3. A GNU Compiler Collecion (gcc)

A GNU Compiler Collection fordítóprogramok gyűjteménye, amelyet C, C++, Objective-C, Java, ADA és Fortran kódok fordítására egyaránt használhatunk. A GCC egy nyílt forráskódú fejlesztés és szigorúan követi az egyes nyelvek szabványait, így ezek használata javasolt például az Microsoft MSVC C/C++ fordítójával szemben. Az egyes Linux disztribúcióknak szerves része a GCC, így nincs szükség különösebb előkészületekre az eszközök használatához. Amennyiben a gcc bináris állomány, tehát maga a fordító program nem lenne megtalálható, a disztribúció csomagkezelő eszközével telepíthetjük. Debian alapú disztribúciók esetén (pl. Debian, Ubuntu, Kubuntu, Edubuntu, stb.) az általános csomagkezelő az **apt−get** alkalmazás, parancssorból a következő módon telepíthetjük a GNU eszközöket, **root** jogosultságú felhasználói fiókból:

```
apt-get install gcc
```

Windows rendszerek esetén összetettebb a feladat. Két lehetőségünk van: a **cygwin** vagy a **MinGW** használata.

A cygwin egy Windows rendszerekre fejlesztett Linux shell. Installálás után elérhetővé teszi a legtöbb parancssoros eszközt, ami az általános Linux shell-ekben elérhető. A cygwin a Linux kernel funkcionalitásait a **cygwin1.dll** nevű dinamikusan linkelhető könyvtáron keresztül teszi elérhetővé, tehát akármilyen szoftvert fordítunk cygwin segítségével, az csak abban az esetben fog más telepített Windows rendszereken működni, ha azon is elérhető a cygwin. Ezzel szemben a MinGW, azaz Minimal GNU for Windows egy olyan környezetet biztosít, amellyel hasonlóan a cygwin-hez a GNU eszközöket

használhatjuk, azonban a minden Windows rendszeren megtalálható **msvcrt.dll** könyvtárra épít, így bármi, amit lefordítunk MinGW-vel Windows-on, az működni fog más Windows rendszereken is.

A MinGW szintén egy nyílt forráskódú fejlesztés, azonban a bináris csomagok disztribútolását a honlapjukon abbahagyták, azaz a MinGW oldalán elérhető disztribúció közel 10 éves, így abban még nem megfelelő verziójú gcc fordító szerepel. A gcc aktuális verziója 4.5. Az aktuális Ubuntu repository-ban elérhető verzió 4.4-es. Azok az eszközök, amelyeket a félév során használni fogunk a 4.2-es verziótól képezik részét a fordítónak, a MinGW honlapjáról letölthető disztribúció 3.5-ös verziójú gcc-t tartalmaz. A megoldás a Qt csomag használata Windows-on. A Qt egy ablakozó alkalmazások készítését lehetővé tevő könyvtár, amely fejlesztését a Trolltech kezdte, jelenleg a Trolltech és a Nokia közösen folytatja. A Qt SDK-ban napra készen tartják a MinGW-t, így Windows-on a MinGW-t a legegyszerűbben a Qt SDK-val együtt telepíthetjük fel. A Qt SDK a következő honlapon tölthető le: `http://qt.nokia.com/downloads/`. Telepítés után a Qt gyökérkönyvtárban találhatjuk a **mingw\bin** mappát, amely tartalmazza a **mingw32–gcc.exe**, **mingw32–g++.exe** valamint **mingw32–make.exe** nevű alkalmazásokat, amelyekkel Windows környezetben használhatjuk a GCC eszközeit.

A félév során erősen javasolt valamely Linux disztribúció használata! A példákat mind Linux rendszeren visszük végig, a Windows-os megoldások csak utalásként jelennek meg.

4. A C/C++ programozás során használt fájl típusok

4.1. Szöveges fájlok

A **.h** kiterjesztésű header fájlok ismertek. C esetén a forráskódok kiterjesztése konvencionálisan **.c**, C++ forráskódok esetén **.cpp** vagy **.cc**. Ezek természetesen csak ajánlások, de sok automatizáló eszköz erre a konvencióra épít, ezért érdemes ezt használni.

4.2. Bináris fájlok

Bináris fájlokról akkor beszélünk, ha azok gépi kódot tartalmaznak. Ezek olyan fájlok, amelyeket már lefordítottunk, vagy bináris formában kaptuk őket, töltöttük le őket, és a linkelés során kapcsolhatjuk a saját kódjainkhoz a bennük implementált funkciókat.

Egy forrásfájl lefordítása során (kivéve azt a speciális esetet, amikor a projektünk egyetlen forrásfájlt tartalmaz, amiben megtalálható a **main** nevű főprogram is), tárgykód áll elő, amely kiterjesztése **.o**. Sok forrásfájlból álló, tehát "igazi" szoftverek esetén felmerül az a probléma, hogy a rengeteg tárgykód kezelhetetlenné és átláthatatlanná teszi a szoftvert. Ha a funkcióinkat több alkalmazáshoz is linkelni szeretnénk, a linkelési parancsban meg kell adnunk az összes tárgykódot tartalmazó fájl nevét, ami nagyon nagyban megnöveli a hibalehetőséget. Ezenfelül, Windows-ban nagyon könnyen átléphetjük a maximális parancs hosszt, és előállhat az a helyzet, hogy a linkelési parancs túl hosszú, és így nem hajtható végre a linkelés.

A probléma megoldására a tárgykódokat könyvtárakba szervezhetjük. A könyvtáraknak két típusa létezik, a statikusan linkelhető és a dinamikusan linkelhető könyvtárak. A statikusan linkelhető könyvtárak nevéből adódóan a bennük tárolt tárgykódra fordított eszközök beépülnek a target objektumba, azaz az alkalmazásba vagy egy újabb statikus vagy dinamikus könyvtárba. A dinamikus könyvtárakat felhasználja a linker, hogy ellenőrizze, minden hivatkozott eszköz létezik-e, de nem építi be a cél objektumba, azaz nem növeli az alkalmazás vagy újabb könyvtár méretét. Alkalmazás futtatása esetén a dinamikus könyvtáraknak elérhetőeknek kell lenniük a futtató rendszer számára, hogy az alkalmazás által hivatkozott eszközöket a futtató rendszer betölthesse az alkalmazás számára.

