

28 - 30 Mart, 4 Nisan 2022

C++ 11 major standartlar (ilk modern C++)
C++ 14 mini
C++ 17 major
C++ 20 major

C embedded için
yapılmış bir dll.
Bu nedenle C++ > C
başlıyor

Const anahtar sözcüğü

Const int*: low level const
Const ptr*: constant point to const
high level const

Const: degristerlenen devenek

int* const ptr: Const pointer to variable

Const int** ptr: pointer to constant

void*: void pointer, can hold the address of any type

```
int main()
{
    int x = 3;
    void* vptr = &x; // ok
    int** ptr = vptr; // error
}
```

→ degerlerden void
pointer türünde okunaklı
değil var. ama
tanı tersi gecerli değil.

* Const değişkenlere ilk değer
vermek zorundadır.

```
int f(); // declaration
int f() { return 42; } // definition
```

const T* ==> T* // hata, örtülü
dönüşüm yok
T* ==> const T* // ok
const T ==> T& // hatalı, örtülü
dönüşüm yok

Const → normal'e örtülü dönüşüm yok
Normalden → Const'a örtülü dönüşüm var.

conventional enum
 scoped enum
 ↓
 enum class

conventional enum(unscoped enum)
eksiklikleri

1) // incomplete type
enum Color;

2) // kontrol dışı tür dönüşümü
int ival;
Color myColor = Black;
ival = myColor; // legal, gençlik hatası

kullanmayın!!

3) enum türünün kapsamı neyse
enumeration constant'ın kapsamı aynı.
enumeration constant'ın ayrı scope'u yok

for gizlemi C'de yok, C++'da heryerde (type deduction)

auto x = 2;

int main()
{

int x;
 int y(3); // direct initialization
 int z = 4; // copy initialization
 int k{1}; // uniform init (brace)

}

uniform initialization (brace init.): C++ ile geldi.

diğer init.'lerin eksikliklerini giderir.

1. uniform (öğrenim kolaylığı için heryerde kullanılabilir)
2. narrowing conversion sentax hatası
3. most vexing parse'in engellenmesi

int = double


 ill deger verme ile
 fonksiyon葛risim konusunda

internal linkage
external linkage

translation unit

bulunduğu yerde anlamı
değişen belimeler:

- static
- using
- auto
- mutable

// zero initialization
int x; // static ve global değişkenlerde 0 ile
başlatılır.
bool y; // false ile başlatılır
int *gp; // nullptr ile hayata başlar

fakat yerel değişken
int y; // garbage value alır
int* z; // garbage reference alır

void func();
void foo(void);

C++ ta ikisi arasında anlam farkı yok.
parametre değişkeni yok demek
ikiside.

dizilere ilk değer verme:

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

value initialization:

```
int main()
{
    int x{}; // value initialization // sıfır ile ✗
    başlama garantisı var
    bool flag{}; // value init. // false değer ile ✗
    başlar
    int *ptr{}; // value init. // nullptr ile hayata ✗
    başlar
    int y; // garbage value ✗
    int z(); // parantezin içi boş olursa
    fonksiyon bildirimi *
```

user defined types:

structures
class
unions
enumerations

#include <cstring> => c'deki string.h'nin C++'daki karşılığı, C'de string.h var ama
~~string sınıfı yok~~
#include <string> => C++

In C programming, a string is a sequence of characters terminated with a null character \0. For example:
char c[] = "c string";

Operator overloading

operator << (cout, "alican"); // aynısı => std::cout << "alican";

Chaining

cout << "ahmet" << "veli" << endl;

namespace scope
class scope
block scope
function prototype scope
function scope

function prototype scope:

void func(int, int); // iki şekilde bildirim de uygun !!!
void func(int x, int y); //function prototype scope
// bu isimler x ve y sadece bu parantez içerisinde bilinirler.

namespace (isim alanı) : C'de global isim alanı tek bir alan. Sizin kullandığınız bütün isimler buraya boşaltılır. Bu da farklı farklı hizmet aldiğiniz kütüphanelerin başlık dosyalarından gelen isimlerin çakışması riskini çok arttırmıştır.

