D Programlama Dili

Ali Çehreli


```
import std.stdio;

void main()
{
    writeln("Merhaba dünya.");
}
```

22 Temmuz 2011; Bahçeşehir Üniversitesi, Beşiktaş

Tanışma

- Hangi forumlardasınız?
- Hangi dilleri kullanıyorsunuz?
- Düzenli olarak katıldığınız toplantılar var mı?

Asıl İzlemeniz Gereken

Andrei Alexandrescu'nun "Three Cool Things About D" isimli sunumu.

- Dilin yaratıcılarından.
- Üç önemli D olanağını anlatmaya çalışır.
- Ama zamanı yetmez.
- Buradaki bazı fikirleri oradan ödünç alıyorum.

Çok İyi Başka Diller de Var

- Burada amaç başka dilleri eleştirmek değil, D'yi tanıtmak.
- Her dilin zayıf ve güçlü tarafları vardır.
- D'nin olanaklarının neden farklı olduklarını anlatabilmek için C'nin ve C++'ın D'den daha zayıf olarak görülen bazı taraflarını göstereceğim.
- C ve C++ kullanıyorum.

Kaynaklar

Türkçe

- Genel: http://ddili.org/
- Kitaplar: http://ddili.org/ders/
  - D Programlama Dili, *Ali Çehreli*
  - GtkD ile Görsel Programlama, *Mengü Kağan* ve *Can Alpay Çiftçi*
  - SDL ile Oyun Programlama, *Faruk Erdem Öncel*
- Makaleler: Andrei Alexandrescu'nun makalesi “Neden D” ve başkaları
- Forum: http://ddili.org/forum/
- Wiki (Kütüphane belgeleri, vs.): http://ddili.org/wiki/

İngilizce

- Yeni: http://d-programming-language.org/
- Eski: http://digitalmars.com/
- Kitap: "The D Programming Language", Andrei Alexandrescu
- Wiki: http://www.prowiki.org/wiki4d/
- vs.

Kişiler ve Tarihçe

- Walter Bright
  - Empire oyununun yazarı
  - İlk C++ derleyicisi: Zortech'in yazarı
  - Digital Mars'ın C ve C++ derleyicilerinin yazarı
  - D1'in yaratıcısı, 2001
  - D'nin yaratıcısı, 2007 (Eskiden D2 deniyordu.)
- Andrei Alexandrescu
  - "C++ şablon sihirbazı" ünvanlı, "Modern C++ Design" ve başka kitapların yazarı
  - Programlama dili uzmanı (Yüksek lisans konusu ve sonra D tasarımı)
  - Phobos aralıklarının (range) yaratıcısı (Bkz. "Eleman Erişimi Üzerine" makalesi)
- Çok sayıda gönüllü
  - github'da dmd (D derleyicisi)
  - github'da Phobos (Standart D kütüphanesi)

Genel Bakış

- Üst düzey sistem dili
- Çok hızlı derleme
- Çok hızlı programlar
- C (ve bazı C++) kütüphaneleri ile bağlanabilme (link)
- Baştan C++'tan türemiş (gibi)
- Java, C#, Haskell, Eiffel ve başka bir çok dilden olanaklar
- Hem çöp toplayıcılı hem değil (RAII, el ile, vs.)
- Güvenli bellek modeli
- Kod güvenliği ve doğruluğu ön planda
- Komite değil, programcılar tarafından geliştirilmiş
- Pratiklik önemli

Derlemeli Bir Dil

- Derleyiciler:
  - **dmd**: Digital Mars'ın derleyicisi
  - **gdc**: gcc'nin D derleyicisi; şimdilik ana sürümle gelmiyor; bir geliştirici sürümünden kurmak gerekiyor
  - **LDC**: Ön tarafta dmd, arka tarafta LLVM; şimdilik D1 ile sınırlı
- Dilin belirli bir alt kümesi, CTFE olanağı (Compile Time Function Execution) sayesinde derleme zamanında işletilebiliyor:

```
    enum menü = menüHazırla([ "Ekle", "Sil", "Çık" ]);
```

  menü, bütünüyle derleme zamanında oluşturulur.

