Nagy számításteljesítményű párhuzamos programozási eszközök - gyakorlati jegyzet

Kovács György

Debreceni Egyetem

Tartalomjegyzék

1 Előszó . . . 1
2 A gyakorlat követelményei . . . 1
3 A GNU Compiler Collecion (gcc) . . . 1
4 A C/C++ programozás során használt fájl típusok . . . 2
4.1 Szöveges fájlok . . . 2
4.2 Bináris fájlok . . . 2
4.3 Csomag disztribúciók . . . 3
5 Első statikus és dinamikus könyvtáraink . . . 3
5.1 Forrásfájlok . . . 3
5.2 A könyvtárak létrehozása . . . 5
5.3 Az alkalmazások fordítása és linkelése . . . 6
5.4 A dinamikus linkelés és a -soname kapcsoló . . . 7
5.5 Csomag disztribúciók készítése . . . 8
5.5.1 Forráskód csomag . . . 9
5.5.2 Bináris csomag . . . 9
5.6 Kiegészítések . . . 9
5.6.1 A dinamikus és statikus könyvtárak mérete . . . 9
5.6.2 Debug és Release build . . . 10
5.6.3 Warningok . . . 11
5.6.4 Fordítási opciók . . . 11
6 Forráskódok menedzselése . . . 14
6.1 GNU make . . . 14
6.2 Egyszerű Makefile létrehozása . . . 14
6.2.1 Fordítási opciók . . . 15
6.2.2 Külső könyvtárak használata . . . 19
6.3 Autotools . . . 20
6.3.1 Egyszerű autotools környezet kialakítása . . . 20
6.3.2 Fordítási opciók . . . 27
6.3.3 Külső könyvtárak használata . . . 31
6.3.4 Disztribúciók létrehozása . . . 31
6.4 CMake . . . 32
6.4.1 Egyszerű CMake környezet kialakítása . . . 32
6.4.2 Fordítási opciók . . . 37
6.4.3 Külső könyvtárak használata . . . 39
6.4.4 Disztribúció létrehozása . . . 40
6.5 qmake . . . 43
6.5.1 Egyszerű qmake környezet kialakítása . . . 43

[1] A szerző email elérzetőségei: e-mail: gyuriofkovacs@gmail.com, url: http://morse.inf.unideb.hu/ gykovacs

6.5.2 Fordítási opciók . . . 45
6.5.3 Külső könyvtárak használata . . . 47
6.5.4 Disztribúció létrehozása . . . 48
6.6 Összefoglalás . . . 49
7 Párhuzamos programozás . . . 50
7.1 Folyamat/Process vs. Szál/Thread. . . . 50
7.2 Adat- és funcionális párhuzamosítás. . . . 50
7.3 Párhuzamos architektúrák. . . . 50
7.4 Flynn osztályok . . . 51
7.5 Amdahl törvénye . . . 51
7.6 Gustafson törvénye . . . 52
8 Esettanulmány . . . 53
9 Automatikus vektorizálás . . . 56
10 OpenMP . . . 61
10.1 Pragmák . . . 61
A Tesztkörnyezet . . . 66
References . . . 79

1. Előszó

A gyakorlataimon leadott anyagok kézirata, NEM LEKTORÁLT, HIBÁKAT TARTALMAZ, HASZNÁLATÁT SENKINEK SEM JAVASLOM!

2. A gyakorlat követelményei

A tárgy teljesítése C/C++ nyelvű beadandó feladat formájában történik.

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 előző példákban bemutatott koncepciót megvalósítva számos link található shared object könyvtárakra.

5.5. Csomag disztribúciók készítése

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

5.5.1. 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 -03 -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 **−I**, **−L** és **−l** 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 **−I**, **−L** és **−l** 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 **−I**, **−L** és **−l** 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 lstinlineose #*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.

6. Forráskódok menedzselése

6.1. GNU make

6.2. Egyszerű Makefile létrehozása

Az előző fejezetben készítettünk ugyan bináris és forrás csomagot is, azonban ha a egy több forrásból és más könyvtárakra is építő csomagot készítünk, a make.sh jellegű fordító script nagyon bonyolulttá, nehezen kezelhetővé válhat. Ennek megoldására dolgozták ki a make script nyelvet, amelyet csomagok fordítására használhatunk. A make egy bináris futtatható program, amelynek paramétere a GNU make scriptnyelven írt Makefile és ez a Makefile tartalmazza a fordításhoz és linkeléshez szükséges szabályokat.

A Makefile-okban definiálható szabályok szerkezete a következő:

```
target : fuggosegek
      vegrehajtando_utasitas
```

Makefile-ok terminológiájában target-nek nevezzük a célt, amit meg akarunk valósítani, a függőségek között azokat a targeteket és fájlokat soroljuk fel, amelyeknek létezniük kell az aktuális target létrehozásához, a végrehajtandó utasítás pedig a target létrehozásához használandó parancs.

Az előző fejezetben fordított forrásállományokhoz a következő Makefile-t hozhatjuk létre:

6.2.1. forráskód: Makefile

```
all: mainStatic mainDynamic mainStaticAndDynamic mainOptional

testStaticLibrary.o: testStaticLibrary.h testStaticLibrary.c
  gcc -I. -c testStaticLibrary.c -o testStaticLibrary.o

testDynamicLibrary.o: testDynamicLibrary.h testDynamicLibrary.c
  gcc -I. -c -fPIC testDynamicLibrary.c -o testDynamicLibrary.o

libtestStaticLibrary.a: testStaticLibrary.o
  ar rcs libtestStaticLibrary.a testStaticLibrary.o

libtestDynamicLibrary.so: testDynamicLibrary.o
  gcc -shared -Wl,-soname,libtestDynamicLibrary.so.1 -o libtestDynamicLibrary.so.1.0.1
      testDynamicLibrary.o
  ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so
  ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so.1

mainStatic: mainStatic.c libtestStaticLibrary.a
  gcc -I. -static mainStatic.c -L. -ltestStaticLibrary -o mainStatic
```

```
mainDynamic: mainDynamic.c libtestDynamicLibrary.so
  gcc -I. mainDynamic.c -L. -ltestDynamicLibrary -o mainDynamic

mainStaticAndDynamic: mainStaticAndDynamic.c libtestStaticLibrary.a
    libtestDynamicLibrary.so
  gcc -I. mainStaticAndDynamic.c -L. -Wl,-Bstatic -ltestStaticLibrary -Wl,-Bdynamic -
      ltestDynamicLibrary -o mainStaticAndDynamic

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

A Makefile-t tartalmazó könyvtárban állva a

```
make target_nev
```

utasítással hozhatjuk létre a megnevezett target-et. A make rendszer tehát lehetőséget biztosít arra, hogy a Makefile-t felhasználva könyvtárunknak csak egyes részeit fordítsuk le. Paraméter nélkül a make csak az első target-ethez tartozó szabályt alkalmazza. Létrehozhatunk további targeteket, amelyeknek csak előfeltételeket adunk meg, így azokkal bizonyos más targetek létrehozását fogjuk össze, megadható továbbá olyan target is, amelyhez nem tartozik előfeltétel. Gyakran használt target-ek az "all", "clean" és az "install". Egészítsük ki előző Makefile-unkat a következő targetekkel:

