuz nesne ile 10 belki 20 maliyet işlem yapıyor olabiliriz.

```
//myclass.h
class A;
class B;
class C;
class Myclass {
public:
    Myclass(A a, B b, C c);
    ~Myclass();
    A get_A();
```

```
    B get_B();
    C get_C();
private:
    class Implementation;
    Implementation* m_imp;
};
//------------------------------
//myclass.cpp
#include "A.h"
#include "B.h"
#include "C.h"

class Myclass::Implementation {
public:
    Implementation(A& _a, B& _b, C& _c) : a(_a), b(_b), c(_c)
    {
    }
    A a;
    B b;
    C c;
};

Myclass::Myclass(A a, B b, C c) {
    m_imp = new Implementation(a,b,c);
}

Myclass::~Myclass() {
    if(m_imp) {
        delete m_imp;
        std::cout << "m_imp deleted\n";
    }
}

A Myclass::get_A() {
    return m_imp->a;
}

B Myclass::get_B() {
    return m_imp->b;
}

C Myclass::get_C() {
    return m_imp->c;
}
```

Composition

Compotion ile oluşturduğumuz nesneler arasında `Has-a` relationship vardır. Örneğin Human adında bir sınıfımız var ve içerisinde Heart adında bir sınıf nesnesi barındırıyor olsun ne zamanki human nesnesi sonlandırılırsa o zaman Heart nesneside sonlanacaktır. Tipik gösterimi aşağıdaki gibidir.

```
#include "Heart"
class Human {
public:
    Human() {
        m_h = new Heart();
    }
    ~Human() {
        if(m_h)
            delete m_h;
    }
private:
    Heart* m_h;
};
```

Hiyerarşi aşağıdaki gibidir.

- association
    - aggregation
        - composition

Her composition bir aggregation, her aggregation da bir association'dır. Fakat bunun tersi doğru değildir.

Aggreagation

Aggreagation ile oluşturduğumuz nesneler arasında da `Has-a` relationship vardır. Fakat compotion'dan farklı olarak nesnenin hayatı sonlandığında sahip olduğu nesnenin'de hayatını sonlandırmaz. Örneğin, Team adında bir sınıfımız olsun ve içerisinde Player adında bir üye sınıf nesnesi tutuyor olsun. Aralarında has-a relationship var `Team has a player`, fakat team ile oluşturduğumuz nesnenin hayatı sonlandığında player hala hayatta olacaktır. Tipik gösterimi aşağıdaki gibidir.

```
class Team {
public:
    Team(Player& player) : m_p(player) {}
private:
    Player m_p;
};
```

Delegating Constructor

Delegating constructor ile birden fazla oluşturulan constructorlar arasında **kod tekrarı yapmamak** için kullanılır. Default member initializer ile kullanılır.

Modern C++'tan önce delegating constructor yardımcı fonksiyon kullanarak gerçekleştirilmeye çalışılırdı.

```
class Myclass {
public:
```

```
    Myclass() { init(); }
    Myclass(int a) { init(a); }
    Myclass(int a, int b) { init(a,b); }
private:
    int ma;
    int mb;
    void init(int a = 0, int b = 0) {
        ma = a;
        mb = b;
    }
};
```

Bu şekilde kod tekrarı yapmadık fakat üye değişkenleride ilk değer olarak başlatmışta olmadık. Atama ile değişkenleri oluşturduk. İşte tam da bu noktada delegating contructor bu sorunu çözmemizi sağlıyor.

```
class Myclass {
public:
    Myclass() : Myclass(0,0) { }
    Myclass(int a) : Myclass(a,0) { }
    Myclass(int a , int b) : ma(a), mb(b) { }
private:
    int ma;
    int mb;
};
```

Gördüğünüz üzere kod tekrarına düşmemekle birlikte ilk değer ile üye değişkenlerini başlatmış olduk.

Raw String Literal

Bir string literal C++'da bir ifade de kullanılacaksa sadece `const char*` olarak kullanılabilir. C'de ise bu `char[]` olarak tanımlanır. String sabitleri escape sequence'lar ile kullanılabilir.

Escape Sequence	Description
\'	single quote
\"	double quote
\?	question mark
\\	backslash
\a	audible bell
\b	backspace
\f	form feed
\n	line feed
\r	carriage return

Escape Sequence	Description
\t	horizontal tab
\v	vertical tab

```
#include <iostream>

int main() {

   const char* ch1 = "FooBar";
   const char ch2[] = "Foo\
Bar";
   char ch3[] = "Foo" "Bar";

   std::cout << ch1 << '\n';
   std::cout << ch2 << '\n';
   std::cout << ch3 << '\n';

   const char* str1 = "\nHello\n World\n";
   const char* str2 = R"foo(
Hello
World
)foo";
   const char* str3 = "\n"
                      "Hello\n"
                      " World\n";
   std::cout << str1 << str2 << str3;
}
// FooBar
// FooBar
// FooBar
// Hello
//  World

// Hello
//  World

// Hello
//  World
```

Namespaces

İsimlerin çakışmasını önlemek amacıyla kullanılır.

- Namespace bir namespace içerisinde olmalıdır.
- Local düzeyde namespace oluşturulamaz.

- Global namespace içerisinde namespace oluşturmak geçerli fakat main içerisinde namespace oluşturulamaz.

Scope kategorileri;

- Namespace scope → C'de file scope
- Class scope → C'de yok
- Block scope
- Function scope
- Function prototype scope

```
namespace ali {
    namespace veli {
        int x, y;
    }
}
ali::veli::x = 13;
ali::veli::y = 13;
```

> Asla başlık dosyalarında using bildirimi veya using namespace bildirimi yapmayın.

Eğer aynı isim alanı birden fazla kullanılırsa, derleyici bu isim alanları içerisindeki ifadeleri birleştirir. Bu kütüphaneleri farklı başlık dosyalarına ayrımak için oldukça faydalıdır. Örneğin statndart kütüphanede vector, string, array, bitset, map vb. gibi bir çok kütüphane `std namespace` isim alanı içerisindedir.

```
namespace enes {
    int x,y;
}

namespace enes {
    int a,b;
}

enes::a = 13; //Geçerli
enes::x = 13; //Geçerli
```

İç içe birden fazla isim alanı kullanılması durumunda;

```
namespace A {
    namespace B {
        namespace C {
        }
    }
}
```

bu şekilde kullanmak yerine;

```
namespace A::B::C {
}
```

Modern C++ ile yanyana yazılabiliyor.

İsimlerin nitelenmeden kullanabilmek için;

- using decleration
- using namespace decleration
- Argument Dependent Lookup(ADL)

bu 3'ünden biri olması gerekmektedir.

Using Decleration

Using bildiriminin bir spoce'u var ve bildirilen ismi o scope içerisine enjekte ediyor.

> Using bildirimi kullanılacaksa en dar scope'da kullanılmalı, eğer kullanılan kapsam yeterli değilse bir üst scope'da o da yeterli değil ise en son global düzeyde yapılmalı.

```
namespace A {
    int x;
}
using A::x;
void func() {
    x = 10;
}
int main() {
    x = 10;
}
```

C++17 ile birlikte using bildirimi ile artık birden fazla tanım yapılabilmektedir.

C++17'den önce `using A::x; using A::b;` şeklinde kullanılırken C++17 ile `using A::x, A::y` şeklinde kullanılabilmektedir.

Using Namespace Decleration

`using namespace` bildirimi kullanıldığı zaman, kullanılan namespace ismi sanki hiç yapılmamış gibi davranır.

```
#include <string>
#include <iostream>
class Myclass {
   void func() {
       using namespace std;
       string str = "Enes";
```

```
        cout << str << endl;
        ///
    }
};
```

using namepace bildirimi yapabilmek için bildirimi yapılacak isimin görünür olması gerekir ve bildirim ya local scope'da ya da bir namespace içerisinde yapılmalıdır.

```
class Myclass {
    using namespace std; // Geçersiz
    //using bildirimi ilk olarak görünür değil, görünür olsa bile class scope
içerisinde bildirilemez.
};
```

Argument Dependent Lookup(ADL)

Fonksiyona argüman olarak gönderilen ifade bir namespace içerisinde tanımlanan türlerden birine `ilişkinse` o zaman bu isim normal arandığı yerin dışında bu isim ait olduğu namespace `içinde de` aranır.

```
namespace enes {
    class Myclass {
        ///
    };
    void foo(Myclass);
    void func(int);
}
int main() {
    enes::Myclass mx;
    foo(mx);  //Geçerli
    func(12); //Geçersiz
}
```

Namespace içerisinde tanımlanan tür eş isimlerinde bir istisna, eğer tür o namespace içerisindeki bir tür eş ismi değilse o namespace içerisinde aranmaz.

```
namespace enes {
    enum Color { White, Red, Green}
    typedef int Word;
    typedef Color Ctype;
    void foo(Word);
    void func(Ctype);
}
int main() {
    enes::Word wx = 15;
    enes::Ctype cx = enes::White;
    foo(wx);  //Geçersiz
```

```
    func(cx); //Geçerli
}
```

Programlamaya ilk giriş kodlarında sıklıkla kullanılan Hello World programında ADL var mıdır?

```
int main(){
    std::cout << "Hello World";
    operator<<(std::cout,"Hello World");
    //görüldüğü üzere std::cout operator left shift fonk. argüman olarak
gönderildi.
    //Böylece operator left shift std isim alanında da arandı ve bulundu.
}
```

Peki `std::cout << "Hello World" << endl;` geçerli midir?

```
int main() {
    std::cout << "Hello World" << endl; //Geçersiz
    operator<<(std::cout,"Hello World").operator<<(endl); //Geçersiz
}
```

Inline Namespace

İç içe namespacelerde en içteki isim alanını bir üstteki isim alanına görünür yapabilmek için using namespace bildirimi yapmak dilin sentaksı açısından bir problem gibi görünmesede hatadır. Bu C++ dilinin çelişkili bir yapısıdır.

```
namespace A {
    namespace B {
        namespace C {
            int x;
        }
        using namespace C;
    }
}
A::B::x = 13; //Geçersiz çünkü C B'de görünür olmuyor
```

Fakat bu yapıyı inline namespace oluşturabiliyoruz.

```
namespace A {
    namespace B {
        inline namespace C {
            int x;
        }
    }
```

```
}
A::B::x = 13; //Geçerli
```

Namespace Alias

```
namespace Enes {
    int x,y;
}
namespace pro = Enes;
pro::x = 13;
pro::y = 13;
```

External linkage: Bir isim birden fazla kaynak dosyasına bağlanıyor fakat aynı varlığı gösteriyorsa external linkage aittir.

> Eğer bir ismi iç bağlantıya almak istiyorsanız neyin ismi olursa olsun bir isimsiz isim alanına alın.

```
namespace {
    int x = 13;
    void func(int) {}
}
int main() {
    x = 20;
    func(x);
}
```

C'de normalde bunu static olarak tanımlayarak iç bağlantıya alıyorduk bu C++'ta da geçerli fakat isimsiz isim alanına almak kodu daha derli toplu gösterir. C++17 ile statik fonksiyon tanımı `deprecated` edildiği için zaten mecburen iç bağlantıya dahil etmek istediğimiz statik fonksiyonları isimsiz isim alanlarına almak zorundayız.

String Sınıfı

String bir tür eş ismidir. std::basic_string<> şablonu ile tanımlanır.

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT>,
              typename Allocator = allocator<charT> >
              class basic_string;
}
namespace std {
    typedef basic_string<char> string;
}
```

String sınıfı verileri heap'te tutar. Fakat küçük yazılar ortalama bir yazıda çok fazla kullanıldığı için `Derleyiciler` string sınıfının veri elemanını heapte tutmak yerine '16-20' arasında derleyiciden derleyiciye değişmekle birlikte bir char dizi elemanında tutarlar. Bu optimizasyon tekniğine `SBO(Small Buffer Optimization)` veya `SSO(Small String Optimization)` denir.

