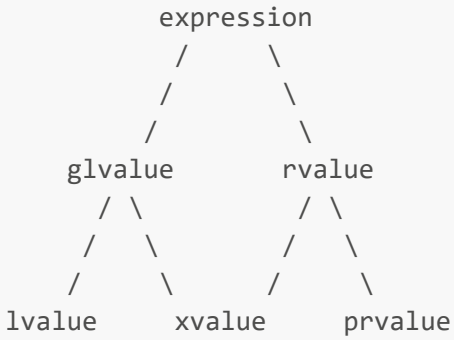
Reference



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();` şekilinde bir kod üretir ve daha sonra `const int& d = temp` olarakoluşturul böylece lvalue referans sol değer ifadesinebağlanır.

> 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, bracet 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)

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(ODR)

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

Sınıfların özel üye fonksiyonları(Special member functions)

- default constructor X();
- deconstructor ~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

Deconstructor(Yıkıcı fonksiyon)

Bir sınıfın sadece bir tane deconstructor'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.

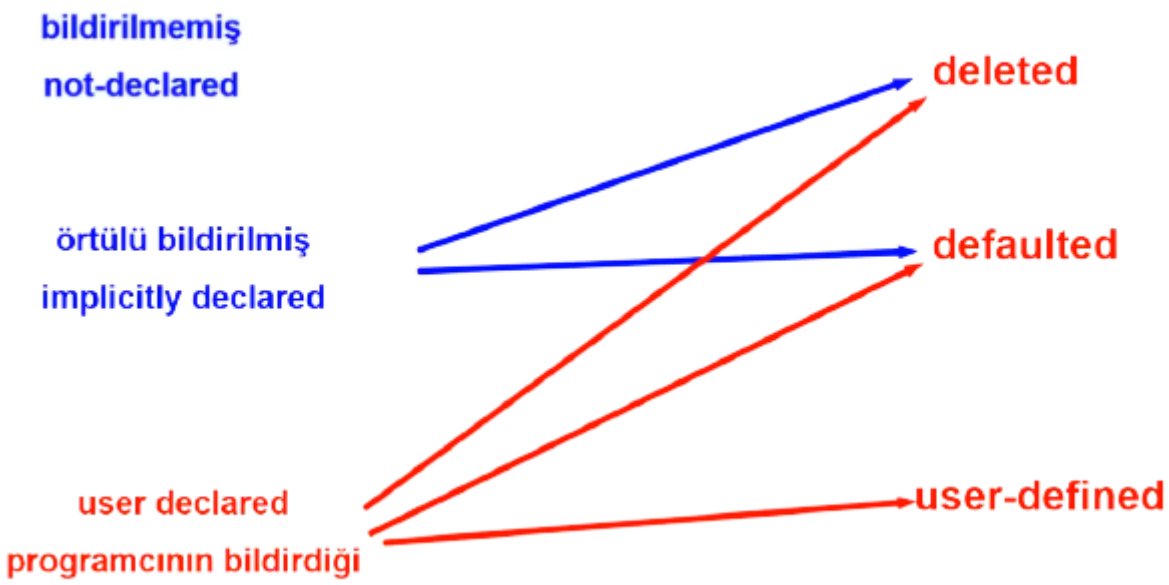
```
class Myclass {
    int mx(3);      // Geçersiz
    // T t = expr;  // Geçerli
    // T t{ expr }; // Geçerli
    // T t(expr);   // Geçersiz
}
```

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 Ctr'yi siz yazacaksınız 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 getirecek isek, 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 ctr'si 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(move(r.x)), y(move(r.y))
    {

    }
//move fonksiyonu, sol taraf değeri türünün sağ taraf değerine dönüştürür.
};
```

```
class Name
{
private:
    char *mp;
    size_t mlen;
public:
    Name() : mlen(0), mp(nullptr) {}
    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(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, p);
    }

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

    size_t length() const
    {
        return mlen;
    }
    ~Name()
    {
        // free edilen kaynağı tekrar free etme!
        if(mp)
        {
            free(mp);
        }
    }
};
```

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.

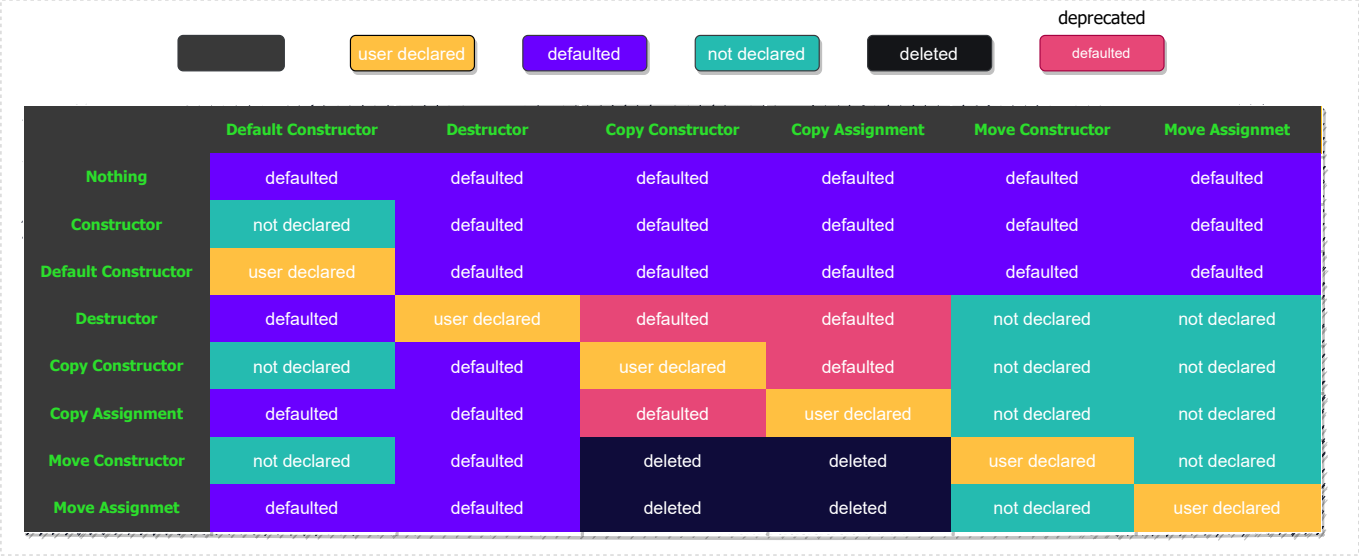
Örneğin;

```
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 parametreliconstructorlar 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 herhangibir 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([ ]) opeatorü

- 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;
}
```

*** operatörü**

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 ctor 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;
        }
        returm *smp;
    }
    void func();
    void foo();
    ///
private:
    Myclass();
    inline static Myclass* smp{nullptr};
};
```

Meyer's Singleton Class

```
class Myclass {
public:
    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)

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 barındıryor 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 başlatmış 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

Escape Sequence	Description
\a	audible bell
\b	backspace
\f	form feed
\n	line feed
\r	carriage return
\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

Sunig 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
```

Unnamed Namespace

Asıl kullanımı isimlerin bağlantı özelliği ile ilgilidir.

İsimlerin bağlantı özellikleri;

1. external linkage
2. internal linkage
3. No linkage olabilir.

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 bağlantoya alıyorduk bu C++'ta da geçerli fakat isimisiz isim alanına almak kodu daha derli toplu gösterir. C++17 ile `deprecated` edilen statik fonksiyon tanımı olmadığı için zaten mecburen iç bağlantıya dahil etmek istediğimiz statik fonksiyonları isimsiz isim alanlarına almak zorundayız.