

## \* Const expr:

→ const: • const değişkenlere ilk değer vermek ZORUNLU / compile time'da değeri bellidir.

• Bir nesnenin const olması, 0 name ile oluşturulan ifadelerin de const expression olarak anlanma gelmiyor.

örn/: `const int x = 10;`

`const int y = foo();`

`int main()`  
{

`int a1[x]{};` ✓

`int a2[y]{};` X → çünkü foo const expression değil

}

\* Modern C++ ile constexpr geliyor. → `constexpr x = 10;` → implicitly const

`constexpr y = 20;`

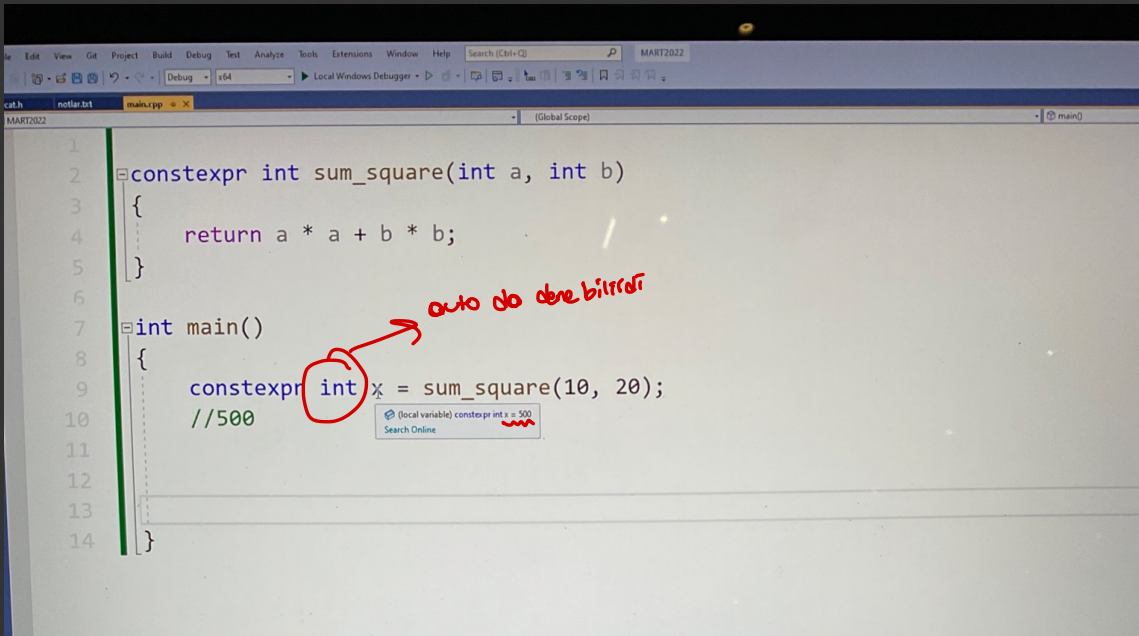
`constexpr z = 30;`

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

\* Aynı şekilde fonksiyonlar da constexpr olabilir.

↳ Bunun avantajı runtime'da yük olarak fonksiyonların değerlerini compile time'da hesaplayabilmek.

örn/:



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

auto da olabilir

(local variable) constexpr int x = 500  
Search Online

### One Definition Rule:

→ Yönlendirici veritiplerin tek tanıtımı olmalı

→ III - form olur eğer ODR olmazsa

→ Header File'da Function Definition Yapmak ODR inline sebep olabilir. !!!

↓  
sadece function  
prototype

→ eğer inline olarak tanıtılırsa  
ODR inline sebep olmaz

→ burada doğru kod yok.

\* 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'nin kendisi

implicitly const

forat

p'nin türü

int\* → const int\* değil

burada const yazılabilir. O zaman p'nin türü const int\* olur.

## \* Function Overloading: → C dilinde olmayan bir özellik

- Aynı isimli birden fazla fonksiyonun aynı scope altında bulunması.
- Tamamen programcının işini kolaylaştırarak işin.
- Compile Time da işin bir mekanizma Run time da bir mekanizma yok.

→ Binding: Eğer fonksiyona yapılan çağrı, hangi fonksiyona ilişik olduğu compile-time da biliniyor: early binding / static binding  
run-time da biliniyor: late binding / dynamic binding

\* Function Overloading olması için → Aynı scope'da bildirilmiş olması.

→ Signature 'ları farklı

↓  
Fonksiyonun Return Değeri Haric parametre sayısı (olduğu parametreler vs...)

\* Function Overload Resolution → Ortada overload varken hangisinin seçileceğinin kararı

→ Bu süreç nasıl işler: 1. Compiler o scope'ta candidate (aday) fonksiyonları bulur.

2. Viability denilerek belirlenir. → sadece bu fonksiyon olmalı, çağrılabilir miydi sorusunun cevabı

→ Argüman sayısı ve parametre sayısı aynı olmalı

→ uygun şekilde tür dönüşümü yapılabilir mi?

\* int\* → overloading çok düşük  
const int\* → level const

\* int → overload değil  
const int

```
1 #include <iostream>
2 #include <cstdlib>
3
4 void func(int double, int);
5 void func(double, long, int);
6 void func(char, float, double);
7
8 int main()
9 {
10     func(10, 20, 5u);
11 }
12
13
14
15
```

error  
mekan

→ Dönüşümün kalitesi belirlenir.