```
cout << "sizeof(int*) = " << sizeof(int*) << '\n';
cout << "sizeof(string*) = " << sizeof(string*) << '\n';
```

Arguments	**Interpretation**
const string & str	The whole string str
const string & str, size_type idx, size_type num	At most, the first num characters
const char* cstr	The whole C-string cstr
const char* chars, size_type len	len characters of the character array chars
char c	The character c
size_type num, char c	num occurrences of character c
const_iterator beg, const_iterator end	All characters in range

Expression	**Effect**
string s	Creates the empty string s
string s(str)	Copy constructor; creates a string as a copy of the existing string str string
s(rvStr)	Move constructor; creates a string and moves the contents of rvStr to it (rvStr has a valid state with undefined value afterward)
string s(str,stridx)	Creates a string s that is initialized by the characters of string str starting with index stridx
string s(str,stridx,strlen)	Creates a string s that is initialized by, at most, strlen characters of string str starting with index stridx
string s(cstr)	Creates a string s that is initialized by the C-string cstr
string s(chars,charslen)	Creates a string s that is initialized by charslen characters of the character array chars
string s(num,c)	Creates a string that has num occurrences of character c
string s(beg,end)	Creates a string that is initialized by all characters of the range [beg,end)
string s(initlist)	Creates a string that is initialized by all characters in initlist (since C++11)
s.~string()	Destroys all characters and frees the memory

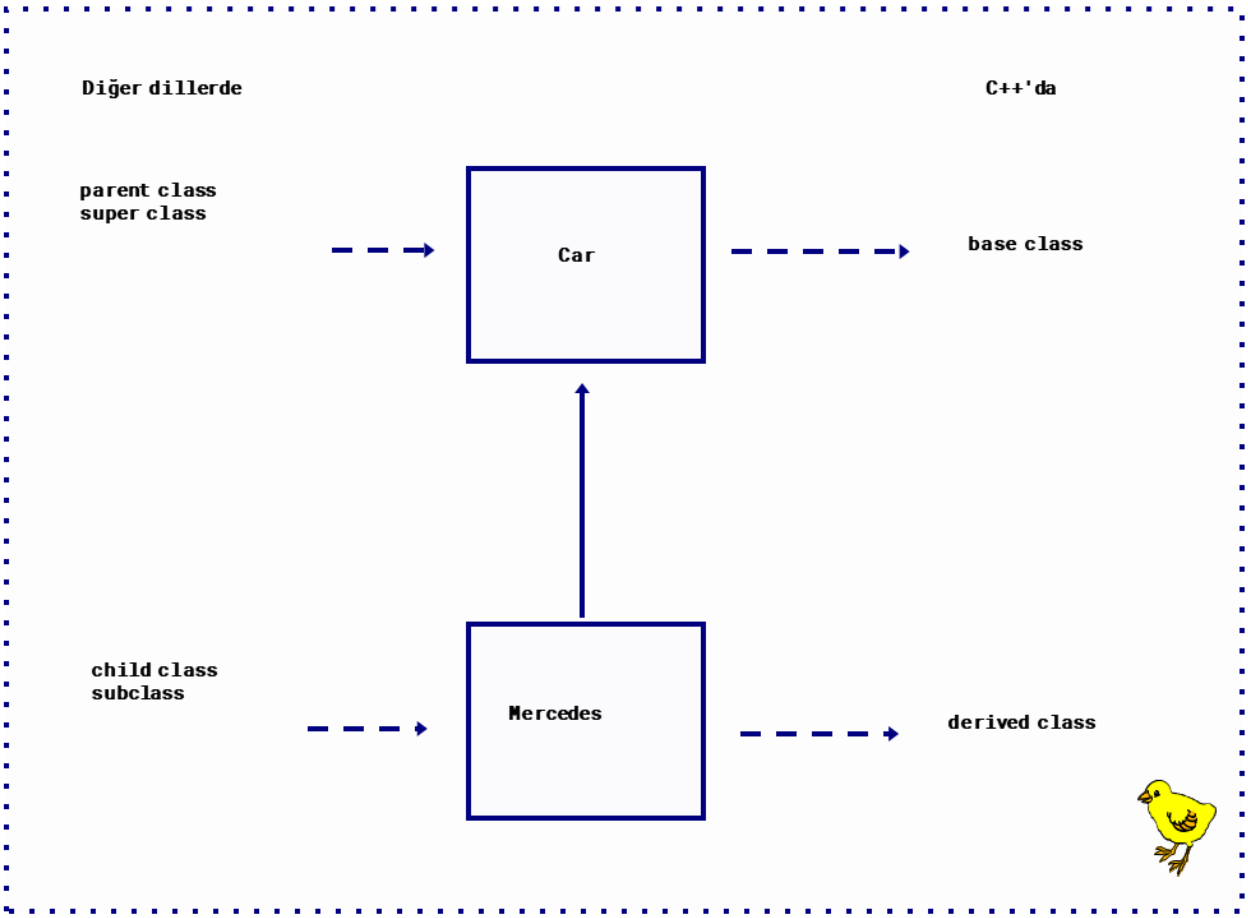
```
int main() {
    string str("Cansu Uca",5); //Data ctor
```

```
    cout << str << '\n';
    string str2("Enes Alp",5,string::npos); // substring
    cout << str2 << '\n';
    string str3(20,'x') // Fill
    cout << str3 << '\n';
    string str4 = {'E','n','e','s',' ','A','l','p'}; // initializer list
    cout << str4 << '\n';
    string str5("Enes Alp"); //cstring
    cout << str5 << '\n';
}
```

Eğer substring parametre söz konusu olduğunda string'te tutulan dizinden daha büyük parametre istenildiğinde string içindeki tüm parametreleri alır 'Undefined Behaviour' olmaz. Fakat bu Date ctor için yapılırsa 'UB' olur.

Inheritance

Aynı arayüzü destekleyen farklı sınıfların aynı türdenmiş gibi kullanılabilmesini sağlayan ve eskiden yazılmış kodları daha sonra yazılacak kodların kullanmasını sağlayan bir mekanizmadır.



C++'da kalıtım

- public inheritance %80 → Java,C#...
- private inheritance

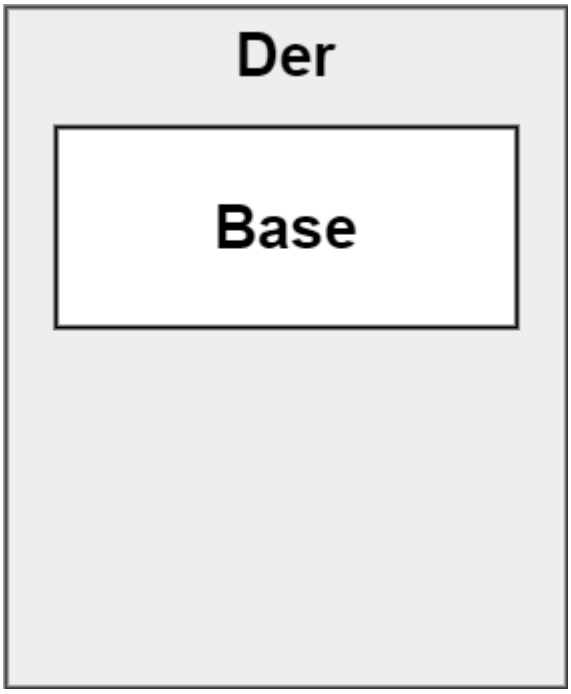
- protected inheritance

olmak üzere 3 farklı şekilde yapılabilir.

```
class Base {
    //...
};

class A : public Base { }; //public inheritance
class B : private Base { }; //private inheritance
class C : protected Base { }; //protected inheritance
class D : Base { }; //private inheritance
struct E : Base { }; //public inheritance
```

Kalıtım tıpkı composition'da olduğu gibi bir sınıf nesnesinin içinde fiziksel olarak başka bir sınıf nesnesi vardır. Ancak composition'da bu içerilen nesneye `member object` denilir. Kalıtımda ise içerilen nesne `base class object` denilir.



Her `der` nesnesi içerisinde bir `base` nesnesi vardır.

```
class Base {
public:
    void func(int) {}
};
class Der : public Base {
public:
    void func() {}
};
int main()
{
    Der myder;
```

```
    myder.func(); //gecerli
    //myder.func(10); //gecersiz
    myder.Base::func(10); //gecerli
}
```

Sentaks hatası olmasının sebebi `namelookup`'tır. İsim arama ile isim ilk olarak 'der' sınıfı içerisinde arandı ve bulundu(bulunduktan sonra ise base içerisinde de aramam yapılmaz çünkü isim arama bir kere yapılır bulunursa sona erer daha sonra context kontrol yapılır). Context kontrol ile `func()` fonksiyonun int parametresi olduğu anlaşıldı ve senktaks hatası oldu. Lakin base'in `func()`fonksiyonu çağrılmak isteniyorsa niteleme yaparak çağrılır(myder.Base::func()).

Namelookup bir kez yapılır bulunması ile sona erer. Türetilmiş sınıflarda ilk olarak isim arama, kullanılan sınıfın içinde bulunmazsa base sınıf içerisinde aranır.

```
void foo(int,int);
class Base {
public:
    void foo(int);
};
class Der: public Base {
    void foo();
public:
void func() {
    foo();
    Base::foo(12);
    ::foo(12,13);
}
};
```

Der'deki `func()` fonksiyonu içerisinde isim arama şu şekilde gerçekleşir. İlk olarak fonksiyon Der sınıfı içerisinde aranır bulunmazsa global'de değil Base içerisinde daha sonra globalde aranır.

Arama sırası şu şekildedir;

1. Namelookup
2. Context Control
3. Access Control

- Türemiş sınıf türünden taban sınıf türüne yapılan dönüşüm upcasting(yukarı doğru dönüşüm) denir.
- Taban sınıfı türünden bir nesneye, türemiş sınıfından bir nesneye atanmaması ve kopyalanmaması gerekir. Akis halde object slicing meydana gelir.
- Her türemiş sınıf nesnesi hayata geldiğinde onun içinde yer alan taban nesnesi hayata gelir.

C++'da kalıtım nesne yönelimli programlamadaki kalıtımı kapsamakla birlikte bunun dışında farklı amaçlarlada kullanılıyor.

> Incomplete type ile kalıtım oluşturulmaz.

3 ayrı kalıtım vardır.

- public
- private
- protected

Türemiş sınıfın constructor'unda eğer biz constructor initializer list ile yazmazsak derleyici her zaman taban sınıfın default ctor'una çağrı yapacak şekilde kod üretecektir.

```
Constructor initializer list ile;
    İlk taban sınıf alt nesnesi
        Daha sonra bildirimdeki sırayla
        : elemanlar için ctor çağrılacaktır.
```

Türemiş sınıftan, taban sınıfa doğru otomatik dönüşüm vardır.

Kalıtımda özel üye fonksiyonlar

Default Constructor

Derleyici taban sınıfın default constructor'una çağrı yapacak şekilde kod üretir.

Eğer bu çağrı;

a) Base'in default constructor'u olmaması

b) Base'in default constructor'u olması fakat private, çağrılamayacak statüde olması

c) Base'in default constructor'u olması fakat delete edilmiş olması

durumlarında ise derleyici default constructor'u `delete` eder.

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

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

```
};

int main()
{
    Der myder;
}
// Output
// Base default ctor
// Der default ctor
```

Copy Constructor

```
Der(const Der& other) : Base(other) {}
```

Move Constructor

```
Der(Der&& other) : Base(std::move(other)) {}
```

Copy Assignment

```
Der& operator=(const Der& other) {
    Base::operator=(other);
    return *this;
}
```

Move Assignment

```
Der& operator=(Der&& other) {
    Base::operator=(std::move(other));
    return *this;
}
```

Taban Sınıfın Üye Fonksiyonu

1. Türemiş sınıflara hem bir arayüz(interface) hem de implementasyon verebilir.
2. Türemiş sınıflara hem bir arayüz hem de default implementasyon verebilir(virtual).
3. Türemiş sınıflara bir arayüz vermez ancak implementasyon verebilir(pure virtual).