- *Hash bang* düzeneği ile kaynak kod bile çalıştırılabilir:

```
#!/usr/bin/env rdmd

import std.stdio;

void main()
{
    writeln("Merhaba dünya.");
}
```

C Ailesinden

- Söz dizimi C ve C++'a çok benzer ama çok sorunsuzdur: Her karakter bir kere okunur; her isim bir kere yüklenir.
- İlintisi (binding) bulunan C kütüphanelerini doğrudan kullanabilir. C kütüphanesi (**`malloc()`**, **`free()`**, vs.), Qt, Gtk, curl, vs.

```
// Örnek ilinti (D binding) dosyası
module benim_ncurses;

extern (C):

enum TRUE = 1;
enum FALSE = 0;

alias void WINDOW;

WINDOW * initscr();
int cbreak();
int noecho();
// ...
```

- Göstergeler kadar alt düzeydir

```
    int i;
    int * p = &i;
    *p = 42;
```

Programlama Yöntemleri

- Emirli (imperative)

```
    auto isimler = [ "Ayşe", "Barış" ];

    foreach (isim; isimler) {
        writefln("Merhaba %s.", isim);
    }
```

- Nesne yönelimli (object oriented)

```
class Kedi : Hayvan
{
    string şarkıSözü() { return "miyav"; }
}
```

- Fonksiyonel (functional)

```
    auto işlem = (int sayı) { return sayı * 2; };
    işlem(42);
```

- Meta programlama (meta programming)

```
T topla(T)(T a, T b)
{
    return a + b;
}
```

Program Doğruluğu

- Temel bir felsefe: "En kolay yöntem, en güvenli ve doğru yöntemdir de." Örnekler:
  - Bütün değişkenler normalde ilklenirler ama istendiğinde ilklenmeyebilirler de
  - Değişkenler normalde iş parçacıkları tarafından paylaşılamazlar (thread-local) ama paylaşılabilirler de
  - Dizi indeksleri normalde denetlenirler ama denetlenmeyebilirler de
  - vs.
- Safe D: Dilin bellek hatalarına izin vermeyen alt kümesi (`@safe`)
- Birim testleri (unit testing)
- Sözleşmeli programlama ("contract programming" veya Eiffel'in "design by contract" kavramı): İşlevlerin giriş ve çıkış koşulları ve yapıların ve sınıfların mutlak değişmezleri
- Hata yönetimi
  - Hata atma düzeneği (exception handling)
  - Sonlandırıcı işlevlerden (destructor) bile hata atılabilir; ek hatalar asıl hataya *bağlı liste* düğümleri biçiminde eklenirler.
  - `scope` deyimi sayesinde RAII türleri tanımlamak gereksizdir

D'nin *Merhaba Dünya* Programı Doğrudur

```
import std.stdio;

void main()
{
    writeln("Hello, world!");
}
```

C'nin ve C++'ın *Merhaba Dünya* Programları Yanlıştır

"Neden D" makalesinde ve "Three Cool Things About D" sunumunda Andrei Alexandrescu'nun *program doğruluğu* ile ilgili olarak söylediği:

"C'nin ve C++'ın *merhaba dünya* programları yanlıştır."

C:

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

C++:

```
#include <iostream>
int main()
{
  std::cout << "Hello, world!\n";
}
```

Merhaba Dünya Programlarının Doğruları

C programının doğrusu:

```c
#include <stdio.h>
int main()
{
  return printf("hello, world\n") < 0;
}
```

C++ programının doğrusu:

```cpp
#include <iostream>
int main()
{
  std::cout << "Hello, world!\n";
  return std::cout.bad();
}
```

Java, Perl, J#, vs. de yanlıştır. Python, C#, vs. doğrudur.

Safe D

`@safe` ile işaretlenmiş olan işlevlerde veya **`-safe`** ile derlenmiş modüllerde bellek hatalarına izin veren olanaklar kullanılamaz. Örneğin şu olanaklara derleyici izin vermez:

- Program içinde *assembler* kullanımı.
- Bir değeri göstergeye dönüştürmek.
- Tehlikeli gösterge kullanımları.
- **`const`**, **`immutable`** veya **`shared`** belirteçlerinin tür dönüşümü yoluyla kaldırılması.