```
all: mainStatic mainDynamic mainStaticAndDynamic mainOptional

clean:
      rm mainStatic mainDynamic mainStaticAndDynamic mainOptional libtestDynamicLibrary.
          so \
      libtestDynamicLibrary.so.1 libtestDynamicLibrary.so.1.0.1 libtestStaticLibrary \
      testStaticLibrary.o testDynamicLibrary.o

install: mainStatic mainDynamic mainStaticAndDynamic mainOptional
  cp main* /usr/bin
  cp libtestDynamicLibrary.so* /usr/lib
  cp libtestStaticLibrary.a /usr/lib
  cp *.h /usr/include
```

Az **all** target mindig a Makefile első target-e, így target név nélkül a **make** utasítás mindent létrehoz. A **clean** targetet a létrehozott fájlok törlésére használják, míg az **install** target elhelyezi a bináris állományokat a megfelelő **/usr/bin** és **/usr/lib** könyvtárakban.

6.2.1. Fordítási opciók

A Makefile-ok egyik erőssége, hogy változókat hozhatunk létre, ezáltal rövidítve a fordítási parancsokat. A változók a környezeti változókhoz hasonlóan működnek, nem kell deklarálni őket, értékadásnál jönnek létre, és hivatkozni az értékükre **$(VALTOZO_NEV)** módon lehet.

A változók használatának szemléltetésére létrehozom a CFLAGS változót, amelyben a források fordításánál használandó kapcsolókat adom meg és a LIB_DIRS változót, amelyben a -L kapcsolóval megadandó könyvtárakat fogjuk össze. Hozzuk létre tehát az alábbi változókat a Makefile első soraiban, és módosítsuk a megfelelő fordítási és linkelési parancsokat (példaként megadom a mainDynamic targethez tartozó parancs módosítását):

```
CFLAGS=-I.
LIB_DIRS=-L.

mainDynamic: MainDynamic.c libtestDynamicLibrary.so
      gcc $(CFLAGS) mainDynamic.c $(LIB_DIRS) -ltestDynamicLIbrary -o mainDynamic
```

Változókhoz hozzáadhatunk további értékeket, azaz sztringeket fűzhetünk hozzájuk a += operátorral, például:

```
CFLAGS+=-O3
```

a CFLAGS változóhoz hozzáfűzi a -O3 kapcsolót.

Változókhoz értéket rendelhetünk a make utasításban is, a következő módon:

```
make CFLAGS=-O3
```

Ebben az esetben a CFLAGS változó értéke a Makefile során az -O3 kapcsoló lesz, függetlenül attól, hogy a Makefile elején értékeket rendeltünk hozzá, azaz a Makefile-ban történt értékadások felül lesznek definiálva.

Az utolsó konstrukció, amit a GNU make-hez kapcsolódóan megnézünk, az a feltételes szerkezet, ugyanis változók értékétől függően feltételesen is beállíthatjuk a fordítási flag-eket. Egy feltételes szerkezet általános formája:

```
ifeq (arg1,arg2)
   utasitasok
[else
   utasitasok]
endif
```

Az fenti feltételes szerkezet az argumentumok egyenlőségét vizsgálja, minden esetben. A [] zárójel az **else** ág opcionalitását jelzi. Felhasználva az utóbbi három eszközt, módosítsuk úgy a Makefile-t, hogy a make hívásban beállított BUILD=release vagy BUILD=debug változó alapján hozzáfűzze a -O3 vagy -O0 kapcsolót a CFLAGS változóhoz a Makefile-ban! A módosított, végleges Makefile az alábbi lesz:

6.2.2. forráskód: Makefile

```
INCLUDE_DIRS:=-I.
LIB_DIRS:=-L.

ifeq ($(BUILD),debug)
    CFLAGS+=-O0 -g
endif
ifeq ($(BUILD),release)
    CFLAGS+=-O3
endif

all: mainStatic mainDynamic mainStaticAndDynamic mainOptional

testStaticLibrary.o: testStaticLibrary.h testStaticLibrary.c
  gcc $(CFLAGS) $(LIB_DIRS) $(INCLUDE_DIRS) -c testStaticLibrary.c -o testStaticLibrary.
      o

testDynamicLibrary.o: testDynamicLibrary.h testDynamicLibrary.c
  gcc $(CFLAGS) $(LIB_DIRS) $(INCLUDE_DIRS) -c -fPIC testDynamicLibrary.c -o
      testDynamicLibrary.o

libtestStaticLibrary.a: testStaticLibrary.o
  ar rcs libtestStaticLibrary.a testStaticLibrary.o

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

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

mainStatic: mainStatic.c libtestStaticLibrary.a
  gcc $(CFLAGS) -static mainStatic.c $(LIB_DIRS) $(INCLUDE_DIRS) -ltestStaticLibrary -o
      mainStatic

mainDynamic: mainDynamic.c libtestDynamicLibrary.so
  gcc $(CFLAGS) mainDynamic.c $(LIB_DIRS) $(INCLUDE_DIRS) -ltestDynamicLibrary -o
      mainDynamic

mainStaticAndDynamic: mainStaticAndDynamic.c libtestStaticLibrary.a
    libtestDynamicLibrary.so
  gcc $(CFLAGS) mainStaticAndDynamic.c $(LIB_DIRS) $(INCLUDE_DIRS) -Wl,-Bstatic -
      ltestStaticLibrary -Wl,-Bdynamic -ltestDynamicLibrary -o mainStaticAndDynamic

mainOptional: mainOptional.c
  gcc $(CFLAGS) mainOptional.c -o mainOptional

clean:
  rm testStaticLibrary.o testDynamicLibrary.o libtestStaticLibrary.a
      libtestDynamicLibrary.so mainStatic mainDynamic mainStaticAndDynamic mainOptional

install: mainStatic mainDynamic mainStaticAndDynamic mainOptional
  cp main* /usr/bin
  cp libtestDynamicLibrary.so* /usr/lib
  cp libtestStaticLibrary.a /usr/lib
  cp *.h /usr/include
```

Ezen új Makefile használata a következő:

- **make BUILD=debug** lefordít mindent debug módban
- **make BUILD=release** lefordít mindent release módban
- **make clean** letöröl minden állományt, amit létrehozott
- **make install** bemásolja az létrehozott könyvtárakat és programokat a megfelelő rendszerkönyvtárakba

A könyvtárak fordítása és linkelése összetett feladat, amely nehezen automatizálható, azonban az egyszerű forráskódok tárgykódra történő fordítását lehet tovább egyszerűsíteni, a következő módon: a make automatikusan tudja, hogy .o kiterjesztésű fájl előállításához a gcc parancsot kell futtatni egy azonos nevű, .c kiterjesztésű forráskód paraméterrel, ezért ilyen egyszerű esetben nem kell kiírnunk a fordítási parancsot, tehát a

```
testStaticLibrary.o: testStaticLibrary.h testStaticLibrary.c
      gcc \$(CFLAGS) -c testStaticLibrary.c -o teststaticLibrary.o
```

szabály egyszerűsíthető az alábbival