A statikus könyvtárak kiterjesztése Windows rendszereken **.lib**, a dinamikus könyvtáraké **.dll**. Linux rendszereken a statikus könyvtárak **.a**-ra végződnek, ami az 'archive' szóra utal, míg a dinamikus könyvtárak **.so** végződésűek, ami a 'shared object' kiterjesztése. A dinamikus könyvtárak egyik fő előnye ugyanis, hogy különböző alkalmazások úgy használhatják ugyanazokat az eszközöket, hogy ugyanazt a shared object fájlt linkelik, így csökkentik a háttértár és memória szükségleteket, valamint ha a shared object-ben módosítunk egy eszközön, nem kell a többi alkalmazást és könyvtárat újrafordítani.

4.3. Csomag disztribúciók

A C/C++ terminológiában a könyvtár fogalma leginkább a Java és C# nyelvek csomag fogalmával azonosítható, tehát a standard nyelvi eszközök használatán túl C/C++ könyvtárak használatával érhetjük el, hogy olyan funkciókat építsünk programjainkba, amelyeket mások írtak. A könyvtár kifejezés azonban magukon a statikus és dinamikus könyvtárakon túl sokszor a header fájlok halmazára is utal. Nagyon fontos megjegyezni, hogy statikus és dinamikus könyvtárakat nem tudunk használni, ha nem rendelkezünk a header fájlokkal, amelyek specifikálják azokat a programozási eszközöket, amelyek ezekben a könyvtárakban lefordítva megtalálhatóak.

A C/C++ könyvtárak két formában érhetőek el, bináris csomagként, és fejlesztői csomagként. A bináris csomagok nevükből kifolyólag tartalmazzák a lefordított forráskódokat, azaz vagy statikus, vagy dinamikus könyvtárakat, esetleg mindkettőt, de emellett tartalmazzák azokat a header-öket is, amelyek specifikálják a lefordított eszközöket. A bináris csomagokban forráskódot, azaz .c, .cc, .cpp fájlokat nem találunk. A fejlesztői csomagok, amelyeket általában **src** szócskával minősítenek, nem tartalmaznak lefordított kódokat, csak header fájlokat, forrásfájlokat és egy olyan script fájlt, amely lehetővé teszi a könyvtár lefordítását, anélkül, hogy tudnánk, a források fordításához milyen egyéb header-ökre, vagy a linkeléshez milyen további könyvtárakra van szükség.

Példaként megnézhetjük a **libpng** könyvtárat, amely a **.png** képek kezeléséhez szükséges függvényeket tartalmazza: `http://www.libpng.org/pub/png/libpng.html`. Az oldalon a 'Source code' bejegyzésnél találjuk a fejlesztői csomagot, amely forráskódot tartalmaz, míg 'Current binaries' bejegyzésnél tölthetjük le a bináris csomagokat, amelyek csak header fájlokat és lefordított statikus és/-vagy dinamikus könyvtárakat tartalmaznak. Mivel a fordítás műveletének célja, hogy a forráskódot olyan gépi kódra fordítsuk, amely az adott hardver utasításkészletének legmegfelelőbb, így a leggyorsabb futást biztosítja, a lefordított kódok a különböző architektúrák között ritkán hordozhatók. A bináris csomagok ebből kifolyólag architektúra függők, azaz a legtöbb fejlesztőcsapatnak a bináris csomagokat az általánosan elérhető architektúrákra külön-külön el kell készítenie.

5. Első statikus és dinamikus könyvtáraink

Az alábbiakban egy nagyon egyszerű példát járunk körbe a statikus és dinamikus könyvtárak használatának bemutatására. Létrehozunk egy statikus és egy dinamikus könyvtárat, header fájlokkal. Mindkét könyvtár egy-egy nagyon egyszerű **printf** függvényt fog tartalmazni, és ezt a két függvényt hívjuk meg aztán három különböző főprogramból. Az első két esetben csak a statikus, majd dinamikus könyvtárban lévő függvényt, a harmadik esetben mindkettőt, tehát statikusan és dinamikusan is linkelünk egyszerre.

5.1. Forrásfájlok

5.1.1. forráskód: testStaticLibrary.h

```
#include <stdio.h>

int printInStaticLibrary();
```

5.1.2. forráskód: testStaticLibrary.c

```
#include <testStaticLibrary.h>

int printInStaticLibrary()
{
   return printf("this is in the static library\n");
}
```

5.1.3. forráskód: testDynamicLibrary.h

```
#include <stdio.h>

int printInDynamicLibrary();
```

5.1.4. forráskód: testDynamicLibrary.c

```
#include <testDynamicLibrary.h>

int printInDynamicLibrary()
{
   return printf("this is in the dynamic library\n");
}
```

5.1.5. forráskód: mainStatic.c

```
#include <testStaticLibrary.h>

int main(int argc, char** argv)
{
   printInStaticLibrary();
   return 0;
}
```

5.1.6. forráskód: mainDynamic.c

```
#include <testDynamicLibrary.h>

int main(int argc, char** argv)
{
   printInDynamicLibrary();
   return 0;
}
```

5.1.7. forráskód: mainStaticAndDynamic.c

```
#include <testStaticLibrary.h>
#include <testDynamicLibrary.h>

int main(int argc, char** argv)
{
   printInDynamicLibrary();
   printInStaticLibrary();
   return 0;
}
```

Öt lefordítandó forrásfájlunk van, a **testStaticLibrary.c** és **testDynamicLibrary.c** források tartalmát statikus és dinamikus könyvtárba kell fordítanunk, és ezt felhasználva a **mainStatic.c**-ből statikusan linkelt, a **mainDynamic.c**-ből dinamikusan linkelt, míg a **mainDynamicAndStatic.c** fájlból statikusan és dinamikusan egyszerre linkelt alkalmazást szeretnénk létrehozni.

