A D programozási nyelv

Ismerkedjünk meg a D nyelvvel, mely nem más, mint a C és C++ hatékony, kiválóan kezelhető utóda.

Az elmúlt évtizedekben a programozási nyelvek egészen látványos fejlődésen mentek keresztül, amely állítás igazsága különösen akkor szembetűnő, ha összehasonlítási alapnak a *UNIX* és *C* nyelv hajnalát tekintjük. Ebben az „ősi" korban a lefordított (*compiled*) nyelvek még éppen csak kezdtek elindulni azon az impozáns pályán, amit később befutottak, s amely fejlődés eredményeképpen mára számos különböző nyelv áll rendelkezésünkre, amelyek a legkülönbözőbb szolgáltatásokat nyújtják, s amelyek mind egy-egy adott területen használhatók igazán jól. Ebben a cikkben egy ilyen új nyelvről a *D*-ről lesz szó, amit a *Digital Mars* fejleszt. A *D* egy lefordított, statikusan tipizált, többparadigmás nyelv, amely leginkább a *C*-re hasonlít. A korábban említett specializációval szemben a *D* fejlesztőinek célja az volt, hogy olyan általánosan használható eszközt adjanak a fejlesztők kezébe, ami egyaránt használható egy rendszermag vagy egy számítógépes játék fejlesztésére. Ezt úgy kívánták elérni, hogy a nyelv tervezése során ötvözték a *C* hajlékonyságát és hatékonyságát az olyan, a produktivitást segítő „nyelvi leleményekkel", mint amelyek a *Ruby*-ban vagy a *Pythonban* megtalálhatók.
Az alapötlet egyébként attól a *Walter Brighttól* származik, aki az első *ANSI C++* fordítóprogramot írta *DOS* alá.
A *D* referenciafordító szabadon letölthető, és egyaránt van *Windows* és *Linux* alatt működő változata. Maga a frontend kettős licensszel (*GPL* és *Artistic*) rendelkezik.
A *D* fordítót, amely tehát *GPL* licenc alapján terjesztett szabad szoftver *GDC*-nek hívják, mégpedig nem véletlenül, ugyanis a háttérben a *GCC* keretrendszert használja. Maga a *D* nyelv számos érdekes tulajdonsággal rendelkezik. Nincs benne előfeldolgozó (*preprocessor*), ugyanakkor végez szemétgyűjtést (*garbage collector*), vannak benne hajlékony elsőosztályú tömbök, kezel szerződéseket (*contracts*), és használhatunk benne *inline* assemblert, hogy csak a leglényegesebbeket említsük. Mindezeket az extra szolgáltatásokat úgy nyújtja, hogy közben megőrzi a régi *C* nyelvvel való *ABI* kompatibilitást, ami azt jelenti, hogy némi extra ráfordítással továbbra is használhatjuk a *C*-ben fáradságos munkával megírt könyvtárainkat. A *D* szabványos könyvtárában szintén megvannak a szükséges C szabványos könyvtárra mutató kapcsolatok.

Helló világ

D-ben a szokásos „Helló világ!" program a következőképpen fest:

```
import std.stdio; // A szabvá-
➥nyos be/kimeneti modul
int main(char[][] args)
{
        writefln("Hello
➥world!");
        return 0;
}
```

A `printf` típusbiztos megfelelője a *D*-ben a `writef` nevű függvény. Ennek a fenti kódban használt `writefln` nevű változata mindössze annyiban tér el kisebb testvérétől, hogy a kiírt szöveg után automatikusan beszúr egy sorvége jelet. A *D* tartalmaz egy automatikus szemétgyűjtőt (*garbage collector*), ami leveszi a programozó válláról a közvetlen memóriakezelés terhét. Ez értelemszerűen nagy előny, hiszen a fejlesztő így sokkal inkább a ténylegesen megoldandó feladatra koncentrálhat, ahelyett, hogy állandóan a különböző helyeken és időkben lefoglalt memóriaterületek állapotával kellene törődnie. A szemétgyűjtés automatizálása ráadásul eleve kiküszöböl egy sor hibatípust, például az „elbitangolt" mutatókat, meg az érvénytelen memóriahivatkozásokat. A szemétgyűjtés persze igényel némi extra teljesítményt a rendszertől, de ha ez lényegesen lassítaná a programunkat, még mindig dönthetünk úgy, hogy kikapcsoljuk ezt a szolgáltatást, és visszatérünk a jó öreg *C* nyelv dinamikus memóriakezeléséhez, vagyis a `malloc` és `free` függvényekhez.