C++ dilinde namespace alanları ile bu riskten kurtuluyor

:: => unary scope resolution operator

```
int main()
{
    ::x; // bu bloğu içine alan ilk namespace alanı içinde arar. Bu da global namespace alanıdır.
}
```

:: => binary scope resolution operator

```
int main()
{
    neco::x; // sol operatörü namespace ise sağ operandını o namespace içinde arar
}
```

#include <cstring> => C'deki string.h'nın C++'daki karşılığı, C'de string.h var ama string sınıfı yok. Char dizisi var

#include < string> => C++

In C programming, a string is a sequence of characters terminated with a null character \0.

For example:

```
char c[] = "c string";
```

function prototype scope:

```
void func(int, int); // iki şekilde bildirim de uygun !!!
void func(int x, int y); //function prototype scope
// bu isimler x ve y sadece bu parantez içerisinde bilinirler.
```

```
#include <iostream>
using namespace std;
main() {
    int a = 21;
    int c ;
    // Value of a will not be increased before assignment.
    c = a++;
    cout << "Line 1 - Value of a++ is :" << c << endl ;
    // After expression value of a is increased
    cout << "Line 2 - Value of a is :" << a << endl ;
    // Value of a will be increased before assignment.
    c = ++a;
    cout << "Line 3 - Value of ++a is :" << c << endl ;
    return 0;
}
```

4 Nisan 2022

Most vexing parse: ilk deger verme ile faktörün
kısıtlaması

maximal munch:

`vector<std::complex<int>> vec6mp;`

pointer → naked (raw) pointer

pointer → Smart pointer

Referans Semantigi: Sını bir yapı, Metcher kod alanında
pointer ile aynı

S bir sayi atomadan telefonlarda adres yoksas referans aralanna gelir.

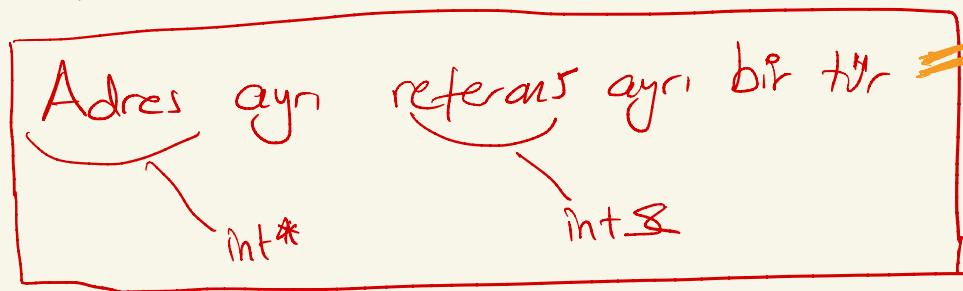
Int x=10;

Int y;

S x; \Rightarrow adres, nummer mit *

int \$x = y; \Rightarrow refers, to int \$x

Ref bir tür, int ref bir tür



Referanslara ilk değer vermek zorlu!!!

narrowing conversion

```
int main()
{
    double dval = 3.4;
    dval = ival; // ok
    int ival = dval; // legal fakat implicit type conversion, ondalık kısım kaybedilir.
    int ival (dval); // legal fakat implicit type conversion, ondalık kısım kaybedilir. Fakat
derleyici uyarı verir.
    int ival {dval}; // sentaks hatalı, uniform init.'de koruma var
}
```

```
int x = 10;
```

```
int main()
{
    int x = x; // tanimsız davranış, = 'in sağında soldaki x'in scope'una girilir, sağdaki x
tanımlanan int x, burada otomatik ömürlü x değişkenine kendi çöp değeri ile ilk
değer verdigimizden tanimsız davranış
}
```

Function redeclaration

baskasının h file'ini değiştirmek yerine

neco.h

```
int foo(int, int, int){  
}
```

neco.cpp

include "neco.h"

```
[ int foo( int, int, int=3) ]
```

function
redeclaration

```
int main()
```

{

```
    foo(10, 23);
```

}

Varsayılan arguman: (default argument)

```
int func( int x=10, int y=20, int z=30)  
{
```

}

```
int main()  
{
```

```
    func();  
    func(3);  
    func(3, 10);  
    func(3, 20, 5);
```

}

C++11 öncesindeki referanslarımıza

a) L value reference diyoruz.

