Kezdő programozók hibái

Azok a buktatók, amikről ha nem tudsz, napokat töprenghetsz hiába programozás-tanulás közben

2011.07.01.
www.programozas-oktatas.hu
Pasztuhov Dániel

Miért írom ezt az útmutatót?

Programozás-oktatói pályafutásom alatt nagyon sokszor találkoztam olyanokkal, akik a nulláról akarnak megtanulni programozni. Egy idő után megfigyeltem, hogy szinte minden esetben belefut a tanítvány ugyanazokba a hibákba. Szinte kivétel nélkül mindenki, sőt, van, aki többször is. Az eredménye persze az, hogy utána órákat – ha nem napokat – a gép előtt görnyed és „guglizik”, hátha rátalál a megoldásra, de általában nem. Így aztán hozzáértő segítségét kell kérnie.

Így aztán arra gondoltam, hogy elkezdem gyűjteni ezeket a hibákat, és összegyűjtök egy csokorra valót tanulságul azoknak, akik éppen most kezdenek programozni tanulni, hogy megspóroljak nekik néhány napnyi szenvedést és guglizást.

Ez persze nem jelenti azt, hogy az anyag teljes lenne. Ezek csak azok a hibák, amiket az én feladataim kapcsán láttam rendszeresen előjönni, nyilván rengeteg másfajta hiba lehetséges még.

Az anyag C-jellegű nyelvek (C, C++, Java, PHP) specialitásaira vonatkozik elsősorban, ugyanis nagyon sokan ilyen nyelveket felhasználva igyekeznek elsajátítani a programozást. Még ha egyébként mondjuk a Pascallal sokkal könnyebb dolguk lenne, de számos egyetem erőlteti, hogy C-jellegű nyelveken kell elkezdeni a programozás-oktatást. A vallásháborúba meg most hadd ne menjünk bele.

Tartalomjegyzék

1. Az alapok csapdái

a. A figyelem

Sajnos sokszor előfordul, hogy szóról szóra elmondom a megoldását az adott problémának, majd rákérdezek újra, és a tanítvány csak néz, mint Rozi a moziban. Az alábbi tulajdonságokra természetesen fel szoktam hívni a hallgatóság figyelmét, ellenben általában ez nem sokat ér.

A figyelem nagyon fontos, főleg, ha meg akarsz tanulni programozni!

b. Az egészosztásról

Adott ez a programrészlet:

```
int c = 45;
int f = 9 / 5 * c + 32;
```

… és a tanítvány nem tudja, miért kap 113 helyett 77-et. A magyarázat az osztás „kétszínűségében” rejlik. Ha mindkét paramétere egész, akkor az osztás matematikai eredményének egészrészét adja (egészként). Ha legalább az egyik paramétere törtszám (float, double), akkor pedig a matematikai eredmény lesz az értéke törtszámként. Tehát helyesen így lehetne (többek között):

```
int c = 45;
float f = 9.0f / 5 * c + 32;
```

c. a = b, azaz b felveszi a értékét…

Nagyon sokan nem tudják elsőre megjegyezni, hogy az értékadás merről merre zajlik. Nos, nem balról jobbra, hanem JOBBRÓL BALRA.

d. if (a = b)…

Az elágazás (if) tanulásánál nagyon gyakori ez a gubanc. A tanítvány feltehetőleg össze akarta hasonlítani a-t és b-t és ha megegyeznek, akkor akart valamit csinálni. Helyesen úgy lett volna, hogy

```
if (a == b)…
```

de a tanítvány még nem tudja, hogy az = és a == operátorok között mi a különbség. Az = értékadás, míg az == összehasonlítás.

A Java már szól azért, hogy ha ilyen hibát elkövetünk, de a C és C++ fordítók nem (feltétlenül), mert hiszen értelmes is lehetne.

Mit is jelent? Előbb a-nak értékül adjuk b értékét, majd a kifejezés értékéről eldöntjük, hogy igaznak számít-e vagy hamisnak (azaz nem nulla vagy nulla). A kifejezés értéke pedig attól függ b milyen értéket vett fel.

e. if, while, for törzsében több utasítás

```
if (a == b)
      egyutasítás;
      mégegyutasítás;
```

A tanítvány azt szerette volna, hogy ha a és b értékei megegyeznek egymással, akkor lefusson mind egyutasítás, mind mégegyutasítás. De a blokkjelet elfelejtette körülük. Általában akkor jelentkezik ez a hiba, amikor először csak egy utasítás áll az if igaz ágában, majd hozzátesz még egyet, csak megfeledkezik a { } párról. És nem érti, miért nem fut le a második utasítás. Azért mert az if (és for, while) esetén a törzs vagy egy utasításból áll, vagy ha több utasításból áll, akkor kapcsos zárójelek közé kell tenni, azaz helyesen:

```
if (a == b) {
      egyutasítás;
      mégegyutasítás;

}
```

f. for végére pontosvessző?

Adott a következő sor:

```
for (i = 6; i < 10; i++); { és a műveletek }
```

… és a tanítvány nem érti, hogy miért csak egyszer fut le, amit a blokkba tett. Hiszen már blokkban van! A rejtély kulcsa a pontosvessző, ami jelen esetben egy üres utasítást jelképez. a for-ciklus lefut négyszer, és lefuttatja az üres utasítást, majd hozzákezd a blokk végrehajtásához – egyszer…

g. Hol van a laptopon a * billentyű?

AltGr+ SHIFT melletti ’-’ (mínusz) jel.

h. #define

Adott a következő program:

```
#define PI 3.14;
int main() {
    printf("%lf", PI);
    return 0;
}
```

… és minden rendben lévőnek is tűnik, hiszen mindenhol van pontosvessző a sorok végén. Miért gond mégis az, hogy a ; előtt) kéne a 3. sorban? Hát ott van!