```
testStaticLibrary.o: testStaticLibrary.h testStaticLibrary.c
```

Az egyetlen kérdés, hogy hogyan adhatunk meg fordítónak szóló kapcsolókat ha nem mi szerkesztjük a fordítási parancsot? A válasz az, hogy a CFLAGS változó egy speciális változó abban az értelemben, hogy a fenti egyszerűsítés esetén a CFLAGS változó értékét a make belerakja a fordítási parancsba, így megfelelően beállított CFLAGS változóval (esetünkben) ugyanazt az eredményt érjük el, mint ha teljesen kiírnánk a szabályt.

6.1. Feladat: Alakítsuk át a fenti létrehozott **Makefile**-t a feltételes szerkezet használatával úgy, hogy a fordítási sorban beállított **WARNINGS=yes**, **BUILD=debug/relese**, **OPTIONAL=yes** változók értelmében a warning-ok, a fordítási mód és az opcionális dinamikusan linkelendő könyvtár használata megfelelően beállítódjon.

6.1. Megoldás:

6.2.3. forráskód: Makefile

```
INCLUDE_DIRS:=-I.
LIB_DIRS:=-L.

ifeq ($(BUILD),debug)
    CFLAGS+=-O0 -g
endif
ifeq ($(BUILD),release)
    CFLAGS+=-O3
endif
ifeq ($(WARNINGS),yes)
    CFLAGS+=-Wall -Wextra
endif
ifeq ($(OPTIONAL),yes)
    O:=-DUSE_DYNAMIC_LIBRARY $(INCLUDE_DIRS) $(LIB_DIRS) -ltestDynamicLibrary
endif

all: mainStatic mainDynamic mainStaticAndDynamic mainOptional

testStaticLibrary.o: testStaticLibrary.h testStaticLibrary.c
  gcc $(CFLAGS) $(LIB_DIRS) $(INCLUDE_DIRS) -c testStaticLibrary.c -o testStaticLibrary.
      o

testDynamicLibrary.o: testDynamicLibrary.h testDynamicLibrary.c
  gcc $(CFLAGS) $(LIB_DIRS) $(INCLUDE_DIRS) -c -fPIC testDynamicLibrary.c -o
      testDynamicLibrary.o

libtestStaticLibrary.a: testStaticLibrary.o
  ar rcs libtestStaticLibrary.a testStaticLibrary.o

libtestDynamicLibrary.so: testDynamicLibrary.o
  gcc -shared -Wl,-soname,libtestDynamicLibrary.so.1 -o libtestDynamicLibrary.so.1.0.1
      testDynamicLibrary.o
  ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so
  ln -fs libtestDynamicLibrary.so.1.0.1 libtestDynamicLibrary.so.1

mainStatic: mainStatic.c libtestStaticLibrary.a
  gcc $(CFLAGS) -static mainStatic.c $(LIB_DIRS) $(INCLUDE_DIRS) -ltestStaticLibrary -o
      mainStatic

mainDynamic: mainDynamic.c libtestDynamicLibrary.so
  gcc $(CFLAGS) mainDynamic.c $(LIB_DIRS) $(INCLUDE_DIRS) -ltestDynamicLibrary -o
      mainDynamic

mainStaticAndDynamic: mainStaticAndDynamic.c libtestStaticLibrary.a
    libtestDynamicLibrary.so
  gcc $(CFLAGS) mainStaticAndDynamic.c $(LIB_DIRS) $(INCLUDE_DIRS) -Wl,-Bstatic -
      ltestStaticLibrary -Wl,-Bdynamic -ltestDynamicLibrary -o mainStaticAndDynamic

mainOptional: mainOptional.c
  gcc $(CFLAGS) $(O) mainOptional.c -o mainOptional

clean:
  rm testStaticLibrary.o testDynamicLibrary.o libtestStaticLibrary.a
      libtestDynamicLibrary.so mainStatic mainDynamic mainStaticAndDynamic mainOptional

install: mainStatic mainDynamic mainStaticAndDynamic mainOptional
  cp main* /usr/bin
  cp libtestDynamicLibrary.so* /usr/lib
  cp libtestStaticLibrary.a /usr/lib
  cp *.h /usr/include
```

6.2. Feladat: Módosítsuk a fenti **Makefile**-t úgy, hogy a dinamikus könyvtár használatát szabályozó **define** direktíva a **mainOptional.c**-be a **config.h** header-ön keresztül kerüljön be!

6.2. Megoldás: A megoldás rafinált, ugyanis **Makefile**-okban nem helyezhetünk el csak úgy shell parancsokat. Létre kell hoznunk két új target-et:

```
config-use:
    echo "#define USE_DYNAMIC_LIBRARY" > config.h
config-nouse:
    echo "" > config.h
```

Ezt követően a **make.sh** esetén született megoldáshoz hasonlóan az **O** változónak nem adjuk értékül a **−DUSE_DYNAMIC_LIBRARY** kapcsolót, viszont egy új, **USING_TARGET** nevű változónak értékül adjuk a **config.h** fájlba író targetek valamelyikét az **if** feltételének teljesülésétől függően:

```
ifeq ($(OPTIONAL),yes)
  O:=$(INCLUDE_DIRS) $(LIB_DIRS) -ltestDynamicLibrary
  USING_TARGET:=config-use
else
  USING_TARGET:=config-nouse
endif
```

Nincs más hátra, mint a **USING_TARGET** változó értékét, azaz a **config.h** helyes kitöltését, mint target-et elhelyezni a **mainOptional** target előfeltételei között:

```
mainOptional: mainOptional.c $(USING_TARGET)
  gcc $(CFLAGS) $(O) mainOptional.c -o mainOptional
```

A fenti módosításokat követően

```
make BUILD=debug WARNINGS=yes OPTIONAL=yes
```

paranccsal a debug fordítást, a warningok használatát és az **makeOptional** alkalmazásban az opcionális dinamikus könyvtár használatát írhatjuk elő.

6.2.2. Külső könyvtárak használata

Amiről eddig nem volt szó, az külső könyvtárak használatának módja. A következő példában a **libpng** könyvtárat fogjuk használni. Az alábbi példaalkalmazás az első parancssori argumentumaként kapott fájlnevet megnyitja, beolvassa az első 8 bájtját, és a **libpng** könyvtár egy függvényével eldönti, hogy a paraméterként kapott fájlnév png szerkezetű fájlt takar-e vagy sem. Az alkalmazás forráskódja tehát:

6.2.4. forráskód: mainPNG.c