Modulok

A *D* nyelv moduljai olyan programegységek, amelyeket az import paranccsal lehet beemelni a főprogramba. A modulok és a forrásfájlok között mindig egy az egyben történő megfeleltetés van, az útvonalelemeket pedig a pont karakter tagolja.
Egy modulban valamennyi szimbólumnak két neve van: a sajátja, illetve egy olyan, aminek az előtagja a modul neve. Ez utóbbit *teljes névnek* (*Fully Qualified Name; FQN*) nevezzük.
A korábban említett `writefln` függvényt például meghívhattuk volna `std.stdio.writefln` alakban is. Vannak esetek, amikor a teljesen meghatározott nevek használata nem opcionális, hanem egyenesen elkerülhetetlen. Ilyen például a statikus import parancs használata, amely beemeli ugyan a modul szimbólumait, de nem helyezi be azokat a globális névtérbe.

Erre akkor van szükség, ha két vagy több olyan modult is használunk, amelyben azonos egyszerű nevek fordulhatnak elő. Ilyen például a `std.string` és az `std.regexp`. Mindkettőben vannak keresésre, helyettesítésre és darabolásra való eljárások, és ezek neve – nem túl meglepő módon – azonos. Mármost ha olyan programot írok is, amelyben mindkét modulra szükségem van, akkor is sokkal valószínűbb, hogy az említett műveleteket az esetek többségében karakterláncokon és nem szabályos kifejezéseken akarom majd elvégezni. Ez pedig azt jelenti, hogy célszerűbb lesz statikusan importálnom az `std.regexp` modult, ami viszont azt jelenti, hogy ha ennek a függvényeit akarom használni, akkor azok teljes nevét kell majd leírnom.
A modulok rendelkezhetnek statikus konstruktorral és destruktorral. A konstruktor szerepét itt a statikus `this()` függvény tölti be, amely a `main()` előtt fut le. A main() visszatérése után kerül aztán sor a statikus `~this()` végrehajtására, amely értelemszerűen a modul destruktora. Mivel a modulok importálása szimbolikusan történik, a *D*-ben nincsenek fejlécállományok. Mindent egyszer, és pontosan egyszer kell deklarálnunk, vagyis sem a függvények, sem az osztályok alakját nem kell előre megadnunk, vagy bármilyen módon kétszer szerepeltetnünk. Ez egyben azt is jelenti, hogy a *D*-ben nem merülhet fel ütközés a tényleges megvalósítás és a deklaráció között, tekintve hogy nem válnak szét.

Az alias és a typedef

A *D* nyelv különbséget tesz `alias` (*álnév*) és *típus* (*type*) között. A *typedef* parancs itt egy teljesen új típust vezet be, amit aztán a típusellenőrző rendszer és a függvények túlterhelését kezelő rendszer tudomásul vesz. (Ezekről később még lesz szó.) Az alias ezzel szemben egy amolyan típushelyettesítő, vagyis opcionális nyelvi szimbólum:

```
alias int size_t; typedef int
➥myint; // nem tudunk implicit
➥módon int alias-ra
➥konvertálni
    someReallyLongFunctionName
➥func;
```

Tömbök

A *D* nyelvben a tömbök minden tekintetben első osztályú típusok. A *D* összesen három tömbtípust különböztet meg: vannak statikus, dinamikus és asszociatív tömbök. A tömbdeklarációkat a fordító jobbról balra haladva értelmezi, vagyis például a `char[][]` szimbólum karaktertömbök tömbjét jelenti:

```
int[] intArray; // Egészek
➥dinamikus tömbje
int[2][4] matrix; // Egy 2x4-es
➥mátrix
```