5.2. A könyvtárak létrehozása

A statikusan linkelendő könyvtár forrását a következő paranccsal fordíthatjuk le:

```
gcc -I. -c testStaticLibrary.c -o testStaticLibrary.o
```

A **-I** kapcsolókkal azokat a könyvtárakat adhatjuk meg a fordítónak, amelyekben a header fájlokat keresnie kell. Esetünkben ez a könyvtár az aktuális könyvtár, amelyre **.**-tal hivatkozunk. A **-I** használatakor a header-öket tartalmazó könyvtár nevét egybe kell írnunk az **-I** kapcsolóval. Például, ha meg szeretnénk adni a **/home/gykovacs** könyvtárat is, akkor a **-I/home/gykovacs** kapcsolót is be kellene raknunk a fordítási parancsba. A **-c** kapcsolóval azt jelezzük a fordítónak, hogy tárgykódot hozzon létre, ne próbáljon linkelni, míg a **-o** kapcsolót követő paraméterrel azt adhatjuk meg, hogy mi legyen az eredményként előálló, lefordult object fájl neve. Miután lefordítottuk a forrásfájlokat, amelyeket statikus könyvtárba szeretnénk szervezni, az **ar** eszközt használhatjuk fel az archívum, azaz a statikus könyvtár létrehozására:

```
ar rcs libtestStaticLibrary.a testStaticLibrary.o
```

Az ar program számára az **r**, **c** és **s** kapcsoló három különböző opciót adnak meg:

- r: ha egy fájl szerepel már az archívumban, akkor cserélje (replace)
- c: archívum létrehozásról van szó (create)
- s: hozzon létre object fájl indexet

Linux rendszereken a statikus és dinamikus könyvtárak egyaránt a **lib** szócskával kezdődnek! Az **ar** program a jól ismert **tar** ősének tekinthető, a paraméterezése, és funkciója is közel azonos: egyszerűen egymás után fűzi a paraméterként kapott fájlokat.

A dinamikus könyvtár létrehozása szintén a forráskód lefordításával kezdődik:

```
gcc -I. -c -fPIC testDynamicLibrary.c -o testDynamicLibrary.o
```

A **-c** kapcsoló hasonlóan a korábbi esethez az object fájl létrehozását jelenti, míg a **-o** kapcsolót követő paraméter a kimenet nevét specifikálja. A **-I.** kapcsoló a header fájlok helyét adja meg, hasonlóan a korábbi esethez. A statikus könyvtárhoz fordított kóddal összehasonlítva a különbség a **-fPIC** kapcsoló, amely nevében az f a flag-re utal, tehát bináris kapcsolóról van szó, a PIC pedig magának a funkciónak a rövidítése, amelyet ezzel a flag-gel bekapcsolunk: Position Independent Code, azaz pozíció független kód. Ez az opció arra utal, hogy a kód nem fog függeni attól, hogy hol helyezkedik el a memóriában. Erre azért van szükség, mert míg a statikus könyvtárakban található kódok beépülnek a későbbi alkalmazásokba, így azokkal szerves egységet alkotva a memóriában

hivatkozási pontként használhatóak, a dinamikus könyvtárak az alkalmazásoktól teljesen független memóriaterületen helyezkedhetnek el, így a memória címeknek függetlennek kell lenni benne az alkalmazás más részeitől.

A fordítást követően a dinamikus könyvtárat a következő paranccsal állíthatjuk elő:

```
gcc -shared -Wl,-soname,libtestDynamicLibrary.so.1 -o libtestDynamicLibrary.so.1.0.1
testDynamicLibrary.o
```

A –**shared** kapcsoló specifikálja, hogy dinamikus könyvtárat kell létrehozni a paraméterként kapott object fájlokból. A –**Wl** kapcsoló lehetővé teszi, hogy a háttérben működő linkernek (ld), paramétereket adjunk át, a következő szintaxissal: a –**Wl,–soname,libtestDynamicLibrary.so.1** sztring a vesszőknél kerül darabolásra, és külön opciónként lesz továbbítva a linkerhez, azaz a linker a –**soname libtestDynamicLibrary .so.1** opciókat kapja meg, ahol a második tag a –**soname** kapcsolót követő paraméter. A –**soname** kapcsoló utáni paraméter specifikálja azt a nevet, amelyen a könyvtárat keresnie kell majd a betöltő rendszernek. Ez a név nem azt a nevet jelenti, amelyen létrejön a könyvtár, hanem egy olyan prefixet definiál, amely a különböző verziójú, de azonos interfésszel rendelkező könyvtárakat összefogja. Amikor egy alkalmazást linkelünk egy dinamikus könyvtárhoz, a –**soname** utáni név fog beépülni az alkalmazásba, mint referencia, és a betöltő rendszer az alkalmazás indításakor ilyen nevű fájlt keres majd. A –**o** kapcsolót követő paraméterrel specifikálhatjuk a létrejövő könyvtár nevét, és ezt követően, a parancs végén felsoroljuk az összefogni kívánt object fájlokat.

Ellenőrizzük le, hogy valóban létrejöttek-e a libtestStaticLibrary.a, valamint a libtestDynamicLibrary.so.1.0.1 statikus és dinamikus könyvtárak.

5.3. Az alkalmazások fordítása és linkelése

A gcc eszközök lehetőséget nyújtanak arra, hogy alkalmazások tehát főprogramokat tartalmazó források fordításánál a fordítást és a linkelést egy lépésben végezzük. Ha visszalapozunk a könyvtárak forrásainak fordításához, láthatjuk, hogy a –**c** kapcsoló célja az volt, hogy ezt a lehetőséget kikapcsoljuk, mivel a könyvtárakhoz tartozó forrásokból csak object kódokat hozhatunk létre.

