A beolvasás és kiírás szerepe a programozásoktatás kezdeti szakaszában

Horváth Győző

horvath.gyozo@inf.elte.hu
ELTE IK

Absztrakt. A kezdő programozásoktatásban kódolás során sokszor a beolvasás és a kiírás arányaiban elnyomja a feladat lényegét megoldó kódot. A kezdő programozó egy feladat megoldásakor lényegében három feladatot kap: a beolvasást, a feldolgozást és a kiírást. A cikk azt próbálja körüljárni, hogy a beolvasás és a feldolgozás milyen szerepet tölt be a kódban, és bizonyos esetekben hogyan lehet egyszerűbbé tenni megvalósításukat.

Kulcsszavak: programozásoktatás, módszertan, beolvasás, kiírás

1. Bevezetés

A programozásoktatásnak már a legelejétől része az, hogy a feladatmegoldást lépésekre bontva, módszeresen tanítsa meg a tanulóknak. A feladat szövegének elolvasása és megértése után a feladatmegoldás lépései közül az első három fontos számunkra: 1) meghatározni és formális eszközökkel leírni, hogy *mi* is a feladat (*specifikáció*), 2) absztrakt nyelven leírni azt, *hogyan* oldjuk meg a feladatot (*algoritmus*), és 3) a feladat előbbi megoldását egy konkrét programozási nyelven *megvalósítani* (*kódolás*). A további lépések (tesztelés, hibakeresés, hibajavítás, dokumentálás, karbantartás) ugyanolyan fontosak, azonban jelen cikk a feladatmegoldási folyamat elejére szeretne fókuszálni, arra, hogy egy feladatszövegtől hogyan jutunk el a konkrét kód megírásáig. A feladat lépésekre bontása fontos módszertani segítség a kezdő programozásoktatás alapvető céljának elérésében, amely során a tanulótól elvárjuk, hogy

- értse meg a feladatot;
- ismerje fel a szövegben az adatokat, miből mit kell előállítani;
- írja le ezen adatoknak a szerkezetét;
- fogalmazza meg a kapcsolatot a bemenő és kimenő adatok között (szövegesen vagy formálisan, ez utóbbi a programozási tételek értő alkalmazása);
- lépésekre tudja bontani a megoldást (ebben is mintát adhatnak a programozási tételek);
- a megoldást adott programozási nyelven implementálni tudja;
- ellenőrizni tudja a megoldás helyességét.

A tanulóknak ebben a folyamatban viszonylag sok új ismeretet kell elsajátítaniuk: a specifikálás formális nyelvét (bár bizonyos körülmények között ez helyettesíthető szöveges vagy rajzos leírással), az algoritmusleíró nyelvet (struktogram, mondatszerű leírás, pszeudokód) és magát a konkrét programozási nyelvet. E cikknek nem célja annak mérlegre tevése, hogy ez mennyire szükséges, hogyan egyszerűsíthető vagy váltható ki, hanem adottnak tekinti a fenti folyamatot.

E cikk kiindulási pontját az a megfigyelés adja, hogy kódolás során – főleg kezdetben – gyakran tapasztalható, hogy a beolvasás és a kiírás (de főként az előbbi) aránytalanul több kódot és nyelvi ismeretet igényel, mint maga a megoldás. Általában a beolvasó utasítás egyszerű, azonban a felhasználóval való kommunikáció, a felhasználó esetleges vagy szándékos tévesztéseinek kiszűrése, valamint az előfeltételek – akár bonyolult – kapcsolatrendszerének ellenőrzése nagyon gyakran

felhízlalja a beolvasó kódot, és olyan nyelvi elemek használatát igényli, amelyet nem feltétlenül a kódolás kezdeti szakaszán kellene bemutatni. Sokszor kell olyan fogalmakat elmagyarázni, mint az input puffer, kapcsolók, típuskonverziók, amelyek megértésére még vagy nem érettek a tanulók, vagy feleslegesen vonják el a figyelmet a lényegtől. Mert a lényeg a feladat megoldása során nem az, hogy hogyan kommunikáljunk biztonságosan, érthetően és udvariasan a felhasználóval, hanem hogy hogyan ábrázolom a feladatban szereplő adatokat és hogyan állítom elő a bemeneti adatokból a kimeneti adatokat, azaz hogyan oldom meg a feladatot. E mellett az, hogy hogyan szerzem meg az adatokat (standard bemenet, fájl, adatbázis, hálózati lekérdezés), valamint hogyan írom ki a végeredményt (standard kimenet, fájl, stb.) külön feladatnak tekinthető. Másképpen megfogalmazva, amikor a tanulók egy feladat kódolásán dolgoznak hirtelen három feladatot kell megoldaniuk: az eredeti feladat mellett a beolvasást és a kiírást is meg kell valósítaniuk.