Valamennyi tömb rendelkezik a `length` (hosszúság), `sort` (rendezés) és a `reverse` (megfordítás) tulajdonságokkal. Az asszociatív tömbök olyan tömbök, amelyekben az elemek indexelése nem egymást követő egészekkel történik, hanem valami mással, például karakterláncokkal, speciális adatszerkezetekkel, vagy tetszőleges egészekkel:

```
import std.stdio;
int main(char[][] args) {
    int[char[]] petNumber;
    petNumber["Dog"] = 212;
    petNumber["cat"] = 23149;
    int[] sortMe = [2, 9, 341,
➥23, 74, 112349];
    int[] sorted = sortMe.sort;
    int[] reversed =
➥sorted.reverse;
    return 0; }
```

A dinamikus és statikus tömböket egyaránt feldarabolhatjuk a `..` operátorral. Ügyeljünk rá, hogy a kezdő index minden esetben hozzátartozik a kijelölt darabhoz, a záró azonban nem. Ha tehát nullától a tömb hosszáig jelölünk ki egy darabot, akkor éppen a teljes tömböt kapjuk vissza.

```
int[] numbers = [1, 2, 3, 4, 5,
➥6, 7];
numbers = numbers[0..2] // Ez
➥az 1-3 indexű elemek tömbje
```

Végezetül érdemes még azt is megemlíteni, hogy a *D* a ~ operátor használja összefűzésre, aminek az az oka, hogy valahol mélyen az összeadás és az összefűzés két különböző művelet, két különböző alapelképzeléssel:

```
char[] string1 = "Hello ";
char[] string2 = "world!";
char[] string = string1 ~
➥string2; // Helló világ!
```

Ez itt egyben egy kiváló példa annak bemutatására is, miként könnyíti meg a D nyelv a programozó életét néhány szintaktikai finomsággal, hogy a valóban megoldandó feladatára tudjon koncentrálni. Itt megfigyelhetjük ugyanis, hogy *D* nyelv semmi egyebet nem tesz, csak némi szintaktikai finomsággal kiegészíti a már meglévő alaptípusokat. Logikailag a karakterlánc tulajdonképpen nem egyéb, mint karakterek tömbje. Éppen ezért a *D*-ben nincs is külön karakterlánc típus, helyette karakterek tömbjét használhatjuk, viszont ehhez kapunk efféle apró segédeszközöket.
A *D*-ben – ahhoz képest, hogy nincs is külön karakterlánc típus – háromféle karakterláncot különböztethetünk meg: van `char`, `wchar` és `dchar` típusú karakterlánc, amelyek egymástól a karakterkódolásban különböznek. Az első ***UTF-8***, a második ***UTF-16***, a harmadik ***UTF-32*** kódolású karaktereket képes tárolni. Mivel ezekhez a típusokhoz a megfelelő kezelőfüggvényeket is megtaláljuk a szabványos könyvtárban bátran kijelenthetjük, hogy a *D* optimális fejlesztőeszköz honosított programok írásához. A karakterláncokkal kapcsolatban a *C*-hez képest lényeges eltérés az is, hogy a *D* karakterláncai „tudják" a saját hosszukat, ami ismét csak számos bosszantó programozási hibát küszöböl ki egy csapásra, hiszen soha többé nem kell keresgélnünk azt a bizonyos záró null karaktert.

Szerződések

A *D* nyelv tartalmaz olyan technikákat, amelyek lehetővé teszik az úgynevezett szerződéseken (*contracts*) alapuló programfejlesztést. Ez utóbbinak elsősorban a programok minőségbiztosításával kapcsolatban van nagy jelentősége. Ha a szerződések magának a nyelvnek a részét képezik, akkor sokkal nagyobb a valószínűsége annak, hogy a programozók használni is fogják azokat, hiszen nem kell őket külön megvalósítaniuk egy külső könyvtár segítségével.
A szerződések legegyszerűbb formája az `assert` parancs használata.