A statikus könyvtárat használó alkalmazás fordítása és linkelése a statikus könyvtárhoz a következő paranccsal történhet:

```
gcc -static mainStatic.c -I. -L. -ltestStaticLibrary -o mainStatic
```

A –**static** kapcsoló specifikálja, hogy statikus linkelést hajtunk végre. A –**I.** hasonlóan a korábbi esetekhez a header fájlokat tartalmazó könyvtár nevét adja meg. A –**L.** a -I-hez hasonlóan működik, de a linkelendő könyvtárakat tartalmazó mappa elérési útját adja meg, ami esetünkben az aktuális mappa. A –**l** kapcsolót hasonlóan használjuk a -I és -L kapcsolókhoz, azaz a paraméterét egybe írjuk vele, a paramétere pedig a linkelendő könyvtár neve, elhagyva a 'lib' szócskát, amellyel minden könyvtár neve kezdődik. Azt követően a -o kapcsolóvan hasonlóan a korábbiakhoz a kimenet nevét adjuk meg. Ha mindent jól csináltunk, előállt a futtatható mainStatic állomány, amelyet futtatva a "this is in the static library" írja a konzolra.

A dinamikus könyvtárak linkelése hasonlóan történik, azonban nincs szükség a –**static** kapcsolóra, mivel ez az alapértelmezett linkelési mód:

```
gcc mainDynamic.c -I. -L. -ltestDynamicLibrary -o mainDynamic
```

A fenti parancs helyes, mégis linkelési hibát kapunk:

```
/usr/bin/ld: cannot find -ltestDynamicLibrary
collect2: ld returned 1 exit status
```

A hiba oka, hogy a dinamikus könyvtárak neve **.so**-ra kell végződjön. A **–ltestDynamicLibrary** kapcsolóval azt specifikáltuk, hogy a dinamikus könyvtár, amelyet linkelni szeretnénk, a libtestDynamicLibrary.so néven létezik, a könyvtárat azonban libtestDynamicLibrary.so.1.0.1 néven hoztuk létre. Shared object könyvtárak esetén a minősítés elemi szükséglet, mivel a legtöbb esetben több verzió is előfordul belőlük egy Linux operációs rendszeren. A problémát linkeléssel oldhatjuk meg, ahogy az a Linux rendszer lelkét képező /usr könyvtárban is történik:

```
ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so
```

A fenti linkelési paranccsal, ahol az **–f** kapcsoló a force-ra utal, azaz ha létezik a link, írjuk felül, míg a **–s** kapcsolóval a szimbolikus link létrehozását kérjük, létrejön a libtestDynamicLibrary.so, ami a libtestDynamicLibrary.so.1.0.1 fájlra mutató szimbolikus link. Mostmár kiadhatjuk a linkelési mainDynamic.c fordítási parancsát. Ha azonban futtatni próbáljuk a mainDynamic futtatható állományt, ismét hibát kapunk:

```
./mainDynamic: error while loading shared libraries: libtestDynamicLibrary.so.1:
cannot open shared object file: No such file or directory
```

A hibaüzenet azt állítja, hogy nem tudja megnyitni a libtestDynamicLibrary.so.1 állományt. Miért is akarja megnyitni? Mivel dinamikus könyvtárról van szó, a benne lévő funkciók, jelen esetben a printInDynamicLibrary() függvény nem fordulnak bele a mainDynamic futtatható állományba, csak egy referencia jelenik meg az mainDynamic-ban, miszerint a printInDynamicLibrary egy libtestDynamicLibrary.so.1 nevű shared object fájlban van, onnan kell betölteni. Kettő probléma van: az első, hogy ilyen fájl egyelőre nincs, a második, hogy ha lenne is, a betöltő rendszer nem tudja, hogy hol keresse. Az első problémát ismét linkeléssel oldhatjuk meg:

```
ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so.1
```

Így már van megfelelő nevű fájlunk, azt azonban nem adtuk meg a rendszernek, hogy ezt a fájlt hol keresse. Ezen információ megadására szolgál az **LD_LIBRARY_PATH** nevű környezeti változó, amelyhez hozzá kell adni azt a könyvtárat, amelyben a betölteni kívánt dinamikus könyvtárakat az betöltő rendszer keresheti:

```
export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:a_konyvtar_eleresi_utja
```

Ha sikeresen beállítottuk a környezeti változó értékét, a mainDynamic is lefut. Windows rendszerekben az LD_LIBRARY_PATH-hoz hasonló szerepet tölt be a PATH változó: a Windows a dll-eket a PATH-ban specifikált könyvtárakban keresi.

A mainStaticAndDynamic alkalmazást statikusan és dinamikusan is linkelni szeretnénk mindkét elkészült könyvtárunkhoz. Ehhez a következő parancsot használhatjuk fel:

```
gcc mainStaticAndDynamic.c -I. -L. -Wl,-Bstatic -ltestStaticLibrary -Wl,-Bdynamic
-ltestDynamicLibrary -o mainStaticAndDynamic
```

A -I, -L, -l kapcsolók működése hasonló a korábbiakhoz. A különbség abban jelentkezik, hogy a linkernek explicit specifikáltuk, hogy melyik könyvtárat hogyan linkelje. A linkernek történő paraméterátadás a fentiekhez hasonlóan a -Wl kapcsolóval történik, a **–Bstatic** kapcsolóval specifikáljuk a statikusan linkelendő könyvtárat, míg a **–Bdynamic** kapcsolóval a dinamikusan linkelendő könyvtárat. Mivel a dinamikus könyvtárakhoz kapcsolódóan az LD_LIBRARY_PATH környezeti változó értékét már beállítottuk, az alkalmazásunk futtatható.