Ebben a cikkben tehát azt vizsgáljuk, hogy mi a beolvasás és kiírás szerepe a feladatmegoldás során, ezeket hogyan tehetnénk egyszerűbbé annak érdekében, hogy a megoldandó feladatra koncentrálhassunk, és megnézzük, a teljes körű bemenet-kimenet kezelésnek hol van a helye.

2. Példa feladat

Tekintsük az alábbi példát, amelyre a cikk hátralévő részében hivatkozunk.

Feladat: adott két pozitív szám, számítsd ki az összegüket!

```
Be: a∈ℕ, b∈ℕ
Ki: c∈ℕ
Ef: a>0 és b>0
Uf: c=a+b
```

Az algoritmus a lehető legegyszerűbb (struktogram és pszeudokód) formában:

```
c:=a+b
```

A pszeudokódos formát sokszor írjuk külön alprogramként:[1]

```
Típus TEgész=Egész
Eljárás Feladat(Konst a, b: TEgész, Vált c:TEgész)
    c:=a+b
Eljárás Vége
```

A megfelelő C++ kód a lehető legegyszerűbb beolvasással viszonylag rövid, de a beolvasás hosszúra nyúlhat bonyolultabb esetekben (ld. 1. függelék):

```
#include <iostream>
using namespace std;
int main() {
    int a, b, c;          // deklarálás
    cin >> a >> b;        // beolvasás
    c = a + b;            // feldolgozás
```

[1] Az algoritmus első sora azt jelzi, hogy a specifikációban szereplő egyszerű és összetett típusokat az eljárás előtt definiálhatjuk. Általában az összetett típusokat szoktuk külön definiálni az egyszerűbb hivatkozhatóság kedvéért, de nincs semmi akadálya annak sem, ha egyszerű típusokat is saját elnevezéssel illetjük. Az algoritmus előállítása így sematikusabbá is válhat.

```
    cout << c << endl;    // kiírás
}
```

3. Beolvasás és kiírás a specifikációban és algoritmusban

A specifikációnak nincsen *közvetlen* köze a beolvasáshoz és kiíráshoz. A be- és kimeneti rész a feladat elvégzéséhez szükséges adatokat és azok szerkezetét írja le. Az előfeltétel a lehetséges bemeneti adatok körét szűkíti le, az utófeltétel pedig a be- és kimeneti adatok kapcsolatát adja meg. Egyik rész sem foglalkozik azonban azzal, hogy az adat honnan és hogyan jön, illetve megy. A specifikáció azzal foglalkozik, hogy a konkrét feladat megoldása (azaz a feldolgozás) *előtt* és *után* milyen állapotoknak kell fennállnia.

Ahogy a példaalgoritmusokból is látszik, az algoritmus sem tartalmaz – legalábbis az általában használt rövidebb formájában – beolvasó és kiíró műveleteket. Az algoritmus írásakor ugyanakkor, legyen az struktogram vagy pszeudokód, számos implicit feltételezéssel élünk: 1) a specifikációból tudjuk, hogy mely nevű adatok a bemenetiek, és melyek a kimenetiek; 2) ugyancsak a specifikációból ismert ezek típusa; 3) tudjuk, hogy az algoritmus kezdetekor a bemeneti adatok helyesen be vannak olvasva. Az implicit feltételezések nyilvánvalóvá válnak, ha nemcsak a lényegi feladat, hanem az egész program algoritmusát felírjuk.[2] Ezt már csak azért is érdemes megtenni, mert ezen keresztül lehet bemutatni a programok általános szerkezetét, miszerint minden program három fő részből áll: *beolvasásból*, *feldolgozásból* és *kiírásból.*

```
Konst ...                   <-- ld. rövid algoritmus
Típus TEgész=Egész          <-- ld. rövid algoritmus
Vált a, b, c: TEgész
Program
    Be: a, b [a>0 és b>0]
    c:=a+b                  <-- ld. rövid algoritmus
    Ki: c
Program Vége
```