```
#include <png.h>

int main(int argc, char** argv)
{
    FILE* fp= fopen(argv[1], "rb");
    char header[8];

    if (!fp)
  return 1;
    fread(header, 1, 8, fp);

    printf("%d\n", png_sig_cmp(header, 0, 8));

    return 0;
}
```

Az alkalmazás tehát 0-t ír a képernyőre, ha az argumentum fájl PNG szerkeztű és 1-t, ha nem az. Ahhoz, hogy le tudjuk fordítani az alkalmazást a korábbi példákhoz hasonlóan, három dolgot kell megadnunk a fordítónak, linkernek:

- a **libpng** könyvtár header-jeinek elérési útját (**-I** kapcsoló);
- a **libpng** könyvtár lib-jeinek elérési útját (**-L** kapcsoló);
- a **libpng** könyvtár lib-jeinek nevét, ügyelve arra, hogy a **lib** szócskát ne írjuk ki és kiterjesztéseket ne használjunk (**-l** kapcsoló).

Ezek a kapcsolók megadhatók explicit elérési utakkal egy adott rendszeren, azonban ez nagyban gyengíti alkalmazásunk hordozhatóságát. Ahhoz, hogy a hordozhatóságot megtartsuk, egy általánosabb megoldást kell találnunk, amelyet a **pkg-config** alkalmazás biztosít számunkra. A **pkg-config** egy interoperábilis alkalmazás. A koncepció a következő: minden forráskód csomag, amelyet feltelepítünk egy rendszerre, installál egy **.pc** kiterjesztésű úgynevezett package-config fájlt. Ezen package-config fájl az installált könyvtár jellemzőit tartalmazza, azaz a header fájlok elérési útját, a library-k elérési útját, a library-k nevét és még néhány egyszerű leíró tulajdonságot. A **libpng** könyvtár .pc fájlja a saját rendszeremen:

6.2.5. forráskód: /usr/lib/pkgconfig/libpng.pc

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/libpng12

Name: libpng
Description: Loads and saves PNG files
Version: 1.2.44
Libs: -L${libdir} -lpng12
Libs.private: -lz -lm
Cflags: -I${includedir}
```

A fenti package-config két fontos változót definiál:

- **Cflags**, amely a fordítónak szóló flag-eket tartalmazza (**-I**),
- **Libs**, amely a linkernek szóló flag-eket tartalmazza (**-L**, **-l**).

A package-config fájlok standard helye a **/usr/lib/pkgconfig** könyvtár, azonban tetszőleges helyen szereplő package-config fájlok is teljesértékűen használhatóak, ha elérési útjuk szerpeel a **PKG_CONFIG_PATH** környezeti változóban. A **pkg-config** alkalmazás parancssori argumentumként egy könyvtár nevét várja (pl **libpng**), és a **--cflags** kapcsoló hatására kiírja azon fordítási flag-eket, amelyekre az argumentumként kapott könyvtár használatához szükség van, a **--libs** kapcsoló hatására pedig azon linker flag-eket, amelyek az argumentumként kapott könyvtár linkeléséhez kellenek. Ezt követően sokkal általánosabb fordítási parancsot kapunk, ha abba beágyazzuk a **pkg-config** hívását:

```
gcc `pkg-config --cflags libpng` `pkg-config --libs libpng` -o mainPNG mainPNG.c
```

A fenti sort hozzáadva az eddig használt **Makefile**-hoz egy megfelelő **mainPNG** targettel, épp a kívánt működést kapjuk, anélkül, hogy bármi rendszerspecifikusat írtunk volna a kódba. Ha a **libpng** könyvtár nincs installálva, a **pkg-config** nem ír ki semmit, így fordítási hibához jutunk, hiszen a fordító nem fogja megtalálni a **png.h** headert.

6.3. Autotools

6.3.1. Egyszerű autotools környezet kialakítása

A korábban bemutatott GNU Makefile-ok tökéletes eszközt nyújtanak a források menedzselésére, azonban létrehozásuk még néhány fájlból álló könyvtár esetén is nehézkes, későbbi módosítása pedig

igazi agyrém. Az autotools eszközök arra használhatóak fel, hogy néhány magasabb szintű konfigurációs állomány segítségével és az autotools eszközökkel GNU Makefile-okat generáljunk, amelyet a korábban megismert make eszközzel használhatunk fordításra.

Az automake eszközrendszer nem a legegyszerűbb, de jelenleg talán a legelterjedtebb módja alkalmazások és könyvtárak forrásainak menedzselésére, Linux rendszereken a leggyakrabban használt eszközrendszer. A rendszer összetett, több konfigurációs állományból és autotools eszközhívással juthatunk el a fordításig, azaz a Makefile-okig létrejöttéig.

Használjuk a korábban létrehozott kódjainkat, azaz a dinamikus és statikus könyvtárat és a négy alkalmazást. Hozzunk létre egy könyvtárat **testLibraries** néven, benne a **srcsl**, **srcdl**, **srcsa**, **srdda**, **srcsda** és **srcoa** könyvtárakat, amelyekbe másoljuk be a korábban használt forrásfájljainkat! Hozzunk létre továbbá egy **m4** nevű könyvtárat a **testLibraries** gyökérkönyvtárban, amelyet segédfájlok tárolására használ majd az autotools rendszer.

6.3.6. forráskód: A könyvtárszerkezet autotools használatához

```
testLibraries
     m4
     srcda
         mainDynamic.c
     srcdl
         testDynamicLibrary.c
         testDynamicLibrary.h
     srcoa
         mainOptional.c
     srcsa
         mainStatic.c
     srcsda
         mainStaticAndDynamic.c
     srcsl
         testStaticLibrary.c
         testStaticLibrary.h
```

Hozzunk létre a **configure.ac** konfigurációs állományt a **testLibraries** gyökérkönyvtárban és **Makefile.am** nevű állományokat a gyökérkönyvtárban és minden alkönyvtárban!

6.3.7. forráskód: Könyvtárszerkezet konfigurációs állományokkal

```
testLibraries
     m4
     srcda
         Makefile.am
         mainDynamic.c
     srcdl
         Makefile.am
         testDynamicLibrary.c
         testDynamicLibrary.h
     srcoa
         Makefile.am
         mainOptiona.c
     srcsa
         Makefile.am
         mainStatic.c
     srcsda
         Makefile.am
         mainStaticAndDynamic.c
     srcsl
         Makefile.am
         testStaticLibrary.c
         testStaticLibrary.h
     Makefile.am
```

```
configure.ac
```

Töltsük fel most a konfigurációs állományokat az alábbi tartalmakkal!

6.3.8. forráskód: configure.ac

```
AC_INIT([testLibraries], [1.0], [gyuriofkovacs@gmail.com])
AC_CONFIG_MACRO_DIR([m4])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
LT_INIT
AC_PROG_CC
AC_PROG_RANLIB
AC_PROG_LIBTOOL

CFLAGS=""

AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile srcsl/Makefile srcsa/Makefile srcdl/Makefile srcda/Makefile
    srcsda/Makefile srcoa/Makefile])
AC_OUTPUT
```

6.3.9. forráskód: Makefile.am

```
SUBDIRS = srcsl srcsa srcdl srcda srcsda srcoa
```

6.3.10. forráskód: srcsl/Makefile.am

```
lib_LIBRARIES = libtestStaticLibrary.a
libtestStaticLibrary_a_SOURCES = testStaticLibrary.c
include_HEADERS = testStaticLibrary.h

AM_CFLAGS = -I.
```

6.3.11. forráskód: srcdl/Makefile.am

```
lib_LTLIBRARIES = libtestDynamicLibrary.la

libtestDynamicLibrary_la_SOURCES = testDynamicLibrary.c
include_HEADERS = testDynamicLibrary.h
AM_CFLAGS = -I.
```

6.3.12. forráskód: srcsa/Makefile.am

```
bin_PROGRAMS = mainStatic
mainStatic_SOURCES = mainStatic.c

AM_CFLAGS = -I. -I$(srcdir)/../srcsl
mainStatic_LDADD = ../srcsl/libtestStaticLibrary.a
```