5.4. A dinamikus linkelés és a -soname kapcsoló

A teljesség kedvéért vizsgáljunk meg még egy példát, amely a -soname kapcsoló, azaz a dinamikus library-t leíró sztring kapcsolatát szemlélteti. A korábbiakban létrehoztunk egy dinamikus könyvtárat **libtestDynamicLibrary.so.1.0.1** néven, "libtestDynamicLibrary.so.1" leíró sztringgel, majd szimbolikus linket hoztuk létre libtestDynamicLibrary.so néven és így már működött a linkelés, mivel az alkalmazás linkelésekor léteznie kell megfelelő .so állománynak. A futtatás azonban már nem működött, mert az alkalmazásba beépült referencia **libtestDynamicLibrary.so.1** nevű dinamikus könyvtárat határozott meg, így újabb linkeléssel létrehoztuk ezt a fájlt is, amit immár dinamikus, a futtatás pillanatában be tudott tölteni a rendszer.

Hozzuk most létre egy másik verzióját a dinamikus könyvtárnak, a fentitől alig különböző linkelési paranccsal és azonos tartalommal:

```
gcc -shared -Wl,-soname,libtestDynamicLibrary.so.1 -o libtestDynamicLibrary.so.1.0.2
```

A fenti paranccsal létrehoztunk egy újabb dinamikus könyvtárat, az előzővel megegyező "libtestDynamicLibrary.so.1" leíró sztringgel, azaz azonos interfésszel, azonban libtestDynamicLibrary.so.1.0.2 néven. Most állítsuk át az libtestDynamicLibrary.so.1 linket, úgy, hogy az újonnan létrejött dinamikus könyvtárra, a **libtestDynamicLibrary.so.1.0.2**-re mutasson:

```
ln -fs libtestDynamicLibrary.so.1.0.2 libtestDynamicLibrary.so.1
```

Ha futtatni próbáljuk a **mainDynamic** alkalmazást, az továbbra is működni fog. Összefoglalva tehát a **mainDynamic** alkalmazás linkelésekor az alkalmazásba egy **libtestDynamicLibrary.so.1** interfészű könyvtárra való referencia épült be. Futtatáskor a futtató rendszer az LD_LIBRARY_PATH-ban specifikált elérési utakon keresi a **libtestDynamicLibrary.so.1** könyvtárat. Miután létrehoztuk a **libtestDynamicLibrary.so.1.0.2**-t könyvtárat és a **libtestDynamicLibrary.so.1** szimbolikus linket feülírtuk, bár linkelés során a **libtestDynamicLibrary.so.1.0.1**-hez linkeltük az alkalmazást, futáskor azonban a **libtestDynamicLibrary.so.1.0.2**-t töltötte be és használta fel, ugyanis ez volt elérhető **libtestDynamicLibrary.so.1** néven.

Hozzuk most létre még egy verzióját a dinamikus könyvtárnak, különböző leíró sztringgel, de az előzőekkel azonos tartalommal.

```
gcc -shared -Wl,-soname,libtestDynamicLibrary.so.2 -o libtestDynamicLibrary.so.1.0.3
```

Az előző példához hasonlóan állítsuk át a libtestDynamicLibrary.so linket úgy, hogy az újonnan létrejött fájlra mutasson:

```
ln -fs libtestDynamicLibrary.so.1.0.3 libtestDynamicLibrary.so
```

Most fordítsuk újra a dinamikusan linkelt alkalmazásunkat:

```
gcc mainDynamic.c -I. -L. -ltestDynamicLibrary -o mainDynamic
```

Az alkalmazásunkba most a libtestDynamicLibrary.so.2 sztring épül be, mint shared object referencia, mivel a libtestDynamicLibrary.so most a libtestDynamicLibrary.so.1.0.3-ra mutat, aminek a leíró sztringjét az -soname kapcsolóval libtestDynamicLibrary.so.2-ként definiáltuk. Így a betöltő rendszer libtestDynamicLibrary.so.2 nevű fájlt keres, ami nincs. Megoldásként létrehozhatunk egy szimbolikus linket libtestDynamicLibrary.so.2 néven, amely a 3 létező könyvtár bármelyikére mutathat. Mindazonáltal ez a példa csak szemléletető jellegű, nem teljesen életszerű, ugyanis a libtestDynamicLibrary.so.1.0.3 leíró sztringje nem felel meg a tényleges fájlnévnek.

Linux rendszerekben a linkek használata nem idegen, ha **ls –l** paranccsal kilistázzuk a /usr/lib rendszerkönyvtár tartalmát láthatjuk, hogy az előzőpéldákban bemutatott koncepciót megvalósítva számos link található shared object könyvtárakra.

5.5. Csomag disztribúciók készítése

Ahogy arról korábban szó volt, bináris és forrás csomagokat disztibútolhatunk.

Forráskód csomag

A forráscsomagban a header fájloknak, a forrás fájloknak, valamint egy script-nek kell megtalálhatónak lennie, amely képes az előbbiek lefordítására, linkelésére. Forráskód csomag disztribútolásához hozzuk létre a fordítást lehetővé tevő make.sh scriptet:

5.5.8. forráskód: make.sh

```
#!/bin/bash

gcc -I. -c testStaticLibrary.c -o testStaticLibrary.o
ar rcs libtestStaticLibrary.a testStaticLibrary.o
gcc -I. -c -fPIC testDynamicLibrary.c -o testDynamicLibrary.o
gcc --shared -Wl,-soname,libtestDynamicLibrary.so.1 -o libtestDynamicLibrary.so.1.0.1
    testDynamicLibrary.o
gcc -static mainStatic.c -I. -L. -ltestStaticLibrary -o mainStatic
ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so
ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so.1
gcc mainDynamic.c -I. -L. -ltestDynamicLibrary -o mainDynamic
gcc mainStaticAndDynamic.c -I. -L. -Wl,-Bstatic -ltestStaticLibrary -Wl,-Bdynamic -
    ltestDynamicLibrary -o mainStaticAndDynamic
```