Ebben a változatban az előbbi feltételezéseink explicitté válnak: látható, hogy miket olvasunk be, és mit írunk ki, látjuk a változók típusait, és az is egyértelmű, hogy a feldolgozáshoz már helyesen beolvasott értékekkel érkezünk, és a számolás után előálló érték(ek)et kell kiírni. Az algoritmust azonban mégsem szoktuk ilyen mértékben részletezni, hiszen a változók neve és típusa "generálható" a specifikációból, a program szerkezete pedig feladatfüggetlenül áll a beolvasás-feldolgozás-kiírás hármasából, és a beolvasás, valamint a kiírás is "generálható" a specifikáció alapján. Ami marad, az a lényeg: az adatok szerkezetének megadása (ha kell, típusdefiníciókkal), és a feladat lépésekre bontása.

Alprogramok használata esetén a program algoritmusa így néz ki:

```
Konst ...                       <-- ld. rövid algoritmus
Típus TEgész=Egész              <-- ld. rövid algoritmus
Vált a, b, c: TEgész
Program
    Beolvasás(a,b)
    Feladat(a,b,c)
    Kiírás(c)
Program Vége
```

[2] A példákat kizárólag pszeudokóddal adjuk meg a továbbiakban.

```
Eljárás Beolvasás(Vált a,b:TEgész)
    Be: a [a>0]
    Be: b [b>0]
Eljárás Vége

Eljárás Feladat(Konst a,b:TEgész, Vált c:TEgész)    <-- ld. rövid algoritmus
    c:=a+b                                           <-- ld. rövid algoritmus
Eljárás Vége                                         <-- ld. rövid algoritmus

Eljárás Kiírás(Konst c:TEgész)
    Ki: c
Eljárás Vége
```

Ebben az esetben is nyilvánvaló, hogy az adatszerkezet-leírásokon és a lényegi feladat algoritmusán kívül minden egyéb mechanikusan származtatható a specifikációból, és ezért nem szoktuk leírni. Annyi előnye van azonban ennek a részletes leírásnak van, hogy határozott ajánlást ad a kód strukturálására.

A fenti algoritmust tovább fejleszthetjük. Mit kellene csinálni, ha eljárások helyett függvényeket alkalmaznánk. Ebben az esetben a főprogram így nézne ki:

```
Program
    (a,b) := Beolvasás
    (c)   := Feladat(a,b)
    Kiírás(c)
Program Vége
```

Az értékadás bal oldalán zárójelbe tett adatokkal azt jelzem, hogy ebben az esetben a beolvasásnak egy összetett adatszerkezetet kellene visszaadnia, amely tartalmazza a beolvasott adatokat, a feldolgozásnak pedig a kimeneti adatok összességét kellene visszaadnia.

Továbbvíve ezt a gondolatot, általánosíthatjuk a részletes algoritmusunkat. Ha a programot egy függvénynek tekintjük, amely a bemeneti adatokból előállítja a kimenetiket, akkor ezt az adatszerkezetekben is jelezhetjük:

```
Konst ...                   <-- feladatspecifikus
Típus TEgész = Egész        <-- feladatspecifikus
Típus TBemenet = Rekord(
        a,b:TEgész)         <-- feladatspecifikus
Típus TKimenet = Rekord(
        c:TEgész)           <-- feladatspecifikus
Változó be: TBemenet
        ki: TKimenet
```

A főprogram algoritmusa így feladatfüggetlen lesz:

```
Program                                 Program
    Beolvasás(be)                           be:=Beolvasás
    Feladat(be, ki)         VAGY            ki:=Feladat(be)
    Kiírás(ki)                              Kiírás(ki)
Program Vége                            Program Vége
```

A függvényes megoldás pedig függvénykompozícióvá egyszerűsíthető:

```
Kiírás(Feladat(Beolvasás))    <===>    Kiírás∘Feladat∘Beolvasás
```

Eljárások esetében a három alprogram így nézhet ki:

```
Eljárás Beolvasás(Vált be:TBemenet)
    Be: be.a [a>0]
    Be: be.b [b>0]
Eljárás Vége

Eljárás Feladat(Konst be:TBemenet, Vált ki:TKimenet)
    ki.c:=be.a+be.b
Eljárás Vége

Eljárás Kiírás(Konst ki:TKimenet)
    Ki: ki.c
Eljárás Vége
```

