CGI programozás C nyelven

Az összetett webes feladatok elvégzését a CGI egyszerűségének megőrzése mellett is felgyorsíthatjuk. A rengeteg hasznos könyvtárnak köszönhetően a parancsfájl-készítő nyelvekről C-re áttérni nem is olyan nagy ugrás, mint azt gondolnánk.

A *Perl*, a *Python* és a *PHP* alkotja a *CGI* alkalmazásprogramozás szentháromságát. A boltok polcai roskadoznak az ezekről a nyelvekről szóló könyvektől, a számítástechnikai sajtó úton-útfélen velük foglalkozik, ahogy számos internetes fórumnak is ezek adják a témáját. Szembetűnő ugyanakkor, hogy mennyire nem érdekel senkit a *C* alapú *CGI* alkalmazások írása. Írásomban éppen ezért a *C* nyelv *CGI* programozásra való használatát fogom tárgyalni, valamint ismertetek néhány olyan helyzetet, amelyben alkalmazása számottevő előnyökkel jár.

Én három ok miatt használok *C*-t az alkalmazásaimban: gyors, nagy tudású és stabil. Bár a szájhagyomány másképp sugallja, saját méréseim alapján egyszerűbb feladatok végrehajtásakor a *C* és a *PHP* sebessége azonos, amikor pedig a feladat bonyolultabbá válik, a *C* lehengerlő győzelmet arat.

A *C* emellett rendkívül nagy tudású nyelv. Maga a nyelv ugyan csak egyfajta váznak mondható, ám az elérhető könyvtárak hihetetlenül széles választékával szinte mindent meg tudunk oldani, amire csak egy számítógép alkalmas lehet. Természetesen a *Perl* sem kezdő játékos ezen a területen, és egy pillanatig sem vitatom, hogy mindkét nyelv rendkívüli sokoldalúságot kínál használójának; bár szerintem a *C* könnyebben bővíthető.

Továbbá, a *C*-ben írt *CGI* programok üzembiztosabbak. Mivel a program teljes egészében lefordításra kerül, nem érzékeny az operációs rendszer által biztosított környezet megváltozásaira, ahogy például a *PHP*. Maga a nyelv is állandónak tekinthető, esetében nem kell olyan gyökeres változásokkal számolni, mint amilyenek a *PHP* használóinak idegeit borzolták az elmúlt években.

Az alkalmazás

Alkalmazásom egy egyszerű eseménylista, a jövőbeli események kezelésére, például üzleti megbeszélések időpontjának rögzítésére vagy akár egy templom eseményeinek ütemezésére használható. Rendelkezik egy – szándékom szerint – jelszóval védett felügyeleti felülettel, valamint egy nyilvános felülettel is, amely – kizárólag – a közeljövő eseményeit listázza ki. Az alkalmazás felületfüggetlen, beállításai futás közben módosíthatók.

1. kódrészlet MySQL séma

```
CREATE TABLE event (
  event_no int(11) NOT NULL auto_increment,
  event_begin date NOT NULL default '0000-00-00',
  name varchar(80) NOT NULL default '',
  location varchar(80) NOT NULL default '',
  begin_hour varchar(10) default NULL,
  end_hour varchar(10) default NULL,
  event_end date NOT NULL default '0000-00-00',
  PRIMARY KEY  (event_no),
  KEY event_date (event_begin)
)
```

Nem írtam saját adattárat, inkább adatbázist használok, a hozzá való csatlakozáshoz szükséges adatokat egy beállító fájlba helyezem el. A felület és a kód szétválasztását különálló fájlcsoporttal biztosítom.

A felügyeleti felület alkalmas az események listázására, szerkesztésére, mentésére és törlésére. Az események listázása az alapművelet, amennyiben más műveletet nem választunk ki. Az új és a meglévő eseményeket egyaránt lehet menteni. A felület tartalmaz egy rácsozatot, ebben jelenik meg az eseménylista, valamint egy részletes képernyőt, amin egy-egy esemény részletei is megtekinthetők.

Az alkalmazás adatbázissémája egyetlen táblából áll, ezt az *1. kódrészlettel* adjuk meg. Az alábbi séma ***MySQL***-hez készült, de természetesen bármilyen adatbázismotorhoz megírható egy ilyen kódrészlet.