6.3.13. forráskód: srcda/Makefile.am

```
bin_PROGRAMS = mainDynamic
mainDynamic_SOURCES = mainDynamic.c

AM_CFLAGS = -I. -I$(srcdir)/../srcdl
mainDynamic_LDADD = ../srcdl/libtestDynamicLibrary.la
```

6.3.14. forráskód: srcsda/Makefile.am

```
bin_PROGRAMS = mainStaticAndDynamic
mainStaticAndDynamic_SOURCES = mainStaticAndDynamic.c

AM_CFLAGS = -I. -I$(srcdir)/../srcdl -I$(srcdir)/../srcsl

mainStaticAndDynamic_LDADD = ../srcdl/libtestDynamicLibrary.la \
  ../srcsl/libtestStaticLibrary.a
```

6.3.15. forráskód: srcoa/Makefile.am

```
bin_PROGRAMS = mainOptional
mainOptional_SOURCES = mainOptional.c

AM_CFLAGS = -I. -I$(srcdir)/../srcdl
mainOptional_LDADD = ../srcdl/libtestDynamicLibrary.la
```

Az autotools rendszer feladat kettős: egyrészt menedzselhetjük vele saját fejlesztésű könyvtárunkat, másrészt olyan konfigurációs scripteket állíthatunk elő vele, amelyek más rendszereken, más architektúrájú és operációs rendszert használó számítógépeken beállítják a fordításhoz szükséges kapcsolókat. Autotools eszközökkel menedzselt könyvtárak első lépése a **configure** nevű script előállítása. Ez a script a rendszer ellenőrzést végzi, és előállítja a fordításhoz használható Makefile-okat. A **configure** script a Makefile-okat a Makefile.am konfigurációs fájlokban megadott információk alapján hozza létre. A **configure** script előállítása mindig a forráscsomagok készítőjének a feladata. Az előző fejezetben láthattuk, hogy a forráscsomagokhoz mindig biztosítani kell olyan állományokat, amelyek lehetővé teszik a forráskód halmaz lefordítását, installálását, esetleg tesztelését. Autotools esetén a **configure** script és a **Makefile.am** állományok szolgálják ezt a célt, azaz ha disztribúcióra kerül a sor, a forráskóddal együtt adnunk kell a **configure** script-et és a **Makefile.am** konfigurációs állományokat.

A fordítással kapcsolatos tényleges információkat, mint például azt, hogy egy target-hez mely forrásfájlok tartoznak, a **Makefile.am** fájlokban adjuk meg. A **configure** script a rendszer ellenőrzését végzi, és a rendszernek, illetve a felhasználó által megadott kapcsolóknak megfelelően előállítja a Makefile-okat a **Makefile.am**-ek segítségével. Hogy pontosan milyen eszközök, könyvtárak jelenlétét kell ellenőriznie, illetve milyen kapcsolókra hogyan kell reagálnia a **configure** script-nek azt a **configure.ac** konfigurációs állományban adhatjuk meg. Vegyük sorra a **configure.ac** tartalmát, és nézzük meg, melyik utasítás mit jelent esetünkben!

Az **AC_INIT** utasítással inicializáljuk a projektünket. Az első paramétere a könyvtár nevét adja meg, második paramétere a verziószámot, harmadik paramétere pedig egy email cím, ahová az esetleges hibajelentéseket küldhetik a könyvtár használói.

Az egész autotools rendszer scriptekből áll, amelyek főleg a konfigurációs állományokban elhelyezett sztringek átírásával, helyettesítésével, azaz makrók feldolgozásával állítják elő azokat a parancsokat és végső target neveket, amelyeket a fordítás során használnak. Az **m4** csomag a Linux rendszereken a legelterjedtebb makró processzor. Könyvtárszerkezetünk gyökerében azért hoztuk létre az **m4** könyvtárat, hogy az **m4** csomag ebben elhelyezhesse a létrehozott temporáris fájlokat. Azt, hogy az **m4** ezt a könyvtárat használhatja, az **AC_CONFIG_MACRO_DIR([m4])** utasítással állítjuk be.

A következő utasítás az **AM_INIT_AUTOMAKE([foreign])** azt specifikálja, hogy könyvtárunk nem lesz része a GNU rendszernek, amely nagyon szigorú disztribúciós szabályokkal rendelkezik (pl. kötelező manual page-ek, stb.), így egy teljes GNU által disztribútult csomaghoz képest kevesebb eszközzel is létrehozhatunk csomagot.

Amikor különböző rendszereken fordítunk GNU eszközökkel, a paraméterezés többnyire megegyezik, (a fordító többnyire ugyanolyan kapcsolókkal rendelkezik, a statikus könyvtárak ugyanúgy archívumba rendezett objekt fájlok), azonban a shared object fájlok különböznek. Linux rendszeren például láthattuk, hogy felhasználtuk a **−fPIC** fordító flag-et, positon-independent-code létrehozására, Windows-on erre nincs szükség, más rendszereknek pedig más további igényeik lehetnek. Hogy a shared object könyvtárak, azaz dinamikusan linkelhető könyvtárak létrehozása közötti különbséget elfedjék az autotools eszközei, kidolgozták az úgynevezett **libtool** eszközt. A **libtool** működése meghaladja ezen tárgy kereteit, azonban tudnunk kell, hogy ha dinamikus könyvtárakkal dolgozunk, a **libtool** használatát specifikálnunk kell a konfigurációs állományainkban. Az **LT_INIT** utasítással kérhetjük a **libtool** eszköz inicializálását.

A következő három utasítás mind a rendszerre telepített eszközöket keresi meg, és megfelelő változókban eltárolja, hogy az egyes eszközök közül, pl. az elérhető C fordítók, melyiket fogja használni. **AC_PROG_CC** a C fordító (C Compiler) jelenlétét vizsgálja, az **AC_PROG_RANLIB** ellenőrzést akkor kell kérnünk, ha valamilyen könyvtárat (akkor is, ha statikusat, akkor is, ha dinamikusat) szeretnénk létrehozni a fordítás során, az **AC_PROG_LIBTOOL** a korábban említett **libtool** jelenlétét és működését ellenőrzi.

Akárcsak a GNU Makefile-ok esetén, autotools használatánál is vannak speciális környezeti változók, ilyen például a **CFLAGS**. Ennek a környezeti változónak az értéke be kerül minden egyes fordítási parancsba. Alapértelmezésben ez tartalmazza a **−g −O2** kapcsolókat, ahol a **−g** azt jelzi, hogy a későbbiekben debuggert szeretnénk használni, az **−O2** pedig egy közepes optimalizálást specifikál. Az alapértelmezett kapcsolókat a **CFLAGS=""** utasítással töröljük.

A következő utasításban fel kell sorolnunk azokat a konfigurációs állományokat (esetünkben csak Makefile-okat), amelyeket a **configure** scriptnek elő kell majd állítania. Vigyázzunk, hogy minden felsorolt fájl esetén legyen a megfelelő helyen egy **Makefile.am**, aminek a tartalma alpján, a rendszernek megfelelően állnak majd elő a végső **Makefile**-k, esetünkben 7 ilyen **Makefile** van: a **AC_CONFIG_FILES ()** paramétereként láthatjuk őket.