A fenti script az összes eddigi parancsot tartalmazza, amelyet a könyvtárak és alkalmazások fordítása során használtunk. Másoljuk a forrás, header és make.sh fájlokat egy újonnan létrehozott testLibAndApp-src nevű könyvtárba, amelyet aztán a következő utasításokkal tömöríthetünk be .tar.gz állományba:

```
tar cvf testLibAndApp-src.tar testLibAndApp
gzip testLibAndApp-src.tar
```

5.5.2. Bináris csomag

A bináris csomagokban header fájlok, dinamikus és statikus könyvtárak, valamint alkalmazások szereplnek. Másoljuk ezeket egy testLibAndApp-Linux-x86_64 nevű könyvtárba, és az előzőhöz hasonló módon csomagoljuk be a könyvtárat:

```
tar cvf testLibAndApp-Linux-x86_64.tar testLibAndApp-Linux-x86_64
gzip testLibAndApp-Linux-x86_64.tar
```

Az előálló csomagok felhasználhatóak más Linux rendszereken, a forráskód csomag fordítható és használható nem 64 bites operációs rendszeren is, a bináris csomagban található könyvtárak és alkalmazások azonban csak 64 bites rendszereken működnek majd.

5.6. Kiegészítések

5.6.1. A dinamikus és statikus könyvtárak mérete

Nézzük meg, milyen fájlok jöttek létre:

```
lrwxrwxrwx 1 gykovacs gykovacs     30 2010-09-28 21:51 libtestDynamicLibrary.so
-> libtestDynamicLibrary.so.1.0.1
lrwxrwxrwx 1 gykovacs gykovacs     30 2010-09-28 21:51 libtestDynamicLibrary.so.1
-> libtestDynamicLibrary.so.1.0.1
-rwxr-xr-x 1 gykovacs gykovacs   7980 2010-09-28 21:51 libtestDynamicLibrary.so.1.0.1
-rw-r--r-- 1 gykovacs gykovacs   1784 2010-09-28 21:51 libtestStaticLibrary.a
-rwxr-xr-x 1 gykovacs gykovacs   8475 2010-09-28 21:51 mainDynamic
-rwxr-xr-x 1 gykovacs gykovacs 741778 2010-09-28 21:51 mainStatic
-rwxr-xr-x 1 gykovacs gykovacs   8625 2010-09-28 21:51 mainStaticAndDynamic
```

A linkek nem szorulnak magyarázatra. Az előállt könyvtárak és futtatható állományok méretbeli különbsége azonban igen. C nyelvű programok esetén észrevehettük, hogy a standard (std) könyvtár header-jeit anélkül include-olhatjuk, hogy bármilyen könyvtárat meg kellene adnunk a linkernek, amelyben a felhasznált függvények megtalálhatók. Gondoljunk csak az stdio.h header-re és a printf függvényre. A printf nem része a C nyelvnek, az std könyvtár tartalma, azonban linkeléskor ezt nem kell megadnunk, mert automatikusan, minden gcc linkeléskor a /usr/lib/libc.so, azaz az std könyvtárat tartalmazó dinamikus könyvtár dinamikusan likelésre kerül. A statikus és dinamikus könyvtáraink közül a statikus lett kisebb méretű. Ennek az az oka, hogy a statikus könyvtár egy egyszerű archívum, ami a tárgykódot és egy indexet tartalmaz, ami összefoglalja, hogy mi található a könyvtárban. A dinamikus könyvtár azonban a dinamikus betölthetőség miatt nagyobb méretű. Az alkalmazások tekintetében már más a helyzet: a statikus alkalmazás nagyságrendekkel nagyobb, mint a dinamikus alkalmazás, míg a dinamikusan és statikusan linkelt alkalmazás hasonló méretű a csak dinamikusan linkelt alkalmazáshoz. Ennek magyarázata a következő: a dinamikus könyvtárhoz linkelt alkalmazás csak egy referenciát tartalmaz a dinamikus könyvtárra, és abból tölti be a megfelelő függvényt. A statikusan linkelt alkalmazásnál azonban a –static kapcsoló nem csak a libtestStaticLibrary statikus linkelését írja elő, hanem az automatikusan linkelt libc könyvtár is statikusan linkelve bekerül az alkalmazásba. Ez magyarázza a jóval nagyobb méretét. A statikusan és dinamikusan is linkelt alkalmazás azért maradt viszonylag kicsi, mert a statikus linkelés csak a libtestStaticLibrary könyvtárra vonatkozott, a libc továbbra is dinamikusan kerül linkelésre, így nem növeli a fájl méretet.

A statikus linkelés előnye tehát, hogy a megfelelő kódrészletek beépülnek az alkalmazásba, de csak azok, amelyekre ténylegesen szükség van. A futtatható állomány azonban nagyobb lesz, így ha több egyszerre futó alkalmazáshoz ugyanazt a könyvtárat linkeltük statikusan, ugyanaz a kódrészlet egyszerre több alkalmazásban is jelen lesz a memóriában.

Dinamikus linkelésnél nem épül be a kód a futtatható állományba, a dinamikus könyvtár a futás pillanatában töltődik be a memóriába, azonban szemben a statikus linkeléssel, ahol csak a ténylegesen használt kódrészletek épülnek be az alkalmazásba, dinamikus linkelésnél az egész dinamikus könyvtár betöltődik, akkor is, ha csak egy kis részére van szükségünk. Viszont ha több alkalmazás is ugyanazt a dinamikus könyvtárat használja, akkor is csak egy példányban fordul elő a könyvtár a memóriában.