Ha szeretnénk elkerülni a folyamatos `be`/`ki` hivatkozást a `Feladat` eljárásban, akkor a következőket kell tennünk:

```
Eljárás Feladat(Konst be:TBemenet, Vált ki:TKimenet)
Vált a,b,c:TEgész
    a:=be.a             <-- "beolvasás"
    b:=be.b             <-- "beolvasás"
    c:=a+b              <-- "feldolgozás"
    ki.c:=c             <-- "kiírás"
Eljárás Vége
```

Összességében azt értük el, hogy a főprogram szerkezete feladatfüggetlen lett, a feladatspecifikumok pedig az egyes alprogramokba kerültek. Érdekes, hogy így a `Feladat` alprogram megismétli a korábbi főprogram szerkezetét, hiszen azon belül jelenik meg a beolvasás-feldolgozás-kiírás hármas. A fő különbség az, hogy ebben az esetben viszont a "beolvasást" és "kiírást" függetlenítettük a külvilágtól, tiszta adatszerkezetekkel dolgozunk csak. Az algoritmusban ennek túl sok előnyét nem tapasztaljuk, hiszen a „`Be: a`" és az „`a:=be.a`" között nincsen sok különbség, a kódban azonban érezhető ez utóbbinak az egyszerűsége.

Annyit jegyezzünk még meg, hogy a program végrehajtása konkrét megvalósítástól függetlenül az alábbi lépésekben történik: 1) az adatokat beolvassuk, így azok egy előre definiált szerkezetben rendelkezésre állnak; 2) ezekből az adatokból a feldolgozás során előre definiált szerkezetű kimeneti adatok készülnek, amiket 3) kiírunk. A folyamat érdekessége, hogy mindegyik lépését az adatok szerkezete alapvetően meghatározza. Az *adatszerkezet* definiálása nagyon fontos része a feladatmegoldásnak, számos tudatos és az algoritmusra és kódra kiható döntést igényel, így ezt negyedik (pontosabb nulladik) feladatként kell a feladat megoldása során tekintenünk. Míg a beolvasás, feldolgozás, kiírás egymástól függetlenül is megoldható, addig ezek mindegyike függ a választott adatszerkezettől. Így a specifikálásra és így az adatszerkezet leírására is külön figyelmet kell fordítanunk.

4. Beolvasás és kiírás a kódolásban

A programozásoktatás elején elsősorban kisebb feladatok fordulnak elő, amelyek a feladatmegoldás és a programozási nyelv alapvető elemeit gyakoroltatják. Ahogy a bevezetőben már volt szó róla, kódolás során a megoldandó feladathoz képest aránytalanul sok kódot kell főként a beolvasáshoz írni. Ebben a fejezetben azt járjuk körbe, hogy milyen lehetőségeink vannak a beolvasás

leegyszerűsítésére. Nemcsak a programozásoktatás elején lehetnek hasznosak ezek a technikák, hanem később is, amikor minél több feladattal és nem a körítésükkel szeretnénk foglalkoznunk.

4.1. A beolvasás elhagyása

Ha a beolvasás aránytalanul sok energiát vesz el, akkor az egyik *legegyszerűbb, ha elhagyjuk*. Ebben az esetben definiáljuk adatszerkezeteinket, és azokat a kódban beégetett értékekkel töltjük fel. A megoldás meghagyja a feladathoz illeszkedő adatszerkezet definiálásának szabadságát, csupán a beolvasást átugorva rögtön a feladatmegoldásra koncentrál.

```
int main() {
    int a, b, c;        // deklarálás
    a = 3;              // beolvasás
    b = 5;
    c = a + b;          // feldolgozás
    cout << c << endl;  // kiírás
}
```

A megoldás hátránya, hogy a program nem általános: új adatok megadásához a programot kell változtatni. Ugyanakkor ebben a fázisban nem feltétlenül az a célunk, hogy általános programot írjunk, hanem az, hogy a tanuló a megadott adatokkal elvégezze a feladatot. Az általánosítást majd megtanulhatja később, pl. önálló beolvasási feladatokkal. A tesztelési folyamat nem sokban különbözik a beégetett és általános esetben:[3]

- beégetett: *új adatok* → (*fordítás* → *futtatás*) → *eredmény ellenőrzése*
- általános: (*futtatás*) → *új adatok* → *eredmény ellenőrzése*

A megoldás további hátránya, hogy automatikusan ebben a formában nem tesztelhető.