```
class Airplane {
public:
    void takeof();          //1. durum
```

```
    virtual void fly();       //2. durum
    virtual void land() = 0; //3. durum
};
```

Eğer bir sınıf en az bir 'virtual' fonksiyon(2. ve 3. durum) içeriyorsa böyle sınıflara ve böyle sınıflardan kalıtım yoluyla oluşturulan sınıflara `polymorphic class(çok biçimli sınıf)` denir.

Eğer en az bir 'pure virtual' fonksiyonu(3. durum) varsa böyle sınıflara `abstract class(soyut sınıf)` denir.

> Abstract sınıflardan nesne oluşturulmaz.

Virtual Dispatch(Sanal Gönderim)

Bir fonksiyon çağrısı

- static binding ~ early binding (compile time)
- dynamic binding ~ late binding

olmak üzere iki farklı ele alınır. Python bir dyanmic binding proglama dilidir. Yani tür run time'da çıkarılır. C, C++, C#... gibi diller ise static binding'tir. C++ aynı zamanda dynamic binding mekanizmasına sahiptir(RTTI).

C++11 ile dile

- override
- final

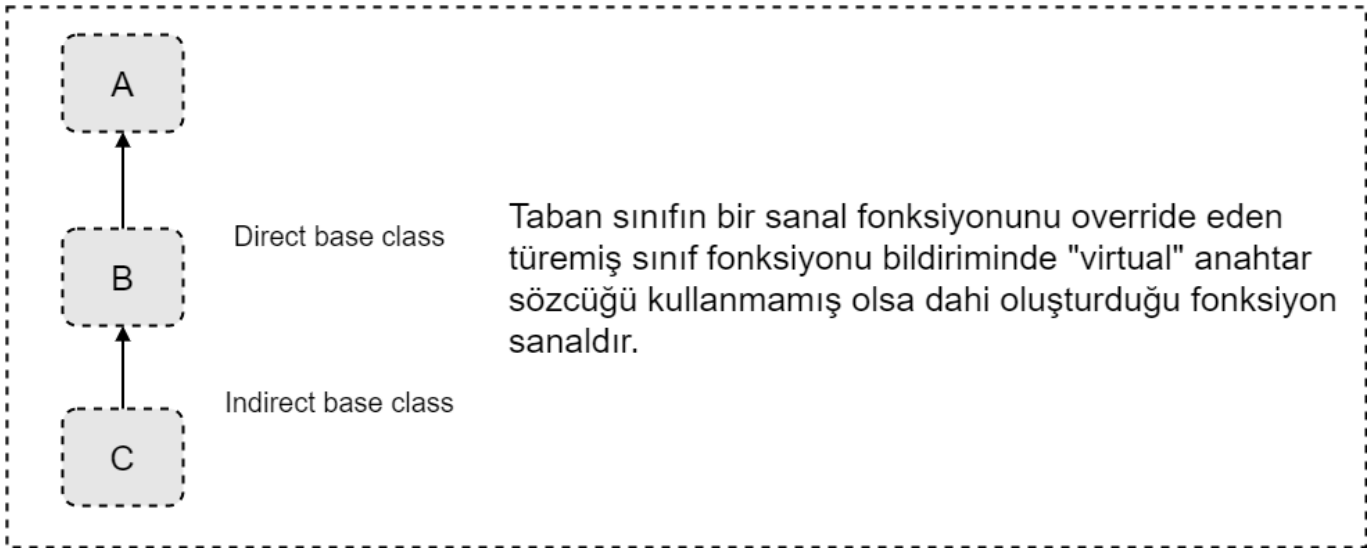
olmak üzere 2 adet contextual keyword(bağlamsal anahtar sözcük) eklendi.

Contextual Keyword: Belirli bir bağlamda kullanıldığı zaman anahtar sözcük etkisi yaparken o bağlamın dışında kullanıldığı zaman ise identifier olarak kullanılabilir.

> Türemiş sınıfın, taban sınıfın bir sanal fonksiyonu ile aynı imzaya sahip ancak geri dönüş değeri türü farklı bir fonksiyon bildirilmesi sentaks hatasıdır.

Object Slicing(Nesne Dilinlenmesi)

Eğer bir sanal fonksiyona yapılan çağrı taban sınıf pointer'ı veya referansı ile yapılıyorsa sanal gönderim mekanizması devreye girer. Fakat çağrı taban sınıf nesnesi ile yapılıyorsa sanal gönderim mekanizmaı devreye girmez ve buna `object slicing` denir(Derleyici bu işlevi derleme zamanında taban sınıf fonksiyonuna bağlar).



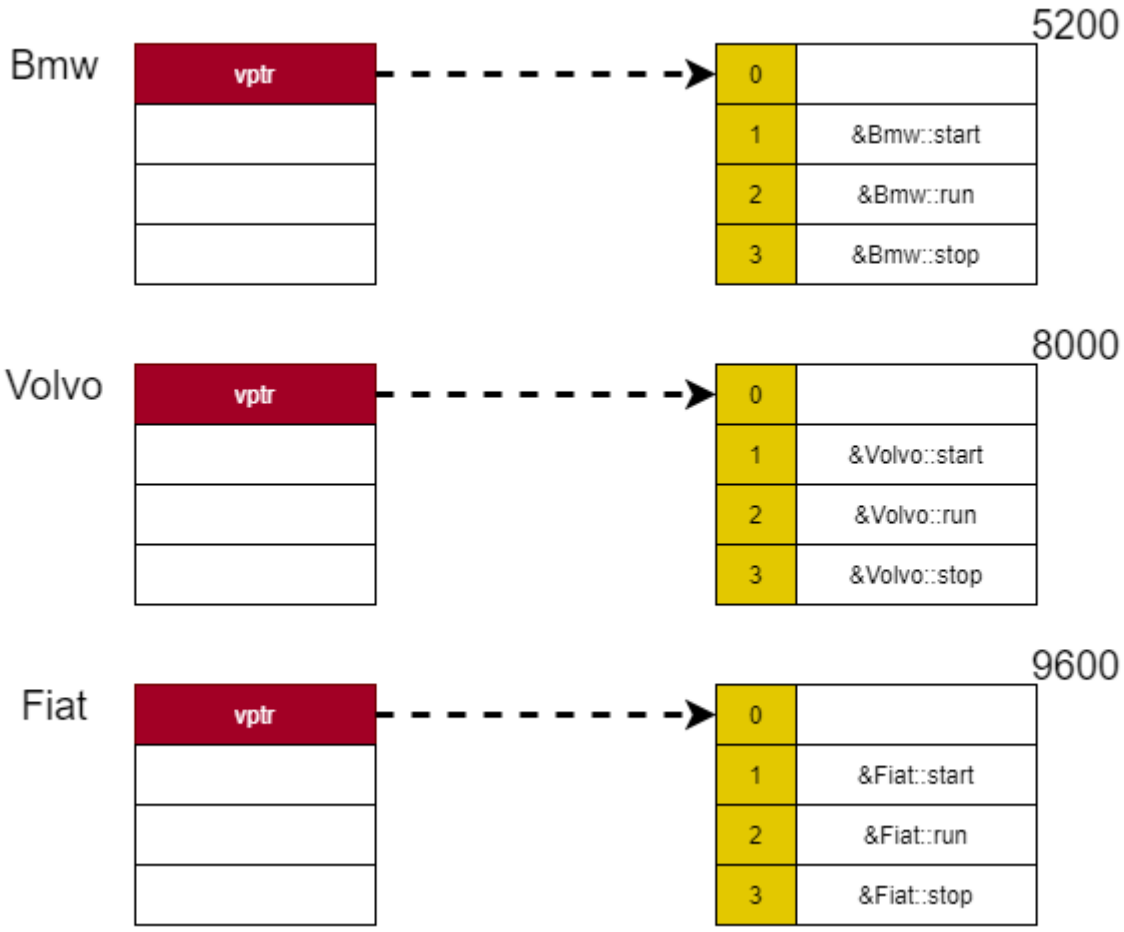
> Bir taban sınıfın üye fonksiyonunu türetilmiş sınıfın private bölümünde oluşturabiliriz.

NVI(Non-Virtual Ineterface Idiom)(Sanal Olmayan Arayüz): Bu idiyom bir taban sınıfın sanal işlevlerinin public olması yerine private ya da protected yapılmasına, taban sınıfın tüm public arayüzünün sanal olmayan işlevler ile oluşturulmasına dayanıyor. Eğer taban sınıfın bir sanal işlevi türemiş sınıflar tarafından çağrılmayacaksa sınıfın private öğesi, türemiş sınıflar tarafından çağrılacak ise sınıfın protected öğesi yapılıyor. Sınıfın public arayüzünde yalnızca sanal olmayan işlevler bulunuyor.

Sanal Gönderimin(Virtual Dispatch) devreye girmediği durumlar;

- Taban sınıfın ctor'u içinde yapılan sanal fonksiyon çağrıları sanal gönderime tabi tutalmaz.

Sanal Fonksiyonlar Nasıl İmplemente Edilir?



Bir Car sınıfımız olsun ve bu Car sınıfından Bmw, Fiat, Volvo sınıflarını kalıtım ile oluşturalım.

```
class Car {
public:
    virtual void start() = 0;
    virtual void run() = 0;
    virtual void stop() = 0;
};
```

//Açıklama eklenecek.

Sanal fonksiyonların oluşturulmasıyla ekstra maliyet olur mu?

1. İşlemci maliyeti
2. Bellek maliyeti

Virtual Constructor

C++ dilinde bir sınıfın constructor'u virtual anahtar sözcüğü ile bildirilemez.

Ancak genel olarak nesne yönelimli programlamada bir sanal constructor ihtiyacı söz konusudur.

Clone Idiom

Bir nesnenin türünü programın çalışma zamanında belirlemesini istiyoruz.

Run time'da Car* tipinde parametre alan fakat gönderilen sınıf nesnesinin Car'dan türetilmiş bir sınıf olduğunu fakat hangi tür olduğunu öğrenmek istediğimiz zaman kullanılacak bir idiom'dır.

```
void car_game(Car* p) {
    p->start();
    Car* pnew = p->clone();
    pnew->start();
}
```

Virtual Destructor

Eğer base'in destructor'u virtual olmaz ise türemiş sınıf nesnesinin dinamik olarak oluşturup bunun adresini taban sınıf pointer'ına atarsak(upcasting)

```
Base* baseptr = new Der;
```

fakat nesnenin görevi bitip delete edilirse, çağrılan sadece base'ın destructor'u olacaktır. Der'in destructor'u çağrılmayacak kaynaklar geri verilmeyecektir.

Bir diğer problem ise new ile oluşturulan türemiş sınıf nesnesi bellekte yer ediniyor daha sonra delete ederken önce base'in destructor'u daha sonra C'deki free gibi olan operator delete() fonksiyonuna operator new ile elde edilen bellekteki adresi gönderiyor. Böylece o bellek alanı geri verilmiş oluyor. `delete baseptr;` taban sınıf pointeri ile taban sınıf pointerini delete operatorunun operandı yaparsanız derleyici operator delete fonksiyonuna taban sınıf nesnesinin adresinini geçecek. New operatörü geri dönüş değeri olarak oluşturulan nesnenin adresini döndürür. `baseptr` bu adresi gösterir fakat delete edilirken baseptr sınıf adresi ile türemiş sınıf adresi ancak aynı yeri gösterirse doğru sonuç olur. Bu da bir zorunluluk olmadığı için `tanımsız davranışa` sebep olacaktır.

Eğer türemiş sınıf nesnesini bir taban sınıf pointeri ile işlemek gibi bir niyetimiz yoksa orta da bir problem yok fakat eğer böyle bir durum söz konusu ise her zaman taban sınıf destructor'unu protected yapın ki yanlışlıkla client bir dinamik türemiş sınıf nesnesini taban sınıf pointerı ile yönetmeye kalkarsa setaks hatası olsun.

```
class Base {
public:
    Base() {}
protected:
    virtual ~Base() {}
};
Base* baseptr = new Der;
delete baseptr; //Sentaks hatası base destructor protected
```

Sonuç olarak;

Taban sınıfların destructor'u ya public virtual olacak ya da protected non-virtual olacaktır.

> Gloabal fonksiyonlar ve sınıfın statik üye fonksiyonları sanal olamaz.

Variant Return Type(Covariance)

Eğer taban sınıfın sanal fonksiyonlarının geri dönüş değeri türü Base* ise türemiş sınıf bu fonksiyonları override ederken geri dönüş değerini Base* yerine Der* yapabilir(eğer Der Base'in türemiş sınıfı ise).

Sadece taban sınıf türünden pointer veya referansı ise geçerlidir.

```
class Base {
public:
    virtual Base* func1();
    virtual Base& func2();
    virtual Base func3();
};

class Der : public Base {
public:
    virtual Der* func1() override; //Geçerli
    virtual Der& func2() override; //Geçerli
    virtual Der func3() override; //Geçerli
    //virtual Der func3() override;  //Geçersiz
};
```

Bu şekilde kullanımı kod yazımını kolaylaştırmaktadır.

Sınıf İçi Using Bildirimi

```
class Base {
public:
    void func(int x) {
        cout << "Base::func(int x) x = " << x << '\n';
    }
};
class Der : public Base {
public:
    void func(double x) {
        cout << "Der::func(double x) x = " << x << '\n';
    }
};
int main() {
    Der myder;
    myder.func(13);  // Der::func(double x)
    myder.func(.13); // Der::func(double x)
}
```

Görüldüğü üzere namelookup ile Der sınıfı içerisindeki `func()` fonksiyonu çağrılır. Fakat biz int olan parametrenin base'in içindeki `func()` fonksiyonu çağırmak istersek;

ya Der içersinde `func()` fonksiyonunu overload ederiz.

```
void func(int x) {
    Base::func(x);
}
```

Bu şekilde bir fonksiyon overloading ile oluşturacağız. Ya da daha iyi bir yöntem, sınıf içinde using anahtar sözcüğü ile bir bildirim yaparsak taban sınıftaki ismin doğrudan türemiş sınıf içinde bilinmesini(visible olmasını) sağlayabiliriz. Yani taban sınıftaki ismi türemiş sınıfın scope'ına enjekte edebiliriz.

```
class Der : public Base {
public:
    using Base::func();
    void func(double x) {
        cout << "Der::func(double x) x = " << x << '\n';
    }
};
```

Bir diğer örnek,

```
class Base {
public:
    Base(int);
    Base(int,int);
    Base(double,double);
};
class Der : public Base {
public:
    using Base::Base;
};
int main() {
    Der myder1(13);
    Der myder1(13,14);
    Der myder1(1.3,.45);
}
```

Der sınıfı içersinde constructor oluşturmamamıza rağmen derleyici using bildirimi ile bunu bizim için gerçekleştirdi.

Fakat Base'in constructor'ları protected yapılması durumunda sentaks hatası olur.

Final Keyword

Tıpkı override anahtar sözcüğünde olduğu gibi bu da bir contextual keyword'tür.

İki ayrı anlamda kullanılır.

- final class
- final override

final class

```
class Base {};
class Der final : public Base {};
class SDer : public Der {}; // sentaks hatası Der final'dır
```

final override

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

Eğer bir override ettiğimiz fonksiyon sizden kalıtım yolu ile elde edilecek sınıflar tarafından override edilmesini istemiyorsanız `final` bağlamsal anahtar sözcüğünü kullanmalısınız.

Private Inheritance

Private kalıtımı eleman(containment) olarak composition'a bir alternatif olarak kullanılır.

```
class A {
public:
    void fa();
private:
    void print();
};

class B {
A ax;
public:
    void fa() {
        ax.fa();
    }
    void foo() {
        //ax.print(); //geçersiz access control hatası
    }
};
```

Bu örnekte sırasıyla;

- A sınıfın public interface'i B'ye eklenmedi.
- B sınıfın public interface'ini seçerek B'ye ekleyebiliriz.
- B sınıfı A sınıfının private bölümüne erişemez.

