16 Mayıs 2022

Özet

temporary object (geçici nesne)
Scope leakage
copy elision
mandatory copy elision
copy elision'ın olduğu 3 yer
temporary materialization
explicit constructor
sınıfların static veri elemanları

---

temporary object (geçici nesne): öyle bir nesne ki value kategori olarak rvalue niteliğinde fakat kodda doğrudan bir ismi yok. İsim vermeden object elde edilir. İsmi olmayan rvalue statüsünde.

```
class Myclass {
public:

};

int main()
{
   Myclass{};  // prvalue expression
}
```

⇒ hayata gelmesi için constructor çağırılır

// parametresi
a) sınıf türünden olan
b) const sınıf türünden ref. olan const class&
c) r value ref. olan int&&
fonksiyonlara argüman olarak geçici nesne gönderebilirim

```
class Myclass {
public:
    Myclass() = default;
    Myclass(int, int);
    void foo();
    void func()const ;
};

void f1(Myclass);
void f2(const Myclass&);
void f3(Myclass&&);
void f4(Myclass&);  // const olmayan sol taraf referansı

int main()
{
    Myclass{12, 45} ; // pr value expr.  // geçici nesne
    Myclass{12, 45}.foo();

    f1(Myclass{});  // call by value
    f2(Myclass{}); // ok  //const lvalue ref.'e rvalue atama
    f3(Myclass{}); // ok
    f4(Myclass{}); // hata // lvalue referansa rvalue ile çağrı yapılamaz  //const lvalue ref.
olsa yapılabilirdi
}
```

buna geçici nesne atanamaz

```
class Date{
public:
    Date(int day, int mon, int year);
};

void func(const Date&);

int main()
{
    Data mydate{16, 5, 2022};
    func(mydate);  // scope leakage
    //  diyelim ki gereksiz yere dosya açtı burada
    sonlanması iyi olur, gereksiz yere kaynak tutulur.
    // destructor hemen burada devreye girmediği için kaynak tutulmaya devam eder.

}
```

*----------

```
int main()
{
  func(Date {16, 5, 2022});  // hemen buranın sonunda geçici nesne için
destructor çağırılır
     // geçici nesne kullanımı ile
    // gereksiz yere kaynak tüketimi engellenir.
    // scope leakage'a neden olmaz
}
```

kapsam sızıntısının iki kötü etkisi (scope leakage)

1) kaynak kullanıyor olabilir, nesne ile işin bitince kaynağın geri verilmesi gerekir
RAII kaynağın destructor tarafından geri verilmesi
2) sonlananı tekrar kullanmak
3) okuyana niyeti belli etmemesi

RAII : resource acquisition is initialization

bu kötü etkileri gidermenin bir yolu
1) scope içine almak

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

}
```

2) Geçici nesne kullanmak

```
int main()
{
    func(Date{ 16, 5, 2022 }); // geçici nesne için destructor çağırılır

    /// code
}
```

eğer geçici bir nesneyi bir referansa bağlıyorsanız
şundan endişe duymanıza gerek yok. O geçici nesnenin scope'u referans devam ettiği sürece devam edecek. O referans dangling hale gelmeyecek.

```
void func()
{
    static Nec mynec; // func fonksiyonu çağırılmazsa mynec nesnesi hayata gelmez !!!
}
```

```
void func(const Nec& rv)
{
    const Nec& r = rv;
    // referansı başka bir ref.'a ilk değer vermek için
    kullandığınızda geçici nesnenin ömrü biter
}

int main()
{
    std::cout << "main basladi\n";
    func(Nec{});

    std::cout << "main sona eriyor\n";
}
```

```
int main()
{
    std::cout << "main basladi\n";
    (void)getchar();

    const Nec &r = Nec{};  // scope sonuna kadar devam eder!!!

    std::cout << "main sona eriyor\n";
}
```

geçici nesneler bir referansa bağlanırsa referansın scope'u devam ettiği sürece ömrü devam eder.

referansı başka bir ref.'a ilk değer vermek için kullandığınızda geçici nesnenin ömrü biter

```
const Nec& foo(const Nec& r)
{
    return r;
}

int main()
{
    std::cout << "main basladi\n";
    (void)getchar();
    const Nec &r = foo(Nec{});
    // yine destructor çağırılır


    const Nec &r = Nec{};  // scope sonuna kadar devam eder!!!

    std::cout << "main sona eriyor\n";
}
```

copy elision: kopyalamanın devre dışı bırakılması, derleyici koda bakarak bir kopyalamayı devre dışı bırakıyor, gereksiz kopyalamadan kaçınmak için optimizasyon. (C++ 17)

derleyiciyi debug modda çalıştırınca birçok optimizasyon yapmaz. Copy elision'ı release'de yapıyor, debug modda optimizasyon olmuyorsa copy elision'ın mandatory olmadığı anlaşılabilir.

mandatory copy elision: (mecburi)

copy elision'ın olduğu 3 yer var

1) mandatory copy elision
fonksiyonun parametresi sınıf türünden ve nesneyi pr value ile çağırmak

ister debug modda, ister gcc., optimizasyon muhakkak yapılıyor