4.2. Segédfüggvények biztosítása

A beolvasás sokszor azért bonyolult, mert nagyon alacsony szintű és kezdőknek komplex nyelvi elemeket kell használni a standard input-output kezelésére, ellenőrzésére, adatok átalakítására. Ezeket az alacsony szintű műveleteket azonban elrejthetjük *magasabb szintű függvények* mögé. A függvénykönyvárat természetesen valahogyan biztosítanunk kell a tanulóknak, erről a megfelelően beállított programozási környezet [1,2] automatikusan gondoskodhat. A segédfüggvények akár szabadon elemezhetőek, bővíthetőek, így a bemenet- és kimenetkezelés tanulásához is hozzájárulhatnak. Továbbra is nagy előny, hogy az adatszerkezetek definiálásának a feladata a tanuló kezében marad. Az alábbiakban pár példa látható ilyen függvények használatára:

```
bool pozitiv(int p) {
    return p > 0;
}

be(a);          // beolvasás
be(a, "a = ", "Pozitív számot kell megadni!", pozitiv);
be_sor(s);
be_sor(s, ';');
ki(a);          // kiírás
ki_sor(s);
```

ahol például az ellenőrzött beolvasás így nézhet ki:

[3] A zárójelben megadott lépések általában egyben elvégezhetőek a programozási környezetekben.

```
template<typename T>
void be(T& p, string message, string errormsg, bool f(T)) {
    bool jo;
    do {
        clog << message;
        cin >> p;
        jo = f(p);
        if (!jo) { clog << errormsg << endl; }
    } while(!jo);
}
```

4.3. Előre megírt beolvasás

Egy másik lehetőség, hogy a tanuló egy *előre elkészített sablon*t kap, amelyben a beolvasási és kiírási logika már megvalósított, neki csupán a feldolgozási részt kell helyesen kitöltenie. Mivel a beolvasás már adott adatszerkezetet tölt fel, ezen megoldások mindegyikében az adatleírás szabadsága hiányzik. Ez egyrészt lehet hátrány, hiszen ezt a képességet nem gyakoroltatja, de lehet előny is, hiszen egyrészt mintát lehet mutatni egy adott probléma esetén az adatábrázolásra, másrészt gyakoroltatja az alkalmazkodás képességét is, amely során meglévő döntéseket kell megértenie és azok mentén dolgoznia (soft skill).

4.3.1. Alprogramokra bontás nélkül

Az első esetben a kód nincs alprogramokra bontva, a deklarálás-beolvasás-feldolgozás-kiírás négyeséből a feldolgozás hiányzik a sablonban. Az ilyen feladatok megfelelő előkészítést igényelnek, és jól használhatóak értékelő környezetekben. Környezettől függően további lehetőség lehet az is, hogy a kód bizonyos részeit elrejtsük vagy szerkeszthetetlenné tegyük. Ha az egész kód szerkeszthető marad, akkor bármilyen nyelvi elem (pl. függvény) alkalmazható. Ha korlátozzuk a hozzáférést, akkor nyelve válogatja, milyen elemek használhatóak, éppen ezért is fontos az, hogy a feladatvégző lássa az írott kód egész kontextusát:

```
#include <iostream>
using namespace std;
int main() {
    int a, b, c;                // deklarálás
    cin >> a >> b;              // beolvasás
    // FELHASZNÁLÓI KÓD ELEJE
                                // feldolgozás
    // FELHASZNÁLÓI KÓD VÉGE
    cout << c << endl;          // kiírás
}
```

4.3.2. Alprogramokra bontással

Ha használunk alprogramokat, akkor továbbra is alkalmazhatjuk az előző alfejezetben írtakat, csupán a feldolgozó függvény belseje nincsen kitöltve.

```
// ...
int feladat(int a, int b);
int main() {
    // ...
    c = feladat(a, b);          // feldolgozás
    // ...
}
```

```
int feladat(int a, int b) {
    // FELHASZNÁLÓI KÓD ELEJE

    // FELHASZNÁLÓI KÓD VÉGE
}
```

Mivel az alprogramok funkcionálisan jól elkülöníthető egységek, ezért megfelelő feladatleírás mellett lehetővé válik a feladat megoldása *csupán egyetlen* függvény implementálásával.

```
int feladat(int a, int b) {
    // FELHASZNÁLÓI KÓD ELEJE

    // FELHASZNÁLÓI KÓD VÉGE
}
```