```
class  B : private A {} // private
class  B : A {}         // private
struct B : A {}         // public
```

- private kalıtımı aslında içerme yoluyla(containment) composition'a bir alternatiftir. Ancak containment her zaman kalıtıma göre çok daha az bir bağımlılık sağlar.

Hangi durumlarda private kalıtımı yapılır?

1. elemanın sanal fonksiyonunu override etmeniz gerekiyorsa
2. elemanın protected bölümüne erişmeniz gerekiyorsa
3. eleman taban sınıf türüne otomatik dönüşüm olması gerekiyorsa

> private kalıtımında is-a ilişkisi yoktur. Base dönüşüm otomatik olmaz.

Restricted Polymorphism

Belirli fonksiyonların polymorphism'den faydanlamasını istiyebiliriz. Bunun için türetilmiş sınıfı private kalıtımı yapıyoruz. Virtual dispatch mekanizmasından yaralanmasını istediğimiz global fonksiyona friendlik veriyoruz.

```
class Base {
public:
    virtual void vfunc();
};
class Der : private Base {
public:
    void vfunc() override;
    friend void foo();
};
void foo();
```

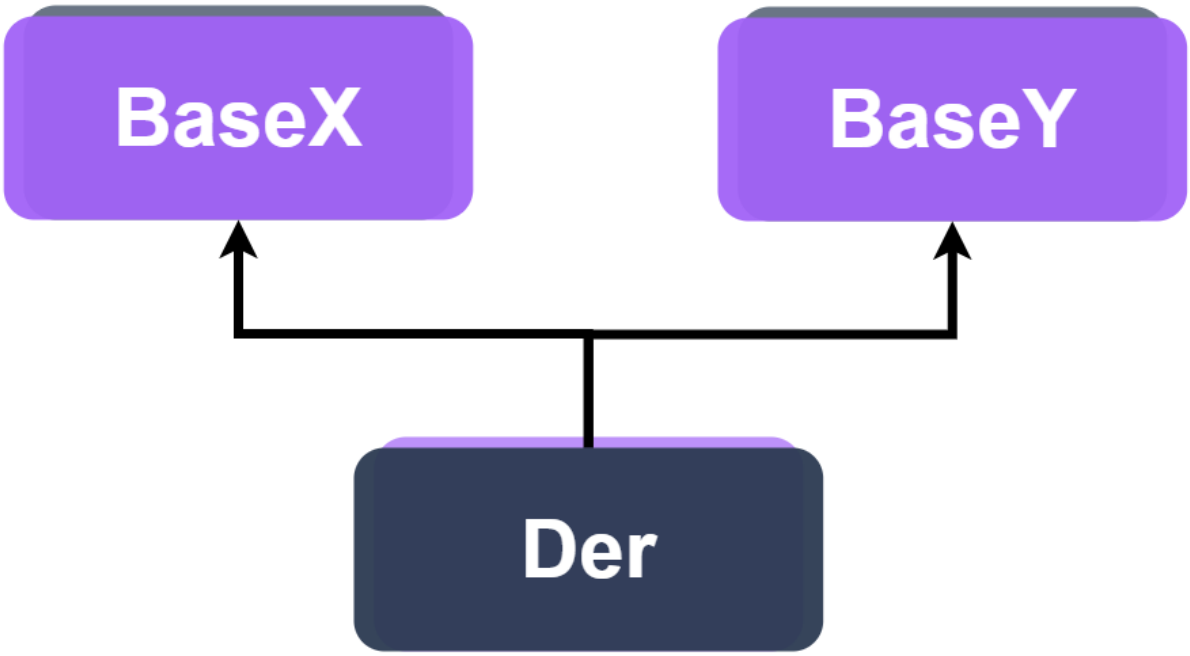
Protected Kalıtımı

Private kalıtımı ile arasındaki tek fark taban sınıfın protected fonksiyonlarına erişmesini istiyorsak istiyorsak o zaman protected kalıtımı tercih edilir.

```
class Base {
public:
    void f1();
```

```
    void f2();
};
class Der : protected Base {
    void fder() {
        void f1();
        void f2();
    }
};
class Myder : public Der {
public:
    void myder() {
        void f1(); //Geçerli fakat Der Base'den private kalıtımı yapsaydı hatalı
olacaktı
    }
}
```

Multiple Inheritance




```
class BaseX {
public:
    void func1();
    void func2();
};
class BaseY {
public:
    void foo1();
    void foo2();
};
class Der : public BaseX, public BaseY {};
```

> Eğer çoklu kalıtım `class Der : public BaseX, BaseY {};` şeklinde yapılsaydı `BaseY` private kalıtımı olmuş olacaktı. Yani virgülle ayrılmış olması aynı aynı kalıtımı yapacağı anlamı taşımamaktatır.

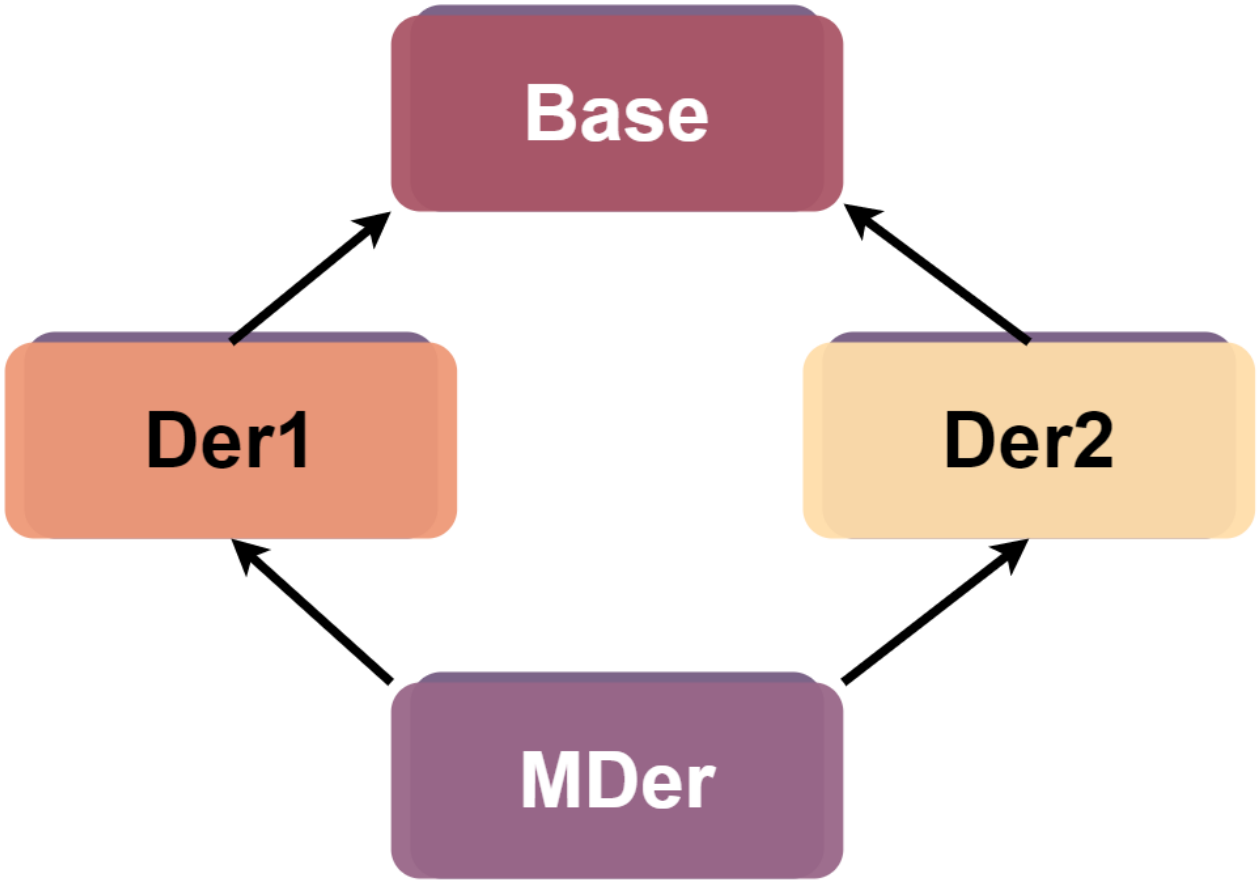
Çoklu Kalıtımda İsim Arama

```
class A {
public:
    void func(int);
};
class B {
public:
    void func(double);
};
class C : public A, public B {};
int main() {
    C cx;
    cx.func(13.5); //Sentaks hatası(ambiguity)
}
```

Çoklu kalıtımda isim arama söz konusu olduğunda `.` operatörünün sağındaki isim ilk olarak türemiş sınıfın scopunda aranır eğer bulunmazsa taban sınıfları içeriside belirli bir sıraya göre aranmıyor `aynı şekilde` aranıyor yani sanki bunlar tek bir scope içerisindeymiş gibi yorumlanıyor fakat scopeları farklı olduğundan dolayı eğer aynı isim kullanılıyorsa function overloading olamayacak `ambiguity` olacaktır. Böyle bir durumda eğer ismi niteleyerek çağrı yaparsak(`cx.B::func(13.5)`) hata olmayacaktır.

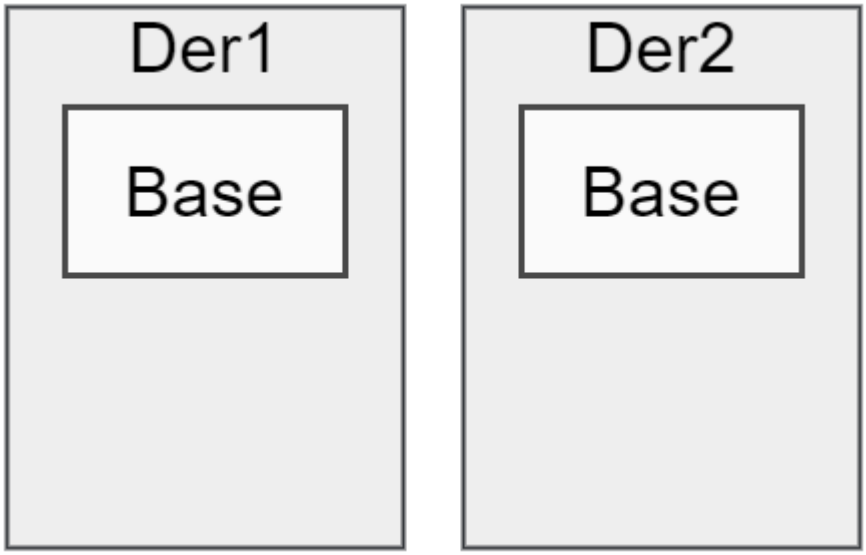
Diamond Formation(Karo/Elmas Formasyonu)

Bir diğer ismi DDD(Dreaful Diamond of Derivation) olarakta bilinir.



Bu formasyonda 2 tane problemler var.

1. Der1 ve Der2'den gelen 2 adet Base nesnesi olacaktır.




```
class Base {};
class Der1 : public Base {};
class Der2 : public Base {};
class MDer : public Der1, public Der2 {};
```

> Diamond formasyonu sadece compile time'a dayalı bir problem değil run time'a yönelikte bir problemi vardır.

2. Modele göre 1 tane taban nesnesi olması gerektiğinden fakat iki tane dolaylı taban nesnesi olduğundan run time'a yönelikte probleme yol açar. 1. türetilmiş sınıf içerisindeki taban nesnesi değiştirildiğinde 2. türetilen sınıf nesnesininde değişmesi gerektiğinden fakat değişmemesinden sorun olur. Taban sınıf nesnesi 1 tane olmalı bunun için de `virtual inheritace` kullanılır.

Virtual Inherintance

Eğer iki sınıftan çoklu kalıtım ile 1 sınıf elde edecekseniz. Elde edilen sınıfların taban sınıflarının 1 tane olmasını istediğiniz zaman virtual inherirance kullanılır.

```
class Device {
public:
  virtual void turn_on() {
    _is_on = true;
    cout << "Cihaz acildi\n";
  }
  virtual void turn_off() {
    _is_on = false;
    cout << "Cihaz kapatildi\n";
  }
  void run() {
    if(is_on()) {
      start();
    } else {
      cout << "cihaz kapali\n";
    }
  }
  bool is_on()const { return _is_on; }
protected:
  bool _is_on;
  virtual void start() = 0;
};

class Fax : virtual public Device {
public:
  void send_fax(){
    if(!is_on()) {
      cout << "cihaz kapali oldugundan gonderilmedi\n";
    }else {
      cout << "fax gonderildi\n";
    }
  }
  void receive_fax() {
    if(!is_on()) {
      cout << "cihaz kapali oldugundan gonderilmedi\n";
    }else {
      cout << "fax alindi\n";
    }
  }
```

```
private:
  void start()  override {
    cout << "Fax calistirildi\n";
  }
};

class Modem : virtual public Device {
public:
  void send_data() {
    if(!is_on()) {
      cout << "cihaz kapali oldugundan gonderilmedi\n";
    }else {
      cout << "data gonderildi\n";
    }
  }
  void receive_data() {
    if(!is_on()) {
      cout << "cihaz kapali oldugundan gonderilmedi\n";
    }else {
      cout << "data alindi\n";
    }
  }
private:
  void start() override {
    cout << "Modem calistirildi\n";
  }
};

class FaxModem : public Fax, public Modem {
private:
  void start() override {
    cout << "FaxModem calistirildi\n";
  }
};
int main() {
  FaxModem fm;
  fm.turn_on();
  fm.run();
  fm.send_fax();
  fm.send_data();
  fm.turn_off();
}
```

Exception Handling

Programın çalışma zamanında oluşacak hataların yönetilmesidir.

Bir fonksiyon yüklendiği işi yapamayacağını anlarsa(saptarsa) ne yapmalı?

C'de iş gören kod ile hata işleyen kod iç içedir. Buda algoritma ile hata kontrolünün iç içe girmesine sebep olur. Bu iki farklı logic yapı okumayı, yazmayı, test etmeyi her şeyi zor duruma sokmaktadır. C++'da bu sıkıntılardan kurtulmak için exception handling mekanizması devreye girmektedir.

Hataya müdahale edecek kod

1. resumptive(hata yakalanacak fakat program sonlanmadan programın devamını sağlar.)
2. terminative(program sonlanır.) olmak üzere iki şekilde ele alınır.