A következő függvényekre mindenképpen szükség van a felügyeleti felület által biztosított szolgáltatások megvalósításához: `list_events()`, `show_event()`, `save_event()` és `delete_event()` (rendre események listázása, megjelenítése, mentése és törlése). Az adatbázis olvasását és írását is külön függvénycsoportba fogom elvonatkoztatni, így az egyes függvények egyszerűbbek maradnak, és egyszerűbb a hibakeresés is. Az adattároló felülethez a következő függvények szükségesek: `event_create()`, `event_destroy()`, `event_read()`, `event_write` és `event_delete` (esemény létrehozása, megsemmisítése, olvasása, írása és törlése). Mivel minden lehetőséget megragadok, hogy megkönnyítsem az

2. kódrészlet A beállítások futás közbeni megadását lehetővé tévő függvény

```
void config_read(char* filename, char** key,
                 char** value) {
  FILE* cfile;
  char tok[80];
  char line[2048];
  char* target;
  int i;
  int length;
  cfile = fopen(filename, "r");
  if (!cfile) {
    perror("config_read");
    return;
  }
  while(fgets(line, 2048, cfile)) {
    if ((target = strchr(line, '='))) {
      sscanf(line, "%80s", tok);
      for(i=0; key[i]; i++) {
        if (strcmp(key[i], tok) == 0) {
          target++;
          while(isspace(*target)) target++;
          length = strlen(target);
          value[i] = (char*)calloc(1, length + 1);
          strcpy(value[i], target);
          target = &value[i][length - 1];
          while(isspace(*target)) *target– = 0;
        }
      }
    }
  }
  fclose(cfile);
}
```

életemet, készítek egy `event_fetch_range()` (eseménytartomány lekérdezése) függvényt is, amivel adott tartományba eső eseményeket tudok kiválasztani. Erre legalább két helyen szükség lesz.
Következőként a rekordjaimat *C* adatszerkezeteknek, az adatbázis-lekérdezések eredményeit pedig láncolt listáknak kell megfeleltetnem. Az elvonatkoztatás lehetővé teszi, hogy – mivel a kódnak csak kis része foglalkozik közvetlenül az adattárolóval – viszonylag kis fáradság árán le tudjam cserélni az adatbázismotort, vagy meg tudjam változtatni az adatábrázolás módját.
A teljes forráskódot természetesen nyomtatásban nem jelentethetjük meg, ezért a ***Makefile*** társaságában a weboldalamról lehet letölteni (lásd az internetes forrásokat).

Eszközök

C használatakor az első leküzdendő akadály a szükséges eszközök beszerzése. Minimálisan egy *CGI*-feldolgozóra szükségünk lesz, ez fogja elérhetővé tenni számunkra a *CGI* adatait. Nagy valószínűséggel valamilyen adatbázis-kapcsolat is jól jön. Nem árt, ha a vezérlő logikát és a felületet valamilyen szinten szét tudjuk választani, így ugyanis az oldal átdolgozása nem jár a kód újabb és újabb újraírásával.
CGI feldolgozási célokra ***Thomas Boutell cgic*** könyvtárát ajánlom (lásd a forrásokat). Használata megdöbbentően

3. kódrészlet HTML sablonfüggvény

```
void html_get(char* path, char* file) {
  struct stat sb;
  FILE* html;
  char* buffer;
  char fullpath[1024];
  /* A fájl és az elérési út neve túllép a rendszer
  ➥korlátján */
  if (strlen(path) + strlen(file) > 1024) return;
  sprintf(fullpath, "%s/%s", path, file);
  if (stat(fullpath, &sb)) return;
  buffer = (char*)calloc(1, sb.st_size + 1);
  if (!buffer) return;
  html = fopen(fullpath, "r");
  fread((void*)buffer, 1, sb.st_size, html);
  fclose(html);
  puts(buffer);
  free(buffer);
}
```

egyszerű, és a CGI felület minden részéhez elérést biztosít. Ha inkább *C++* irányultságúak vagyunk, a ***cgicc*** könyvtárakat használhatjuk (lásd a forrásokat). Szerintem a ***Boutell*** könyvtárral könnyebb dolgozni.
A ***MySQL*** a ***UNIX*** alapú webes fejlesztéseknél gyakorlatilag szabványnak tekinthető, ezért ebben az esetben én is nála maradok. Minden komolyabb adatbázismotornak van *C* felületkönyvtára, vagyis mindenki olyan adatbázist választ, amilyet csak szeretne.
Jómagam elkészítem a saját felületfüggetlen eljárásaimat, de aki úgy gondolja, a ***libxml*** és a ***libxslt*** segítségével ugyanezt jóval kifinomultabb szinten oldhatja meg.