Ez ellenőrzi, hogy a neki átadott érték igaz-e, és ha nem, akkor kivételt dob. Az `assert` utasításoknak opcionálisan szöveges üzeneteket is átadhatunk argumentumként, amelyekkel „olvasmányosabbá" tehetjük a program kimenetét. A függvényekhez kétféle szerződés is tartozhat. Az egyik a függvény kódjának végrehajtása előtt, míg a másik ez után végez ellenőrzéseket. Az első az `in`, a második az `out` kulcsszóval jelölt blokk, de függetlenül az érvényesülési idejüktől mindkettőt a függvény kódja előtt kell megadni. Az `in` szakaszban megadott feltételeknek maradéktalanul teljesülniük kell a függvény végrehajtása előtt, ellenkező esetben a rendszer egy `AssertError` típusú kivételt dob. Az out blokk kicsit máshogy működik. Ez a függvény kimenetét kapja meg, és ellenőrzi, hogy a kód valóban úgy működött-e, ahogy azt elvártuk tőle. A hívó fél csak akkor kapja meg a visszatérési értéket, ha ez a vizsgálat sikeres volt. Végezetül ha egy programot a `release` opció bekapcsolásával fordítunk le, akkor a fordítóprogram kiveszi belőle az összes `assert` blokkot. Ez értelemszerűen gyorsítani fogja a program futását.

```
int sqrt(int i)
in {
            assert(i > 0);
   }
out(result) { // A visszatérési
➥érték minden
              // esetben
➥átkerül a result változóba
assert((result * result) ==i);
}
body
{...}
```

A szerződések egy másik formája az úgynevezett egység teszt (***unit test***). Ennek a vizsgálati módnak a célja annak eldöntése, hogy egy adott függvény, vagy függvények egy csoportja valóba a specifikációk szerint működik-e a különböző bemenő paraméterek mellett. Tegyük fel például, hogy van a programunkban egy ilyen, meglehetősen haszontalan függvény:

```
int add(int x, int y) { return
➥x + y; }
```

Az ezzel kapcsolatos egységtesztet végző kódot ugyanebben a modulban kell elhelyeznünk. Ha ezután a fordításkor engedélyezzük a unittest opciót, akkor a kérdéses ellenőrzés mindannyiszor lefut, valahányszor beemeljük valahova a kérdéses modult és lefuttatunk belőle bármilyen függvényt. Esetükben az ellenőrzőkód nagyjából a következőképpen fog kinézni:

```
unittest {
            assert(add(1, 2)
➥== 3);
            assert(add( -1,
➥-2) == -3);
}
```

Feltételes fordítás

Mivel a ***D*** nyelvnek egyáltalán nincs előfeldolgozója, a feltételes fordítást vezérlő utasítások is magának a nyelvnek a részét képezik. Ezzel egy csapásra két probléma is megoldódik. Egyrészt nem tudjuk elkövetni mindazokat a hibákat, amelyeket eddig az előfeldolgozók, illetve azok kissé eltérő viselkedése okozott, másrészt gyorsul maga a fordítási folyamat is, hiszen kevesebb program kevesebb lépésben állítja elő a végső kódot. A ***D*** `version` utasítása meglehetősen hasonlít a *C*-ből jól ismert `#ifdef` direktívához. Ha megadunk egy version azonosítót, csak akkor fordítódik bele a végső programba az ehhez tartozó kód, ellenkező esetben nem:

```
version(Linux)
import std.c.linux.linux;
else version(Win32)
import std.windows.windows;
```

A `debug` utasítás meglehetősen sokban hasonlít a version-hoz, de nem kell föltétlen azonosítónak tartoznia hozzá. A nyomkövetéshez, hibakereséshez használt kiegészítő kódot elhelyezhetjük egy globális `debug` feltételben, vagy rendelhetünk hozzá egy külön azonosítót is:

```
debug writefln("Debug:
➥something is happening.");
debug (socket) writefln("Debug:
➥something is
    happening concerning
➥sockets.");
```

A statikus `if` parancs állandók fordítás közben történő vizsgálatát teszi lehetővé:

```
const int CONFIGSOMETHING = 1;

void doSomething()
{
            static if
➥(CONFIGSOMETHING == 1)
            { .... }
}
```