```
try {}
catch(int x) { /*execption handler*/ }
```

Eğer bir hata oluşur ve bu oluşan hata yakalanmazsa("uncought exception") program `terminate` fonksiyonunu çağırır ve program sonlandırılır. Buda kayıplara yol açabilir. Bu yüzden her zaman exception yakalanması gerekir.

```
typedef void(*terminate_handler)(void);
//using terminate_handler = void(*)(void);

terminate_handler set_terminate(terminate_handler);
```

set_terminate' e eğer bir fonksiyon adresini geçerseniz std::terminate fonksiyonu bizim oluşturduğumuz fonksiyonu çağıracaktır.

```
void f1() {
    throw 1;
}
void my_terminate() {
    cout << "terminate cagrildi\n";
    cout << "my_terminate cagrildi\n";
    cout << "abort cagrildi\n";
    abort();
}

int main() {
    set_terminate(my_terminate);
    f1(); // Hata yakalanmadı terminate cağrıldı o da my_terminate'i çağırdı.
}
```

Statndart kütüphanenin kendi hata sınıf hiyerarşisi vardır.

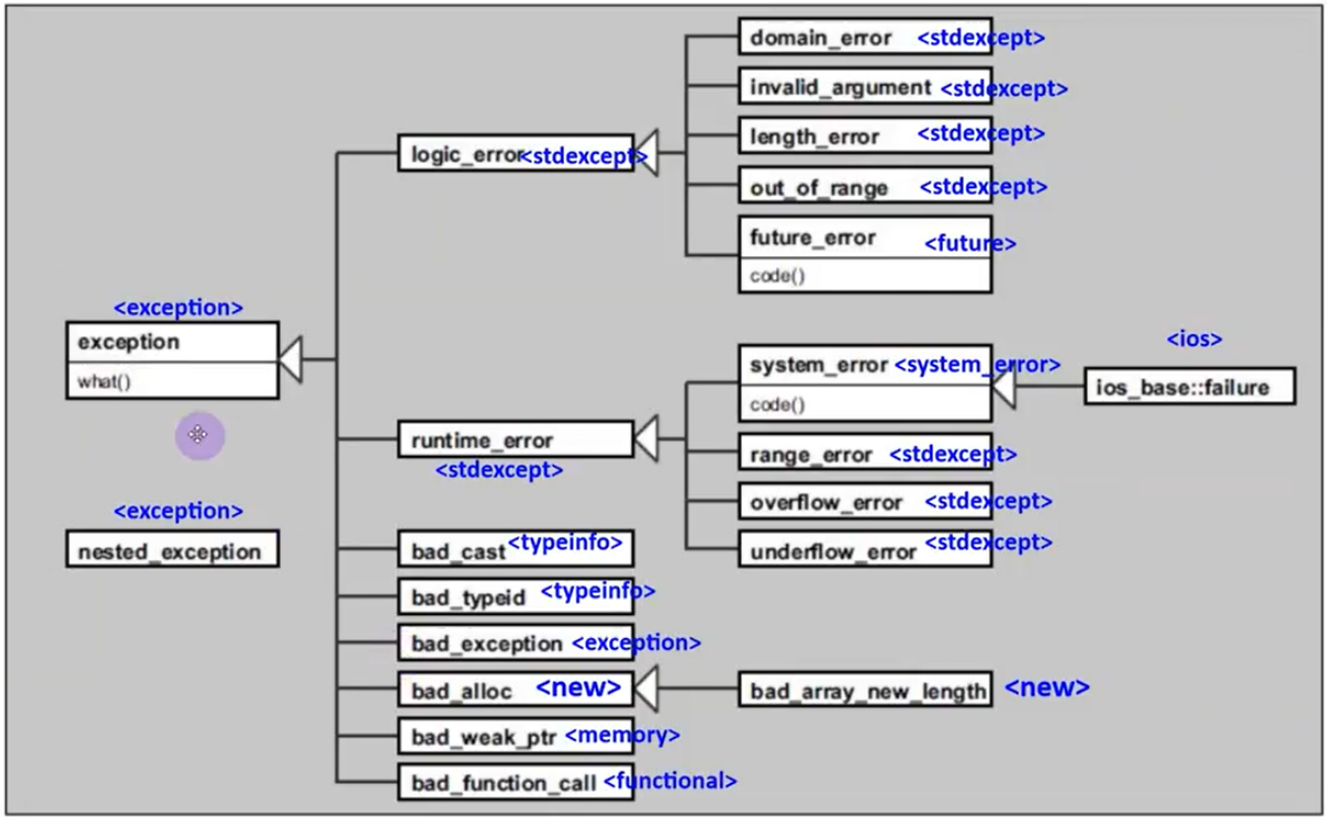


Figure 4.1. Hierarchy of Standard Exceptions

> Hataların işlendiği catch bloklarının özelden genele doğru sıralanması bir kodlama paternidir(desen,kalıp). Aksi takdirde alt taraftaki cath'e gelmeden daha genel olan varsa ona girecektir.

Gönderilen hatanın ne olursa olsun yakalanmasını istiyorsanız(catch all) bunun için catch içerisine elipsis(...) atomu koyarak yapabilirsiniz.

```
try { }
catch(...) { /*exception handler*/ }
```

```
void func() {
    int x = 13;
    throw x;
}
```

throw ifadesi bir lvalue expression olsa dahi yukarıya gönderilen nesne x'in kendisi değildir. Böyle olsaydı problem olurdu çünkü x'in hayatı func'ın bitmesiyle sonlanacaktı böylece throw edilen nesne hayatı bitmiş nesne olacaktır. Dolayısıyla hiç bir zaman aslında siz nesne gösteren bir ifade oluştursanız dahi derleyici bu nesnenin kendisini yukarıya gönderecek kod üretmiyor `throw T{x};` şeklindeki gibi derleyici bir geçici nesne oluşturuyor.

```
class Myclass {
    void func() {
        throw Myclass{};
    }
}
```

Mandatory copy elision ile nesneyi throw ettiği zaman func içerisinde nesne oluşturulmayacaktır. Böylece ilave bir maliyet olmayacaktır.

Exception'ı yakaladığımızda ne yapacağız?

1. Hatayı yakaladım. Gereken işlemleri yaptım ve programı sonlandırdım.
2. Hatayı yakaladım. Programın devamını sağladım.
3. Hatayı yakaladım. Kısmi müdahalede bulundum. Yakaladığım hata nesnesini tekrar gönderdim.("aynı" hata nesnesini yeniden göndermeye `rethrow statement` denir.)
4. Kendi müdahelemi yaptım. Ama beni çağıran kodların(onlara hitap eden) daha yüksek seviyeli exception döndürdüm. Buna `translate` denilmektedir.

```
void func() {
    std::cout << "func cagrildi\n";
    throw std::out_of_range{"aralik hatasi\n"};
}
void foo() {
    std::cout << "foo cagrildi\n";
    try {
        func();
    }
    catch(const std::exception& ex) {
        cout << "hata foo içinde yakalandi:" << ex.what() << "\n";
        throw; //rethrow statement
    }
}
```

> Exception asenkron bir mekanizma değil senkron bir mekanizmadır. Yani bir thread'ten gönderilen bir hataya başka thread'ten yakalayamayız.

> Eğer program bir hata nesnesi yakalayamadığı için terminate işlevinin çağrılmasıyla sonlanırsa(uncought exception) bu ana kadar oluşturulmuş yerel sınıf nesneleri için destructor çağrılma garantisi yoktur.

Stack unwinding(yığının geri sarımı) yapısı ile eğer hata yakalanırsa bu ana kadar oluşturulmuş yerel sınıf nesnelerinin kaynakları geri verilir.

Constructor ve Exception

Eğer bir constructor bir sınıf nesnesini hayata getiremeyeceğini anlarsa bu durumda tek yol exception throw etmektir. Fakat sınıfın ctor'u içerisinde exception throw edildiğinde sınıf nesnesi hala hayata gelmediği için

destructor çağrılmayacaktır bu yüzden bir kaynak edinimi yapacak olan sınıfın üye elemanları `smart pointer` ile oluşturulması gerekir.

> Dinamik ömürlü nesnelerin ctor'unda exception gönderilmesi durumunda zaten operator new mecbur çağrıldı ama sınıfın destructor'u çağrılmayacak ve operator delete'inde çağrılmayacağını düşünebilirsiniz fakat operator delete çağrılacaktır.

```
class Myclass {
public:
    Myclass {
        cout << "ctor\n";
        throw 1;
    }
    ~Myclass {
        cout << "destructor\n";
    }
    char buffer[1024];
};

void foo() {
    Myclass* p = new Myclass;
    delete p;
}

int main() {
    try {
        foo();
    }
    catch(int) {
        ///
    }
}
```

Destructor ve Exception

Destructor ya hiç exception throw etmeyecek ya da ederse içinde bunu yakalaması gerekmektedir.

Exception Safety

- Basic Guarantee
- Strong Guarantee
- Nothrow Guarantee

Basic Guarentee(Temel Garanti)

- Programın durumu değişebilir.
- Program tutarlı(devam edebilecek) bir durumda kalacak.
- Kaynak sızıntısı olmayacak
- Fonksiyon hiçbir nesneyi geçersiz durumda bırakmayacak.

Strong Guarantee(Sıkı Garanti)

- Programın durumu değişmeyecek.
- commit or rollback

  Ya işinin gör ya da hiçbir şey yapılmamış durumda bırak. İşlev çağrılmadan önce durum nasılsa işlev çağrıldıktan sonrada durum aynı olmalıdır.
- Program tutarlı(devam edebilecek) bir durumda kalacak.

Nothrow Guarantee(Hata Göndermeme Garantisi)

- Fonksiyon işini yapma garantisi veriyor.
- Hata gönderilirse kendi yakalayıp işini görecek.

```
void func()noexcept; //exception throw etmeyeceğinin garantisini yapar.
```

noexcept operatoruda vardır ve uneveluated context'tir.

```
int foo();
int main() {
    auto b = noexcept(foo());
}
```

Eğer fonksiyon noexcept garantisi veriyorsa noexcept true değilse false değeri döndürür.

```
void f1(int)noexcept(true);
void f2(int)noexcept; // f1 ile aynı

void f3(int)noexcept(false);
void f4(int); // f3  ile aynı

void f5(int)noexcept(true);
void f6(int)noexcept(noexcept(f1(10)));
```

RTTI(Run Time Type Idendification)

Programın çalışma zamanında nesnenin türünün ne olduğunu anlamaya yönelik bir araçtır.

> sizeof operatorünün operandının bir ifade olması durumunda operator'u parantez kullanmadan da oluşturabiliriz.

```
int x = 10;
int* ptr = &x;
```

```
sizeof(x);    /* ile */   sizeof x;    /* aynı */
sizeof(*ptr); /* ile */   sizeof *ptr; /* aynı */
sizeof(int);  /*geçerli*/ sizeof int;  // sentaks hatası
//sizeof int sentaks hatası çünkü bir ifade değil
```

dynamic_cast

down_cast işleminin çalışma zamanında güvenli bir şekilde yapılıp yapılamayacağını sınar.

> dynamic_cast operatörünün operandı polimorfik bir türden olmalıdır.

```
struct Base {
    virtual ~Base();
};
class Der : public Base {};
void func(Base* baseptr) {
    if(Der* derptr = dynamic_cast<Der*>(baseptr)) {
        //....
    }
}
```

Typeid Operatorü

başlık dosyasındadır.

```
int x = 5;
typeid(x);
```

typeid operatorü type_info sınıfına referans eder bunu derleyici yapar. type_info sınıfı türünden nesne oluşturamayız çünkü ctor'u yoktur mecburen typeid operatorünü kullanıyoruz.

Her tür için bir type_info nesnesi vardır.

```
cout << typeid('A').name() << '\n';
cout << typeid(13).name() << '\n';
cout << typeid(2.5).name() << '\n';
class Myclass {};
Myclass mx;
cout << typeid(mx).name() << '\n';
string str{"Enes"};
cout << typeid(str).name() << '\n';
```

```
int x = 10;
if(typeid(x) == typeid(int)) {
```

```
    cout << "evet esit\n";
} else {
    cout << "hayir esit degil\n";
}
```

```
class Base {};
class Der : public Base {};
Der myder;
Base* baseptr = &myder;
cout << typeid(*baseptr).name() << '\n'; // class Base
```

Ancak eğer Base'e bir tane sanal fonk. eklersek

```
class Base {
public:
virtual void foo() {}
};
class Der : public Base {};
Der myder;
Base* baseptr = &myder;
cout << typeid(*baseptr).name() << '\n'; // class Der
```

Böylece bu yapıyı biz RTTI gibi kullanabileceğiz.

