

* Const expr:

→ const: • const deklarasyonu ile değer vermek ZEVENLU / compile time'da değerini bellidir.

• Bir nesnenin const olması, o nesne ile oluşturulan ifadelerin de const expression olarak anılmasına gelmeyen.

ör%: const int x = 10;

const int y = foo();

```
int main()
```

```
{
```

int a1[x] {} ✓

int a2[y] {} : X → const foo const expression değil

```
}
```

* Modern C++ ile constexpr geliyor. → constexpr x = 10; → implicitly const
constexpr y = 20;
constexpr z = 30;

x + y + z → 60 olur. #define .. 60 ile aynı.
bu da
constexpr int

* Aynı şekilde fonksiyonlar da constexpr olabilir.

↳ Bu da avantajı runtime'da yük olmayan fonksiyonları
değerlerini compile time'da hesaplayabilmek.

ör%:

The screenshot shows a Microsoft Visual Studio IDE window with the following code in a file named 'main.cpp':

```
1 //constexpr int sum_square(int a, int b)
2 {
3     return a * a + b * b;
4 }
5
6 int main()
7 {
8     constexpr int x = sum_square(10, 20); //500
9     // (local variable) constexpr int x = 500
10    //Search Online
11 }
12
13
14 }
```

A red arrow points from the text "auto da derebilir" to the line "constexpr int x = sum_square(10, 20);". A red circle highlights the word "constexpr" in the same line. A tooltip "Search Online" is visible near the bottom of the highlighted line.

One Definition Rule:

- 4ozilmesi gereklilerin tek definitioni olmali
- Ill-formed olur eger ODR olmazsa

→ Header File'da Function Definition yapmak ODR'ineline sebeplidir. !!!

↓
sadece function
prototype

→ eger inline olarak tenitilirse
ODR'ineline sebeplidir

```
int foo(int)
{
    return 1;
}
```

→ Ayni header altında, syntax hatalı. Fakat,
 farklı header'a olساğdı, syntax hatalı olmazdı.

↳ konularla ilgili kod şırrı.

```
int foo(int)
{
    return 1;
}
```

```
inline void func()
```

→ Inline olarak tenitilmiş fonksiyon header olusmeye konabilir.
Fakat inline olmasa, ODR'iniz olur!

* Tipik olarak constexpr fonksiyonlar header olusurlar. Cunki compile-time de kodu girmesi gerekiyor!

* Constexpr functionlar implicitly inline fonksiyonlardır.

* Constexpr ile decltype:

```
constexpr int x = 10
int main()
{
    decltype(x) } → x'in türü const int!
}
```

→ int x { }

```
int main()
{
    constexpr int * p = &x;
    *p = 367;
}
```

p'in kendisi
implicitly const

fakat p'in türü
int * → const int * değil

Junya da const ypsilebilir. O zaman p'in türü const int * olur.



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

    constexpr int* p = &x;
}
```

→ Bu syntax notesı, Cunkı \times const değil X

```
int x{};

int main()
{
    constexpr int* p = &x;
}
```

→ Fakat global ömrili değişkenlerin adresleri olan türdeki .CONSTANT EXPRESSION (statisch ömrili)

```
int x{};

int main()
{
    constexpr int* p = &x;
    *p = 567;
}
```

p'ının türü int*, const int* değil X

bıraya
const eklememiz
gerekir!

* Function Overloading: → C dilinde olmaz bir konu

- Aynı isimli birden fazla fonksiyonun aynı scope altında bulunması.
- Tercümen programının ismini takip etmek için.
- Compile Time'da nesnenin bir mekanizma Run time'da bir mekanizmaya Static Binding

→ Binding: Eğer fonksiyona uygun çağrı, hangi fonksiyona iliskin olduğu compile time'da biliinirse: early binding / static binding

run time'da bilinirse: late binding
dynamic binding

* Function overloading olması için → Aynı scope'ta bildirilmesi olması.

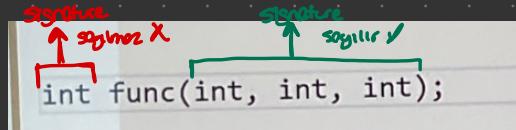
→ Signatureları farklı

↓
Fonksiyonun Return Degeri Hanesi parametrik yapisi (oldugu parametrelere vs...)

* Is it Function Overloading?

→ Aynı scope ✓

→ Farklı signature.



⇒ return degeri hane, gelen parametrelere.

* Aynı scope, aynı signature + Aynı return degeri = FUNCTION REDDECLARATION (NOT OVERLOAD)

Aynı scope, aynı signature + FARKLI RETURN DEGERI = SYNTAX ERROR X

```
int foo(int);
int foo(const int);
```

→ OVERLOAD DEĞİL X

→ TOP LEVEL CONST OLDUĞU İÇİN → FUNCTION REDDECLARATION

```
int foo(int *p);
int foo(const int *p);
```

→ LOW LEVEL CONST OLDUĞU İÇİN → FUNCTION OVERLOADING ✓

* Typedef ile ter�itirme konusu

```
#include <iostream>

typedef int itype;

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

→ FUNCTION REDDECLARATION X

* Default argument, eger aynı signatürdeki veildigile → FUNCTION DECLARATION X
eğer farklı signatürdeki veildigile → FUNCTION OVERLOADING ✓

* Signed / Unsigned birden fazla → FUNCTION OVERLOADING ✓

```
void func(char){}
void func(signed char){}
void func(unsigned char){}
```

* Function Overload Resolution:

- Overload'un olması, bu örn her zaman legal olmamı garantierne X

- Case 1:

- Overloader var, fakat hizmeti legal değil çağrıldığında → SYNTAX ERROR X

- Case 2:

- Overloader var fakat compiler hangisini seçeceğini bilmiyor → AMBIGUITY X

```
enum Color {Red, Black};    ↴ NOT VIABLE
void func(int);
void func(int, int);        ↴ VIABLE
void func(int, double);    ↴ candidates
void func(char*);          ↴ NOT VIABLE
void func(int, Color);     ↴ NOT VIABLE

int main()
{
    func(12, 56);
}
```

1. Adım: Candidate Fonksiyonlar belirlenir. Derleyici, argümanları gözlemeğen, **ÖTEKİLERİN Fonksiyonlarını listeden siler!** visible olmasının gerekliliği!