Futás közben megadott beállítások

Fontos tényező, hogy az adatbázis-kapcsolat beállításait futás közben is meg lehessen adni. A beállító függvény egy fájlnév és egy a beállító kulcsokat megadó karakterlánctömb alapján feltölt egy tömböt a megfelelő beállítási értékekkel. (Lásd a *2. kódrészletet*.) Ezután tetszőleges, az általam kiválasztott kulcsokkal tölthetek fel egy megfelelő karakterlánctömböt, az eredményt az értéktömbben kapom meg.

A felhasználói felület

A felhasználói felület két részből áll. Én, mint programozó, elsősorban a beviteli űrlapokkal és az ***URL***-karakterláncokkal foglalkozom. Mindenki más inkább arra figyel, hogy az űrlapomat körülvevő oldal hogyan néz ki, magát az űrlapot adottnak veszi. Ha mindkét fél elégedettségét biztosítani akarjuk, akkor külön kell választanunk az oldalt az űrlaptól és a programtól.
PHP és ***Perl*** alá számtalan sablonkönyvtár létezik, ám *C* alá nem nagyon találunk ***HTML*** sablonokat. A legjobb az, ha a *C* kód a kimenetnek csak a lehető legkisebb részét foglalja magába, a maradékot pedig ***HTML*** formátumú fájlokba helyezzük el, melyeket mindig a megfelelő időpontban jelenítünk meg. Erre alkalmas függvényt látunk a *3. kódrészletben*.
A kimenet kiadása előtt közölnünk kell a webkiszolgálóval és a böngészővel, hogy mit is szeretnénk átadni; a `cgiHeaderContentType()` pontosan ezt a célt szolgálja.

4. kódrészlet save_event(), CGI adatok feldolgozása

```
struct event* e;
e = event_create();
cgiFormInteger("eventno", &e->event_no, 0);
cgiFormStringNoNewlines("name", e->name, 80);
cgiFormStringNoNewlines("location",
                        e->location, 80);
/* Processing date fields */
cgiFormInteger("beginyear",
               &e->event_begin->year, 0);
cgiFormInteger("beginmonth",
               &e->event_begin->month, 0);
cgiFormInteger("beginday", &e->event_begin->day,
➥0);
cgiFormInteger("endyear", &e->event_end->year, 0);
cgiFormInteger("endmonth", &e->event_end->month,
➥0);
cgiFormInteger("endday", &e->event_end->day, 0);
/* Process begin & end times separately */
cgiFormStringNoNewlines("beginhour",
                        e->event_begin->hour, 10);
cgiFormStringNoNewlines("endhour",
                        e->event_end->hour, 10);
event_write(e);
cgiHeaderLocation(cgiScriptName);
```

A tartalom típusa (`content type`) `text/html`, ezért ezt használom átadott értékként. Minden oldal megjelenítése előtt az alábbi általános lépéseket kell követnünk:

```
cgiHeaderContentType("text/html");
html_get(path, pagetop.html);
A tartalom előállítása.
html_get(path, pagebottom.html);
```

Az űrlapok feldolgozása

Megoldottuk az oldalak és az űrlapok megjelenítését, most tehát gondoskodnunk kell az űrlap tartalmának feldolgozásáról. Numerikus és szöveges elemek beolvasására egyaránt szükség lehet, ezért a ***cgic*** könyvtár két függvényét veszem igénybe, ezek a `cgiFormStringNoNewlines()` és a `cgiFormInteger()`.
A ***cgic*** könyvtár tartalmazza a fő függvény megvalósítását, rám csak az int `cgiMain(void)` – túlnyomó részt ebben történik az űrlap feldolgozása – megírásának feladata hárul.
Ha a `show_event` függvénnyel egyetlen rekordot akarok megjeleníteni, akkor a ***CGI*** `eventno` átadott érték `event_no` (ez az elsődleges kulcs) értékét veszem.
A `cgiFormInteger()` függvény egy egész értéket kérdez le, és ha nincs megadva ***CGI*** átadott érték, akkor alapértelmezettet állít be.
A `save_event` szintén jó néhány adatot igényel az űrlapból. A dátumok bevitele fogas kérdés, ugyanis három adatrészből állnak: évből, hónapból és napból. Kezdő és záró dátumra egyaránt szükség van, vagyis összesen hat mező tartalmát kell feldolgozni. Szükség van még az esemény nevére, kezdő és záró időpontjára (ezek karakterláncok, ugyanis önmagukban is lehetnek események, mint például napkelte és napnyugta) és helyére. A *4. kódrészlet* mindennek a megvalósítását szemlélteti.