Ez a minta módszertanilag is számos előnnyel bír. Egyrészt nagyon jól mutatja, hogy a feladat megoldása markánsan elkülönül a beolvasástól és kiírástól, igazából nem is kellenek azok hozzá. A függvények – mint interfészek – paramétereiken és visszatérési értékükön keresztül kommunikálnak a külvilággal, nem érdekelve őket az, hogy az adat honnan érkezik és hova megy. Jól gyakoroltatja a modularitást, a felelősségi körök szétválasztását, és jól tesztelhető is, hiszen a hívó környezet határozza meg a bemeneti adatokat, és értékeli a kimenőket. Programozási nyelve válogatja, hogy ezt a fajta interfészt hogyan érdemes megvalósítani, milyen nyelvi elemek engedélyezettek. Legrugalmasabb megoldást az nyújtja, ha kihasználjuk az adott nyelv moduláris szolgáltatásait (modulok, osztályok, függvények). C++ esetében például a felhasználói kód egy fejléc állományban is megvalósulhat, amit a futtató/tesztelő környezet beemel és használ:

```
// feladat.h
#include <vector>           // tetszőleges include-ok
typedef vector<int> Szamok; // tetszőleges típusok és konstansok
int feladat(int a, int b) {
    // FELHASZNÁLÓI KÓD
}

// program.cpp
#include "feladat.h"
int main() {
    int a, b, c;            // deklarálás
    // ...
    c = feladat(a, b);      // feldolgozás
    // ...
}
```

4.3.3. Bemeneti-kimeneti adatszerkezet

Az algoritmusok vizsgálatánál láttuk azt a lehetőséget, amikor a bemenet és a kimenet egy összetett adatszerkezetként jelenik meg. Az előre megadott beolvasásnál elképzelhető egy olyan variáns is, amely ilyen adatszerkezetekkel dolgozik. Működhet ez alprogramokkal és azok nélkül is természetesen, az alábbi példák azonban alprogramokra mutatják be a lehetőségeket. C++ esetén egy ilyen feldolgozás így nézhet ki:

```
typedef int Egesz;
struct Input { Egesz a, b; };
struct Output { Egesz c; };
Output feladat(Input input) {
```

```
    int a = input.a;        // "beolvasás"
    int b = input.b;
    int c = a + b;          // "feldolgozás"
    Output output;          // "kiírás"
    output.c = c;
    return output;
}
```

Modern nyelvi elemekkel a "beolvasás" és "kiírás" a következőképpen egyszerűsíthető a `feladat` függvényben:

```
Output feladat(Input input) {
    auto [a, b] = input;    // "beolvasás"
    int c = a + b;          // "feldolgozás"
    return { c };           // "kiírás"
}
```

Ugyanez például TypeScriptben a következőképpen néz ki:

```
type Input = {a: number, b: number};
type Output = {c: number};
export function feladat({a, b}: Input): Output {  // "beolvasás"
    const c: number = a + b;                       // "feldolgozás"
    return { c };                                  // "kiírás"
}
```

4.4. Strukturált bemenet

Konzolos programok esetén a bemenet gyakorlatilag a bekért adatok egymásutánisága. Ennek struktúráját tehát a bekérő program logikája határozza meg. Ha ezeket az adatokat fájlba mentjük, akkor igazából számok és szövegek halmazát látjuk, de ennek értelmet a bekérő logika ad, ránézésre egy strukturálatlan állományról van szó. A beolvasás ráadásul fordítva történik: adott adatok egymásutánisága, ezeket kell beolvasni a megfelelő adatszerkezetekbe. A strukturálatlan bemenet beolvasásának egyik előnye, hogy meghagyja az adatábrázolás szabadságát. Hátránya a bonyolult logika a beolvasáshoz, illetve a bemeneti adatok nehezen átlátható szerkesztése.

A hátrányokon sokat segít a strukturált bemenet használata. Ez gyakorlatilag valamilyen ember által értelmezhető formátum használatát jelenti. A manapság népszerű szöveges adatleíró formátumok közül érdemes kiemelni a JSON [3] és a YAML [4] formátumot.[4] Példánk, illetve egy összetettebb struktúra így írható le:[5]

YAML	JSON

`a: 3  # első szám`
`b: 5  # második szám`
	`{`
	`  "a": 3,`
	`  "b": 5`
	`}`

`oktatok:`	`{ "oktatok": [`

[4] Az XML formátum inkább gépi feldolgozásra való, mint emberi szerkesztésre. Más formátumok, mint pl. a HOCON [5], izgalmasak, de támogatottságuk és ismerettségük kicsi.