C++ 11 sonrası ile eklenen

b) R value reference

c) forwarding reference (universal reference)

primary value categories (in C++)

PR value (expression)

L value (expression)

X value

bir de bunların birleştirilmesinden elde edilen value category var

(L value) U (X value) : glvalue

(PR value) U (X value) : R value

expression (ifade) :

sabitlerin isimleri ve operatörlerle yaptıkları bileşime denir.

10

x + 10

x + 10 > 20

*

statement (expression sonuna noktalı virgül konursa statement olur):

10;

x + 10 > 20;

*

expression's

a) data type

b) value category (değer kategorisi)

value category (veri türü) (C dilinde)

L value expression

R value expression

```
int main()
{
    int x = 10; // x ifadesinin türü nedir ? int
    x + 1.2; // ifadenin türü nedir ? double
    x + 1.2f; // ifadenin türü nedir ? float
    &x; // ifadenin türü nedir ? int* ??????
    // zaten ref. olabilmesi için ilk değer verilmeli
}
```

&x : adres ise türü => int*
&y : referans ise türü => int&
decltype ile tür bilgisi elde edilebilir

(float) 3; // çırkin yazım
3f; // böyle yazın

* bir ifadenin rvalue mu lvalue mu anlamak için adres operatörünün operatörü yapın.
sentax hatası varsa rvalue yoksa lvalue'dur.

pointer'a reference:

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

int*& r = ptr; // *& operatör değil declarator, hangi türden değişkene referans
olacaksa önce o yazılır int*, ptr kendisi de bir değişkendir. r int* türünden nesneye
referans, r demek ptr demek

```
++* r; // x = 11;
```

```
int main()
{
    int x{ 24 };

    int& r1 = x; // r1 demek x demek
    int& r2 = r1; // r2 demek x demek

    ++r1;
    ++r2; // x = 26;
}
```

```

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

    // öyle bir pointer değişken tanımlayın ki ismi p olsun ve *p ifadesi a dizisinin kendisi olsun.
    int (*p)[5] = &a; // doğru !!! diziye pointer, bu baya enteresan bilmediğim birşey

    int* ptr = a; // yanlış, bu dizinin ilk elemanını tutar

    *p => dereferencing ile a dizisine erişirim
    *ptr => dereferencing ile a dizisinin ilk elemanına erişirim

    for(int i = 0; i < 5; ++i)
        printf("%d %d\n", a[i], (*p)[i]);

    // öyle bir isim olsun ki r, r demek a demek olsun
    int (&r)[5] = a; // r demek a demek
}

```

diziye referans oluşturanın çok daha kolay yolu var

```

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

    auto& r = a; // int(&r)[5] = a;
    // dizinin hepsini tutar, burdaki parantez declarator
    , öncelik parantezi değil,

    auto r = a; // int* r = a; // bu da ilk elemanı tutar
}