5.6.2. Debug és Release build

Interpreteres nyelveknél nem jön elő ez a kérdés, C/C++ programozásnál azonban nagyon fontos. Két fordítási módot különböztethetünk meg alkalmazások és könyvtárak fordításánál: a Debug fordítást és a Release fordítást. Debug módon történő fordítás esetén a tárgykódok a változók Debug-rendszer általi követését lehetővé tevő szimbólumokat tartalmaz, és hogy a függvényhívásokat is pontosan

lássuk, a helyettesíthető függvénykódok sem kerülnek be a függvényhívás helyére. Release fordításnál nincs már szükség nyomkövetési információkra, amelyek feleslegesen növelik a könyvtár és az alkalmazás méretét, és arra sincs szükségünk, hogy szintén ezekket az információkat felhasználva pontosan tudjuk, melyik függvény fut éppen, azaz a függvények kódja beírható a függvényhívás helyére (nem minden esetben, például rekurziónál). Az optimalizálás miatt a Release fordítás tovább tart, de általában egy nagyságrenddel gyorsabb kódot eredményez. Hogyan adhatjuk meg, hogy a fordítás Debug vagy Release legyen? A gcc fordító abban a formában, ahogy az előzőekben használtuk, mindig Debug fordítást végez! A Release fordítást, azaz az optimalizálás fokát kapcsolókkal adhatjuk meg. Az alapértelmezett kapcsoló az **–O0** az optimalizálás 0 fokára utal, tehát nincs optimalizálás. További lehetséges kapcsolók: **–O1**, **–O2**, **–O3**. Az **–O3** kapcsolóval specifikálható a lehető legjobb optimalizálás. A statikusan linkelt alkalmazás Release fordítása tehát az alábbi paranccsal érhető el:

```
gcc -O3 -static mainStatic.c -I. -L. -ltestStaticLibrary -o mainStatic
```

5.6.3. Warningok

A fordítás során, a fordító által kiírt warningokat a kóduk javítására használhatjuk fel. Az igazán jó kódok fordítása nem generál warningokat! A fordító által generált warning üzeneteket a **–Wall** és **–Wextra** kapcsolókkal kérhetjük. A **–w** kapcsoló kikapcsolja a warningokat. A **–Werror** minden warningot fordítási hibává tesz, azaz warning esetén a fordítás hibával megáll és nem csak egy figyelmeztetést ír ki. A statikus alkalmazás fordítása esetén a warningokat a következő módon kapcsolhatjuk tehát be:

```
gcc -O3 -Wall -Wextra mainStatic.c -I. -L. -ltestStaticLibrary -o mainStatic
```

5.6.4. Fordítási opciók

Fordítási opciók használatáról akkor beszélünk ha

- valamely fordítási kapcsolókat (warning-ok, optimalizálás) a fordítás során úgy szeretnénk ki/be kapcsolni, hogy a forráskódot nem változtatjuk;
- összetett könyvtárak vagy alkalmazások esetén a forráskódnak csak egy részét szeretnénk lefordítani;
- a könyvtár vagy alkalmazás olyan csomagra épít, amellyel nem rendelkezünk, így ezen csomagtól függő kódrészleteket ki szeretnénk hagyni a fordításból. Alkalmsazásunk vagy könyvtárunk funkcionalitása ugyan csökken, de mégis lefordítható.

Az utolsó két esetben az opcionális fordítás lényege, hogy elérjük, bizonyos kódrészletek csak akkor képezzék részét a lefordítandó alkalmazásnak, ha azt kérjük. Erre a célra preprocesszor direktívák használhatóak fel. Hozzuk létre az alábbi fájlt **mainOptional.c** néven, ami arra utal, hogy ilyen, opcionálisan fordítandó kódrészletet fog tartalmazni.

5.6.9. forráskód: mainOptional.c

```
#ifdef USE_DYNAMIC_LIBRARY
   #include <testDynamicLibrary.h>
#else
   #include <stdio.h>
#endif
int main(int argc, char** argv)
{
   #ifdef USE_DYNAMIC_LIBRARY
```

```
        printInDynamicLibrary();
    #else
        printf("dynamic library is not available\n");
    #endif
    return 0;
}
```

A fenti kódrészletben az **#ifdef**, **#else**, **#endif** direktívákat használtuk. Működésük a következő: ha a USE_DYNAMIC_LIBRARY változó definiált, akkor az **#ifdef** és **#elif** kulcsszavak közötti rész bekerül a fordítandó kódba, ha a változó nem definiált, akkor az **#else** és **#endif** kulcsszavak közötti rész lesz része a lefordítandó kódnak. Természetesen a **#elif** ág használata nem kötelező.

Alapértelmezésben a USE_DYNAMIC_LIBRARY természetesen nem definiált. Lefordítva és linkelve az alkalmazást a korábbihoz hasonló ismert

```
gcc mainOptional.c -o mainOptional
```

Az alkalmazás lefordul, azonban ha futtatjuk, láthatjuk, hogy a "dynamic library is not available" sztring jelenik meg a konzolon. Ennek oka, hogy a USE_DYNAMIC_LIBRARY változó nincs definiálva, így a dinamikus könyvtár include-olása valamint a könyvtárban lévő függvény hívása nem kerül bele a fordítandó kódba. Mivel nem használjuk a dinamikus könyvtárat, nincs szükség az **-l**, **-L** és **-I** kapcsolók megadására.

Ahhoz, hogy a fordítás során a dinamikus könyvtárat is használja a fordító, definiálnunk kell a USE_DYNAMIC_LIBRARY változót, lehetőleg anélkül, hogy módosítanánk a forráskódot. Erre használható a **-D** kapcsoló a fordítási parancsban. A **-D** kapcsoló után, azzal egybeírva kell megadnunk azon változó nevét, amelyet definiálni szeretnénk:

```
gcc mainOptional.c -I. -L. -ltestDynamicLibrary -DUSE_DYNAMIC_LIBRARY -o mainOptional
```

A fenti paranccsal lefordítva és futtatva az alkalmazást, a "this is in the dynamic library" sztring jelenik meg a konzolon. Természetesen ebben az esetben meg kell adnunk a megfelelően beállított **-l**, **-L** és **-I** kapcsolókat is.

