A Perl Hibakereső

Perl kódunkba print utasításokat tűzdelve is kereshetünk hibákat, de egy teljes értékű hibakereső ennél sokkal több információt szolgáltat.

A hibakeresés minden nyelvben bosszantó, ám szükséges rossz, akár saját kódunkat javítjuk, akár másvalakiét, aki éppen a rendszerünkön dolgozik. Bármi, ami megkönnyíti a hibakeresést, komoly előnyt jelenthet. A *Perl* parancssoros hibakeresővel rendelkezik, amely jelentősen megkönnyítheti a hibakeresés folyamatát. Cikkünkben áttekintjük a hibakereső használatának alapjait és bemutatunk néhány hasznos trükköt.

Hibák elkerülése figyelmeztetések és Strict kulcsszó használatával

A *Perl* maga elképesztő mennyiségű hibát képes küszöbölni, pusztán azzal, hogy a programunk elején bekapcsoljuk a figyelmeztetéseket és a szigorú szabályértelmezést (`strict`). Amennyiben programunk tartalmazza a `use warnings;` sort egy tucatnyi általános hibát máris elfoghatunk. Megtalálhatjuk például a csak egyszer használt változóneveket, melyek rendszerint elírások, a beállításuk előtt felhasznált skaláris változókat, vagy a újradefiniált alprogramokat.
A hibakereső üzeneteket kicsit beszédesebbé tehetjük, ha a `use diagnostics;` sort is a kódba szúrjuk, amely minden egyes figyelmeztetéshez leírást is mellékel. A másik megoldás, ha a figyelmeztetések jelentésére rákeresünk a `man perldiag` paranccsal.
Amennyiben 5.6-os változatnál régebbi *Perlt* használunk , a figyelmeztetések használata helyett a -w kapcsolót kell alkalmaznunk a script első sorában, valahogy így: `#!/usr/bin/perl -w`.

Végül sok általános hibát elkaphatunk a `use strict;` kifejezéssel, amely tulajdonképpen néhány nem teljesen biztonságos programozói egyszerűsítést tilt le. A `strict` kulcsszó által bevezetett korlátozások a következők:

- A változókat használat előtt vagy definiálnunk kell a `my` illetve `our` kulcsszó valamelyikével, vagy importálni kell őket a `use vars` kulcsszóval, vagy a csomagnevet is tartalmazó teljes nevet kell használnunk.
- A csupasz szavak csak alprogramnevek lehetnek, karaktersorozatok nem (például: `$string = blabla;`).
- A hivatkozások nem lehetnek szimbolikusak; lásd a széljegyzetet.

Mint láthatjuk, a figyelmeztetések és a `strict` kulcsszó a *Perl* néhány olyan képességét korlátozzák, amelyeket egyébként jó célra is lehet használni, de gyakran sikerül velük visszaélni. Ezek a parancsok megkönnyítik a hibakeresést, hiszen a Perl maga kapja el a hibákat.

Szimbolikus hivatkozások

A szimbolikus hivatkozások abban különböznek a szokásos "kemény" hivatkozásoktól, hogy itt a változó egy másik változóra hivatkozik. A szimbolikus hivatkozások olyankor jönnek létre amikor a programozó egy karaktersorozatot használ azonosítónak. A következő két sor például alapesetben érvényes *Perl* kód:

```
$name = "username";
$$name = "da";       # $username beállítása
```

Az ilyen kóddal könnyen előfordulhat, hogy az értelmező azt hatja végre amit kiadtunk és nem azt amire gondoltunk. Könnyen használhatunk szimbolikus hivatkozást ott, ahol kemény hivatkozást akartunk megadni, illetve elkeverhetjük a hivatkozott változónevet, hiszen soha nem szerepel a kódban. Sokkal biztonságosabb és a ugyanezt a hatás érhető el, ha az ilyen változókat inkább hashben tároljuk majd a `strict` kulcsszó használatával bekapcsoljuk a szigorú változófigyelést.

Mi a print utasítások hátránya?

Sokan kérdezhetik, miért baj az, ha `print` utasításokat szórunk szét a kódunkban? Semmi probléma nincs ezzel a hibakereső technikával, csak éppen az interaktív hibakeresővel sokkal hatékonyabban tudunk dolgozni. Futtatáskor a program és a környezet valamennyi összefüggését vizsgálni tudjuk, nem csak azokat melyekre előre gondoltunk, és tisztábban láthatjuk pontosan mit is csinál a kód. Reményeim szerint a cikk végére mindenki egyet ért majd velem abban, hogy az a kevéske idő amit a hibakereső megtanulásába fektettünk bőségesen megtérül.

