* Inline Functions: → inline expansion'dan farkını bil.

→ Inline Expansion:

- Bir optimizasyon tekniği (compiler yapar)
- Fonksiyonun çağırıldığı yerin geri dönüşünü tutup, linker'a iletmek yerine direkt çağırıldığı yerde execute edilir.
- Compiler eğer verim artışı olacaksa, inline expansion yapar
- Fonksiyonun inline olması ve inline expansion farklıdır! Inline anahtar sözcüğü ile fonksiyon tanıtmadan da expansion yapılabilir!

- Bir fonksiyon inline olarak tanımlanıp, header'a yazılırsa, ODR kuralının ihlal edilmediğini garantiler

- Eğer header'daki tüm fonksiyonlar, inline tanımlanırsa, bu durumda cpp dosyası olmadan modül oluşturulabilir. Böyle modüllerden oluşan kütüphanelere, header-only library denir.

- Inline fonksiyon tanıtarak, compiler'a inline expansion ŞANSI verirsin. (Garantilemez). Her fonksiyon gibi

- C++ 17 ile inline variables eklendi
  → tanıtıldığı header'ı evleyen tüm dosyalarda sadece 1 adet tanım olacağını garantiler.

- Inline fonksiyonlar, farklı header dosyalarında → token by token, birebir aynı şekilde tanıtılsa bile, syntax hatası olmaz. Linker sadece bir tanesini seçer! ODR ihlali olmaması garantisi!

Fakat, aynı header dosyasında olması syntax hatası!

```
inline int foo(int x, int y)
{
    ///code
    return x * y + 3;
}

inline int foo(int x, int y)
{
    ///code
    return x * y + 3;
}
```

→ Aynı header dosyasında syntax hatası

→ Farklı header dosyasında sorun yok!

- Global fonksiyonlarda olduğu gibi, member fonksiyonlar da inline olarak header da tanımlanabilir. Bu durum ODR'ı ihlal etmez.

```
class Murat {

public:
    void set(int x);
private:
    int mx;
};


inline void Murat::set(int x)
{
    mx = x;
}
```

→ ODR violation yok

- Direct Class Definition içerisinde de fonksiyon tanımlanabilir. Implicitly Inline olur.

```
class Murat {

public:
    void set(int x)
    {
        mx = x;
    }
private:
    int mx;
};
```

→ Set implicitly inline fonksiyon oldu

⑧ C'de Derlenmiş Kodların C++'da Kullanılmasında Yaşanan Hata:

```
#include <iostream>

int sum_square(int, int);

int main()
{
    int x = sum_square(3, 6);

    std::cout << "x = " << x << "\n";

}
```

C kodu bize linker hatası verecek!

- Çünkü external objelere referans C'de underscore ile verilirken, C++'da o fonksiyonun prototipine göre de isimlendirilir.
  ↳ Linker o referansı bulamaz X Notasyon farkı var!

```
extern "C" int sum_square(int, int);

int main()
{
    int x = sum_square(3, 6);

    std::cout << "x = " << x << "\n";

}
```

```
extern "C" {
    void f1();
    void f2();
    void f3();
    void f4();
}
```

→ Bu şekilde de yapılabilir!

* Constructors: - Sınıf nesnesini hayata getiren üye fonksiyon / Ömrünü sonlandıran da destructor

- Bir sınıfın constructor ve destructorları Global veya Static olamaz!!
- Constructorın adı class adı ile Aynı
- Constructorların return kavramı yoktur.
- Constructorlar Overload edilebilir.

* Bir nesnenin, argüman almadan çağrılabilen constructor'ı → Default ctor
→ Ya argüman almayacak, ya da default argument alacak.

→ Special Member Functions: - Default Constructor
- Destructor
- Copy Constructor
- Move Constructor
- Copy Assignment
- Move Assignment

⇒ Derleyici biz yazmasak bile, bu fonksiyonları kendi generate edebilir. / Buna default da denir.

* Constructor ve Destructor CONST OLAMAZ.
- Constructor ve Destructor PRIVATE OLABİLİR ama Client ulaşamaz. Protected olabilir
- Sınıfın içinde inline tanımlanabilir.
- Constructor, non-static olduğu için, this pointer kullanılabilir.

* Destructors:

- Constructor gibi ama başında ~ (tilda) var.
- Global olamaz, static olamaz, const olamaz.
- Parametresi OLAMAZ!! Overload edilemez!

* Statik Ömürlü Nesnelerde Constructor / Destructor. Statik ömür → ya global olarak tanımlanacak.
→ ya da static anahtarıyla

1. Global olarak tanımlanır:

```
};                                  // Bu bir class'ın hem ctor hem de dtor'u var

Nec nx;
Erg ex;                             // main fonksiyonundan önce construct edilir
                                    // " " bitmeden hemen önce destruct edilir.
int main()
{
    std::cout << "main basladi\n";
    std::cout << "main sona eriyor\n";
}
```

Fakat
1. Nec ctor
2. Erg ctor
3. Erg dtor
4. Nec dtor

→ en son construct edilen, ilk destruct edilir.