2. Adım: Belirlenen argümanına göre, **Çağrılmasının legal olan, VIABLE Fonksiyonlar belirlenir.** VIABLE OLMAGANLAR listeden silinir!

→ VIABLE olması için:
• Argüman sayısı ile parametre değişken sayıları aynı
(Default argümanları da!)
• Her argümandan geçerli parametreye tır
dönüştürülmeli yepilebilir.

Eğer 2. adımdan sonra bir de **iki VIABLE candidate** kalmışsa

3. Adım:
• Ya Fonksiyonlardan biri seçilebilir olurken ya da ambiguity hatası verilir!
• Artık döşeme kriterine basılır!

② Dönüşmen Kriteri:

- Argümandan, parametreye dönüşümün farklı kriterler var.
- En düşük kalite, **variable dönüşüm!**

```
void func(int, ...);
```

→ İki VIABLE olsaydından biri variable ise, diğer fonksiyon koztur!

User Defined Conversions: - C++ oylan bir dönüştürme

- Normalde legal olmayan bazı değişmeler, bir fonksiyonun bildirimini dikkatinde, desleyenin o fonksiyona gelen argümanı ile değiştirilebilir. Bu da user defined conversion denir!

```
struct Data {
    Data(int);
};

void func(Data);
// argümanı dönüştür

int main() // dönüştür var!
{
    func(12);
}
```

Variadic Conversion dan abla işi!

Standard Conversion: - Daha fazla kural var!

Mehfûl kuralı:

- Character literal 'A' yapıda char

c'de int

```
int main()
{
    char c = 12; // + operatör almışlığı x=char
    auto x = +c; // amaç + operatör olunca, integral promotion, x=int!
}
```

- 3 farklı katagoriye incelenir:
 - Exact match
 - Promotion
 - Conversion
- exact match > promotion > conversion

Exact match: - Argümanın töre ile parametrenin töre bare bir aynı!

- Array Decay de exact match! (Array to pointer conv)
- Low level const exact match!

```
void func(const int*); // => exact match!

int main()
{
    int x = 10;
    func(&x);
}
```

- Function to pointer conversion exact match! →

```
void func(int (*)(int)); // => exact match!

int foo(int);

int main()
{
    func(foo);
```

- Promotion: integer altindaki türlerin int'e dönüsmesi

singed / unsigned short
singed / unsigned char → INT = integral promotion!
bool

Floating, sadexe double'a dönüşümde promotion. → sadexe float sadexe double'a dönüşümde!

- Conversion: - Eğer tür aynı olursa conversion gerçekleşse, ambiguity!

```
void func(double);  
void func(unsigned);  
ambiguity!
```

```
int main()  
{  
    func(10);  
}
```

```
void func(long double);  
void func(char);  
ambiguity!
```

```
int main()  
{  
    func(10.);  
}
```

* Örnekler:

```
// const overloading
```

```
void func(int *);  
void func(const int *);
```

```
int main()  
{  
    const int cx = 5;  
    int ival = 10;  
  
    func(&cx);  
    func(&ival);  
}
```

• const nesne adresiyle çağırırsanız const olan
const olmayan nesne adresiyle çağırırsanız, CONST olmaz!

```
void foo(int &);  
void foo(const int &);
```

return semantikinde de
ayrıdır!

④ Nullpointer Conversion:

```
void foo(int *p);  
void foo(int);
```

```
int main()  
{  
    foo(0);  
}
```

Eğer int parametresi function
olmasa, 0, nullpointer'a dönüs!

```
void foo(void *p);  
void foo(int *);
```

→ Ambiguity!

```
int main()  
{  
    foo(nullptr);  
}
```

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

→ Overload ✓

```
int main()  
{  
    func(0);  
}
```

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

→ Ambiguity!

```
int main()  
{  
    int x = 10;  
    func(x);  
}
```

```
void func(void *);  
  
int main()  
{  
    int ival{};  
    func(&ival);  
}
```

→ variable

```
void func(bool);
```

→ Also visible

→ Pointer to bool conversion

Nullptr = false
Addrs = true gib!

```
int main()  
{  
    int ival = 10;  
    func(&ival);  
}
```

✳ void func(bool);
void func(void *);

→ Bu senyada
void * onerilir!

```
int main()  
{  
    int ival = 10;  
    func(&ival);  
}
```

```
void func(int, double, int);  
void func(double, long, int);  
void func(char, float, double);
```

Boyle birde dekilde kodun seccilebilmesi icin:
- En az bir argumentinde olusturulmali
- Diger argumentlerde, esit yada daha kota olmalidir!

```
|func(10, 20, 5u); → 1. seccim!  
| 10
```

exact
match!

```
int main()  
{  
    func(10, 20, 5.f); → Ambiguity!  
}
```

func 1
exact match

func 3
promotion

func 1 konumuz X

func 1, 1. argumentde otur ama 3. argumentde otur
degil!