A scope parancs

A `scope` utasítás működési logikáját a ***D*** nyelv tervezése során úgy alakították ki, hogy segítségével egy-egy objektum hatókörét le lehessen tisztítani, illetve logikusan csoportosítani lehessen a kód sikeres vagy sikertelen végrehajtását:

```
void doSomething()
{
    scope(exit) writefln("We
➥exited.");
    scope(success) writefln("We
➥exitednormally.");
    scope(failure) writefln("We
➥exited due to an
➥exception.");

...

}
```

A `scope` utasításokat a rendszer fordított sorrendben fogja végrehajtani. Létezik egy ***DMD*** nevű, a szkriptekre jellemző ***D*** szintaxis is, amellyel a héjprogramokra jellemző módon hivatkozhatunk a ***D*** fordítóra, és ennek a `-run` opciót megadva úgy futtathatunk egy D programot, hogy annak kódja a szabványos bemenetről érkezzék. Ennek – az „igazi" szkriptekhez hasonlóan – természetesen az az értelme, hogy ezzel a mechanizmussal „önlefordító" ***D*** programokat írhatunk. Mindössze arról kell gondoskodnunk, hogy a ***D*** szkript elején a legelső sorban ott legyen a következő bűvös kódsor:

```
#!/usr/bin/dmd -run
```

Típuskikövetkeztetés (type inference)

A ***D*** nyelv lehetővé teszi, hogy úgynevezett automatikus deklarációval

határozzuk meg egy változóhoz azt a típust, ami neki a legjobban megfelel:

```
auto i = 1; // Ez egy int
➥változó lesz
auto s = "hi"; // Ez pedig
➥char[4]
```

A típus meghatározását maga a fordító végzi el, ami egyben azt is jelenti, hogy a választás bizonyosan a legmegfelelőbb lesz.

A foreach parancs

Az olvasók közül néhányan már bizonyára találkoztak ezzel a paranccsal más nyelvekben. A `foreach` működési logikájának lényege valahogy a következőképpen fogalmazható meg: „hajtsuk végre egy műveletsort a megadott tömb összes elemén”. Bár ez elsőre talán nem teljesen nyilvánvaló, ez az utasítás nem azonos azzal, hogy „hajtsunk végre megadott számú alkalommal egy műveletsort egy tömb elemein, amely megadott szám különben a tömb hossza”. Utóbbi esetben magának a programozónak kell gondoskodnia arról, hogy az a bizonyos megadott szám mindig megegyezzen a tömb elemszámával, míg a `foreach` mechanizmus leveszi ezt a terhet a fejlesztő válláról. Itt maga a fordítóprogram végzi el a szükséges vizsgálatot, és „teszi elérhetővé” a műveletsor számára a tömb összes elemét:

```
char[] str =
➥"abcdefghijklmnop";
foreach(char c; str)
➥writefln(c);
```

Mi több, akár a feldolgozott elemek indexét is megszerezhetjük, ha deklarálunk egy megfelelő változót a `foreach` ciklusban:

```
int [] y = [5, 4, 3, 2, 1];
foreach(int i, int x; y)
writefln("number %d is %d", i,
    x);
```

Végül ha ez könnyebbséget jelent, akár a feldolgozandó változók típusáról is elfeledkezhetünk, és használhatjuk a típus interfészt:

```
foreach(i, c; str)
```

Ez egyben megnyitja az utat a legkülönfélébb, a fordítóprogram szintjén megvalósítható optimalizálási eljárások alkalmazása előtt, hiszen a bemutatott megoldásnak éppen az a lényege, hogy annyi feladatot bízunk magára a fordítóra, amennyit csak lehet. Mindeközben a programozó mindezt könnyebbségként élheti meg, hiszen olyan hajlékony nyelvi szolgáltatást kap, amely lehetővé teszi számára, hogy a konkrét feladatra, és nem a nyelvi finomságokra koncentráljon.

Kivételek

Alapszabály, hogy a *D* nyelv a kivételeket hibakezelésre és nem hibakódok küldésére használja. Az alkalmazott módszer a ***try-catch-finally*** hármas, amely lehetővé teszi, hogy hiba utáni takarítókódot kényelmesen illesszük be a `finally` blokkba. Azokban az esetekben, amikor egy ilyen blokk kialakítása nem elég, elővehetjük a `scope` parancsot, ami ezekben a helyzetekben is nagy hasznunkra lehet.