* Fakat, farklı dosyalarda hangi constructor önce çağrılır belli değil.
↓
Static Initialization fiasco

2. Statik Yerel Değişken: → Bir fonksiyonun statik yerel değişkeni

```
void foo()
{
    static int cnt{};
    static Nec nx;
    std::cout << "foo fonksiyonuna yapilan " << ++cnt << ". cagri\n";
}
```

foo çağırılmadan bu hayata gelmez!

- Bu nesnenin construct edilmesi için tanımlandığı fonksiyonun çağırılması gerekir.
- Fakat statik ömürlü olduğu için

```
Microsoft Visual Studio Debug Console
main basladi
Nec default ctor  this = 001653C4
foo fonksiyonuna yapilan 1. cagri
foo fonksiyonuna yapilan 2. cagri
foo fonksiyonuna yapilan 3. cagri
foo fonksiyonuna yapilan 4. cagri
foo fonksiyonuna yapilan 5. cagri
main sona eriyor
Nec destructor this = 001653C4

D:\KURSLAR\MART2022\Release\MART2022.exe (process 3540) exited with code 0.
Press any key to close this window . . .
```

Fonksiyon kaç kez çağırılırsa, çağırılsın, yalnızca 1 kez nesne girişte constructor'ı çıkışta destructor'ı çağırır.

```
main()
{
    std::cou
    foo();
    foo();
    foo();
```

* Otomatik Ömürlü Sınıf Nesneleri: → Bir fonksiyon içinde tanımlanan nesneler otomatik ömürlüdür.

- Fonksiyon kaç kez çağırılırsa, bu sınıf nesneleri o kadar construct ve destruct edilir.
- Array elemanı class olabilir.
  - → tüm array elemanları için tek tek constructor / destructor çağırılır.

```
void foo()
{
    Nec nx;
}

int main()
{
    for (int i = 0; i < 10; ++i) {
        foo();
    }
}
```

Hayata gelir

Bu noktada destructor çağırılır! / scope'u bitirse!

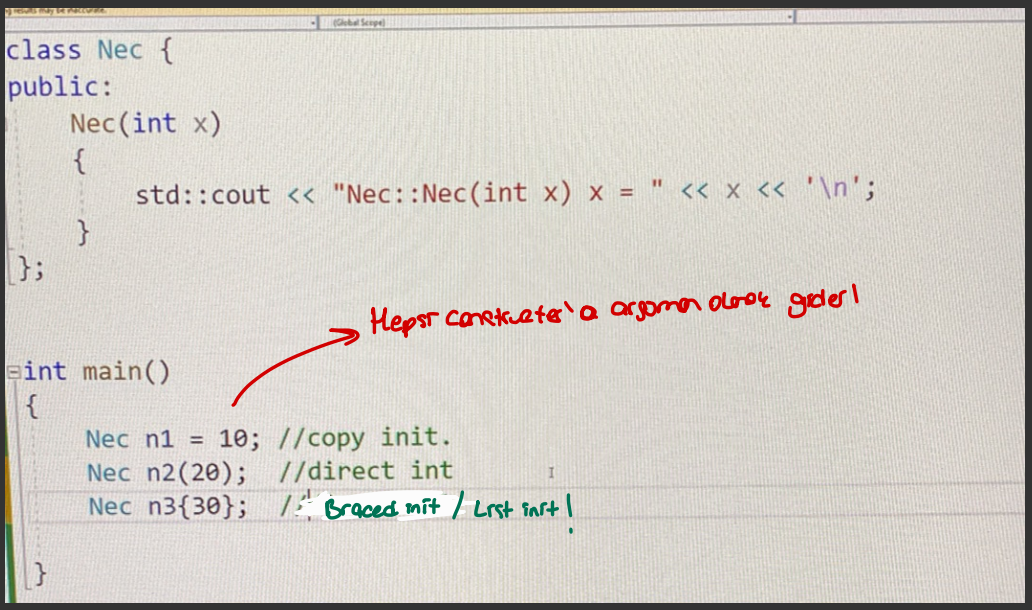
10 kez hayata gelip silinecek!

```
class Nec {
public:
    Nec()
    {
        std::cout << "Nec default ctor  this = " << this << "\n";
    }

    ~Nec()
    {
        std::cout << "Nec destructor this = " << this << "\n";
    }
};

int main()
{
    Nec ar[5];
}
```

Her elemanı için ayrı bir sınıf constructor / destructor edilir!

```
class Nec {
public:
    Nec()
    {
        static int x{};
        std::cout << x++ << ' ';
    }
};

int main()
{
    Nec ar[100];
}
```

→ 0'dan 100'e yazdırma. Milenyal Sayısı!


class Nec {
public:
Nec(int x)
{
std::cout << "Nec::Nec(int x) x = " << x << '\n';
}
};
Hepsi constructor'a argüman olarak gider!
int main()
{
Nec n1 = 10; //copy init.
Nec n2(20); //direct int
Nec n3{30}; //Braced init / List init!
}