A hibakereső indítása

A hibakeresőt a parancssorból indíthatjuk el, a *Perlnek* átadva a `-d` kapcsolót :

```
perl -d filename.pl
```

Amennyiben `CGI.pm`-el készített ***CGI*** programot vizsgálunk, az átadandó paraméterek mellé egyszerűen tegyük be a `-d` kapcsolót is a parancssorba:

```
perl -d filename.pl param=value param2=value
```

A parancssoros megoldáson kívül a ***Perl*** hibakeresőt bizonyos ***IDE***-k (fejlesztői környezetek) részeként is használhatjuk, ilyen a ***GNU Emacs*** vagy a ***Activestate Komodo*** programja, valamint léteznek ***GUI*** hibakereső felületek is, például a ***ddd*** vagy a ***ptkdb***. A rövidség kedvéért e cikkben csak a parancssorral fogok foglalkozni, de az alapelvek a ***GUI*** hibakeresőben is hasonlóak lesznek.
Ha parancssoros hibakeresőt használunk, sokat segíthet a `Term::ReadLine` modul telepítése , ez ugyanis lehetővé teszi a lépkedést a parancs-történetben.
Következzen a cikkünkben használt példaprogram. Másoljuk a következőket egy ***sample.pl*** nevű fájlba:

```
#!/usr/bin/perl
use warnings;
use strict;
my $name = “Pengu”;
foreach (1..20) {
  &shout($name);
}
sub shout {
  my $name = shift;
  print “*** $name ***\n”;
}
```

Legfontosabb hibakereső parancsok

Az hibakeresés megkezdéséhez a következő hét alapvető parancsot kell ismerünk:

- ***s***: egy lépés végrehajtása a következő sorig, belelépve az alprogramokba.
- ***n***: egy lépés végrehajtása a következő sorig, átugorva az alprogramokat.
- ***r***: folyamatos végrehajtás, amíg az adott alprogramból vissza nem térünk.
- ***c <sor száma>***: folyamatos végrehajtás a megadott sorig.
- ***l <sor száma, tartomány vagy alprogram neve>***: forráskód listázás.
- ***x <kifejezés>***: a ***<kifejezés>*** kiértékelése és olvasható kiíratása.
- ***q***: kilépés a hibakeresőből.

Tesztprogramunkat lefuttatva a hibakeresőben próbáljuk ki a fentieket:

```
perl -d sample.pl
```

Most a hibakereső indulási információit kell látnunk:

```
Default die handler restored.
Loading DB routines from perl5db.pl version 1.07
Editor support available.
Enter h or h h for help or
man perldebug for more help:
main::(sample.pl:6):     my $name = “Pengu”;
  DB<1>
```

Ez a program indulása előtti állapot. Az utolsó előtti sorban a hibakeresés állapotáról olvashatunk hasznos információkat: láthatjuk, hogy a main csomagban vagyunk, a ***sample.pl*** fájl 6. sorában, végül megmutatja azt a sort amit éppen futtatni fogunk.
Az utolsó sorban láthatjuk a parancssort és a parancs sorszámát (amely a begépelt parancsokkal folyamatosan növekszik) valamint „kacsacsőröket”, melyek száma a beágyazott parancsokat jelzi. Ezzel egyelőre nem kell foglalkoznunk.
Gépeljük az `s` parancsot a parancssorba majd üssük le az ENTER billentyűt, ezzel végrehajtva a program egyetlen sorát:

```
  DB<1> s
main::(sample.pl:8):     foreach (1..20) {
  DB<1>
```