```

// call by reference

```

void func(int *a)
{
    *a = 999;
}

```

```

int main()
{
    int x = 10;

    func(&x);
}

```

// call by reference (daha pratik yol)

```

void func(int &r)
{
    r = 999;
}

```

```

int main()
{
    int x = 10;

    func(x);
}

```

const correctness: kod kalitesini belirleyen en önemli faktör

void func(int* p); // mutator, gönderilen değişkeni değiştirmeye yönelik
void foo(const int* p); // gönderilen nesne değişimeyecek, salt okumaya yönelik erişim

```
int main()
{
    int x = 10;
    int* const p = &x; // p'yi değiştirmeme
    int const* ptr = &x; // const int* ptr = &x, gösterdiği nesne değiştirilemez
```

int& const r = x; // hiçbir anlamı yok, kullanma, zaten referanslar başka nesneye bağlanamaz !!!

```
const int& cr = x; // doğru,
cr = 10; // hata
}
```

```
int* foo()
{
    // code
    return &g;
}
```

foo() fonksiyonunun referans semantığı karşılığı nedir ?

```
int& func()
{
    // code
    return g; // referans döndüğü için
func() demek g demek !!
}
```

```
int main()
{
    func() = 999; // lvalue expr., atama g
değişkenine yapıldı
```

```
int& r = func(); // r demek g demek
r = 999; // g'yi de 999 yaptım
}
```

Adres döndüren fonksiyonlar (functions returning pointers)

```
int g = 10;

int* foo()
{
    // code
    return &g;
}

int main()
{
    int* p = foo();

    std::cout << *p << "\n",
    *p = 444;

    *foo() = 56; // g değerini 56 yapar
}
```

Otomatik ömürlü nesne adresi geri döndürülmez !!!

```
// error: returning address of local variable
int &foo()
{
    int x;
    std::cout << "bir sayı giriniz: ";
    std::cin >> x;

    return x; // undefined behaviour,
    // otomatik ömürlü nesneye referans dönmek tanımsız davranış
}

// error: returning address of local variable
int* foo()
{
    int x;
    std::cout << "bir sayı giriniz: ";
    std::cin >> x;

    return &x; // & bir şey atanmadan kullanılıncı adres yoksa reference
    // anlamına gelir , undefined behaviour,
    // otomatik ömürlü nesneye referans dönmek tanımsız davranış
}
```

assembly kod için yukarıdaki iki fonksiyon aynı !!

Adres döndüren fonksiyon:

3 ihtimal var

- 1) static ömürlü bir nesne adresi döndürebilir
 - a) global değişken
 - b) static yerel değişken
 - c) string literalı
 - 2) Dinamik ömürlü bir nesne döndürebilir
 - 3) çağrıdan aldığı adresi döndürebilir
- Otomatik ömürlü nesne adresi geri döndürülmez !!!

pointer array:

```
int g1 = 10;  
int g2 = 20;  
int g3 = 30;  
  
int main()  
{  
    int* pa[] = { &g1, &g2, &g3}; // pointer array  
  
}
```

void func(T *);
void foo(T &) aynıdır

referans dizisi diye birşey yok !!!

Fonksiyonunun niyetini anlamak !!

void func(T *ptr); // mutator, nesnenizi değiştirecek
bu fonksiyonun referans semantığında anlamı ne ?

cevap:

void func(T &r);

void foo(const T *p) == void foo(const T &r) // nesneyi değiştirmeyecek !!

pointer ile referans farkı:

- 1) pointer'lara ilk değer vermek mecburi değil, ama referanslara mecburi
- 2) pointer'lar const olmak zorunda değil
- 3) pointer'lar dizilerde tutulabilir ama referanslar tutulamaz
- 4) pointer to pointer olur ama reference to ref. olmaz
- 5) nullptr var ama nullref yok

reference bir nesnenin yerine geçen isimdir

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

    &r; // &x demek

}
```



```
int* foo();
int& func();