5. kódrészlet A küldés gombok kezelése

```
char* command[5] = {"List", "Show",
                    "Save", "Delete", 0};
void (*action)(void)[5] = {list_events,
  show_event, save_event, delete_event, 0};
int result;
cgiFormSelectSingle("do", command, 4, &result, 0);
action[result]();
```

A 4. kódrészletben a `cgiHeaderLocation()` is szerepel, ennek feladata a felhasználó új oldalra irányítása. Miután elmentettem az elküldött adatokat, mindig az eseménylista oldalt szeretném megjeleníteni. Meghatározott karakterlánc helyett a ***libcgic*** által biztosított változók egyikét, a `cgiScriptName`-et használom, ezzel azt érem el, hogy a program neve működésének megzavarása nélkül megváltoztatható.
Végül kell valamilyen megoldás az adatok elküldésére szolgáló (***submit***) gombok kezelésére. Ezek a legösszetettebb beviteli megoldások, az elindítandó függvény ugyanis esetükben az értéküktől függ, illetve szükség esetén alapértelmezett értéket is kell választani. A ***cgic*** könyvtárnak van egy függvénye, a `cgiFormSelectSingle()`, amely pontosan ezt végzi el. Működéséhez a lehetséges értékeknek egy karakterlánctömbben kell szerepelniük. Egy egész értékbe helyezi a megfelelő átadott érték tömbbeli indexét; ha pedig nem talál egyezést, akkor az alapértéket használja.
A függvénymutatókról bővebb tájékoztatás a források között található. Aki nem békült még meg a függvénymutatókkal, az `switch` utasítással is megoldhatja mindezt. Jómagam inkább a függvénymutatókat szeretem, ez a megoldás ugyanis tömörebb; régebbi kódjaimban ugyanakkor természetesen a `switch` utasítást is használtam.

Az adatbázisrendszer

A ***MySQL*** kezelése ***C*** alól nagyon hasonló a ***PHP*** alóli kezeléshez – már amennyiben ez mond valamit. A karakterláncokban szereplő problémás karakterek, például a kettős idézőjelek és a visszaperjelek eltávolítására a ***MySQL*** megfelelő függvényeit kell használnunk, de más eltérést nem nagyon fogunk találni. A show_event() függvény esetében az elsődleges kulcs alapján kell lekérdeznünk egy rekordot. A hibakezelés miatt a kód meghízik ugyan, de három alaputasítás adja a lényeget. A `mysql_query()` hívásakor lefut a ***MySQL*** utasítás, aminek keletkezik valamilyen eredménye. Ezután a `mysql_store_result()` lekérdezi az eredménykészletet a kiszolgálótól. Végül a `mysql_fetch_row()` egyetlen `MYSQL_ROW` változót emel ki az eredménykészletből.
A `MYSQL_ROW` változót karakterláncok tömbjeként (`char**`) kezelhetjük. Ha az adatok valamelyike numerikus, és numerikusként is szeretnénk használni, akkor át kell alakítanunk. Jelenlegi programunkban például a dátumot három különálló számérték formájában érdemes kezelni. Mivel a kapott adat ***ÉÉÉÉ-HH-NN*** szerkezetű, a `sscanf()` függvénnyel választhatjuk szét az összetevőit. (*6. kódrészlet*)
Az adatok adatbázisba írása érdekesebb, itt ugyanis ügyelni kell a problémás karakterek kiszűrésére.
Ezt a *7. kódrészlet* szemlélteti.

6. kódrészlet Adatok lekérdezése MySQL-ből

```
MYSQL_RES* res;
MYSQL_ROW row;
int beginyear;
int beginmonth;
int beginday;
if (mysql_query(db, sql)) {
  print_error(mysql_error(db));
  return;
}
if((res = mysql_store_result(db)) == 0) {
  print_error(mysql_error(db));
  return;
}
if ((row = mysql_fetch_row(res)) == 0) {
  print_error("Nincs ilyen számú esemény.");
  return;
}
sscanf(row[0], "%d-%d-%d", &beginyear, &beginmonth,
       &beginday);
```

Az `escapedname` ugyanazt a karakterláncot tartalmazza, mint a `name`, de a *MySQL* különleges karakterei nélkül; így már nyugodtan beilleszthetjük egy *SQL*-utasításba. Fontos, hogy a felhasználók által megadott karakterláncokat mindig tisztítsuk meg, ellenkező esetben rosszindulatú személyek kihasználhatják hanyagságunkat, és kellemetlen dolgokat művelhetnek az adatbázisunkkal.