```
void foo(Myclass x)
{
}

int main(){
    foo(Myclass{});  // pr value ile hayata getirmek
     // copy const. çağırılmaz
}
```

→ assignment değil dikkat!!

veya

```
int main(){
    Myclass x = Myclass{};  // önce default const.   geçici nesne
oluşturulur, sonra x nesnesi copy  const. ile oluşturulur.
   Gerçekleşen :  copy const. çağırılmaz
}
```

2) return value optimization RVO (mandatory)

fonksiyonun geri dönüş değeri bir sınıf türünden
pr value ile fonksiyon return etti. Fonksiyon çağrısını bir nesneyi hayata getirmek için kullanmak

```
Myclass foo()
{
   return Myclass{};
}

int main()
{
   Myclass x = foo();
   // beklenti : foo() çağırıldığında return için bir temporary object oluşturulacak, bu temp. object için default const. çağırılacak,  bu  da x nesnemizi hayata getirecek
   // gerçek : bir kez default const. çağırılmış
   // assembly düzeyinde bakarsak fonks. geri dönüş değerinin yazıldığı bir yer var.  Ona return slot diyelim. Derleyici bunu gördüğünde önce foo() içindeki nesneyi hayata getirip ondan sonra kopyalamak yerine doğrudan Myclass x nesnesinin hayata getiriyor.

   // derleyici burada sadece bir x yaratılmış diye optimize eder.
   // return value optimization RVO
}
```

3) named return value optimization NRVO (not mandatory)

```
Myclass foo()
{
    Myclass x;
    return x;
}

int main()
{
    Myclass x = foo();  // derleyici burada sadece bir x yaratılmış diye optimize
eder.
    // named return value optimization NRVO
    // debug modda derlersen
}
```

---

temporary materialization : prvalue karşılığı bir nesnenin oluşturulması demek

Myclass{} => nesnenin kendisi değil ama bir nesneyi initialize ediyor.

Myclass x = Myclass{}; // x nesnesinin Myclass{} ile init. et, temporary materialization yok

temporary materialization için Myclass{} ifadesini bir referansa bind ederseniz, o zaman prvalue expr.'dan nesne oluşturur.

```
Myclass foo()
{

    return Myclass{};
}

int main()
{
    Myclass x = foo(); // pr value nesne demek değil // temporary materialization yok

    const Myclass& r = Myclass {}; // geçici nesne bir referansa (sol veya sağ taraf
referansına ) bağlanırsa, temporary materialization var.

    Myclass&& x = foo(); // temporary materialization var, ref. atanmış,

    Myclass x = Myclass{Myclass{Myclass{}}};
    // temp. materialization yok.

}
```

Mülakat sorusu : kodu yorumla

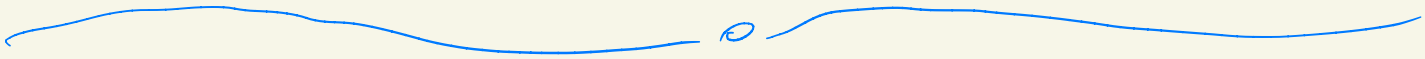
```
Myclass foo()
{
    Myclass x;
    // code

    return x;
}

Myclass m = foo();
```

cevap:
en iyi ihtimalle name return value optimization olacak. Ne copy constructor ne move constructor çağırılacak. Bir kez default const. çağrılacak. Ama bu mecburi değil. Aradaki kodların ne olduğuna bağlı olarak
derleyici bu optimizasyonu yapabilir veya yapmayabilir. optimizasyon olup olmayacağı derleyici ve kodların ne olduğuna bağlı.
2) sınıfın move const. varsa x otomatik ömürlü
olduğundan hayata gelecek m nesnesi için move const. çağırılacak ama move const. yoksa copy const. çağırılır

1) default const. çağırılması
2) move const. varsa çağırılması
3) move const. yoksa copy const. çağırılması



```
class Myclass{
public:
    Myclass() = default;
};

int main()
{
    Myclass m;
    m = 10; //  sentax hatası, türleri farklı
    // int türünden class türüne örtülü dönüşüm yok

}
```

conversion constructor (dönüştüren kurucu işlev) : special member function değil !!

adeta tür dönüştürme için kullanılıyor

```
class Myclass{
public:
    Myclass()
    {
        std::cout << "Myclass default const. \n";
    }