Az **AC_OUTPUT** állítja elő végül a **AC_CONFIG_FILES**-ban felsorolt fájlokat.

Vegyük most sorra a **Makefile.am** fájlokat és tartalmukat.

A **testLibraries** gyökérkönyvtárban található **Makefile.am** egyedül azt adja meg, hogy mely további alkönyvtárakban vannak feldolgozandó **Makefile**-ok. Esetünkben ez az 6 könyvtár a statikus és dinamikus alkalmazáshoz, valamint a statikusan, dinamikusan és statikusan és dinamikusan linkelt alkalmazások forráskódját tartalmazó könyvtárak.

A statikus könyvtárhoz tartozó **Makefile.am** válotozók értékadásait tartalmazza és target specifikációkat tartalmaz. Autotools **Makefile.am**-ek esetén a konvenció a

```
where_PRIMARY= targets ...
```

változónév és értékadás forma. A **PRIMARY** rész adja meg, hogy a fordítás során a target-ek nevén milyen fájl jöjjön létre. A lehetőségek:

- **_PROGRAMS** esetén bináris futtatható állományok jönnek létre;
- **_LIBRARIES** esetén statikus könyvtárak;
- **_LTLIBRARIES** esetén libtool, azaz dinamikus könyvtárak;
- **_HEADERS** esetén nincs szükség fordításra, azt vezérelhetjük azonban, hogy a megfelelő header-ök installálása hová történjen;

- **_SCRIPTS** esetén scriptek jönnek létre;
- **_DATA** esetén pedig a felsorolt fájlok resource fájlok, fordítás nem történik, azonban azt vezérelhetjük, hogy ezen változókon keresztül, hogy hová insatllálódjanak.

A **where** rész az értékadás bal oldalán azt specifikálja, hogy installáláskor mely könyvtárba kerüljenek a létrejött target-ek. Ajánlott azonban a konvenciók használata: futtatható állományok a **bin** könyvtárba, míg statikus és dinamikus könyvtárak egyaránt a **lib** könyvtárba kerüljenek installálás után. Ennek megfelelően ha megnézzük a statikus könyvtárhoz tartozó **srcsl/Makefile.am** állományt, láthatjuk, hogy egyetlen target-et definiáltunk, amely **lib** könyvtárba kerül majd installáláskor, a neve pedig **libtestStaticLibrary.a**. Ezt követően az egyes target-ekhez tartozó források megadása következik, amelyet a

```
targetNev_SOURCES = forras_fajlok
```

formában rendelhetünk hozzá az egyes target-ekhez. Esetünkben ez egyetlen forrásfájl. Ügyeljünk arra, hogy a target nevekben szereplő '.' karaktereket, ha a target név a fentihez hasonlóan értékadás bal oldalán szerepel, minden esetben '_' karakterrel kell helyettesítenünk. Ahhoz, hogy egy statikus vagy dinamikus könyvtárat használni tudjunk, minden esetben szükségünk van header-ökre, így nem szabad elfelejtkeznünk arról, hogy az **include** könyvtárba kerülő header-öket is explicit módon megadjuk a **include_HEADERS =** változó beállításával. Egyetlen utasítás maradt, amelyben a fordító kapcsolóit állítjuk be: az **AM_CFLAGS** változó az aktuális Makefile-ból származó fordításokra van csak hatással, szemben a **CFLAGS** változóval, amely értéke a projekt fordítása során minden fordítási parancsba bekerül. Az **AM_CFLAGS** értéke esetünkben egyetlen **−I.** include könyvtár megadás, ugyanis emlékezzünk, hogy a forrásfájljainkban az **include** utasításokban a header-öket **<>**-ek között adtuk meg, ami azt jelenti, hogy a fordító csak a fordítási parancsban **−I** kapcsoló paramétereként megadott helyeken keresi a megfelelő header-öket.

A dinamikus könyvtárhoz tartozó **Makefile.am** szerkezete azonos, egyedül a target név és a target **PRIMARY**-je különbözik, a target név értelemszerűen más (libtool használata esetén a dinamikus könyvtárak kiterjesztése **.la**), a **PRIMARY**-t viszont **LTLIBRARIES**-re módosult, ugyanis dinamikus könyvtárat hozunk létre, a libtool használatával, ennek jele az **LT** a **LIBRARIES** előtt.

Lássuk a statikusan linkelt alkalmazáshoz készült **srcsa/Makefile.am**-et. Az előzőek fényében a **mainStatic** target és forrásfájljainak megadása világos. A forrásokon túl azonban ebben az esetben könyvtárat is rendelünk a target-hez

```
targetNev_LDADD = konyvtar
```

formában. A könyvtárat ezúttal statikus elérési úttal adjuk meg, azonban teljesen dinamikussá is tehetnénk a konfigurációs script-eket, a **configure** által beállított változókkal. Erre ebben az egyszerű tutorial-ban nem térünk ki. A fordító kapcsolók a statikus könyvtár header-jének elérési útjában különböznek csak a statikus és dinamikus könyvtárnál használt kapcsolóktól.

A dinamikus könyvtárhoz linkelt alkalmazáshoz tartozó **srcda/Makefile.am** szerkezete azonos, az egyetlen különbség, hogy a statikus helyett a dinamikus könyvtár elérési útjait és nevét adjuk meg. Máris látható az autotools használatának előnye, a GNU Makefile-okhoz képest, ugyanis nem kell foglalkoznunk a linkelés módjának beállításával, a statikus könyvtárakat statikusan, a dinamikusakat dinamikusan linkeli az előálló Makefile alapján a linker.

Az statikus és dinamikus könyvtárhoz is linkelt alkalmazás **srcsda/Makefile.am**-je megintcsak azonos a másik két alkalmazásával, az egyetlen különbség, hogy a fordító kapcsolók között mindkét könyvtár elérési útjait megadjuk és a targethez linkelendő könyvtárak felsorolásában is szerepel mind a statikus, mind a dinamikus könyvtár.

Most, hogy értelmeztük a konfigurációs állományokat, lássuk, hogyan fordíthatjuk le a könyvtárakat és az alkalmazásokat!