```
void car_game(Car* carptr) {
    carptr->start();
    carptr->run();
    if(typeid(*carptr) == typeid(Mercedes)) {
        auto pm = static_cast<Mercedes*>(carptr);
        pm->open_sunroof();
    }
    carptr->stop();
}
```

RTTI'da dynamic_cast veya typeid operatorünü kullanarak türü run timeda öğrenebiliyoruz peki bir maliyet söz konusu ise hangisi daha az maliyetli olacaktır?

typeid daha az maliyetli olacaktır. dynamic_cast tür çıkarımını yapmak için tüm türemiş sınıflara bakması gerekiyor kakat typeid sadece tek karşılaştırma yapacaktır.

uneveluated context'te olan operatorler

```
int foo() {
    cout << "foo cagrildi\n";
    return 1;
```

```
}
auto sz = sizeof foo();             // 1
decltype(foo()) x = 5;              // 2
auto p = typeid(foo()).name();      // 3
constexpr auto b = noexcept(foo()); // 4
```

Template(Şablon)

Derleyiciye kod yazdırma aracıdır(Meta kod).

Template çeşitleri

1. Function Template
2. Class Template
3. Variable Template (Modern C++)
4. Alias Template (Modern C++)

Bir şablon oluştururken "template<>" şeklinde açısal parantez içerisine bir type belirterek kullanılır.

```
template<typename T> // T bir tür olmak üzere
template<class T> // T bir tür olmak üzere
```

> template'den sonra gelen açısal parantez içerisine yazılan class veya typename anahtar sözcüğü arasında hiç bir fark yoktur.

```
//template type parameter
template<typename T>
```

```
//template non-type parameter
template<int SIZE>
```

Hem type parameter hem de non-type parameter birlikte kullanılabilir.

```
template<typename T, int SIZE>
```

Non-type parameter şeklinde C++20'e kadar sadece ya adress türü ya da bir tam sayı olmak zorunda gerçek sayı türünden olmamakta idi. Gerçek sayı türünden sabit olma özelliği C++20 ile eklendi.

C++17'e kadar deduction sadece fonksiyon şablonları için yapılıyordu. C++17 ile birlikte sınıf şablonları için deduction kısıtlıda olsa dile eklendi(CTAD(Class Template Argument Deduction)).

```

```

```
template<typename T>
void func(T x) {
    //...
}
func(12);
func(12.5);
func('0');
func("enes");
func(); // Sentaks hatası tür çıkarımı yapılamıyor
```

Template argument deduction ile `func` fonksiyonundaki T türünün çıkarımı yapılıyor. Eğer tür çıkarılmazsa sentaks hatası olur.

Derleyicinin template ile tür çıkarımını nasıl yaptığını öğrenmek istediğimizde küçük bir hile ile bunu öğrenebiliriz.

```
template<typename T>
class TypeTeller;

template<typename T>
void func(T x) {
    TypeTeller<T> X;
}

int a[10]{};
func(a); // Derleyicinin verdiği hata T = int* şeklinde olacaktır.
```

Bu şekilde `func` fonksiyonuna gönderilen parametrenin türünün ne olduğunu anlayabiliriz. Derleyici `incomplete type` olan `TypeTeller`'in tanımını görmek istiyecek ve göremediği için bize nasıl bir çıkarım yapıldığını söyleyecektir.

```
//Array decay uygulanır
template<typename T>
void func(T);

//Array decay uygulanmaz referans ile çıkarım yapılır
template<typename T>
void func(T&);

//Forwarding reference/ Universal reference
template<typename T>
void func(T&&);
```

Templatelerde çıkarım auto ile aynı çıkarımı yapmaktadır.

> Bir istisna eğer initializer_list ile çıkarım yapılmak istenirse auto ile çıkarım yapılabilir fakat template ile çıkarım sentaks hatası olacaktır.

```
auto a = {1,2,4,5}; // çıkarım initializer_list olacaktır

template<typename T>
void func(T x) {}
func({1,2,4,5}); // sentaks hatası
```

```
//Template parametre paketi
template<typename... Types>
class TypeTeller;

template<typename T, typename U>
void func(T(*fp)(U)) {
    TypeTeller<T,U> x;
}

int foo(double);
func(foo); //çıkarım T => int, U => double
```

> Fonksiyon çağrı operatorunu overload eden sınıflara functor veya function object denir.

```
// expilicit template deduction
template<typename T>
T foo();

foo<double>();
```

> template fonksiyonlar implicitly inline'dır.

Fonksiyon şablonları aynı isimli başka işlev şablonları ile veya gerçek fonksiyonlar ile overload edilebilir.

```
template<typename T>
void func(T) {
    cout << "T turu:" << typeid(T).name() << "\n";
}

void func(int) {
    cout << "void func(int x)\n";
}
```

Öyle bir şey yapalım ki func işlevi sadece int türden argüman ile çağrılabilsin.

```
template<typename T>
void func(T) = delete;
```

```
void func(int);
```

Eğer bir fonksiyon şablonu ya da sınıf şablonu içinde template parameteresine bağlı bir nested type kullanıyorsak başına "typename" anahtar sözcüğünü koymak gerekir.

```
template<typename T>
void func(T x) {
    typename T::MyType mt;
}
```

Perfect Forwarding(Mükemmel Gönderim)

Benim yerime benim istediğim fonksiyon çağrılsın ancak benim gönderdiğim argumanın value category'si, const'ness değişmesin.

```
void func(int&) {
    cout << "void func(int&)\n";
}

void func(int&&) {
    cout << "void func(int&&)\n";
}

void func(const int&) {
    cout << "void func(const int&)\n";
}
//Perfect Forwarding
template<typename T>
void foo(T&& x) {
    func(std::forward<T>(x));
}

int a = 13;
const int b = 19;
func(a);  //int&
func(b);  //const int&
func(12); //int&&
```

(Explicit) Full Specialization

```
Fonsiyon şablonları(fakat çok özel durumlar dışında kullanmayın)
Sınıf şablonları
```

Partial Specialization

```
Sınıf şablonları
```

(Explicit) Full Specialization

```
template<typename T>
T Max(T x, T y) {
    cout << "primary template\n";
    return x > y ? x : y;
}

template<> // diamond template deme eğilimi vardır
const char* Max(const char* p1, const char* p2) {
    cout <<"explicit specialization for const char*\n";
    return std::strcmp(p1,p2) > 0 ? p1 : p2;
}

auto a = Max(12,45);
auto b = Max(15.3,24.5);
auto c = Max("ali","deniz");
```

```
template<typename T>
void func(T) {
    cout << "1";
}
template<>
void func(int*) {
    cout << "2";
}

template<typename T>
void func(T*) {
    cout << "3";
}
template<>
void func(int*) {
    cout << "4";
}

int* p = nullptr;
func(p); // 4 çağrılır.
```

4 çağrılmasının nedeni 2 numaralı `func(int*)` fonksiyonu 1 numaralı `func(T)` fonksiyonun full specialization'una katılır. 4 numaralı `func(int*)` fonksiyonu ise 3 numaralı `func(T)` fonksiyonun full specialization'una katılır. func fonksiyonuna int* türünden çağrı yapılırsa derleyici daha specific olan fonksiyonu seçecek böylelikle 3 numaralı func fonksiyon şablonu seçilecek daha sonra bununda full specialization yapıldığı 4 numaralı func fonksiyonu çağrılacaktır.

Şablonun varsayılan argüman alması

```
template<typename T = int>
class Myclass {
public:
    Myclass() {
        cout << "type t is: " << typeid(T).name() << "\n";
    }
};

Myclass<double> mx;
Myclass<> my; // Varsayılan argüman kullanılacak(int)
```

Döngü kullanmadan 1'den 100'e kadar olan sayıları yazalım.

```
// 1. yöntem
struct A {
    A() {
        static int x = 1;
        cout << x++ << " ";
    }
};

// 2. yöntem
template<int n>
struct B : B<n-1> {
    B() {
        cout << n << " ";
    }
};
template<>
struct B<0> {
    B(){}
};

A a[100];
B<100> b;
```

Partial(Kısmi) Specialization

Explicit specialization'da tek bir tür için alternatif kod verilirken partial specialization'da belirli özelliği sağlayan türler için alternatif kod verir. Örneğin template argümanının pointer türü olması durumunda alternatif kod yazarsak;

```
template<typename T>
struct Myclass {
    Myclass() {
```

```
        cout <<"primary template for type: " << typeid(T).name() << "\n";
    }
};

template<typename T>
struct Myclass<T*> {
    Myclass() {
        cout <<"partial spec. for T*\n ";
    }
};

Myclass<int> x1;    //primary
Myclass<double> x2; //primary
Myclass<int*> x3;   //partial spec.
Myclass<long> x4;   //primary
Myclass<long*> x5;  //partial spec.
```

typedef bildirimlerini template hale getiremiyoruz fakat using bildirimi ile template hale getirebiliriz. Buna `alias template` denir.

```
template<typename T, typename U>
class Myclass {};

template<typename T>
using Eno = Myclass<T,T>;

template<typename T>
using Iclass = Myclass<T,int>;

template<typename T>
using epair = std::pair<T,T>;

template<typename T>
using gset = std::set<T,std::greater<T>,allocator<T>>;
```

```
template<typename T>
using Ptr = T*;
int ival{};
Ptr<int> ip = &ival; // int* ip = &ival;
```

Variadic Template

C'de variadic fonksiyonların bazı dezavantajı vardır.

1. Type safe değildir.
2. Gönderilecek argüman sayısı belirtilmek zorundadır.

```
template<typename... Types> // template parameter pack
void func(Types... args);   // function parameter pack
```

Fonksiyon parametre paketinde kaç tane öğe var bunu bulmak istediğimiz zaman sizeof operatorunu kullanabiliriz fakat bu bildiğimiz sizeof operatoru değil modern C++ ile dile eklenen `sizeof...` operatorudur.

```
template<typename... Types>
void func(Types... args) {
    sizeof...(Args);
    constexpr auto n = sizeof...(args);
}
```

Pack Expansion

Fonksiyon parametre paketinin sonuna üç nokta(`...`) koyduğumuzda derleyici bunu virgülerle ayrılmış bir şekilde açar.

```
void foo(int,int,double);
template<typename... Args>
void func(Args... args) {
    foo(args...);
}
func(1,2,3.2);
```

Eğer `foo()` fonksiyonuna çağrı doğru olmasaydı sentaks hatası olacaktı fakat func fonksiyonuna gönderilen parametreler (int,int,double) olduğu için sentaks hatası yoktur.

Eğer func'un içerisinde foo'yu referans ile çağırmak istersek

```
template<typename... Args>
void func(Args... args) {
    foo(&args...); // foo(&p1,&p2,&p3)
    foo(args)...;  // foo(p1),foo(p2),foo(p3)
}
```

```
template<typename T,typename U, typename F>
struct Myclass {};

template<typename... Args>
void func(Args... args) {
    Myclass<Args...> x;
    //Myclass<int,double,float> açılımı olabilir
}
```

```
template<typename... Args>
void func(Args... args) {
    foo<Args>(args)...;
    //foo<T1>(p1), foo<T2>(p2), ...
}
```

Perfectly forward etmek istediğimiz zaman

```
template<typename... Args>
void func(Args... args) {
    foo(std::forward<Args>(args)...);
}
```

```
template<typename T>
void func(T) {
    cout << "non-variadic\n";
}

template<typename... Types>
void func(Types...) {
    cout << "variadic\n";
}

func(1,2);          //variadic
func(1,5.3,"alis"); //variadic
func(13);           //non-variadic
```

```
int x = 10;
int y = 20;
int z = 30;
int a[] = {(x++,0),(y++,0),(z++,0),};
// a'nın 3 elemanı var hepsi '0' olacak fakat x,y,z 1 artmış olacaktır.
```

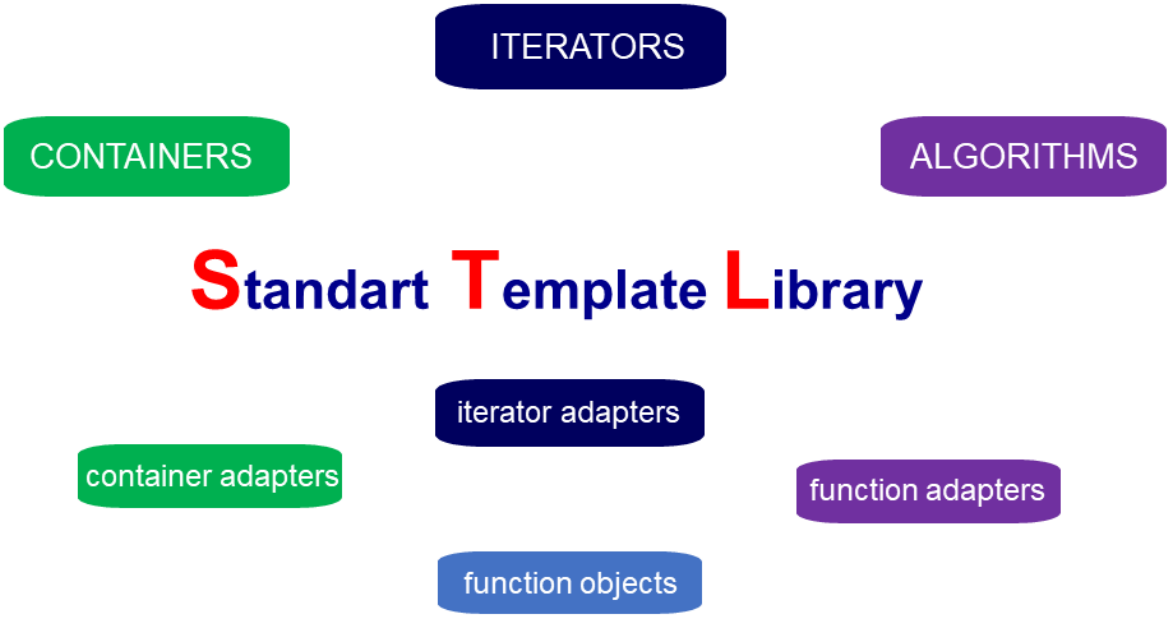
Yukarıdaki örnek ile yola çıkarak. Print fonksiyonu yazalım gönderilen her parametreyi ekrana yazsın.