[5] Érdemes megjegyezni, hogy a YAML 1.2-es verziójától a JSON leírás a YAML része.

```yaml
-
  nev: Zsakó László
  neptun: zslzsl
-
  nev: Szlávi Péter
  neptun: szpszp
```

```json
    { "nev": "Zsakó László",
      "neptun": "zslzsl" },
    { "nev": "Szlávi Péter",
      "neptun": "szpszp" }
  ]
}
```

A strukturált adatok beolvasásának mikéntje már programozásinyelv-függő. Ezekhez a népszerű formátumokhoz általában létezik olyan függvénykönyvtár, amely a beolvasásukat egyszerűvé teszi. C++ esetében mindkét formátumhoz található külső függvénykönyvtár [6,7], TypeScript esetében pedig az a kiváló helyzet, hogy a JSON a JavaScript (és így a TypeScript) nyelv adatszerkezeteinek sorosított formája, YAML→JSON átalakító pedig található hozzá [8]. Általában elmondható, hogy szerencsés, ha a strukturált formátum könnyen leképezhető az adott nyelv adatszerkezeteire vagy része azoknak.

A 2. függelék azt mutatja, hogyan lehet C++-ban JSON adatot beolvasni. Az alkalmazott JSON függvénykönyvtár az elemi típusokat és a `vector` típust automatikusan felismeri, összetett struktúráknál azonban a fejlesztőnek (az előkészítő tanárnak vagy a tanulónak) kell megírnia a konvertáló kódot. Ennek elkészítéséhez a függvénykönyvtár ismerete szükséges. A függelék példája egy szándékosan összetettebb beolvasást tartalmaz.

TypeScript esetében nincs szükség plusz könyvtárak használatára:

```typescript
const input: Input = JSON.parse(textarea.value);
```

A strukturált bemenet – ahogy neve is mutatja – már tartalmazza az adatábrázoláshoz szükséges döntéseket. Ha a bemeneti adat ilyen formában adott, akkor az adatábrázolást nem gyakorolja a tanuló. Strukturált bemenetet automatikus kiértékelő környezetekben is lehet használni megfelelő előkészítés mellett [1,2].

4.5. A beolvasás és kiírás mint külön feladat

A cikk elején említettük, hogy kódoláskor a tanuló igazából három feladattal találkozik: az adatok beolvasásával, az eredmény kiszámításával és a kiírással. Eddig főleg azzal foglalkoztunk, hogyan fókuszálhatnánk jobban a feladat megoldására a beolvasás helyett. A beolvasás azonban külön feladatként is megjelenhet: újabb nyelvi elemeket, fogalmakat és technikákat lehet rajta keresztül tanítani, bonyolultabb esetekben pedig programozási tételek is megjelennek benne. Egy tömb beolvasása például egy másolás programozási tétel, míg az előfeltétel-vizsgálatokban tetszőleges tétel (gyakran eldöntés) előfordulhat. Feladatként fel lehet adni, hogy egy adott bemeneti állományt adott szerkezetbe olvasson be a tanuló, vagy az adatszerkezetet is maga találja ki. Így a beolvasás maga válik feldolgozássá, a beolvasás eredményét kell kiíratni. Ennek egyébként olyan pozitív hozadéka is lehet, hogy rászoktatjuk a tanulókat arra, hogy a beolvasás eredményét is ellenőrizni kell.

A kiírásról eddig nem nagyon esett szó, mert annak komplexitása általában sokkal kisebb, mint a beolvasásé. Kevesebb nyelvi elem is szükséges hozzá. Így ilyen jellegű feladatok viszonylag korán, sőt a legelején előjöhetnek. Ezeknél a feladatoknál beégetve adott egy adatszerkezet, és azt kell kiíratni. Kezdetben ezek egyszerű típusok és egyetlen utasítással kiírathatók, logikai értékeknél megjelenhet az elágazás, tömböknél pedig a ciklus. Az alkalmazott programozási tétel – ha egyáltalán kell hozzá – szinte mindig másolás.