A parancs ismétléséhez üssünk ENTER-t és ismételgessük tetszés szerint, hogy lássuk a program valóban lépésenként fut. Valahányszor egy `print` utasítást hagyunk el, az kimenet a hibakereső adatokkal együtt megjelenik a képernyőn.
Most próbáljuk ki az alparancsokat átugró utasítást (n) és nyomjunk néhány ENTER-t. Ahogy végiglépkedünk a cikluson, láthatjuk, hogy az alprogram eredményét azonnal visszakapjuk, és nem lépkedünk végig az alprogram utasításain.
Most próbáljuk ki az alprogramból való visszatérést (`r`). De ne indítsuk el most rögtön, akkor ugyanis a program végéig fog futni, hiszen a főprogramból fogunk „visszatérni”. Először csináljunk néhány ismétlést az `s` paranccsal míg az alprogramba nem lépünk. Ezután az `r` parancsot kiadva a következőket kell látnunk:

```
  DB<1> s
main::(sample.pl:8):     foreach (1..20) {
  DB<1>
main::(sample.pl:9):       &shout($name);
  DB<1>
main::shout(sample.pl:13):         my $name =
➥shift;
  DB<1> r
*** Pengu ***
void context return from main::shout
main::(sample.pl:8):     foreach (1..20) {
  DB<1>
```

Figyeljük meg a `void context return from main::shout` sort. Ha a ciklusban lekérdeznénk a visszatérési értéket, itt megnézhetnénk. ***Perlben*** a függvények és az alprogramok a hívási környezettől (skalár, tömb vagy `void`) függően különböző értékeket adhatnak vissza. A ***Perl*** hibakereső egyik hasznos képessége az `r` parancs, amely megmutatja a hívó környezettípusát. Könnyen megtalálhatjuk vele bizonyos hibákat, például amikor konstans értéket várunk, de véletlenül az alprogramunk tömböt ad vissza.
Következik az `l` parancs. Próbáljuk ki:

```
  DB<1> l
8==>     foreach (1..20) {
9:         &shout($name);
10       }
11
12       sub shout {
13:        my $name = shift;
```

```
14:         print “*** $name ***\n”;
15        }
  DB<1>
```

Az `l` önmagában egy oldalnyi forráskódot listáz, a következő végrehajtandó sortól kezdve, szöveges nyíllal jelölve meg az éppen végrehajtandó sort. Kilistázhatunk egy tartományt is ha sorszámokat adunk meg: `l 200-230`. Továbbá, az alprogramok nevét megadva azok listáját is megkaphatjuk: `l shout.`

A `c` parancs egy adott sorig folytatja a végrehajtást, így előreugorhatunk a minket érdeklő tetszőleges pontra:

```
  DB<1> c 14
main::shout(sample.pl:14):  print “*** $name ***\n”;
  DB<1>
```

A parancssorba gépelve bármilyen *Perl* kifejezést végre tudunk hajtani, beleértve a futó kódot megváltoztat utasításokat is. Akár kézzel is beállíthatunk változókat a programban. Az `x` parancs kiértékeli, majd olvasható formában kiírja a kifejezés értékét. Valamennyi kimenet elé egy sorszámot fűz, minden visszafejthető elemet ***visszafejt (dereference)*** valamint a visszafejtés minden szintjét egyel beljebb kezdi. Példaképpen alább beállítottunk egy `@sample` nevű tömböt, majd kiírattuk:

```
  DB<1> @sample = (1..5)
  DB<2> x @sample
0  1
1  2
2  3
3  4
4  5
  DB<3>
```

Figyeljük meg, hogy a hasheket kulcsokkal és értékekkel együtt jeleníti meg, mégpedig soronként egyet. A hasheket a `\` jellel kezdve tudjuk helyesen megjeleníteni, ez a jel ugyanis a hasht hash-hivatkozássá alakítja így az helyesen fejtődik vissza. Ez a következőképpen néz ki:

```
  DB<4> %sample = (1 .. 8)
  DB<5> x \%sample
0  HASH(0x83d53bc)
   1 => 2
   3 => 4
   5 => 6
   7 => 8
  DB<6>
```

Ha elégedettek vagyunk az eredménnyel, lépjünk ki hibakereső gyakorlatunkból a `q` paranccsal.

További négy hibakereső parancs

Sok ember kizárólag a fenti parancsokat használja a *Perl* hibakeresőben. Amikor azonban már kényelmesen tudjuk használni őket, a következő négy paranccsal még hatékonyabbá tehetjük a hibakeresést, különösen ha objektum orientált kódot készítünk:

- */<minta>*: listázás a szabályos kifejezés következő egyezésénél.
- *?<minta>*: listázás a szabályos kifejezés előző egyezésénél.
- *S*: a programban hozzáférhető valamennyi alprogram és metódus listázása.
- ***m <objektum vagy csomag>***: az adott objektum vagy csomag összes metódusának listázása.