```
template<typename... Args>
void print(Args... args) {
    int a[] = {(cout<<args<<" ",0)...)};
}
```

Derleyiciler kullanılmayan varlıklar için sentaks hatası verme eğilimindeler bu yüzden kodu tekrar yazacak olursak

```
template<typename... Args>
void print(Args... args) {
    std::initializer_list<int>{(cout<<args,0)...};
}
```

Standart Template Library(STL)



- OOP library değildir.
- Çok user friendly değil.
- Generic olduğu için basit bit hata olması durumunda çok fazla uyarı mesajı vermektedir.
- Modern C++ ile dile eklenen araçlarla(move schematic, perfect forwarding,lambda expressions, variadic templates,...) ile birlikte eskisine göre çok daha verimli hale geldi.
- Hata yapma riski daha az.

> Hiçbir STL algoritması aldığı range'in doğruluğunu/geçerliliğini sınamaz.

Iterator

Iterator'un kategorileri vardır.

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Iterator

Iterator'un kategorisi iteratorun interface'inde neler var neler yok onu belirliyor. Iterator'un yeteneğini gösteriyor. Mesela ++,--,* gibi operatorleri var mıdır?

- std::input_iterator_tag
- std::output_iterator_tag
- std::forward_iterator_tag
- std::bidirectional_iterator_tag
- std::random_iterator_tag

İterator kategorileri	operasyonlar	
Output Iterator	Copy constructible ++it it++ *it it-> (sol taraf değeri)	ostream_iterator, ostream_buf_iterator
Input Iterator	Copy constructible ++it it++ *it it-> (sağ taraf değeri) İt1 == it2 it1 != it2	istream_iterator, İstreambuf_iterator
Forward Iterator	Copy constructible default constructible ++it it++ *it it-> (sağ taraf değeri)(sol taraf değeri) İt1 == it2 it1 != it2	forward_list, unordered_set unordered_multiset, unordered map, unordered_multimap
Bidirectional Iterator	Copy constructible default constructible ++it it++ --it it--- *it it-> (sağ taraf değeri)(sol taraf değeri) İt1 == it2 it1 != it2	list, set, multiset, map, multimap
Random Iterator	Copy constructible default constructible ++it it++ --it it--- *it it-> (sağ taraf değeri)(sol taraf değeri) it1 == it2 it1 != it2 it + n n + it it – n it += n it -= n it1 – it2 it[n] it1 < it2 it1 <= it2 it1 > it2 it1 >= it2	vector, deque, array, string, C arrays

Copy gibi bir iterator konumundan başlayarak bir aralığa yazma(set) işlemi yapan STL algoritmalarının geri dönüş değeri en son yazılan konumdan sonraki konumudur.

Neden algoritmaların parametreleri kaplar değildir?

1. Container ile olsaydı bütün parametreler üzerinde işlem yapılacaktır fakat biz belli aralıktaki değerler için algoritmaları çağırabiliriz.

Modern C++'tan önce containerların iterator veren begin() ve end() fonksiyonları sadece sınıfın üye fonksiyonlarıydı fakat modern C++ ile birlikte bunlar global olarakta verildi.

```
std::vector<std::string> svec;
begin(svec); // ADL ile std namespace içerisinde de aranır
end(svec);   // ADL ile std namespace içerisinde de aranır
```

Find Algoritması:

```
template<typename InIter, typename Value>
InIter Find(InIter beg, InIter end,const Value& key) {
    while(beg != end) {
        if(*beg == key)
            return beg;
        ++beg;
    }
    return beg;
}
```

```
vector<int> ivec{1,5,7,8,45,6};
if(auto iter = Find(ivec.begin(),ivec.end(),6); iter != ivec.end()) {
    cout << "bulundu " << (iter-iter.begin()) << ". indisli oge\n";
} else {
    cout << "bulunamadi\n";
}
```

Find_if Algoritması:

```
template<typename InIter, typename UnPred>
InIter FindIf(InIter beg, InIter end, UnPred f) {
    while(beg != end) {
        if(f(*beg))
            return beg;
        ++beg;
    }
    return beg;
}
```

```
#include "nutility" // test kodları içerisinde
int is_ok(int val) {
    return val%5;
}
vector<int> ivec;
rfill(ivec,100,Irand{0,500});
```

```
print(ivec);
if(auto iter = FindIf(ivec.begin(),ivec.end(),is_ok); iter != iter.end()) {
    cout << "bulundu " << (iter-iter.begin()) << ". indisli oge\n";
} else {
    cout << "bulunamadi\n";
}
```

Count Algoritması:

```
template<typename InIter, typename Value>
int Count(InIter beg, InIter end, const Value& key) {
    int cnt{};
    while(beg != end) {
        if(*beg == key)
            ++cnt;
        ++beg;
    }
    return cnt;
}
```

```
int arr[]{5,78,45,6,25,5,48,5,9};
auto n = Count(begin(arr),end(arr),5);
cout << "arr dizisi icerisinde "<< n << " adet 5 vardir\n";
```

Count_if Algoritması:

```
template<typename InIter, typename UnPred>
int CountIf(InIter beg, InIter end, UnPred f) {
    int cnt{};
    while(beg != end) {
        if(f(*beg))
            ++cnt;
        ++beg;
    }
    return cnt;
}
```

```
#include "nutility" // test kodları içerisinde
class DivPred {
public:
    DivPred(int val) : mval(val) {}
    bool operator()(int x) const {
        return x%mval == 0;
    }
private:
```

```
    int mval;
};
vector<int> ivec;
rfill(ivec,100'000,Irand{0,5000}); //nutility fonksiyonu
int val;
cout << "kaca tam bolunenleri saymak istersiniz: ";
cin >> val;
cout << CountIf(ivec.begin(),ivec.end(),DivPred{val});
```

Copy Algoritması:

```
template<typename InIter, typename OutIter>
OutIter Copy(InIter beg, InIter end, OutIter destbeg) {
    while(beg != end) {
        *destbeg++ = *beg++;
    }
    return destbeg;
}
```

```
vector<string> svec{"enes","burak","gorkem","ozan"};
list<string> ilist(4);
Copy(svec.begin(),svec.end(),ilist.begin());
```

Copy_if Algoritması:

```
template<typename InIter, typename OutIter, typename UnPred>
OutIter CopyIf(InIter beg, InIter end, OutIter destbeg, UnPred f) {
    while(beg != end) {
        if(f(*beg))
            *destbeg++ = *beg;
        ++beg;
    }
    return destbeg;
}
```

```
bool is_ok(const std::string& str) {
    return str.size() == 6;
}
vector<string> svec{"enes","burak","gorkem","ozan"};
list<string> ilist(4);
CopyIf(svec.begin(),svec.end(),ilist.begin(),is_ok);
```

Boş bir container'ın begin fonksiyonun geri dönüş değeri end fonksiyonunun geri dönüş değerine eşittir. Bu iteratorlar asla dereference edilmemelidir aksi halde "undefined behaviour" olur.

```
vector<string> svec;
cout << "svec.size() = " << svec.size() << "\n";
cout << boolalpha << svec.empty() << "\n";
auto iter_beg = svec.begin();
auto iter_end = svec.end();
cout << boolalpha << (iter_beg==iter_end) << "\n";
cout << *iter_beg; // UB(Undefined Behaviour)
```

Neden global begin ve end fonksiyonları eklendi?

- En önemli nedini primitive tür dizileri içindir.

```
template<typename T, size_t size>
T* Begin(T(&r)[size]) {
    return &r[0];
}
template<typename T, size_t size>
T* End(T(&r)[size]) {
    return &r[size];
}

int a[] = {2,3,4,8,9};
Begin(a); // std::begin(a)
End(a);   // std::end(a)
```

Öyle bir şey yapalım ki copy algoritması ile hedef iterator adresine eklem yoluyla değerleri kopyalayalım.

```
vector<string> svec{"oguz","akif","alp","ahmet"};
vector<string> dvec;
copy(svec.begin(),svec.end(),dvec.begin()); //UB(Undefined Behaviour)
```

yukarıdaki kodda "UB" olmasının nedeni dvec nesnesinin boş container olması ve bu boş container'a copy algoritmasındaki içerik operaotoru(*) ile erişmek istemek sebep olmuştur. Peki öyle bir şey yapalım ki bu kod "UB" olmadan ekleme yoluyla kopyalansın.

```
template<typename C>
class BackInserterIterator {
public:
    BackInserterIterator(C& c) : _c(c) {}
    BackInserterIterator& operator++() { return *this; }
    BackInserterIterator& operator++(int) { return *this; }
    BackInserterIterator& operator*() { return *this; }
    BackInserterIterator& operator=(const typename C::value_type& val) {
        _c.push_back(val);
        return *this;
    }
```

```
    BackInserterIterator& operator=(typename C::value_type&& val) {
        cout << "move\n";
        _c.push_back(std::move(val));
        return *this;
    }
private:
    C& _c;
};
template<typename T>
BackInserterIterator<T> BackInserter(T& type) {
    return BackInserterIterator<T>{type};
}

std::list<int> ilist{1,2,4,5,7,8};
std::vector<int> ivec;
BackInsertIterator<std::vector<int>> bv(ivec);
Copy(ilist.begin(),ilist.end(),BackInserter(ivec)); //std::copy çalışmaz.
Print(ivec); //nutility header
```

Yukarıdaki kodta görüldüğü üzere ivec container'ı boş olmasına rağmen ekleme yoluyla kopyalama yaptık. Dikkat ederseniz std::copy fonksiyonunu değilde bizim yazdığımız Copy fonksiyonunu çağırdım bunun nedeni std::copy geçilen OutIter türününe bazı kontroller eklemektedir. Bu örneği standart olan `back_insert_iterator` sınıf şablonuyla `back_inserter` fonksiyonunu kullanarak yazalım.

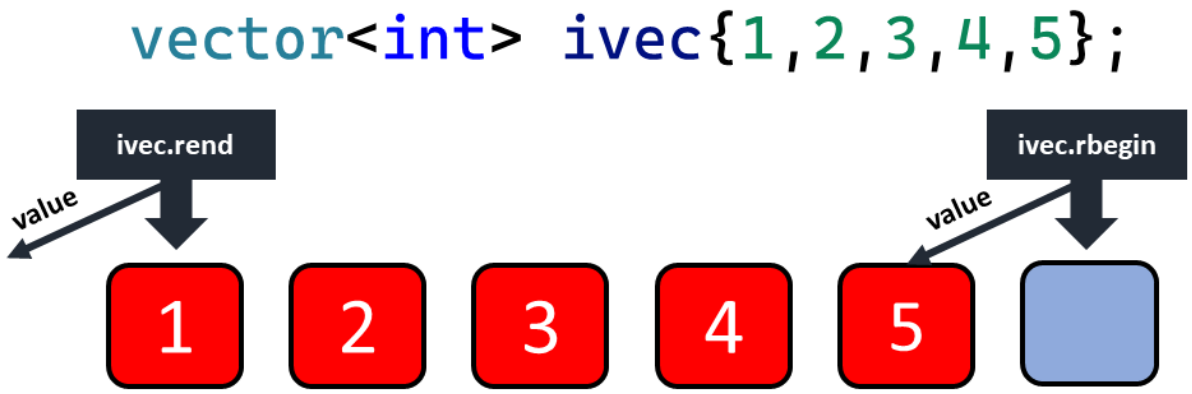
```
std::list<int> ilist{1,2,4,5,7,8};
std::vector<int> ivec;
std::copy(ilist.begin(),ilist.end(),back_inserter(ivec));
```

Copy'de sondan ekleme olduğu gibi baştan ekleme yapmakta mümkündür. Bunu front_inserter fonksiyonu ile yapabiliriz. Fakat bunu yapmak için fonksiyonun parametresi olan container'ın push_front fonksiyonu olmak zorundadır.

> push_back fonksiyonu sondan eklemenin constant time olması durumunda seçilmiştir.

Reverse Iterator

Sınıfların rbegin ve rend fonksiyonları reverse iterator nested type'ı türünden iterator döndürürler.



Containerların iteratorleri	Kullanımı
iterator	myvec.begin(); myvec.end()
const_iterator	myvec.cbegin(); myvec.cend()
reverse_iterator	myvec.rbegin(); myvec.rend()
const_reverse_iterator	myvec.crbegin(); myvec.crend()

Iteratorleri manipule eden global şablonlar;

- advance
- distance
- next (C++11)
- prev (C++11)
- iter_swap

Advance

Iteratorleri arttırmak amacıyla kullanılır. Bunun asıl nedeni template kodlarda iterator olarak hangi iterator kategorisi olduğunu bilmeden arttırma işlemini yapabilmektir. Örneğin input_iterator kategorisinde "+=" operatoru yoktur. Arttırma için "++" operatoru kullanılır. Random iterator için ise "+=" operatoru vardır. Derleyici advance fonksiyonunun implementasyonunu yaparken gelen iterator kategorisini en düşük iterator kategorisi olarak varsayarak implemente etseydi.bu sefer bir döngü içersinde sürekli dönen "++"

operatorü(tüm iterator kategorileri "++" operatorunu overload eder) ile bir kod yazması gerekirdi bu da verimli bir durum olmazdı. Bunun yerine derleyici `tag-dispatch` mekanizmasını kullanarak gelen iterator kategorisine göre o iterator kategorisini işleyerek koda çağrı yapmaktadır.