    ~Myclass()
    {
        std::cout << "Myclass destructor this : " << this << \n";
    }
    Myclass(int)     // conversion constructor (dönüştüren kurucu işlev)
    {
        std::cout << "Myclass(int x) x  = " << x <<  "this :" << this << "\n";
    }
};

int main()
{
    Myclass m;
    (void)getchar();
    m = 10; //  ok  // burada int parametreli Myclass(int) constructor çağırıldı ve hemen ardından this nesnesi için destructor çağırıldı. Bu this nesnesi sadece bu atamanın yapılması için kullanıldı.
    std::cout << "main devam ediyor\n";

    // derleyicinin ürettiği pseudo code:
    /*
        m = Myclass{10};
    */

    // m nesnesi için destructor çağırılır
}
```

user-defined conversion (programcı tarafından oluşturulan dönüşüm): eğer bir conversion normal olarak yok ise fakat bizim yazdığımız bir fonkiyonunun kullanılması ile yapılabiliyorsa buna user-defined conversion denir.

```
class A{
public:

};

class B{
public:
    B();
    B(A); // A'dan B'ye dönüşüm var çünkü B'nin A parametreli const.'ı var.
};

class C{
public:
    C(B);  // B'den C'ye otomatik dönüşüm var çünkü C'nin B parametreli const.'ı var.

};

int main()
{
    A ax;
    B bx;

    bx = ax;  // nasıl geçerli oluyor ?
    // çünkü B sınıfının A türünden parametreye sahip bir const. olduğu için B sınıfının B(A) bir conversion const.'dır.

    cx = ax; // sentax hatası, neden sıralı dönüşüm yapmadı. Çünkü iki tane user defined conversion + user defined conversion arka arkaya çağırılamaz

    cx = static_cast<C>(ax) ; // ikinci dönüşümü örtülü değil de açık olarak tür dönüşümü operatörü ile yapıldığı için ok.
}
```

implicit conversion: (örtülü dönüşüm) derleyicinin durumdan vazife çıkarması ile yapılır.

explicit conversion : bir dönüşüm tür dönüştürme operatörleri kullanılarak yapılır.

```
class Myclass{
public:
    Myclass();
    Myclass(int);
};

void func(Myclass);
void foo(const Myclass&);
void foo2(Myclass&);

Myclass bar()
{
    return 10; // diyelim ki bilmeden kullandım
    // hata yok, conv. constr. devreye girer.
    // fakat tehlikeli durum !!!
    // niyetiniz dışında hata yapmaya neden olur.
}

int main()
{
    Myclass m = 10;  // conversion const.
    Myclass m2(10); // ok
    Myclass m3{10}; // ok

    func(10);
    foo(20);
    foo2(20);  // hata, pr value lvalue'a bağlanamaz

    int ival{4};
    func(static_cast<Myclass>(ival));
    // bunu kullanmak çok daha açıklayıcı !!!
    // conv. const. bunu yapması yerine niyetini explicit olarak belli et !!!
    func(ival);  // örtülü dönüşüm tehlikeli !!!

}
```

Dönüşüm yapacaksan implicit olarak değil explicit olarak yap.

*----------

explicit constructor :

```
class Myclass{
public:
    explict Myclass(int);   // explicit only
};

Myclass::Myclass()  // explicit anahtar sözcüğü sadece bildirimde olmalı,
```

tek parametreli constructor ikna edici bir neden olmadığı sürece explicit olarak bildirilmeli

```
int main()
{
    int x(10); // direct init.
    int x = 10; // copy init.
}
```

*----------

```
class Myclass{
public:
    explicit Myclass(int);
};

int main()
{
    Myclass m = 10;  // hatalı
    Myclass m2(10);
    Myclass m3{10};
}
```

Dikkat !!
constructor'ın explicit olmasının bir başka etkisi
copy init.'de kullanılamaması
fakat direct veya brace init'de hata vermez

```
class Myclass {
public:
    Myclass(int, int);
};

int main()
{
    Myclass m = {12, 45}; // geçerli
}
```

*----------

```
class Myclass {
public:
    explicit Myclass(int, int);
};

int main()
{
    Myclass m = {12, 45}; // sentax hatası
}
```

*----------

class definition içinde bildirilen isimlere sınıfların member'ları deniliyordu.

Member'lar üç kategoriye ayrılıyor
a) data member
b) member function
c) member type (nested type)'ları

data member'lar ikiye ayrılır
a) non-static data members
b) static data members

member functions
a) non-static member functions
b) static member functions

sınıfların static veri elemanları:
class variable olarak geçer (instance variable değil)
global değişkenlere bir alternatif
global değişkene bütün kodlar erişebilir fakat static eleman bir sınıf içerisindedir.
global access control yok
fakat static access control var ve public, private, protected olabilir.

```
class Myclass{
    static int x;
};

int main()
{
    Myclass m;

    m.x = 10; // ok, her ne kadar m nesnesine ait izlenimi veriyorsa da, m ile alakası yok, m sadece namelookup amaçlı kullanılır.
}
```