A `/` és `?` jelekkel előre és hátrafelé kereshetünk és jeleníthetünk meg kódrészleteket, tetszőleges karaktersorozat vagy szabályos kifejezés egyezését vizsgálva. A keresendő szöveg elé nem kell szóközt tenni:

```
  DB<6> /name
6:       my $name = “Pengu”;
```

Az `S` és `m` parancsok akkor lehetnek hasznosak, ha az elérhető alprogramokat szeretnénk megismerni: az `S` a programban elérhető összes alprogramot kilistázza. Ezek éppen fordított sorrendben jelennek meg mint ahogy a `use` vagy `require` kulcsszóval beillesztettük őket, és a hibakeresőből beillesztett alprogramokat is tartalmazzák így például a `Term::ReadLine`-t is. Az `m` parancs az objektumban vagy egy egész csomagban elérhető valamennyi metódust listázza.
Nézzünk egy példát:

```
  DB<7> use CGI
  DB<8> $q = new CGI
  DB<9> m $q
AUTOLOAD
DESTROY
XHTML_DTD
_compile
_make_tag_func
_reset_globals
_setup_symbols
add_parameter
all_parameters
[...]
```

Műveletek, Töréspontok és Figyelőpontok

A műveletek, töréspontok és figyelőpontok, még ennél is nagyobb vezérlést adnak a kezünkbe a hibakereső és a futó program felett. Ezeket elképzelhető, hogy szívesebben használjuk egy grafikus *Perl* hibakereső felület, például a ***ddd***, ***ptkdb*** vagy ***Activestate Komodo*** alól. A *Perl* hibakeresőjét sokan bírálják amiatt, hogy a parancsok kiadásához meg kell jegyeznünk a megfelelő parancs gyorskombinációkat, ráadásul a parancsokon belül további megjegyezendő gyorskombinációkkal kerülünk szembe.
Ráadásul *Perl 5.8* alatt a jobb belső összhang érdekében néhány billentyűkombinációt megváltoztattak. Gyakran előfordul, hogy az ember kénytelen 5.8-as és korábbi verziókat is használni, így aztán sokszor egyszerűbb ***GUI*** alatt dolgozni. A parancsokat a parancssor alapján mutatom be, az alapelvek természetesen mindenhol ugyanazok.
A ***művelet (action)*** célja, hogy kódrészletet szúrjunk be a programunkba a forráskód megváltoztatása nélkül. Hasznos lehet, ha a kód élesben fut, és ki akarunk próbálni valamit. Akkor is jól jön, ha éppen a hibakeresés közepén vagyunk, és nem szeretnénk egy változtatás miatt újraindítani az egész hibakeresőt.

Műveletet a következő formában adhatunk meg:
```
a <line-number> <code>.
```

Nézzünk egy példát:
```
DB<10> a 9 $index = $_;
```

A fenti sorral egy új parancsot adtunk a `foreach` ciklushoz, amely tárolja a folyamatosan növekvő indexszámlálót. Ha kilistázzuk a programot, a műveleteket tartalmazó sorok mellett egy a betűt láthatunk. A művelet mindig előbb hajtódik végre mint a sor amelyhez kapcsolódik. A beállított műveleteket az `L` paranccsal tudjuk listázni, törölni pedig úgy lehet őket, hogy az a és a sorszám után nem írunk semmilyen parancsot. Az eddigiek a Perl 5.6 és korábbi verzióira vonatkoztak; ***Perl 5.8*** alatt a műveletet az A parancs és a művelet sor száma törli.
A töréspontok és figyelőpontok folyamatos végrehajtás során adják vissza a vezérlés a hibakezelőnek, így például a korábban bemutatott `r` vagy `c` kiadása után. Akkor lehetnek hasznosak, ha éppen egy adott számú ciklusismétlődés után megjelenő problémát szeretnénk vizsgálni, és nem szeretnénk mindig kézzel végiglépkedni a cikluson.
Töréspontot adott sorra vagy alprogramra állíthatunk be és feltételhez is köthetjük. A töréspont megadása a b paranccsal történik a következő formában:
```
b shout
```

Ha kilistázzuk a programot, a `shout` alprogram első sorának sorszáma mellett láthatjuk a `b` betűt. A végrehajtás folytatásához üssük be a `c` parancsot, és az az alprogram belsejében fog megállni.
Amennyiben követtük az előző példát és a 9. sorban létrehoztunk a műveletet, be tudunk állítani egy olyan töréspontot, amely éppen a ciklus adott körbefordulásánál áll meg:
```
b shout ($index eq 8)
```