A másik lehetőség koncepciójában különbözik ettől. Hozzunk létre egy üres **config.h** állományt a **mainOptional.c** mellett, és helyezzük el az **#include <config.h>** utasítást a **mainOptional.c** forráskód elejére, a USE_DYNAMIC_LIBRARY minden használata elé. Ekkor ha a **config.h** header-t üresen hagyjuk, a USE_DYNAMIC_LIBRARY változó nyilván definiálatlan lesz, ha azonban elhelyezzük benne a

```
#define USE_DYNAMIC_LIBRARY
```

direktívát, a változó definiálttá válik. Természetesen ügyelnünk kell ebben az esetben is a fordítási parancsban a megfelelő **-l**, **-L** és **-I** változók beállítására.

Célunk úgy változtatni most a **make.sh** scriptet, hogy bizonyos környezeti változók **yes** értékével a warningokat, optimalizálást és a **mainOptional** alkalmazás esetén a dinamikus könyvtár használatát be tudjuk kapcsolni. Legyenek ezek a változók rendre a **WARNINGS**, **OPTIMALIZE**, **OPTIONAL**.

5.6.10. forráskód: make.sh

```
#!/bin/bash

M="-O0 -g"
LIB_DIRS=-L.
INCLUDE_DIRS=-I.
```

```
if [ x${WARNINGS} = xyes ]; then
    W="-Wall -Wextra"
fi
if [ x${OPTIMALIZE} = xyes ]; then
    M="-O3"
fi
if [ x${OPTIONAL} = xyes ]; then
    O=`echo -DUSE_DYNAMIC_LIBRARY $INCLUDE_DIRS $LIB_DIRS -ltestDynamicLibrary`
fi

FLAGS=`echo ${W} ${M}`

set -x verbose #echo on

gcc $INCLUDE_DIRS -c testStaticLibrary.c -o testStaticLibrary.o $FLAGS
ar rcs libtestStaticLibrary.a testStaticLibrary.o
gcc $INCLUDE_DIRS -c -fPIC testDynamicLibrary.c -o testDynamicLibrary.o $FLAGS
gcc --shared -Wl,-soname,libtestDynamicLibrary.so.1 -o libtestDynamicLibrary.so.1.0.1
    testDynamicLibrary.o $FLAGS
gcc -static mainStatic.c $INCLUDE_DIRS $LIB_DIRS -ltestStaticLibrary -o mainStatic
    $FLAGS
ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so
ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so.1
gcc mainDynamic.c $INCLUDE_DIRS $LIB_DIRS -ltestDynamicLibrary -o mainDynamic $FLAGS
gcc mainStaticAndDynamic.c $INCLUDE_DIRS $LIB_DIRS -Wl,-Bstatic -ltestStaticLibrary -Wl
    ,-Bdynamic -ltestDynamicLibrary -o mainStaticAndDynamic $FLAGS
gcc mainOptional.c -o mainOptional $FLAGS $O
```

A fenti **make.sh** fájlban az **M** változó tárolja a **debug** Az **M** változó tartalmazza a debug/release fordítási mód kapcsolóit. Alapértelmezése a debug fordítás. Az **M** alapértelmezésének beállítása után három **bash if** utasítással ellenőrizzük a **WARNINGS**, **OPTIMALIZE** és **OPTIONAL** környezeti változók értékét. Ha valamelyik **yes**, akkor beállítjuk az **W**, **M** és **O** változók értékét a megfelelő módon. Az elágaztató utasítások feltételében azért szerepel az **x** karakter, mert ha a környezeti változó értéke üres sztring lenne, szintaktikailag helytelen utasítást kapnánk. A környezeti változók beállítása után a warningok és a fordítási mód kapcsolóit berakjuk a **FLAGS** változóba, majd a **FLAGS** változó értékét betesszük minden fordítási sorba. A **mainOptimal** fordítási sorába betesszük az opcionális fordítás kapcsolóit is. A **set -x Istinlineose *#echo on*** utasítás bekapcsolja a shell script-ben kiadott parancsok futás előtti kiírását.

A fenti **make.sh** segítségével kényelmesen konfigurálhatjuk a fordításunkat a környezeti változók beállításával. Például az alábbi

```
export WARNINGS=yes
./make.sh
```

utasítások hatására jól látható, hogy a fordítási parancsokba bekerült a **-O3** kapcsoló. A fentitől elegánsabb, ha a környezeti változók értékét csak a **make.sh** futtatására korlátozzuk:

```
WARNINGS=yes ./make.sh
```

A

```
WARNINGS=yes OPTIMALIZE=yes OPTIONAL=yes ./make.sh
```

parancs bekapcsolja a warning-ok használatát, optimalizálást ír elő és a **mainOptional** alkalmazás esetén az opcionálisan dinamikus könyvtár használatát állítja be.

Az **mainOptional** esetén az dinamikus könyvtár használatának korábban már említett másik módja, ha a megfelelő ***#define*** utasítást a **config.h**-ba írjuk bele. Ehhez a **make.sh**-ban az **OPTIONAL** változó vizsgálatát a következő módon kell módosítani:

```
if [ x${OPTIONAL} = xyes ]; then
  echo "#define USE_DYNAMIC_LIBRARY" > config.h
  O=`echo $INCLUDE_DIRS $LIB_DIRS -ltestDynamicLibrary
else
  rm "" > config.h
fi
```

Az **if** igaz ágában a **USE_DYNAMIC_LIBRARY** változót definiáló direktívát a **config.h** állományba írjuk és az **O** változóból kihagyjuk a –**DUSE_DYNAMIC_LIBRARY** kapcsolót, az **else** ágban pedig egy üres **config.h**-t állítunk elő. Jegyezzük meg, hogy ebben az esetben a **makeOptional.c** forrás elején include-olni kell a **config.h** headert.