C++11'in Bazı Yenilikleri ve D'deki Karşılıkları

Ali Çehreli

30 Haziran 2012; Tütev, Ankara

C++11

Kısaltmalar:

- İlk standart: **C++98**
- 2003'teki *technical report*'un eklenmiş hali: **C++03**
- 2011'deki standart: **C++11**

Denemek için:

- Yeni bir Visual C++
- g++ 4.7.0 veya daha yenisi, **`-std=c++0x`** seçeneği ile:

```
g++ -std=c++0x deneme.cpp
```

- vs.

Otomatik Tür Çıkarsama

Eski Yetersizlik

- Türlerin açık açık yazılmasının gereksiz olduğu durumlar:

```
isim_alanim::Sinif * p = new isim_alanim::Sinif();

for (std::list<int>::const_iterator it = l.begin(); it != l.end(); ++it) {
    // ...
}
```

- **`sizeof`** bir *ifadenin türünün büyüklüğünü* verir. Onun bir adım öncesi olan ve bir *ifadenin türünü* veren **`typeof`** standart değildi:

```
typeof(bir_ifade) a;    // standart degildi
```

Otomatik Tür Çıkarsama

Eski Çözüm

- Tür isimlerini **`typedef`** ile kısaltmak (bu aslında başka zamanlarda da yararlıdır):

```
typedef std::list<int> Sayilar;
typedef Sayilar::const_iterator SayilarIt;

for (SayilarIt it = l.begin(); it != l.end(); ++it) {
    // ...
}
```

- Derleyicilerin ek olanaklarından yararlanmak:

```
__typeof__(bir_ifade) a;
```

Otomatik Tür Çıkarsama

Yeni Çözüm

- Bütünüyle etkisizleşmiş olan **`auto`** anahtar sözcüğü C++11 ile yeni bir hayat buldu ve *türünü otomatik olarak çıkarsa* anlamını kazandı:

```
auto * p = new isim_alanim::Sinif();

for (auto it = l.begin(); it != l.end(); ++it) {
    // ...
}
```

- **`typeof`** değil ama **`decltype`** geldi:

```
decltype(bir_ifade) a;
```

Otomatik Tür Çıkarsama

D'deki Karşılığı

auto C'deki anlamını yeniden kazanmıştır: otomatik yaşam süreçli değişken. Otomatik tür çıkarsama için özel bir anahtar sözcük yoktur.

```
import std.stdio;

auto foo(T)(T t) {
    T[] sonuç;
    return sonuç;
}

auto bar() {
    struct S { /* ... */ }

    return S();
}

void main()
{
    auto a = 42;                    // otomatik yaşam süreçli bir int
    immutable i = 2.75;             // değişmez bir double
    const c = foo(0x1_0000_0000);   // long dilimine referans
    shared s = "merhaba";           // paylaşılabilen bir string
    auto b = bar();                 /* Voldemort türü; ismi söylenemez!
                                     * (aslında deneme.bar.S)
                                     */

    // vs.

    immutable tablo = [ "abc" : 1.5, "çde" : 5.25 ];  // double[string]

    foreach (anahtar, değer; tablo) {    // anahtar: string, değer: double
        // ...
    }
}
```

Aralıklar için `for` Döngüsü

Eski Yetersizlik

- Amaç *bütün elemanlar için* demek olduğunda **`for`** döngüsü gereksiz tekrar içerir:

```
for (auto it = l.begin(); it != l.end(); ++it) {
    // ...
}
```

- Farklı topluluklar için farklı yazılması gerekebilir. Örneğin, dizilerde indeksleri arttırarak ve [] işleciyle:

```
int dizi[5] = { 1, 2, 3, 4, 5 };

for (size_t i = 0; i != 5; ++i) {
    ++dizi[i];
}
```

Aralıklar için `for` Döngüsü

Eski Çözüm

`std::for_each` algoritması:

- Serbest işlevle:

```
void karesini_goster(int sayi)
{
    cout << sayi * sayi << '\n';
}

// ...

    for_each(v.begin(), v.end(), karesini_goster);
```

- operator() işleciyle:

```
template <class T>
struct KareGosterici
{
    void operator() (T sayi) const
    {
        cout << sayi * sayi << '\n';
    }
};

// ...

    KareGosterici<int> gosterici;
    for_each(v.begin(), v.end(), gosterici);
```