int main()
{
    foo(); // geri dönüş değeri adres olan fonk. çağrı ifadeleri prvalue expression
    func(); // geri dönüş değeri sol taraf referansı olan fonksiyonlara yapılan çağrı
            iifadeleri lvalue expression
}
```



Onutulmamasi gerekenler : 1) const depreklere ilk değer vermek zorunlu

2) const depreklər, pointer'lardan normal depreklər, pointer'lara örtülü tür dönüşümü yok. Tam tersi var.

3) TÜR AİKARMI C'de yok iken C++'da həyəde var.
auto x = 3;

4) Brace initialization C++ ile geldi. a) normal C++-da həts var
b) most vexing parse ; example

5) Yerel değişkenlere oturam yapılmaz ise garbage value ile başlar. Static ve global değişkenler ise sıfır, garbage value ile başlar.

Braze mit. ile başlarsa sıfır, nullptr ile başlarsa

int x{};

int x(); // fonksiyon bildirimi !!!

6) C'de string sınıf yok, sonu null karakter ile biter char dizesi var.

7) void func(int x, int y); // function prototype scope
void func(int, int); x ve y sadece burada bilinir

8)	namespace scope	function scope
	class scope	function prototype scope
	block scope	

9) C'de sadece global isim alanı varken C++'da namespace deye bir kavram var.

10) c=a++; // a will not be increased before assignment
c=++a; // a will be increased before assignment

11) most vexing parse: ilk değer verme ile fonksiyon çağrısını kabul etmek
maximal munch: <int>
bosluk olursa >> okusılır.

12) referans sunu bir yapı, derleyicinin ürettiği kod açısından pointer ile aynı. Referanslara ilk değer vermek zorundadır.

ilk değer verilenle pointer atanma gelir.

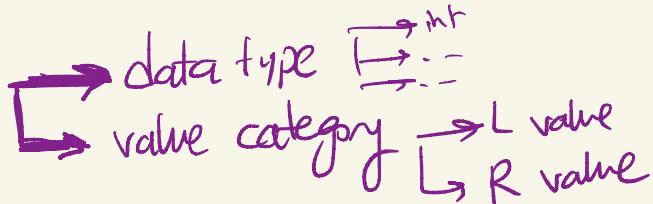
$\text{int } \&x = a;$; x 'in türün int ref (int&)

$\&x$: adres türü ise \Rightarrow int&

$\&y$: referans türü ise \Rightarrow int&

13) expression's

$x+3$



value category'ları anlamak için adres operasyonlarında
operands yapı,

14)

$\text{int } x\{10\};$
 $\text{int } \&y = x;$ // x denet y denek

pointer'a reference:

$\text{int } x = 10;$

$\text{int } \&\text{ptr} = \&x;$

$\text{int } \&\text{r} = \text{ptr}; \Rightarrow$ pointer'a referans

diziye pointer:

$\text{int } a[5] = \{1, 2, 3, 4, 5\};$

$\text{int } (\&p)[5] = \&a;$

$\text{int } \&\text{ptr} = a;$

$\&p \Rightarrow$ a dizisine erişim

$\&\text{ptr} \Rightarrow$ a dizisinin ilk elemanına erişim

diziye referans: $\text{int } (\&r)[5] = a;$ // r denet a dizi

buradaki parameter deklarasyonu
öncelik parameteri değil!

drıye referans oluşturmanın daha kolay yolu:

int a[5] = {1, 2, 3, 4, 5};

auto & r = a; // int(&r)[5] = a;

- 15) Const correctness :: kod kalitesini belirleyen en önemli faktör
- void func(int* ptr) \Rightarrow mutator, gürdenle nesneyi değiştirme
void foo(const int* ptr) \Rightarrow Sade okunma yordamı, gürdenle nesne değiştirilemeyecek

int* const ptr = &x; \Rightarrow ptr 'yi degistirmeme

int const* ptr = &x; \Rightarrow gürdenle nesne degistirilemez.

int & const r = x; // higher olarak yok, odań referansları
başa nesneye bağlanamaz

- 16) Adres döndüren fonksiyonlar. (functions returning pointers)

```
int* foo()
{
    return &r;
}
```

```
int fo()
{
    return &r;
}
```


foo denebilir r denebilir!

Otomatik "müraci" nesne adresi döndürilemez! Hata verir.

- 17) Pointer array:
- ```
int a=3;
int b=4;
int c=5;
```
- $\&\text{int}^*\text{n}[3] = \{\&a, \&b, \&c\};$

Referans drıası diye bir sey yok!!!