Kısıtlı olarak izin verilen bir olanak:

- **`T1*`** türü, **`T2*`** türüne ancak güvenli yönde dönüştürülebilir. Örneğin **`T* → void*`** veya **`int* → short*`** olur.

Birim Testleri (Unit Testing)

Programcının en büyük yardımcısı

```
string tekrarla(string dizgi, int adet)
{
    // ...
}

unittest
{
    assert(tekrarla("abc", 2) == "abcabc");
    assert(tekrarla("ğ", 5) == "ğğğğğ");
    assert(tekrarla("a", 0) == "");
}
```

Bir yapının birim testleri:

```
struct Dizi
{
    @property int uzunluk();
    void ekle(int eleman);
}

unittest
{
    auto dizi = Dizi();
    assert(dizi.uzunluk == 0);

    dizi.ekle(42);
    assert(dizi.uzunluk == 1);
}
```

Sözleşmeli Programlama
(Contract Programming)

İşlevlerde giriş koşulları için **in**, çıkış koşulları için **out** blokları:

```
string tekrarla(string dizgi, int adet)
in {
    assert(!dizgi.empty);
}
out (sonuç) {
    assert(sonuç.length == (dizgi.length * adet));
}
body {
    string sonuç;
    foreach (i; 0 .. adet) {
        sonuç ~= dizgi;
    }

    return sonuç;
}
```

Yapılarda ve sınıflarda mutlak değişmezler için **invariant** blokları:

```
class Zaman
{
    int saat;
    int dakika;

    invariant()
    {
        assert((saat >= 0) && (saat <= 23));
        assert((dakika >= 0) && (dakika <= 59));
    }

    // ... 'saat' ve 'dakika'yı değiştiren çeşitli işlevler ...
}
```

RAII gibi scope

Dosya kopyalayan bu program ya başarıyla kopyalar ya da geride yarım dosya bırakmaz.

```
import std.exception;
import std.file;

void main(string[] parametreler)
{
    enforce(parametreler.length == 3,
            "Kullanım: trcopy <kaynak> <hedef>");

    auto araDosya = parametreler[2] ~ ".ara_dosya";
    scope(failure) if (exists(araDosya)) remove(araDosya);
    copy(parametreler[1], araDosya);

    rename(araDosya, parametreler[2]);
}
```

(Kabul: **`exists()`** veya **`remove()`** da başarısız olabilir.)

- **`scope(failure)`**: Hatayla çıkıldığında
- **`scope(success)`**: Başarıyla çıkıldığında
- **`scope(exit)`**: Herhangi bir yolla çıkıldığında

Modül Sistemi

- **#include** değil, **import**
- **#ifndef** başlık korumalarına gerek yoktur.
- Bildirim ve tanım ayrımı yoktur.
- Çoğu durumda tanım sırasının önemi yoktur.

```
module deneme;        // bu modülün ismi

import std.stdio;     // standart giriş/çıkış modülünü kullanıyor

void main()
{
    selamVer();       // önceden bildirmeye gerek yok
}

void selamVer()
{
    writeln("Merhaba dünya.");
}
```

Temel Türler

C'den ve C++'tan farklı olarak türlerin uzunlukları standarttır (**real** hariç).

```
// Mantıksal ifade türü:
bool b;

// Karakter türleri:
char  c_8;
wchar w_16;
dchar d_32;

// Tamsayı türleri:
byte  b_8;
short s_16;
int   i_32;
long  l_64;
// cent c_128;  ileride kullanılmak üzere

/* Yukarıdakilerin işaretsiz olanları:
 *   ubyte, ushort, uint, ulong, ucent */

// Kesirli sayı türleri:
float  f_32;
double d_64;
real   r_platforma_bağlı;    // benim ortamımda 128 bit

// Kısıtlı olarak 'void':
void[] baytlar;

/* Yakında kütüphanede halledilecek olan karmaşık sayı türleri:
 *   cfloat, cdouble, creal, ifloat, idouble, ireal */
```