Osztályok

Mint minden más objektumközpontú nyelvben a *D*-ben is lehetőségünk van arra, hogy objektumokból osztályokat alkossunk. Az egyik lényeges eltérés ezen a területen a `virtual` kulcsszó hiánya, ami például a C++-ban megvolt. Az ok egyszerű: a virtualitást a fordító saját hatáskörben, automatikusan kezeli. A *D* nyelv az egyszeres öröklődési paradigmát használja, vagyis vannak benne felületek (***interface***) és közvetlenül nem példányosított osztályok (***mixin***) is. Ezekről később még részletesebben is lesz szó. Az osztályok átadása hivatkozással (***reference***) és nem érték szerint történik, így a programozónak nem kell azzal törődnie, hogy a program bármely pontján mutatóként kezelje azokat. Ezen kívül a *D*-ben nincs se `->` se `::` operátor. Ehelyett a pontot (`.`) használja a nyelv minden olyan helyzetben, amikor szerkezetek vagy osztályok elemeit kell megcímezni.
A programok valamennyi osztályának őse az `Object` osztály, vagyis ez az öröklődési rendszer gyökere.

```
class MyClass {
        int i;
        char[] str;
        void doSomething() {
➥... };
}
```

Az osztályok tulajdonságainak meghatározása során több különböző függvénnyel kapcsolatban is használhatjuk ugyanazt a nevet:

```
class Person {
        private char[] PName;
        char[] name() {return
➥Pname;}
        void name(char[] str)
        { // elvégezzük
➥azokat a műveleteket amelyek
➥ahhoz szükségesek
          // hogy az összes
➥előfordulási helyén frissítsük
➥a kérdéses nevű objektum
➥tartalmát
        PName = name; }
    }
```

Az osztályok rendelkezhetnek konstruktorral és destruktorral, amelyek neve a D-ben `this` és `~this`:

```
class MyClass {
        this() {
➥writefln("Constructor
➥called");}
        this(int i) {
           writefln
➥("Constructor called with
➥%d", i);
        }
        ~this() { writefln
➥("Goodbye");}
```

Öröklődés esetén valamennyi osztály hozzáférhet az alaposztálya konstruktorához:

```
this(int i) {
        super(1, 32, i); //
➥super az a alaposztály
                         //
➥konstruktora
    }
```

Új osztályt akár egy másik osztály vagy függvény belsejében is deklarálhatunk. Ezek az osztályok aztán alapértelmezésként hozzáférnek minden az adott érvényességi körben elérhető változóhoz is. Szintén lehetőség van az operátorok túlterhelésére, ami ebben a szituációban sokkal egyértelműbb operátorhasználatot eredményez, mint ahogy azt a *C++*-ban megszokhattuk.
Az osztályok rendelkezhetnek úgynevezett invariánsokkal. Ezek olyan

szerződések, amelyeket a rendszer a konstruktor lefutása után, a destruktor lefutása előtt, illetve valamennyi nyilvános taghoz való hozzáférés során ellenőriz, de a `release` fordítási opció hatására eltávolítja őket a kódból:

```
class Date
{
int day;
int hour;
invariant
    {
        assert(1 <= day && day
➥<= 31);
        assert(0 <= hour &&
➥hour < 24);
    }
}
```

Ha ellenőrizni akarjuk, hogy két osztályhivatkozás ugyanarra az osztályra mutat-e, használhatjuk az is operátort:

```
MyClass c1 = new MyClass;
MyClass c2;
if(c1 is c2)
       writefln("These point to
➥the same thing.");
```

Felületek (interface)

Az interfész függvények olyan halmaza, amelyeket minden, az adott osztályból származtatott osztálynak meg kell valósítania:

```
interface Animal {
          void eat(Food what);
          void walk(int
➥direction);
          void makeSound();
}
```