5. Összefoglalás

A programozásoktatásban manapság sokszor természetesnek vesszük, hogy a feladatmegoldás szerves részét képezi az adatok beolvasása és kiírása ezek bonyolultságával együtt. Bár a

végeredményt tekintve ez teljesen jogos elvárás, a cikkben igyekeztem rámutatni, hogy ennek nem feltétlenül kell így lennie. A három feladat elválasztható egymástól, és fokozatosan bevezethetőek. Nehézség szerint ugyanis legegyszerűbb a kiírás, aztán az egyszerűbb feladatok, végül a beolvasás. Ahogy a feladatoknál is fokozatosan haladunk a nehezebb példák felé, úgy a beolvasásnál is a bonyolultabb vizsgálatokat érdemes a vége felé megismertetni. A megoldás három nagy részfeladatának nem kell minden feladatnál együtt járnia, szétválasztásuk arra is lehetőséget ad, hogy a lényegi anyagra rámutató feladatokkal többet foglalkozzunk. A legtöbb feladatnál hagyjuk el a beolvasást, vagy tegyük nagyon egyszerűvé, és csak néha, pl. kifejezetten ezt gyakoroltató vagy beadandó feladatoknál részletezzük. A cikk számos módszert bemutatott arra, hogy hogyan is lehet a beolvasást egyszerűbbé tenni. Az egyes módszerek lehetővé teszik azt is, hogy a beolvasást fokozatossan ismertessük meg, például először beégetett adatokkal, majd előre elkészített kódok használatával, csak beolvasási feladatokkal. Legtöbb feladat esetében azonban a beolvasás-kiírás párosát érdemes teljesen el is hagyni. A legtöbb módszer működhet hagyományos fejlesztőkörnyezetekben, de a tanulást támogathatjuk speciális, automatikus értékelést vagy szerkesztési lehetőségeket kínáló környezetekben is.

Irodalom

1. Horváth Győző, Menyhárt László: *Webböngészőben futó programozási környezet megvalósíthatósági vizsgálata*, INFODIDACT 2016, Zamárdi, 2016.
2. Horváth Győző: *A web-based programming environment for introductory programming courses in higher education*, ICAI 2017, Eger, 2017
3. *JSON*
   http://json.org/ (utoljára megtekintve: 2017.11.11.)
4. *YAML*
   http://yaml.org/(utoljára megtekintve: 2017.11.11.)
5. *HOCON*
   https://github.com/lightbend/config/blob/master/HOCON.md (utoljára megtekintve: 2017.11.11.)
6. *JSON for C++*
   https://github.com/nlohmann/json (utoljára megtekintve: 2017.11.11.)
7. *YAML for C++*
   https://github.com/jbeder/yaml-cpp (utoljára megtekintve: 2017.11.11.)
8. *YAML for JavaScript*
   https://github.com/nodeca/js-yaml (utoljára megtekintve: 2017.11.11.)

Függelék

1. Példa egy bonyolultabb beolvasásra C++-ban

```
#include <iostream>
using namespace std;
int main(){
  // deklarálás
  int a, b;
  int c;

  // beolvasás
  bool jo;
  string sv;
  do {
```

```
    clog << "Kérem az első számot: ";
    cin >> a;
    jo = cin.good() && (a>0);
    if (!jo) {
      clog << "Pozitív számot kell megadni!" << endl;
      cin.clear();
      getline(cin, sv);
    }
  } while(!jo);

  do {
    clog << "Kérem a második számot: ";
    cin >> b;
    jo = cin.good() && (b>=0);
    if (!jo) {
      clog << "Nemnegatív számot kell megadni!" << endl;
      cin.clear();
      getline(cin, sv);
    }
  } while(!jo);

  // feldolgozás
  c = a + b;

  // kiírás
  cout << c << endl;

  return 0;
}
```

2. Példa JSON beolvasására C++-ban

```
#include <iostream>
#include <vector>
#include "json.hpp"
using json = nlohmann::json;
using namespace std;

typedef vector< vector<int> > matrix;
struct Pont { int x, y; };
struct Input {
    int a, b;
    vector<int> szamok;
    vector<Pont> pontokTombje;
    matrix m;
};
void from_json(const json& j, Pont& p) {
    p.x = j.at("x").get<int>();
    p.y = j.at("y").get<int>();
}
void from_json(const json& j, Input &input) {
    input.a            = j.at("a");
    input.b            = j.at("b");
```

```
    input.szamok       = j.at("szamok")      .get< vector<int> >();
    input.pontokTombje = j.at("pontokTombje").get< vector<Pont> >();
    input.m            = j.at("m")           .get<matrix>();
}
void feldolgozas(Input input) {
    auto [a, b, szamok, pontokTombje, m] = input;
}
int main() {
    json j;                 // beolvasás
    cin >> j;
    Input input = j;
    feldolgozas(input);     // feldolgozás
    // ...                  // kiírás
}
```