Dizgiler Unicode'dur

Hem kaynak kodda hem dizgilerde:

```
    bool ASCII_geçmişte_kaldı = true;

     string söz    = "Görüşelim dünya.";    // UTF-8
    wstring söz_16 = "Görüşelim dünya.";    // UTF-16
    dstring söz_32 = "Görüşelim dünya.";    // UTF-32
```

Dizgi hazır değerlerinin kodlamaları açıkça belirtilebilir:

```
    kullan("merhaba"w); // UTF-16
    kullan("merhaba"d); // UTF-32
```

const ve immutable

C ve C++ gibi başka dillerdeki **const** bazen yetersizdir: **const** olan bir referansla erişilen değerin değişip değişmeyeceğinden emin olunamaz.

- **const**, C'deki ve C++'taki anlamdadır: "bu referans yoluyla erişildiğinde değiştirilemez". Başka yerde değiştirilebilir.

```
int seneSonuNotu(const ref Öğrenci öğrenci)
{
    /* öğrenci burada değiştirilemez ama çağrıldığı yerde
     * değişiyor olabilir.
     * ...
     */
}
```

- **immutable**, "kesinlikle değiştirilemez" anlamındadır.

```
    immutable raporNumarası = 42;
    string s = "merhaba";   // immutable(char)[]
```

  - Eş zamanlı programlamada kilitlenmesi gerekmez.
  - Kesinlikle değişmediği bilindiğinden derleyicinin daha fazla eniyileştirme olanağı vardır.

İkisi de geçişlidir:

```
int seneSonuNotu(const ref Öğrenci öğrenci)
{
    öğrenci = new Öğrenci;   // ← DERLEME HATASI
    öğrenci.numara = 100;    // ← bu da DERLEME HATASI
    // ...
}
```

Nesne Yaşam Süreçleri

Seçime bağlıdır:

- Çöp toplayıcı
- RAII
- Açıkça programcı tarafından

Yapılar (struct)

C++'takinden farklı olarak C köklerine çok daha yakındır.

```
struct Nokta
{
    double x;
    double y;

    void yansıt()
    {
        x = -x;
        y = -y;
    }
}
```

- Değer türüdür.
- Normalde C++'taki gibi hemen sonlandırılır ama **new** ile oluşturulduğunda çöp toplayıcı tarafından da sonlandırılabilir.
- Nesne yönelimli programlamayı desteklemez.

Sınıflar (class, interface)

C++'taki **struct** ve **class**'a yakındır.

```
interface Hayvan
{
    void ilerle(int mesafe);
}

interface Şarkıcı
{
    string söyle();
}

class Kedi : Hayvan, Şarkıcı
{
    void ilerle(int mesafe)
    {
        foreach (i; 0 .. mesafe) {
            writeln("tıpış");
        }
    }

    string söyle()
    {
        return "miyav";
    }
}
```

- Referans türüdür.
- Normalde çöp toplayıcı tarafından sonlandırılır ama **scoped()** ile oluşturulduğunda C++'taki gibi kapsamdan çıkılırken sonlandırılabilir.
- Nesne yönelimli olanaklar içindir; çoklu kalıtım yoktur ama birden fazla **interface**'ten türetilebilir.

C Dizileri (?)

C dizilerinin bir eleştirisi:

- C'de dizi hem vardır hem yoktur. Tek değişken göstergesi dizi yerine geçer. Bir gösterge ve bir uzunluk, anlaşmaya bağlı olarak dizi olarak kullanılır.
- Bazen değer türüdür (örneğin bir yapı üyesi olduğunda)
- Bazen referans türüdür (örneğin işlev parametresi olduğunda)
- Diziler üzerinde hiçbir işlem tanımlı değildir (belki **`memmove()`**). Örneğin eleman eklemek için **`realloc()`** ile açıkça yer ayırmak gerekebilir.
- Söz dizimi *ters yüzdür*

```
char alan[1][2];  // 1 adet char[2] mi, 2 adet char[1] mi?
printf("%zu", sizeof(alan[0]));       // 1 mi yazar, 2 mi?
```

(Neyse ki C++'ta **`std::vector`** var.)