Aralıklar için `for` Döngüsü

Yeni Çözüm

Diziler dahil, bütün toplulukların bütün elemanları üzerinde ilerlemek için aralık **`for`**'u (range **`for`**):

```
    for (auto eleman : v) {
        karesini_goster(eleman);
    }

// ...

    int dizi[5] = { 1, 2, 3, 4, 5 };

    for (int & sayi : dizi) {
        ++sayi;
    }
```

Aralıklar için `for` Döngüsü

Yeni Çözüm (devam)

`begin()` ve **`end()`** işlevini tanımlamış olan kullanıcı türleriyle de uyumludur. Bu işlevlerin o aralığa uygun bir erişici (iterator) döndürmeleri gerekir:

```
#include <iostream>

struct Ikiserli
{
    int bas_, son_;

    Ikiserli(int bas, int son) : bas_{bas}, son_{son} {}

    struct const_iterator {
        int deger_;
        explicit const_iterator(int deger) : deger_{deger} {}

        bool operator!= (const const_iterator that) const {
            return deger_ != that.deger_;
        }
        void operator++ () { deger_ += 2; }
        int operator* ()   { return deger_; }
    };

    const_iterator begin() const { return const_iterator(bas_); }
    const_iterator end() const   { return const_iterator(son_); }
};

int main()
{
    Ikiserli aralik{10, 20};

    for (auto sayi : aralik) {
        std::cout << sayi << '\n';
    }
}
```

Aralıklar için `for` Döngüsü

D'deki Karşılığı

`foreach` döngüsü diziler, dilimler, eşleme tabloları, ve kullanıcı türleri ile uyumludur:

```
    auto dizi = [ 1, 3, 5, 7 ];

    foreach (sayı; dizi) {
        // ...
    }

    auto tablo = [ "abc" : 1.5, "çde" : 5.25 ];  // double[string]

    foreach (anahtar, değer; tablo) {
        // ...
    }
```

Aralıklar için `for` Döngüsü

D'deki Karşılığı (devam)

Kullanıcı türleri için iki yöntem vardır:

- Çoğu durum için en basit olarak **`InputRange`** işlevleri:

```
import std.stdio;

struct İkişerli
{
    int baş, son;
    bool empty() const @property { return baş >= son; }
    int front() const @property  { return baş; }
    void popFront() { baş += 2; }
}

void main()
{
    auto aralık = İkişerli(10, 20);

    foreach (sayı; aralık) {
        writeln(sayı);
    }
}
```

- **`opApply()`** işlevi:

```
class Okul {
    Öğrenci[] öğrenciler;
    Öğretmen[] öğretmenler;
// ...
    int opApply(int delegate(ref Öğrenci) işlemler) { /* ... */ }
    int opApply(int delegate(ref Öğretmen) işlemler) { /* ... */ }
}
// ...
    foreach (Öğrenci öğrenci; okul) { /* ... */ }
    foreach (Öğretmen öğretmen; okul) { /* ... */ }
```

İlkleme Listeleri

Eski Yetersizlik

Dizilere ayrıcalık:

```
const int d[] = { 1, 7, 42 };       // derlenir
const vector<int> v = { 1, 7, 42 }; // C++03 derleme hatasi
```

İlkleme Listeleri

Eski Çözüm

1. Diziden ilklemek:

```
    const int d[] = { 1, 7, 42 };
    const vector<int> v0(d, d + ELEMAN_ADEDI(d));
    // veya:
    const vector<int> v1(DIZI_BASI(d), DIZI_SONU(d));
```

2. İşlevden döndürmek:

```
vector<int> kur()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(7);
    v.push_back(42);
    return v;
}
// ...
    const vector<int> v = kur();
```

3. **boost::assign::list_of**'un ilginç ve hoş söz dizimi ile:

```
#include <boost/assign.hpp>
using namespace boost::assign;
// ...
    const vector<int> v = list_of(1)(7)(42);
```

4. vs.