A létrehozott konfigurációs állományokból első feladatként a már sokszor említett **configure** script-et kell létrehoznunk. Erre több mód is kínálkozik. Korábbi autotools eszközöknél több különböző script hívására volt szükség a **configure** legfinomabb beállításához. Ez a hívási lánc **aclocal** => **autoheader** => **autoconf** => **automake** most is használható, azonban a lépések értelmezése, paraméterezése és magyarázata messzire vezetne, ezért helyette egy kis projekteknél jól használható script-et indítunk, amely a fenti lépéseket megfelelő alapértelmezett paraméterekkel foglalja magában. Adjuk ki tehát a **testLibraries** gyökérkönyvtárban az alábbi parancsot:

```
~/testLibraries$ autoreconf --install
```

A hívás számos fájlt és könyvtárat hoz létre mind a gyökérkönyvtárban, mind az alkönyvtárakban:

```
~/testLibraries$ ls
aclocal.m4      config.h.in  configure.ac  ltmain.sh    Makefile.in  srcdl   srcsl
autom4te.cache  config.sub   depcomp       m4           missing      srcsa
config.guess    configure    install-sh    Makefile.am  srcda        srcsda
```

A létrejött fájlokkal nem kell foglalkoznunk, az autotools automatikusan hozza létre és használja őket. Látható azonban, hogy a futtatható jogokkal rendelkező **configure** script is létrejött. Futtassuk le előbb a −−**help** kapcsolóval.

```
~/testLibraries$ ./configure --help
```

Látható, hogy a **configure** script számos kapcsolóval rendelkezik, amelyekkel a **Makefile.am**-ekben használt változóknak adhatunk értéket. A legfontosabb talán a −−**prefix** kapcsoló, amely azt a könyvtár prefix-et adja meg, amely a targetek megadásánál a **where** rész elé kerül. Ha tehát a **configure** script-et −−**prefix=/home/gykovacs** kapcsolóval hívjuk meg, akkor a későbbi **make install** hatására a bináris programok a **/home/gykovacs/bin**, a könyvtárak a **/home/gykovacs/lib** könyvtárba kerülnek. A −−**prefix** alapértelmezése a **/usr** könyvtár, ami általában megfelelő.

Ha most a **configure** script-et a −−**help** kapcsoló nélkül indítjuk, lefut a rendszer ellenőrzése, láthatjuk, hogy számos alapvető dolgot ellenőriz a script, mint például a C fordító működése, próbafordítással. A script a **Makefile.am**-ek felhasználásával előállítja a **Makefile**-okat minden olyan helyen, amit felsoroltunk a **configure.ac**-ben.

Kiadva a **make** parancsot a testLibraries gyökérkönyvtárban, statikus és dinamikus könyvtárunk valamint az alkalmazások is lefordulnak. Érdemes megnézni a könyvtárak linkelési parancsát, ami esetünkben egy-egy **libtool** hívás, amelynek a korábban parancssori linkelésnél használt **gcc** hívás is paramétere.

Lássuk, milyen fontos target-ek jöttek létre a Makefile-okban:

- **make** paraméter nélkül ekvivalens a **make all** hívással, azaz minden könyvtárat és alkalmazást elkészít;
- **make clean** letörli a fordítás során létrehozott már nem szükséges .o object fájlokat;
- **make distclean** letörli a **configure** script által létrehozott fájlokat;
- **make install** installálja a lefordított könyvtárakat és alkalmazásokat a **Makefile.am**-ekben megadottaknak megfelelően;
- **make uninstall** letörli az installált fájlokat;

- **make dist** létrehozza a könyvtárak és alkalmazások forráskódként történő disztribútolására alkalmas .tar.gz csomagot;
- **make distcheck** létrehozza a könyvtárak és alkalmazások forráskódként történő disztribútolására alkalmas .tar.gz csomagot, majd kitömöríti, és teszteli, hogy tényleg fordítható-e, azaz minden fájl benne van-e, ami szükséges.

Indítsuk most a **make**-et a **distcheck** targettel. Ha hibátlanul működik minden, előállt a testLibraries-1.0.tar.gz csomag. A csomag neve és verziószáma a **configure.ac**-ban foglaltaknak megfelelően áll elő. Ha belenézünk .tar.gz állományba, kitömörítve, vagy Midnight Commander használatával, megtalálhatjuk a forrásainkat, a konfigurációs állományokat, valamint a **configure** scriptet és a **configure** létrehozása során létrejött fájlokat. Nagyon fontos tehát, hogy autotools eszközökkel történő disztribúció során mindig adnunk kell a **configure** scriptet, azonban ha jól építjük fel a könyvtárunkat menedzselő konfigurációs fájlokat, a végső csomag automatikusan előáll. Ha valaki letölt egy autotools eszközkkel disztribútolt csomagot, akkor találnia kell benne egy **configure** script-et, amelyet aztán a korábbiakhoz hasonlóan esetleges kapcsolókkal futtatva kaphatjuk meg a **Makefile**-okat, amelyeket aztán felhasználhatunk a fordításhoz.

6.3.2. Fordítási opciók

Lássuk most, hogyan adhatunk a **configure** script-hez egy saját kapcsolót, a debug/release fordítás vezérlésére, jelenleg ugyanis minden **−O0** optimalizálással fordult le.

Mivel a **configure** script-hez szeretnénk új opciót adni, azt a **configure.ac**-ban kell specifikálni, hiszen a **configure.ac** alapján áll elő a **configure** script.

Adjuk hozzá az alábbi sorokat a **configure.ac** fájlhoz, a **AC_PROG_LIBTOOL** és a **CFLAGS=""** utasítások közé!

```
AC_ARG_ENABLE([debug],
[  --enable-debug Turn on debugging],
[case "${enableval}" in
  yes) debug=true ;;
  no)  debug=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-debug]) ;;
esac])
AM_CONDITIONAL(DEBUG, [test x$debug = xtrue])
```

A fenti sorok tulajdonképpen két utasítás kiadását takarják. Az **AC_ARG_ENABLE** három paramétert vár, az első az opció neve, amely kapcsolónévként a **−−enable−** prefix után megjelenik. Esetünkben a "debug" nevű opciót szeretnénk bekapcsolni, így az előálló **configure** script a **−−enable−debug** kapcsoló jelenléte esetén fogja a megfelelő beállításokat elvégezni. Az **AC_ARG_ENABLE** második paramétere egy sztring, ami a **configure −−help** hívás esetén megjelenik. A haramdik paraméter egy többirányú elágazás utasítás **bash** szintaktikával. A haramdik paraméter tehát az az utasítás, ami a **−−enable−elsoParameter** kapcsoló esetén végrehajtódik. A beépített **enableval** változó étéke alapján a **debug** változó értékét **true**-ra vagy **false**-ra állítjuk. Ha az **enableval** értéke nem **yes** vagy **no**, akkor hibaüzenetet írunk ki a konfigurálás során.

A következő utasításban a Linux bash-ban megszokott **test** programmal ellenőrizzük, hogy az őt követő logikai feltétel teljesül-e, azaz a **debug** változó értéke egyenlő-e **true**-val. A **debug** változó értékét a **$**-el érhetjük el, az **x**-re pedig azért van szükség, mert mivel sztringhelyettesítés után hívódik meg a **test** program, ha a **debug** változó üres, akkor **x** nélkül egy szintaktikailag helytelen **test** paraméterezést kapunk. Makró környezetekben jól bevált szokás a sztringek egyenlőségének ilyen formájú ellenőrzése. **x** helyett tetszőleges karaktersorozat állhat az egynelőség jel két oldalán. Ha az **AM_CONDITIONAL** második paramétere igaz, akkor a **DEBUG** változóba is igaz érték kerül, ellenkező esetben hamis

érték. A továbbiakban az így létrejött **DEBUG** változót használjuk fel a fordítási kapcsolók megfelelő megadására.

A létrejövő DEBUG változó az **Makefile**-ok előállítása során használható, hiszen a **configure** script futásakor állítódik be az értéke, így tehát a **Makefile.am**-ekben kell megadnunk olyan feltételes szerkezeteket, amelyekkel az egyes **Makefile**-ok **−O0 −g** vagy **−O3** fordítási flag-ekkel jönnek létre.