Daha fazla bilgi için bkz. Walter Bright'ın "C's Biggest Mistake" ("C'nin En Büyük Hatası") adlı makalesi.

Sabit Uzunluklu Diziler

- Uzunluğu değiştirilemez.
- Değer türüdür.
- İndeks hatası derleme zamanında yakalanabilir.

```
int[3] dizi= [ 10, 42, 100 ];
assert(dizi.length == 3);
dizi[0] = 11;
int a = dizi[5]; // derleme zamanı hatası
```

Söz dizimi doğaldır:

```
// Her zaman için Tür[uzunluk]
char[2] dizi;               // 2 adet char

char[2][1] alan;            // 1 adet char[2]
writeln(alan[0].sizeof);    // 2 yazar
```

D zamanla güçlendiğinden şimdilik dilin iç olanağı olan diziler yakında kütüphane olanağı olacaklar.

Dinamik Diziler

- Eleman adedi değişebilir.
- Referans türüdür.
- *Ekleme* işlemi için **~** işleci kullanılır

```
int[] dizi= [ 10, 42, 100 ];
dizi.length += 10;
assert(dizi.length == 13);
dizi[0] = 11;
int a = dizi[20]; // çalışma zamanı hatası

dizi ~= 7;
auto yeniDizi = dizi ~ dizi;
```

Aslında *dilimlerle* aynı şeydir. Daha fazla bilgi için bkz. Steven Schveighoffer'in "D Dilimleri" makalesi.

Dilimler (Slices)

Çok hızlı, çok kolay ve çok güvenli.

```
    int[] dizi = [ 10, 20, 30, 40 ];
    int[] dilim = dizi[1..3];           // 20 ve 30
```

Bir dizginin palindrom olup olmadığını bildiren bir işlev:

```
bool palindrom_mu(string dizgi)
{
    return ((dizgi.length < 2)
            ||
            ((dizgi[0] == dizgi[$-1]) &&
             palindrom_mu(dizgi[1..$-1])));
}

unittest
{
    assert(palindrom_mu("abccba"));
    assert(!palindrom_mu("abca"));
}
```

main'in parametrelerini işleyen bir program:

```
void main(string[] parametreler)
{
    // Program ismini atla
    işle(parametreler[1..$]);
}
```

Eşleme Tabloları (Associative Arrays)

Çok hızlı bir *hash table* gerçekleştirmesidir.

string'den **string**'e dönüştüren bir tablo:

```
    string[string] renkler = [ "red" : "kırmızı",
                               "blue" : "mavi",
                               "green" : "yeşil" ];

    writeln(renkler["red"]);   // 'kırmızı' yazar
```

string'den **double**'a dönüştüren bir tablo:

```
    double[string] evrenselSabitler;

    evrenselSabitler["pi"] = 3.14;
    evrenselSabitler["e"] = 2.72;
```

toHash() üye işlevini tanımlayan yapılar ve sınıflar da indeks türü olarak kullanılabilirler.

Eskiden dilin iç olanağıydı; şimdi bütünüyle kütüphane olanağı.

Şablonlar (Templates)

- İşlev şablonları

```
auto min(SolTür, SağTür)(SolTür soldaki, SağTür sağdaki)
{
    return sağdaki < soldaki
           ? sağdaki
           : soldaki;
}
```

- Yapı ve sınıf şablonları

```
struct Nokta(T = long, int boyut = 2)
{
    T[boyut] koordinatlar;

    void yansıt()
    {
        foreach (ref koordinat; koordinatlar) {
            koordinat = -koordinat;
        }
    }
}

alias Nokta!(double, 3) Nokta3D;

// ...

    Nokta3D merkez;
```

Türden Bağımsız Programlama

- Şablonlar çok güçlü ve çok kolay
- Şablon kısıtlamaları (C++0x'e giremeyen *concepts* olanağı)

```
struct Eksili(T)
    if (isInputRange!T)
{
    // ...
}
```