CGI programok hibakeresése

A *C* programok hibáinak felderítése elég kellemetlen tevékenység, ugyanis általában szegmentációs hibát okoznak, és semmilyen jelzést nem kapunk a hiba forrásáról. A hibakereső programok más alkalmazások esetében kiválóan megfelelnek, ám a *CGI* programoknál a bemenő adatok fogadásának módja miatt más a helyzet.
Ilyenkor siet segítségünkre a *cgic* könyvtárban található `capture` nevű *CGI* program, mely a neki küldött *CGI*-bemeneteket egy fájlba rögzíti. A fájl nevét a `capture` forráskódjában adhatjuk meg. Ha *CGI* programunkon hibakeresést kell végeznünk, akkor `cgiMain()` függvényének elejére adjunk hozzá egy `cgiReadEnvironment(char*)` hívást. Ügyeljünk arra, hogy a `filename` átadott érték megegyezzen a rögzítéshez megadottal. Ezután a problémás adatot a kérésünkben szereplő űrlap vagy parancsfájl alapértelmezett műveletévé téve küldhetjük el rögzítésre. Végül a *GDB*-vel vagy kedvenc hibakeresőnkkel bogozhatjuk ki, hogy kódunk pontosan milyen hibát okozott.
A későbbi hibakeresést és fejlesztést bizonyos lépésekkel leegyszerűsíthetjük. Bár ezek minden program fejlesztésére érvényesek, *CGI* programozásnál különösen kifizetődők. Ne feledjük, egy-egy függvénynek egyetlen, és csakis egyetlen feladatot adjunk, továbbá tesztelését minél hamarabb kezdjük el, és a lehető leggyakrabban ismételjük meg.
A kód írása közben érdemes minden egyes függvényt azonnal kipróbálni, és ellenőrizni, hogy valóban a tőle elvárt módon működik-e. Azt is érdemes megvizsgálni, hogy hibás adatok megadására hogyan válaszol, ugyanis nagyon valószínű, hogy élete során a függvény előbb-utóbb valóban fog hibás adatokat kapni. Ha minderre jó előre gondolunk, akkor sok kellemetlenséget spórolhatunk meg magunknak.

7. kódrészlet A felhasználó által megadott adatok kezelése MySQL-lel

```
char name[11];
char escapedname[21];
cgiFormStringNoNewlines("name", name, 10);
mysql_real_escape_string(db, escapedname, name,
                         strlen(name));
```

Telepítés

A fejlesztésre és a futtatásra használt számítógép az esetek nagy részében nem ugyanaz. A fejlesztői rendszert ezért próbáljuk a lehető legnagyobb mértékben azonossá tenni a termelési rendszerrel. Én például a programjaimat általában *Linux* vagy *OpenBSD* alatt fejlesztem, futtatásuk viszont szinte mindig *FreeBSD* alatt történik.
Amikor felkészülünk a termelési rendszer felépítésére vagy telepítésére, akkor különösen fontos, hogy tisztában legyünk a könyvtárváltozatok közötti különbségekkel. Azt, hogy kódunk mely dinamikus könyvtárakat használja, az `ldd` paranccsal tekinthetjük meg. Érdemes ellenőrizni, mert meglepetéseket okozhatnak a könyvtárak miatt felmerülő további függőségek.
Ha a könyvtárváltozatok közel állnak egymáshoz, ami általában a fő változatszámok megegyezésében nyilvánul meg, akkor különösebb gondokra nem kell számítanunk. A külső üzemeltetésű weboldalra telepített programoknál gyakori a fejlesztői és a termelési rendszer által tartalmazott változatok eltérése.
Ilyenkor én saját, helyi változatot szoktam fordítani a könyvtárból. Eltávolítom a könyvtár megosztott változatát, és a rendszer által tartalmazott helyett a helyi változatot építem be. Természetesen ilyenkor a futtatható fájl megnő, ám a fennhatóságunkon kívüli könyvtáraktól való függése is megszűnik.
Miután lefordítottuk a futtatható fájlt a termelési rendszeren, az `ldd` ismételt kiadásával ellenőrizzük, hogy az összes dinamikus könyvtár megtalálható-e. A könyvtárak helyi változatainak beépítésekor különösen gyakran fordul elő, hogy elfeledkezünk a dinamikus változat eltávolításáról, amit persze futás közben nem fog találni a program (és az `ldd`). A fordítást sose kapkodjuk el, újra és újra fordítsunk és ellenőrizzünk, míg végül el nem tűnnek a könyvtárak megtalálásával kapcsolatos hibák.