İlkleme Listeleri

Yeni Çözüm

{} karakterleri ile gruplamak **initializer_list** parametresi olarak belirir:

```
class Dizi
{
    std::vector<int> v_;

public:

    Dizi(const std::initializer_list<int> & degerler)
        :
        v_{degerler}
    {
        // ...
    }
// ...
};

// ...

    Dizi d0 = { 1, 7, 42 };
    Dizi d1({ 1, 7, 42 });
    Dizi d2{ 1, 7, 42 };
```

İşlevlerle de kullanılabilir:

```
void foo(const std::initializer_list<double> & parametreler)
{
    // ...
}
// ...
    foo({ 1.5, 2.5 });

    // {} karakterleri ilkleme anlaminda oldugu icin
    // islev cagirirken yasal degil:
    foo{ 1.5, 2.5 };    // ← derleme HATASI
```

İlkleme Listeleri

Yeni Çözüm (devam)

`initializer_list`'in elemanlarına *aralık* ***`for`***'u ile de erişilebilir:

```
class Dizi
{
public:

    Dizi(const std::initializer_list<int> & degerler)
    {
        for (auto deger : degerler) {
            // deger'i kullan
        }
    }
// ...
};

// ...

    Dizi d0 = { 1, 7, 42 };
    Dizi d1({ 1, 7, 42 });
    Dizi d2{ 1, 7, 42 };
```

İlkleme Listeleri

D'deki Karşılığı

Belirsiz sayıda parametreli işlevler (variadic functions):

```
class Dizi
{
    int[] değerler;

    this(int[] değerler...) {
        this.değerler = değerler;
    }
// ...
}

// ...

    auto d0 = new Dizi([ 1, 7, 42 ]);  // Diziyle
    auto d1 = new Dizi(1, 7, 42);      // Ayrık parametrelerle
    auto d2 = new Dizi(1);             // Tek parametreyle
```

foreach ile de kullanılabilir:

```
class Dizi
{
    this(int[] değerler...) {
        foreach (değer; değerler) {
            // değer'i kullan
        }
    }
// ...
}
```

İsimsiz İşlevler (lambda functions)

Eski Yetersizlik

Küçük işlemler için başlı başına işlev tanımlamak bazen fazlaca külfetli olur.

Korkunç söz dizimleri:

```
    set<ii>::iterator it = find_if(tes.begin(),tes.end(),bind2nd(comp(),ha));
// ...
    VBI iter = find_if(vb.begin(), vb.end(),
                       boost::bind(&Binding::value, _1) == 12);
// ...
    return boost::bind(&BolmeIslemi::hesapla, dorde_bolucu, _1);
```

İsimsiz İşlevler (lambda functions)

Eski Çözüm

Kodun okunaklılığını arttırmak için yazılan ve belki de tek kere kullanılacak olan işlev nesneleri:

```
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

struct tam_kati
{
    int bolen_;

    tam_kati(int bolen) : bolen_{bolen} {}

    bool operator() (int sayi) { return (sayi % bolen_) == 0; }
};

int main()
{
    vector<int> v{ 1, 10, 22, 30 };

    auto it = find_if(v.begin(), v.end(), tam_kati(11));

    if (it != v.end()) {
        cout << *it << '\n';
    }
}
```

İsimsiz İşlevler (lambda functions)

Yeni Çözüm

İsimsiz işlevler *hazır değer* (literal) olarak kod içinde tanımlanabilirler:

```
    auto it = find_if(v.begin(), v.end(),
                      [] (int sayi) { return (sayi % 11) == 0; });
```

Değişkenlere atanabilirler:

```
    auto kistas = [] (int sayi) { return (sayi % 11) == 0; };
    auto it = find_if(v.begin(), v.end(), kistas);
```

Söz dizimi:

- **[]** parantezleri (capture specification) kapsamdaki değişkenleri *yakalamak* içindir:

```
    int bolen = 0;
    cin >> bolen;

    auto kistas = [&] (int sayi) { return (sayi % bolen) == 0; };
```

- Dönüş türü otomatiktir. Belirtmek gerektiğinde işlev parantezlerinden sonra yazılır:

```
    auto a = [] (int a) -> int { return a + 7; };
    cout << a(42);
```

İsimsiz İşlevler (lambda functions)

Yeni Çözüm (devam)

- **`[]`** Hiçbir değişkeni yakalama
- **`[&]`** Kullanılan bütün değişkenleri referans olarak yakala
- **`[=]`** Kullanılan bütün değişkenleri kopya olarak yakala
- **`[=, &a]`** **`a`**'yı referans olarak yakala; diğerlerini kopyala
- **`[b]`** Yalnızca **`b`**'yi kopyala
- **`[this]`** Bu sınıfın **`this`** göstergesini kopyala (bütün üyelerin erişimini açar)