- **`static if`** çok kullanışlı

```
void baytlarınıGöster(T)(ref T değişken)
{
    // ...
    static if (isArray!T) {
        writeln("elemanlar:");
        // ...
    }
    // ...
}
```

- **`static assert`** dile dahil

```
    static assert((T.sizeof % 4) == 0,
                  "Türün büyüklüğü dördün katı olmalıdır");
```

Phobos: Standart Kütüphane

- Çok sayıda modül içeriyor; hâlâ gelişiyor
  - **`std.algorithm`**: çok sayıda türden bağımsız algoritma
  - **`std.range`**: aralıklar
  - **`std.array`**: dizileri aralık olarak kullandıran olanaklar, vs.
  - **`std.string`**: dizgi olanakları
  - **`std.parallelism`**: koşut programlama
  - **`std.concurrency`**: eş zamanlı programlama
  - vs.
- Algoritmalarla toplulukları bir araya getirmek için *aralık soyutlamasını* kullanan ilk kütüphane

Koşut İşlemler için `std.parallelism` Modülü

Bağımsız işlemleri aynı anda işletmek için.

- Tek çekirdek üzerinde 4 saniye:

```
    auto öğrenciler =
        [ Öğrenci(1), Öğrenci(2), Öğrenci(3), Öğrenci(4) ];

    foreach (öğrenci; öğrenciler) {
        öğrenci.uzunBirİşlem();
    }
```

- Dört çekirdek üzerinde 1 saniye:

```
    foreach (öğrenci; parallel(öğrenciler)) {
```

Eş Zamanlı Programlama

Birbirine bağlı işlemleri aynı anda işletmek için (threads).

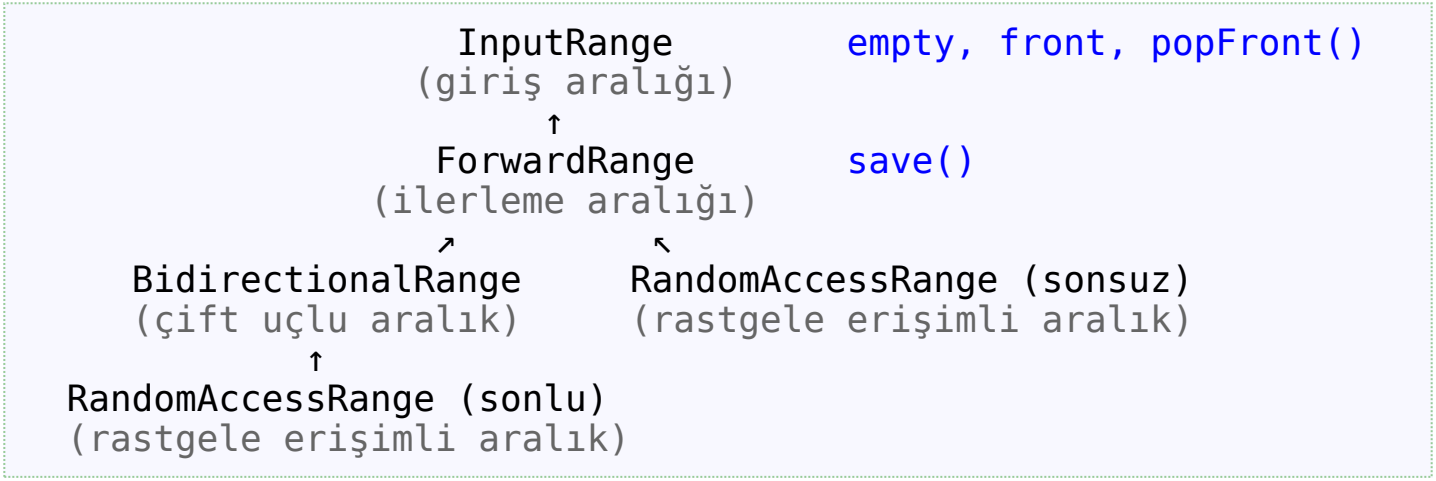
- En dertli programcılık sorunlarından birisidir.
- Verilerin doğrudan paylaşılmalarına dayanan eş zamanlı programlamada programın doğruluğunun kanıtlanamayacağı kanıtlanabilir. (Buna rağmen D bu yöntemi de destekler.)
- D bellek modelinde veriler normalde paylaşılamazlar (thread-local). Yalnızca değişmez oldukları için **`immutable`** veriler ve açıkça **`shared`** ile işaretlenen veriler paylaşılabilirler.
- Veri paylaşımı için en güvenli yöntem *mesajlaşmadır* (messaging).
- **`std.concurrency`** modülü mesajlaşma yöntemini uygular.