```
template<typename Iter>
void Advance_impl(Iter& r, int n, std::random_access_iterator_tag) {
    r += n;
}

template<typename Iter>
void Advance_impl(Iter& r, int n, std::bidirectional_iterator_tag) {
    while(n--) {
        ++r;
    }
}

template<typename Iter>
void Advance(Iter& r,int n) {
    Advance_impl(r,n,typename Iter::iterator_category{});
}
```

```
//#include "nutility"
list<string> slist;
rfill(slist,10,rname);
print(slist);
auto iter = slist.begin();
Advance(iter,5);
cout << *iter << "\n";
```

Distance

Random iterator'un altındaki iterator kategorileri için "-" operatoru yoktur. Bu yüzden iki ietartor arasındaki farkı almak için farklı bir yöntem kullanmamız gerekir. Fakat biz hangi iterator olursa olsun iki iterator arasındaki farkı `distance` fonksiyonu ile bulabiliriz.

```
vector<int> ivec{1,2,3,4,17,5,7,8};
auto iter_found = find(ivec.begin(),ivec.end(),17);
cout << distance(iter_found,ivec.begin()) << "\n";
```

size üye fonksiyonu olmayan tek container forward_list container'dır. Bu yüzden distance fonksiyonu ile size değerini buluyoruz.

```
forward_list<int> mylist{1,2,3,4,5};
cout << "size = " << distance(mylist.begin(),mylist.end()) << "\n";
```

Next & Prev

```
list<int> ilist{1,3,5,7,9,11};
next(ilist.begin());   // 3  (next(ilist.begin(),1))
next(ilist.begin(),5); // 11
```

iter_swap

Aynı veya farklı containerlarin iterator konumundaki öğeleri takas etmek için kullanılır.

```
vector<int> ivec{1,3,5,8};
print(ivec);
iter_swap(ivec.begin(),prev(myvec.end()));
print(ivec);
```

Global size fonksiyonu vardır.

```
vector<int> ivec(5);
list<double> dlist(5);
int a[] = {1,2,3,4,5};
std::cout << "ivec.size() = " << ivec.size() << '\n';
std::cout << "dlist.size() = " << dlist.size() << '\n';
std::cout << "size a = " << sizeof(a)/sizeof(*a) << '\n';
std::cout << size(ivec) << " " << size(dlist) << " " << size(a) << '\n';
constexpr auto s = size(a); // diziler için size sabit ifadesi döndürür.
```

Lambda Expression

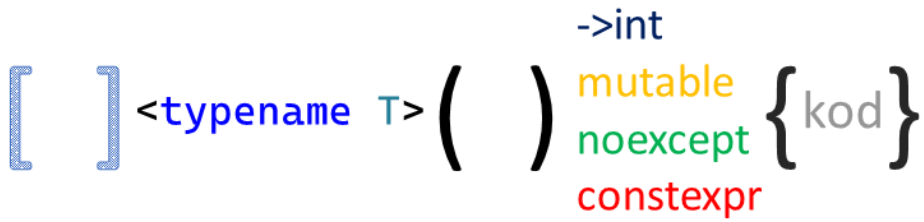
Lambda ifadesi nedir?

Derleyici bir lambda karşılığı;

- Derleyici ifadenin bulunduğu yerde bir sınıf tanımlar.
- Tanımladığı sınıfa bir üye `operator()` işlevi yazar.
- Koddaki lambda ifadesini oluşturduğu sınıf türünden bir geçici nesne(temprorary object) dönüştürür.

closure type: Derleyici lambda ifadesi karşılığı oluşturduğu sınıf türü.

closure object: Derleyici lambda ifadesi karşılığı oluşturduğu sınıf nesnesi.



- Lambda ifadeleri daha iyi okunabilir, daha iyi anlaşılabilir bir kod oluşturuyor.
- Çağrılan fonksiyonların kodunu görebilmek için kaynak dosyasında bir başka yere gitmek zorunda kalmıyoruz.
- İşlev göstericilerine göre daha verimli bir kod oluşturuyor(inline olarak açabilir).
- Bir fonksiyondan geri dönüş değeri olarak alabiliriz.
- Fonksiyon çağrısı için bir değişken oluşturmak zorunda değiliz. Lambda ifadesini yazdığımız yerde fonksiyonu çağırabiliriz.(IIFE)
- std::fonction yoluyla bir başka fonksiyona argüman olarak geçebiliriz.
- Bir fonksiyon şablonuna argüman olarak gönderebiliriz.(Algoritmalara argüman olarak geçebiliriz)

Derleyici lambda ifadesini

```
[](){}(); // Derleyici sanki lambda_1_2()(); şeklinde kod üretir.
class lambda_1_2 {
public:
    constexpr void operator()const {}
};
```

Lambda ifadesinin parametre prantezi bazı durumlarda yazılmayabilir.

1. Yazılacak lambda ifadesinin parametre değişkeni olmazsa
2. Parametre parantezi ile kodun kodun yazıldığı süslü parantez arasına gelen anahtar sözcükler yoksa

```
[](){ cout << "Hello World!\n"; };
[]{ cout << "Hello World!\n"; };
```

```
auto f = [](int x){ return x*5; };
class lambda_1_2 {
public:
    constexpr int operator() const {
        return x*5;
    }
};
```

lambda ifadesinin operator çağrı fonksiyonun geri dönüş değerini biz belirleyebiliriz. "Tralling return type" sentaksını kullanarak yaparız.

```
[](int x)->double{ return x*5; }
[]()->char{ return 1; }
```

Lambda ifadelerinden önce STL algoritmalarına callable olarak ya fonksiyon olarak tanımlayacak ve o fonksiyon adresini geçerdik ya da bir function object oluşturmamız geremekteydi.

```
class LenPred {
public:
    LenPred(size_t len) : mlen(len) {}
    bool operator()(const std::string& s)const {
        return s.lenght()==mlen;
    }
};
std::vector<string> svec("Enes","Ozan","Emre","Ece","Ahmet");
copy_if(begin(svec),end(svec),ostream_iterator<string>{cout," "},LenPred(4));
```

Yukarıdaki function object ile çağrı yapılan "copy_if" algoritmasını lambda ile oluştururusak.

```
copy_if(begin(svec),end(svec),ostream_iterator<string>{cout," "},
    [](const string& s){ return s.length()==4; });
```

Köşeli parantez içerisine yazılan ifadeye `capture close` denir. Derleyici lambda ifadesi karşılığı yazacağı sınıf içerisine bu "capture close" olan değişekni bir veri elemanı olarak ekler.

```
int a = 10;
auto f = [a](int x) { return x*a; };
```

```
class lambda_1_1 {
public:
    lambda_1_1(int a) : a(a) {}
    int operator()(int x) { return x*a; }
private:
    int a;
};
```

Global değişkenler ve statik yerel değişkenler capture edilemez.

```
int g = 10;
int foo() {
    static int x = 10;
    [g]() {}; // sentaks hatası
    [x]{};    // sentaks hatası
    []{++x};  // geçerli
    []{++g};  // geçerli
}
```

Kopyalama yoluyla yakalama	[x]	[x,y,]	[x,y,z]
Referans yoluyla yakalama	[&x]	[&x,&y,]	[&x,&y,&z]
Hem kopyalama hem referans yoluyla yakalama	[&x ,y]	[x,&y]	[x,&y,&z]
Alaynını, hepsini kopyalama yoluyla yakama		[=]	
Alaynını, hepsini referans yoluyla yakama		[&]	
Hepsini kopyalama yoluyla, x'i referans yoluyla		[=,&x]	
Hepsini referans yoluyla, x'i kopyalama yoluyla		[&,x]	

Lambda init capture

Taşıma sematiği ile capture oluşturmak için ilk değer vererek capture edebiliriz. unique_ptr sınıfı türünden bir sınıfı kopyalama yoluyla yakalamak söz konusu değildir çünkü zaten kopyalamaya kapalı bir sınıftır. Peki ben derleyiciye bir sınıf kodu yazdırmak istesem ve derleyici tarafından yazılan sınıfın veri elemanının unique_ptr olmasını istesem ve bu da capture edilen nesnenin kayanğını çalsın istiyorum.

```
auto ptr = make_unique<string>("Enes Alp");
auto f = [uptr](){}; // Sentaks hatası uptr kopyalamaya kapalı
auto fx = [uptr = move(uptr)](){}; // Geçerli lambda init capture
```

> Lambda init capture asıl kullanılma amacı taşıma semantiğini desteklemesidir.

C++14 ile generic lambda eklendi.

```
auto fx = [](auto x, auto y){return x+y;};
```

C++17 ile constexpr anahtar sözcüğü eklendi.

```
auto f = []()constexpr{};
```

C++20 ile template lambda eklendi.

```
auto fx = [](int x, int y){return x+y;};
auto fy = [](auto x, auto y){return x+y;};
auto fz = [](auto x, decltype(x) y){return x+y;};
auto ft = []<typename T>(T x, T y){return x+y;};
```

C++20'den önce stateless lambda ifadelerinin default constructorları delete edilmiş durumdadır.

```
auto fx = [](int x){return x*2;};
decltye(fx) fy; // C++'den önce Sentaks hatası
// default c'tor deleted.
```

> Herhangi bir değişkeni capture etmeyen ve herhangi bir statik değişkeni değiştirmeyen lambda ifadelerine *stateless lambda* denir.

Stateless bir lambda ifadesinden fonksiyon pointeri türü elde etmek istediğimizde kullanılan bir idiom vardır. Buna positive lambda idiom denilmektedir.

```
int(*f)(int) = +[](int x){return x+5;};
```

işaret operatorunun operandı bir sınıf nesnesi olmayacağından derleyici lambda ifadesini fonksiyon pointeri türüne dönüştürmektedir.

funtional başlık dosyasının içerisinde aritmetik işlemlerin hemen hemen hepsinin bir function object sınıfı vardır.

```
cout << plus<int>()(10,20)     << "\n"; //30
cout << multiplies<int>()(2,5) << "\n"; //10
cout << devides<int>()(10,5)   << "\n"; //2
cout << negate<int>()(10)      << "\n"; // -10
```

```
template<class InIter,class OutIter,class UnOp>
OutIter Transform(InIter beg, InIter end, OutIter destbeg, UnOp f) {
    while(beg != end) {
        *destbeg++ = f(*beg++);
    }
    return destbeg;
}
vector<int> ivec{1,2,4,5,9,11,13,15};
list<int> ilist(ivec.size());
Transform(ivec.begin(),ivec.end(),ilist.begin(),
    [](int x){return x*2;});
```

```
template<class InIter1, class InIter2, class OutIter, class BinOP>
OutIter Transform(InIter1 beg1,InIter1 end1, InIter2 beg2, OutIter destbeg, BinOp
f) {
    while(beg1 != end1) {
        *destbeg++ = f(*beg1++,*beg2++);
    }
    return destbeg;
}
//#include "nutility"
vector<string> namevec;
vector<string> surnamevec;
rfill(namevec,20,rname);
rfill(surnamevec,20,rfame);
print(namevec);
print(surnamevec);
list<string> person_list(namevec.size());
Transform(begin(namevec),end(namevec),begin(surnamevec),end(surnamevec),
begin(person_list),[](const string& s1, const string& s2){return s1 + ' ' + s2;});
print(person_list);
```

"Call by copy" olan bir fonksiyonuna sağ taraf değeri nesnesi geçersek bu fonksiyon kopyalama yerine doğrudan o nesneyi oluşturabilir(copy elision).

```
class Myclass {
public:
    Myclass(string s) : ms(move(s)) {}
private:
    string ms;
};
```

Yukarıdaki Myclass sınıfının constructor'u "string&" değilde "string" olarak seçilmesinin nedeni sağ taraf değeri geçilmesi durumunda bir kopyalanma yapılmasın(copy elision) direk o string nesnesini oluştrusun diyedir. Bundan sonra ise s'i taşıma haline getiriyoruz(bundan bir avantaj elde ediliyorsa) böylece kopyalama maliyetini azaltıyoruz.

```
template<class InIter, class UnOp>
UnOp ForEach(InIter beg, InIter end, UnOp f) {
    while(beg != end) {
        f(*beg++);
    }
    return f;
}
//#include "nutility"
vector<int> ivec;
rfill(ivec,20,rand);
ForEach(begin(ivec),end(ivec),[](int x){cout << x+1 << " " ;});
```

```
class Summer {
public:
    Summer(int val) : mval(val) {}
    int operator()(int x)const {
        ++mcnt; return x+mval;
    }
    int get_count()const{return mcnt;}
private:
    int mcnt;
    int mval;
};

auto f = ForEach(begin(ivec),end(ivec),Summer(19));
f.get_count(); /* Bu şekilde ForEach ile işlem yapıldıktan sonra
function object değişmemiş ve bu function objectten bir takım
verileri elde etmek istediğimiz zaman for_each algoritmasının
geri dönüş değerini kullanırız.*/
```