Függvények

A *D* nyelvben nincs `inline` kulcsszó, mivel a fordító maga dönti el, hogy melyik függvény legyen ***inline***. A programozónak tehát az optimalizálásnak ezzel a formájával egyáltalán nem kell törődnie.
A függvényeket túl lehet terhelni, vagyis megadhatunk két különböző függvényt ugyanazzal a névvel, ha a bemenő paramétereik eltérőek. A fordítóprogram elég okos ahhoz hogy a paraméterlista alapján eldöntse, pontosan melyik változatra is gondoltunk:

```
void func(int i) // can
➥implicitly take
                 // longs and
➥shorts too
{...}


void func(char[] str)
{...}


void main()
{
          func(23);
          func("hello");
}
```

A függvényparaméterek lehetnek `in`, `out`, `inout` vagy `lazy` típusúak. Az alapértelmezés természetesen az `in`. Az `out` típusú paraméterek egyszerű kimeneti pontok:

```
void func(out int i)
{
         I += 4;
}


void main()
{
     int n = 5;
     writefln(n);
     func(n);
     writefln(n);
}
```

Az `inout` típusú paramétereket írni és olvasni egyaránt lehet, de nem lehet belőlük új másolatot készíteni:

```
void func(inout int i)
{
      if(i >= 0)
                 ...
else
                 ...
}
```

A `lazy` típusú paramétereket a program csak akkor számítja ki, ha szükség van rájuk. Tegyük fel például, hogy meghívunk egy ehhez hasonló függvényt:

```
log("Log: error at
➥"~toString(i)~" file not
➥found.");
```

Vegyük észre, hogy valahányszor meghívjuk ez a függvényt, a program összemásolja a karakterláncokat és átadja neki azokat. A `lazy` tárolási osztály ezzel szemben azt jelenti, hogy a karakterláncokat csak akkor fűzzük össze és adjuk át, ha ennek a műveletnek az eredményére valóban szükség van. Nyilván nem kell magyarázni senkinek, hogy ez mitől növeli a program hatékonyságát. A *D* nyelv lehetőséget ad a függvények egymásba ágyazására, vagyis egy függvény tartalmazhat egy másikat.

```
void main()
{
       void func()
        {
            ...
        }
}
```

A beágyazott függvényeknek írási és olvasási joguk is van a befoglaló függvény változóihoz:

```
void main()
{
     int i;
     void func()
     {
                  writefln(i +
➥1);
     }
}
```

Sablonok (template)

A *D* nyelv egy teljesen újratervezett és rendkívül hajlékony sablonrendszert használ. A kezdők kedvéért talán érdemes megjegyezni, hogy a sablonok példányosításához a `!` operátort kell használnunk. Ez egyrészt kiküszöböli azt a számos félreértést, ami a `<>` operátorral való példányosítás körül felmerült, másrészt a felismerése is könnyebb egy hosszú kódban. Példaként lássunk egy egyszerű másoló sablont:

```
template TCopy(t) {
         void copy(T from, out
➥T to)
         {
              to = from;
         }
}

void main()
{
        int from = 7;
```

```
        int to;
        TCopy!(int).copy(from,
➥to);
}
```

A sablonok deklarációihoz álneveket (alias) is rendelhetünk:

```
alias TFoo!(int) temp;
```

A sablonokat specializálhatjuk különböző típusokra, a fordítóprogram pedig automatikus képes kikövetkeztetni, hogy mikor melyik sablontípusról van szó:

```
template TFoo(T) { .... }
➥// #1
template TFoo(T : T[]) { .... }
➥// #2
template TFoo(T : char) { ....
➥}  // #3
template TFoo(T,U,V) { .... }
➥// #4
alias TFoo!(int) foo1;
➥// Az #1-et példányosítja
alias TFoo!(double[]) foo2;
➥// A #2-t példányosítja úgy

// hogy T egy double típusú
➥érték
alias TFoo!(char) foo3;
➥// A #3-at példányosítja
alias TFoo!(char, int) fooe;
➥// Ez a megoldás hibás. Nem
➥stimmel az

// argumentumok száma
alias TFoo!(char, int, int)
➥foo4; // A #4-et
➥példányosítja
```