std.concurrency Modülü

```
import std.stdio;
import std.concurrency;

void main()
{
    auto işçi = spawn(&işParçacığı);

    işçi.send(42);
    işçi.send("merhaba");
    işçi.send(Sonlan());
}

struct Sonlan
{}

void işParçacığı()
{
    bool tamam = false;

    while (!tamam) {
        receive(
            (int mesaj) {
                // ...
            },

            (string mesaj) {
                // ...
            },

            (Sonlan mesaj) {
                tamam = true;
            });
    }
}
```

Aralıklar

- Algoritmalar bir kere yazılmalıdırlar ve her toplulukla o topluluğa en uygun biçimde işlemelidirler.
- Bunun için bir *eleman erişimi soyutlaması* gerekir.
- C++'ın STL'si çok güçlü bir soyutlamadır ama hem yeterli değildir hem de sorunları vardır: güvenlikten yoksunluk, kullanım güçlüğü, tanım güçlüğü, ve C++ diline fazla bağlılık.
- Andrei Alexandrescu aralık soyutlamasını "Eleman Erişimi Üzerine" adlı makalesinde ortaya atar.



Giriş Aralığı: `InputRange`

Üç üye işlev herhangi bir türü giriş aralığı yapmaya yeter:

- **`empty`**: Aralık boş olduğunda **`true`** değerini vermelidir.
- **`front`**: Aralığın başındaki elemanı vermelidir.
- **`popFront()`**: Aralığı başından daraltmalıdır.

```
[ 10, 11, 12 ]
    [ 11, 12 ]
        [ 12 ]
            [ ]
```

```
struct FibonacciSerisi
{
    int baştaki = 0;
    int sonraki = 1;

    static immutable bool empty = false;  // ← sonsuz aralık

    @property int front() const
    {
        return baştaki;
    }

    void popFront()
    {
        int ikiSonraki = baştaki + sonraki;
        baştaki = sonraki;
        sonraki = ikiSonraki;
    }
}
```

InputRange Kullanım Örneği

```
    writeln(take(cycle(take(FibonacciSerisi(), 5)), 20));
```

Çıktısı:

```
[0, 1, 1, 2, 3, 0, 1, 1, 2, 3, 0, 1, 1, 2, 3, 0, 1, 1, 2, 3]
```

Açık yazılımı:

```
    auto seri                        = FibonacciSerisi();
    auto başTarafı                   = take(seri, 5);
    auto tekrarlanmışı               = cycle(başTarafı);
    auto tekrarlanmışınınBaşTarafı = take(tekrarlanmışı, 20);

    writeln(tekrarlanmışınınBaşTarafı);
```

Hepsi tembeldir. Asıl işlemler **`popFront()`** sırasında aralık *baş tarafından daraltılırken* halledilir.

Aralık Kullanan Başka Bir Aralık

```
struct Eksili(T)
    if (isInputRange!T)
{
    T aralık;

    this(T aralık)          { this.aralık = aralık; }

    @property bool empty() { return aralık.empty; }

    @property auto front() { return -aralık.front; }

    void popFront()         { aralık.popFront(); }

    static if (isForwardRange!T)
    {
        Eksili!T save()     { return Eksili!T(aralık.save()); }
    }
}

Eksili!T eksili(T)(T aralık)
{
    return Eksili!T(aralık);
}
```

```
    writeln(eksili(take(FibonacciSerisi(), 5)));
```

Çıktısı:

```
[0, -1, -1, -2, -3]
```

Sorular?