Uyarı: Referans olarak alınan değişkenler kapsamdan çıkıldığında geçersizdir:

```
    std::function<void (void)> f;  // "void döndüren void alan işlev" anlamında
    f = [&] () { cout << "Merhaba"; };

    {
        int b = 7;    // yerel değişken
        f = [&] () { b = 42; };
    }

    f();    // HATA: Geçersiz olan b'ye erişir.
```

İsimsiz İşlevler (lambda functions)

D'deki Karşılığı

C'deki işlev göstergelerinin eşdeğeri olan **function** ve kapsamdaki değişkenlere erişim sağlayabilen **delegate**:

```
import std.array;
import std.stdio;
import std.algorithm;

void main()
{
    int[] dizi = [ 1, 10, 22, 30 ];

    auto sonuç = find!((sayı) { return (sayı % 11) == 0; })(dizi);

    if (!sonuç.empty) {
        writeln(sonuç.front);
    }
}
```

Daha kısa olarak:

```
    auto sonuç = find!(sayı => (sayı % 11) == 0)(dizi);
```

İsimsiz İşlevler (lambda functions)

D'deki Karşılığı (devam)

Çöp toplayıcı yerel değişkenleri gerektiği kadar canlı tutar:

```
import std.stdio;
import std.string;

alias int delegate() Temsilci;

Temsilci temsilciYap(int i)
{
    int araHesap = i * 5;
    string dizgi = format("%s", i);

    Temsilci temsilci = () { return dizgi.length + araHesap; };
    return temsilci;

    /* Yukarıdaki son iki satırın eşdeğeri:
     *
     *   return () => dizgi.length + araHesap;
     */
}

void main()
{
    auto temsilci = temsilciYap(10);
    auto sonuç = temsilci();
    writeln(sonuç);
}
```

Döndürülen temsilcinin yerel **araHesap** ve **dizgi** değişkenlerini kullanıyor olması hata değildir; yerel kapsam temsilci yaşadığı sürece geçerlidir.

Sabit İfadeler

Eski Yetersizlik

C++'ta sabit ifadeler aritmetik işlemler gibi çok basit ifadelerle kısıtlıdır:

```
    int dizi[7 * 24] = {};
```

Derleyici tanımlarını görebiliyor olsa bile sabit değer döndüren işlevler kullanılamazlar:

```
int haftadaGun()
{
    return 7;
}

int gundeSaat()
{
    return 24;
}

// ...

    int dizi[haftadaGun() * gundeSaat()] = {};    // ← derleme HATASI
```

Sabit İfadeler

Yeni Çözüm

Programcı ifadenin sabit olduğunu **`constexpr`** anahtar sözcüğü ile garanti edebilir:

```
constexpr int haftadaGun()
{
    return 7;
}

constexpr int gundeSaat()
{
    return 24;
}

// ...

    int dizi[haftadaGun() * gundeSaat()] = {};    // derlenir
```

Sabit İfadeler

D'deki Karşılığı

D'nin CTFE (Compile Time Function Execution) olanağı *son derece* güçlüdür: Derleme zamanında işletilebilen her işlev sabit değerler için kullanılabilir:

```
string menü(string başlık, string[] seçenekler, int genişlik) {
    string sonuç = ortalanmışSatır("-= " ~ başlık ~ " =-", genişlik);

    foreach (seçenek; seçenekler) {
        sonuç ~= ortalanmışSatır(". " ~ seçenek ~ " .", genişlik);
    }

    return sonuç;
}

string ortalanmışSatır(string yazı, int genişlik) {
    string girinti;

    if (yazı.length < genişlik) {
        foreach (i; 0 .. (genişlik - yazı.length) / 2) {
            girinti ~= ' ';
        }
    }

    return girinti ~ yazı ~ '\n';
}

void main() {
    enum tatlıMenüsü = menü("Tatlılar",
                            [ "Baklava", "Kadayıf", "Muhallebi" ], 30);

    string beklenenDeğer = "       -= Tatlılar =-\n"
                           "         . Baklava .\n"
                           "         . Kadayıf .\n"
                           "        . Muhallebi .\n";

    assert(tatlıMenüsü == beklenenDeğer);
}
```

rvalue Referansları

Eski Yetersizlik

İsimleri atama işlecinin solunda ve sağında yer alabilmelerinde gelir:

- **lvalue (sol değer)**: *Genellikle*, adresi alınabilen değişkenlerdir (örneğin, isimli değişkenler).
- **rvalue (sağ değer)**: *Genellikle*, adresi alınamayan değişkenlerdir (çoğunlukla isimsiz, geçici değişkenler)

.