Sebesség: CGI kontra PHP

Általános nézet, hogy a *CGI* felületet használó programok lassabbak, mint a valamilyen kiszolgáló modul – mint a *mod_php* és a *mod_perl* – által biztosított nyelvet használók. Mivel a webes alkalmazások fejlesztését *PHP*-ben kezdtem, ezt fogom a *C*-ben írt *CGI*-k sebességének vizsgálatakor kiindulási alapként venni. Nyilván mindebből a *C* és a *Perl* közötti sebességkülönbségekre nézve nem következik semmi.
Az összehasonlításnál a külső adatbázis-felületet vettem (`events.cgi` és `events.php`), ugyanis mindkettő azonos

eljárást alkalmaz a felület elkülönítésére. A belső felületet nem teszteltem, ugyanis a külső hívások mellett a belsők szerepe szinte elenyésző.

Az ***Apache Benchmark*** segítségével mindkét változatot tízezer lekérdezéssel terheltem le, amilyen gyorsan a kiszolgáló képes volt azokat kezelni. A ***C*** alapú változat átlagos végrehajtási ideje 581 ms, a ***PHP*** alapú változaté pedig 601 ms volt. Mivel a két érték közel esett egymáshoz, feltételeztem, hogy a méréseket megismételve némi szórást észlelek majd. Így is volt, de végeredményként elmondhatom, hogy a ***C*** alapú változat az esetek nagyobb hányadában volt gyorsabb a ***PHP*** alapúnál.

Munkáim során egy összetettebb felületelválasztó könyvtárat használok, ez a libtemplate (lásd a forrásokat). A könyvtárból ***PHP*** és ***C*** változat egyaránt létezik. Amikor az eseményütemező változatait a ***libtemplate*** segítségével hasonlítottam össze, akkor a ***C*** alapú változat határozottan jobb válaszidőkkel futott. A ***C*** alapú változat átlagos futási ideje 625 ms volt; nem sokkal több, mint az egyszerűbb változaté.

A ***PHP*** alapú változat átlagos futási idejére 1957 ms-ot kaptam. Azt is érdemes megjegyezni, hogy a terhelés a ***PHP*** alapú változat futása közben a ***C*** alapú változat futásakor mértnek nagyjából kétszerese volt. A tesztek végrehajtása közben más komolyabb alkalmazás nem futott a gépen, és más felhasználók sem terhelték.

A két ***C*** alapú változat egymáshoz közeli futtatási ideje arra utal, hogy az idő nagy része a program betöltésével telik. Ha ez megtörtént, a futtatás már gyorsan megy. A ***PHP*** ezzel szemben viszonylag lassan fut. Természetesen a ***PHP*** esetében is be kell tölteni a programot a memóriába, ám itt a fordítást – amelyen a ***C*** alapú program már túlesett – is el kell végezni.

Összefoglalás

Megfelelő eszközökkel és némi tapasztalattal a ***C*** alapú ***CGI*** alkalmazások fejlesztése nem nehezebb, mint a ***Perl*** vagy ***PHP*** alapúaké. Jómagam mindkettővel rendelkezem, és egyértelműen a ***C***-t választottam a ***CGI***-k fejlesztésére.

A ***C*** előnyei főként bonyolultabb műveletek elvégzésekor és a hosszú távú stabilitás terén mutatkoznak meg. A ***PHP***-val szemben teljesen érzéketlen a kiszolgálót érintő változásokra. Hacsak nem töröljük a gépről valamelyik megosztott könyvtárat, például a ***libc***-t vagy ***libmysqlclientet***, a ***C*** alapú változatot rendkívül nehéz tönkretenni. A ***C*** alapú programok futásuk gyorsasága miatt összetettebb adatfeldolgozó műveletek végrehajtásakor egyértelműen jobb választásnak bizonyulnak.

Linux Journal 2005. április, 132. szám

A cikk forrásai: ➲ ***www.linuxjournal.com/article/8058***

Clay Dowling a Lazarus Internet Development (www.lazarusid.com) elnöke. A programozás mellett hobbija a sörfőzés és a borkészítés. A clay@lazarusid.com címen érhető el.