Igen, de más is van ott. A kérdéses sorból ugyanis az előfeldolgozás után a következő sor keletkezik (miután a define-t behelyettesítettük):

```
    printf("%lf", 3.14;);
```

Vagyis keletkezett bele egy nem várt pontosvessző, ami előtt kéne, hogy legyen még egy bezárójel.

Egyszóval: #define végére nem kell (tilos!) a pontosvessző.

2. Függvények nyalánkságai

a. Függvény meghívása és a deklarációja

Sokan összekeverik a kettőt. Adott a függvény deklarációja:

```
int fuggveny(int a, long b);
```

valamint a függvény meghívása:

```
x = fuggveny(c, d);
```

Néha természetesen összekeveredik, így a meghívásba tesznek típusnevet… Az ökölszabály: ha függvényt deklarálunk, van típusnév, ha meghívjuk, nincs.

b. A függvény meghívásakor, ha változót használunk, akkor annak nevének meg kell egyeznie a (formális) paraméter nevével.

Természetesen nem. Például az alábbi kóddal jól szemléltethető:

```
int max(int a, int b) { return a > b ? a : b; }
```

…

```
int a,b,c,d, ab,cd;
```

…

```
ab = max(a,b);
cd = max(c,d);
```

Hát már miért ne számolhatnánk ki c és d maximumát?

c. Visszatérési érték

Éppen megtanulta a tanítvány a függvények használatát, máris gondban lehet. Nem érti, hogy mikor meghívja a függvényt a visszatérési érték az mi a csuda, és hogy hogy kell azt kezelni. Gyakran láthatunk ilyesmit:

```
egyfuggveny(p1, p2, p3);
valtozo = egyfuggveny(p1, p2, p3);
```

Azt gondolja, hogy az egyfuggveny valahogy magába szívja a visszatérési értéket, és bent tartja, amíg le nem írjuk a függvény nevét (rosszabb esetben teljes meghívását) újra, és akkor végre kiereszthetti magából. Látszólag egyébként jól is működik a dolog. A szépséghiba az, hogy a függvény kétszer fut le.

A másik lehetséges probléma, ha nem is tárolják a visszatérési értéket, csak gondolják, hogy majd visszajött valahogy, és majd felhasználjuk… valahogy.

d. Visszatérés logikai értékkel

Ha egy függvénynek meg kell vizsgálnia egy feltételt és annak megfelelően vissza kell térnie egy igaz vagy hamis értékkel, azt sokan így csinálják:

```
if (a<b)
      return true;
else
      return false;
```

Pedig egyszerűbben is lehetne:

```
return a < b;
```

e. Hányszor lehet leírva a return egy függvényben?

Van, aki szerint egyszer, de ő téved. A return akárhányszor le lehet írva a függvényben, de ha a vezérlés ráfut, akkor kilép a függvényből. Olyan, mint a rexasztal. Több lyuk is van rajta, de ha arra jár egy golyó, beleesik.

f. Függvényen belül lehet függvényt definiálni?

Nem. Bár jogos a felvetés, mert pl. Pascalban ez megengedett. C-ben nincs függvényen belül függvénydefiníció. Javaban is meg lehet oldani, de ez nem kezdőknek való téma.

g. Prímtulajdonság vizsgálata

Gondoljunk arra a feladatra, hogy el kell dönteni, hogy egy kártyacsomagban van-e piros lap. Végignézzük az összes lapot. Ha az, amit nézünk piros, akkor kész vagyunk, tudjuk, hogy van a pakliban piros lap. Ha nem piros, akkor még nem tudunk semmit, hiszen később lehet még piros. A tanítványok legtöbbje ezt elrontja, és azt írja a programba, hogy ha piros, akkor kész vagyunk, mert van a pakliban piros, ha nem piros, akkor pedig tudjuk, hogy nincs benne, valahogy így:

```
for (i = 0; i < max; i++) {

      if (piros)
            return true;
      else
            return false;

}
```

A prímtulajdonság vizsgálata címet pedig azért kapta, mert ugyanez a helyzet akkor, ha meg kell állapítani egy számról, hogy prím-e. Ha találunk osztót 2 és a szám-1 (vagy a szám fele vagy a szám négyzetgyöke) között, akkor azonnal tudjuk, hogy nem prím. Ha az adott szám nem osztója, akkor bizony tovább kell menni.

h. switch-case

A leggyakoribb probléma az, hogy a case-ágak végén lemarad a break.

3. A C++ nyalánkságai

a. Az osztály definíciója végére pontosvessző kell!

Nincs mit magyarázni rajta. Ha elfelejted, nézheted órákig, de nem hinném, hogy rá fogsz jönni hamarabb.

b. Egy objektum függvényének meghívása

… esetén nehezen esik le az aktuális objektum fogalma. Mármint az az objektum, amire a tagfüggvény futása esetén a this mutatni fog.

c. Osztály tagfüggvényének átadni a tagváltozókat is

Az objektum-orientált programozás nem tartozik a gyorsan megérthető részek közé. Az egyik gyakori hiba, hogy – struktúrált programozásból váltva – a tanítvány át akarja adni a tagfüggvénynek a tagváltozókat is, valahogy így:

```
class Valami {
      int belsovaltozo;
public:
      void tagfuggveny(int belsovaltozo, int parameter) {…}
};
```

Természetesen a tagfüggvény hozzáfér a tagváltozóhoz, így erre semmi szükség… sőt, mindenféle objektum-orientált tervezési elvet sért, ha így teszünk.

Még az elején static-ra állítottam a pályát tároló tömböt -ki tudja már miért-, és így nem tudtam mentés-betöltéskor serializálni. Ezért nem frissült a pálya betöltés után.