C++'ta kopyalama (by-copy) temeldir:

```
vector<int> sayilariOlustur()
{
    vector<int> sayilar;
    // ...
    return sayilar;
}

// ...

    vector<int> s = sayilariOlustur();

// (Evet, RVO ve NRVO eniyileştirmeleri uygulanabilir; gözardı ediyorum.)
```

Oysa, döndürülen nesnenin içeriği hedefe aktarılabilse (move) büyük hız kazancı sağlanabilir.

rvalue Referansları

Eski Yetersizlik (devam)

Geçici nesne vector'ün ara belleğine kopyalanmak zorundadır:

```
vector<Yapi> v;
v.push_back(Yapi(42, "abc"));
```

Oysa, bir daha hiçbir biçimde kullanılamayacak olan geçici nesnenin içeriği vector'ün ara belleğindeki elemana aktarılabilse büyük hız kazancı sağlanabilir.

rvalue Referansları

Yeni Çözüm

C++11, rvalue referanslarını belirlemek için **T&&** söz dizimini getiriyor:

```
struct Yapi {
    vector<int> v_;

    Yapi(const vector<int> & v)  // lvalue ile kurma
        : v_{v} {}

    Yapi(vector<int> && v)       // rvalue ile kurma
        : v_{std::move(v)} {}

    Yapi(const Yapi & diger)    // lvalue'dan kopya
        : v_{diger.v_} {}

    Yapi(Yapi && diger)         // rvalue'dan aktarma
        : v_{std::move(diger.v_)} {}

};     // Dikkat: v_{diger.v_} kopyalar; aktarmaz!

vector<int> vectorYap() {
    vector<int> v;
    // ...
    return v;
}

Yapi YapiYap() {
    return Yapi(/* ... */);
}

int main() {
    vector<int> v{ 1, 2, 3 };

    Yapi y0{v};             // v'yi kopyalayarak kurar
    Yapi y1{vectorYap()};  // döndürülen vector'ü aktararak kurar
    Yapi y2{y0};            // y0'ı kopyalayarak kurar
    Yapi y3{YapiYap()};    // döndürülen nesneyi aktararak kurar
}
```

rvalue Referansları

D'deki Karşılığı

D bu sorunu tasarım aşamasında çözmüştür:

- Sınıflar referans türü olduklarından sınıflarda zaten böyle bir sorun yoktur.
- Yapılar bellekte serbestçe kaydırılabilen türlerdir. (D, kendi içine referans bulunduran yapıları yasaklar.) Bu sayede aktarma (move) derleyici tarafından zaten uygulanır.
- Yapı nesneleri ancak gerçekten gereken durumlarda kopyalanırlar. O zaman kullanıcının tanımlamış olduğu kopya sonrası (post-blit (BLock Transfer)) işlevi çağrılır.

rvalue Referansları

D'deki Karşılığı (devam)

Örnek:

```
struct Yapı {
    int[] dizi;

    this(const ref int[] dizi) {  // lvalue ile kurma
        this.dizi = dizi.dup;
    }

    this(int[] dizi) {            // rvalue ile kurma
        this.dizi = dizi; // <-- ucuz çünkü dilimler referans türleridir
    }

    this(this) {                  // kopya sonrası (post-blit (BLock Transfer))
        dizi = dizi.dup;
    }
}

int[] diziYap() {
    int[] dizi;
    return dizi;
}

Yapı YapıYap() {
    return Yapı(/* ... */);
}

void main()
{
    int[] dizi = [ 1, 2, 3 ];
    auto y0 = Yapı(dizi);         // dizi'yi kopyalayarak kurar
    auto y1 = Yapı(diziYap());    // döndürülen vector'ü aktararak kurar
    auto y2 = y0;                 // y0'ı kopyalayarak kurar
    auto y3 = YapıYap();          // döndürülen nesneyi aktararak kurar
}
```