Könnyen elképzelhető milyen hatékonyak lehetnek ezek a műveletek és töréspontok, ha egy hosszabb, összetett feltételes kifejezéseket és külső adatforrásokat tartalmazó programban keresünk hibákat.
A töréspontokat a `L` parancs listázza, törölni őket ***Perl 5.6*** és korábbi verzióiban a `d` paranccsal lehet. ***Perl 5.8*** alatt a `B` utasítás törli a töréspontokat.
A figyelőpontokra talán még jobb elnevezés a kifejezésfigyelés. Ezek akkor állítják le a programot, ha egy adott kifejezés megváltozik. ***Perl 5.6*** alatt a `w` paranccsal a következő módon állíthatjuk be őket:
```
w $name
```

A figyelőpontokat az `L` listázza és valamennyit törölhetjük ha a `w` utasítást paraméter nélkül adjuk ki. ***Perl 5.8*** alatt a w paranccsal vehetünk fel és a W paranccsal törölhetünk figyelőpontokat.

A Perl hibakereső testreszabása

Az első dolog amit tudnunk kell, hogy a ***Perl*** hibakereső nem más mint egy ***Perl*** könyvtár, amely kihasználja a ***Perl*** értelmezőbe épített programhurkokat. Ha akarjuk akár a teljes hibakeresőt is lecserélhetjük, ha lemásoljuk valahová az eredeti állományt, majd kódunk `BEGIN` ciklusában hivatkozunk rá:
```
cp /usr/lib/perl5/5.6.1/perl5db.pl ~/myperl5db.pl
```

A kódunk elejére a következőt írjuk:
```
BEGIN { require "~/myperl5db.pl" }
```

A fenti megoldást például akkor érdemes használni, ha a hibakereső 5.6-os változatának formátumát és működését jobban kedveljük mint az 5.8-asét.
Ezen kívül a `-d` parancskapcsolóval is megadhatunk másik hibakeresőt. A 5.6 maga is tartalmazz a ***Dprof profiler***-t amely a hibakereső hurkokat használja. Ezt a következőképpen használhatjuk:
```
perl -d:DProf mycode.pl
```

Akár saját programunkban is használhatjuk a hibakereső hurkokat. Kódunkban közvetlen töréspontot helyezhetünk el a `$DB::single = 1;` változó beállításával ami például akkor jöhet jól, ha a BEGIN blokkban szeretnénk hibát keresni. Normál esetben ugyanis ez a blokk az előtt fut le, hogy a hibakeresőtől megkapnánk a parancssort. Ezen kívül a hurkokat arra is felhasználhatjuk, hogy egy adott kódot futtassunk bármely alprogram lefutásakor. Ezekről, és a többi hibakereső hurokról további érdekességeket tudhatunk meg a `perldebug` kézikönyvoldalból.
A hibakereső belső változókészlettel rendelkezik, amelyet szintén megtalálunk a `perldebug` kézikönyvoldalon. A változók átírására a saját könyvtárunkban, vagy az aktuális könyvtárban elhelyezett `.perldb` beállításállományt használhatjuk. A beállításfájlban található ***Perl*** kód a hibakereső indulásakor fut le. A következő utasításokkal például saját parancsokat készíthetünk:
```
$DB::alias{'quit'} = 's/^quit(\s*)/q/';
```

Ez a sor lehetővé teszi, hogy a `quit` begépelésével is kiléphessünk. A `perldebug` kézikönyvoldalain néhány további hasznos `alias` tippet találunk.
Számos hibakereső opciót állíthatunk be közvetlenül a hibakeresőben az `O` paranccsal. Én ezek közül csak egyet használtam, amely a lapozóprogramot változtatja meg:
```
O pager=|less
```

Ezzel a megoldással minden, egy lapnál hosszabb kiíratást kedvenc lapozóprogramunkon keresztül tekinthetünk meg, ha egy pipe karaktert írunk a kiadott parancsok elé.

Linux Journal 2005. március, 131. szám

Daniel Allen (da@coder.com)
Egy 1200-baudos modem, ingyenes helyi tárcsázási lehetőség és egy MIT vendég azonosító jóvoltából ismerte meg a UNIX rendszert, amikor ezek a dolgok még léteztek. 1995 óta elhivatott Linux felhasználó.
A Prescient Code Solutions, szoftvertanácsadó cég elnöke.