Függvénysablonok

Ha egy sablonnak mindössze egyetlen tagfüggvénye van és semmi más, akkor a következő módon is deklarálhatjuk:

```
void TFunk(T) (T i)
{
        ...
}
```

Függvénysablonok implicit példányosítása

Lehetőségünk van arra, hogy a függvénysablonokat implicit módon példányosítsuk, mikor is az argumentumok típusát a fordító fogja kikövetkeztetni:

```
TFoo!(int, char[]) (2,"foo");
TFoo(2, "foo");
```

Osztálysablonok

Ha olyan sablon kell létrehoznunk, amelynek egyetlen eleme egy osztály, akkor a következő egyszerűsített szintaxist is alkalmazhatjuk:

```
class MyTemplateClass (T)
{
            ...
}
```

Nem példányosított osztályok (mixins)

A nem példányosított osztály (***mixin***) olyan, mint mikor fogjuk egy sablon kódját, és egyszerűen átmásoljuk egy osztályba. Ez a művelet nem keletkeztet saját hatókört:

```
template TFoo(t)
{
            t i;
}

class test
{
            mixin TFoo!(int);
            this()
            {
                    i = 5;
            }
}

void main()
{
             Test t = new Test;
             writefln(t.i);
}
```

Összefoglalás

A ***D*** egy kifejezetten ígéretes programozási nyelv, amely tulajdonságai alapján valószínűleg igen széles körben lesz majd alkalmazható. A nyelv legnagyobb előnye, hogy ötvözi az új megoldásokat a hagyományokkal. Az olyan nyelvi szolgáltatások, mint a tömbök, a héjprogramokra emlékeztető kiegészítő szintaxis, vagy a típusinterfészek a ***D***-t leginkább a ***Ruby***-hoz vagy a ***Pythonhoz*** teszik hasonlóvá. Ugyanakkor a rendszerprogramozók előtt is nyitva áll az út, hogy segítségével alacsony szintű műveleteket valósítsanak meg, hiszen továbbra is megvan benne az inline assembler, illetve számos olyan funkció, amelyekre a hardverközeli programozás során lehet szükségünk. A D nyelv ráadásul számos eltérő programozási paradigmát képes egyszerre támogatni. Használhatunk objektumközpontú megközelítést, hiszen ott vannak az osztályok és az interfészek, de támaszkodhatunk az általános programozás (***generic programming***) logikájára is, hiszen rendelkezésünkre állnak sablonok és példányosítás nélküli osztályok (***mixin***). És persze ott van a régi eljárás alapú megközelítés is (***procedural programming***), amikor csupán függvényeket és tömböket használunk. Van a rendszerben automatikus szemétgyűjtő eljárás, amely egyrészt leveszi a programozó válláról azt a terhet, hogy magának kelljen gondoskodnia a memóriakezelésről, a területek lefoglalásáról, felszabadításáról és állapotuk követéséről, másrészt meghagyja azt a lehetőséget, hogy ha a hatékonyság érdekében mégis erre lenne szükségünk, akkor megtehessük. Találunk aztán a ***D*** nyelvben olyan az imperatív nyelvekre – például a ***LISP***-re – jellemző megoldásokat is mint a `lazy` tárolási osztály, amely jelentősen növelheti egyes kódok hatékonyságát. Összességében a nyelv viszonylag stabil, jól kidolgozott, bár néha még fel-felbukkan benne egy új szolgáltatás, vagy egy régi megoldás javítása.
Röviden minden jel arra mutat, hogy a ***D*** nyelv készen áll az éles bevetésre.

Linux Journal 2007., 155. szám

Ameer Armaly 18 éves, vak és jelenleg középiskolában tanul. Legkedvesebb időtöltései közé tartozik a gitározás, a programozás és a tudományos-fantasztikus művek olvasása.

KAPCSOLÓDÓ CÍMEK

A D nyelv specifikációja és a letölthető fordítóprogram:
➲ www.digitalmars.com/d
GDC: dgcc.sf.net
Számos nyílt forrású projekt és oktatóanyag található a köetkező helyen:
➲ www.dsource.org