Hozzunk létre a **testLibraries** gyökérkönyvtárban egy **common.mk** nevű fájlt az alábbi tartalommal:

6.3.16. forráskód: common.mk

```
if DEBUG
MYCFLAGS = -O0 -g
else
MYCFLAGS = -O3
endif
```

A **common.mk** tartalma könnyen végiggondolható. Ha a **DEBUG** változó tartalma igaz, a **MYCFLAGS** változó értékeként beállítjuk a **−O0** kapcsolót, ellenkező esetben a **MYCFLAGS** tartalma a teljes optimalizálást jelentő **−O3** kapcsoló.

A következő és egyben utolsó lépés, hogy ezt a kódrészletet minden **Makefile.am**-be beletegyük, amelyre a **−−enable−debug** kapcsolót értelmezni szeretnénk. Ehhez egyszerűen minden könyvtárban (a gyökérkönyvtárban nem szükséges) a **Makefile.am** elejére írjuk be a **common.mk** "include"-olását megvalósító parancsot, és emellett minden **Makefile.am**-ben módosítsuk az **AM_CFLAGS** beállítását a **MYCFLAGS** értékének hozzáadásával, azaz például a statikus **srcsl** könyvtárhoz tartozó **Makefile.am** a következőt fogja tartalmazni:

6.3.17. forráskód: srcsl/Makefile.am

```
include ../common.mk

lib_LIBRARIES = libtestStaticLibrary.a
libtestStaticLibrary_a_SOURCES = testStaticLibrary.c
include_HEADERS = testStaticLibrary.h

AM_CFLAGS = ${MYCFLAGS} -I.
```

Miután a fenti módosításokat végrehajtottuk, hozzuk létre újra a **configure** script-et a **autoreconf −−install** hívással. Ha most lefuttatjuk a **configure** script-et a **−−help** paraméterrel, látnunk kell az új, saját **−−enable−debug** kapcsolónkat:

```
~/testLibraries\$ ./configure --help
...
  --disable-dependency-tracking  speeds up one-time build
  --enable-dependency-tracking   do not reject slow dependency extractors
  --disable-libtool-lock  avoid locking (might break parallel builds)
  --enable-debug Turn on debugging
...
```

Ha most a **configure** script-et a **−−enable−debug** kapcsolóval hívjuk és utána kiadjuk a **make** utasítást, a fordítási parancsokban megjelenik a **−O0** kapcsoló, ha azonban a **−−enable−debug** nélkül hívjuk, a **−O3** kapcsoló jelenik meg a fordítási parancsokban.

Ügyeljünk arra, hogy az autotools eszközökkel is csak akkor történik fordítás, ha szükséges, azaz valamely forrás módosul. Ha ki szeretnénk próbálni a **configure** különböző hívásai közötti különbséget, a fordítás előtt adjuk ki a **make clean** parancsot, hogy a korábbi fordítás eredményeként előállt fájlok törlődjenek. Ezt követően fog a **make** utasítás hatására ténylegesen fordítani a rendszer.

6.3. Feladat: A fenti példa alapján módosítsuk a **configure.ac** és **common.mk** állományokat úgy, hogy lehetőség legyen benne a warning-ok bekapcsolására valamint az opcionálisan a **mainOptional** fájlba fordítandó dinamikus könyvtárbeli függvényhívás bekapcsolására!

6.3. Megoldás:

6.3.18. forráskód: configure.ac

```
AC_INIT([testLibraries], [1.0], [gyuriofkovacs@gmail.com])
AC_CONFIG_MACRO_DIR([m4])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
LT_INIT
AC_PROG_CC
AC_PROG_RANLIB
AC_PROG_LIBTOOL

AC_ARG_ENABLE([debug],
[  --enable-debug Turn on debugging],
[case "${enableval}" in
  yes) debug=true ;;
  no)  debug=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-debug]) ;;
esac],[debug=false])
AM_CONDITIONAL(DEBUG, [test x$debug = xtrue])

AC_ARG_ENABLE([warnings],
[  --enable-warnings Turn on warnings],
[case "${enableval}" in
  yes) warnings=true ;;
  no)  warnings=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-warnings]) ;;
esac],[warnings=false])
AM_CONDITIONAL(WARNINGS, [test x$warnings = xtrue])

AC_ARG_ENABLE([optional],
[  --enable-optional Turn on the optional function call in mainOptional],
[case "${enableval}" in
  yes) optional=true ;;
  no)  optional=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-optional]) ;;
esac],[optional=false])
AM_CONDITIONAL(OPTIONAL, [test x$optional = xtrue])

CFLAGS=""

AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile srcsl/Makefile srcsa/Makefile srcdl/Makefile srcda/Makefile
    srcsda/Makefile srcoa/Makefile])
AC_OUTPUT
```

6.3.19. forráskód: common.mk

```
if DEBUG
MYCFLAGS = -O0 -g
else
MYCFLAGS = -O3
endif
if WARNINGS
MYCFLAGS += -Wall -Wextra
endif
if OPTIONAL
MYCFLAGS += -DUSE_DYNAMIC_LIBRARY
endif
```

Az autotools magasszintű eszközöket biztosít arra, hogy a **config.h**-n keresztül történő ***#define*** bekapcsolást véghez vigyük. Elsőként távolítsuk el a **common.mk**-ból a **−DUSE_DYNAMIC_LIBRARY** bekapcsolását. Adjuk értékül a **configure.ac** megfelelő opciójában az új **DEFINE_DIRECTIVE** változónak a **config.h**-ba belerakni kívánt direktívát sztringként. A **configure.ac**-ben ki kell még adnunk az **AC_SUBST** parancsot, a **DEFINE_DIRECTIVE** paraméterrel. Ezen parancs azt jelzi az automake környezetnek, hogy a **DEFINE_DIRECTIVE** változót is helyettesíteni kell az értékével a konfigurációs fájlokban. Hozzuk létre a **config.h.in** template-et, amely a **DEFINE_DIRECTIVE** változó értékére hivatkozik, majd adjuk hozzá a **config.h** header-t a létrehozandó konfigurációs fájlok listájához, azaz az alábbi módon módosulnak az állományaink:

6.3.20. forráskód: configure.ac

```
AC_INIT([testLibraries], [1.0], [gyuriofkovacs@gmail.com])
AC_CONFIG_MACRO_DIR([m4])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
LT_INIT
AC_PROG_CC
AC_PROG_RANLIB
AC_PROG_LIBTOOL

AC_ARG_ENABLE([debug],
[  --enable-debug Turn on debugging],
[case "${enableval}" in
  yes) debug=true ;;
  no)  debug=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-debug]) ;;
esac],[debug=false])
AM_CONDITIONAL(DEBUG, [test x$debug = xtrue])

AC_ARG_ENABLE([warnings],
[  --enable-warnings Turn on warnings],
[case "${enableval}" in
  yes) warnings=true ;;
  no)  warnings=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-warnings]) ;;
esac],[warnings=false])
AM_CONDITIONAL(WARNINGS, [test x$warnings = xtrue])

AC_ARG_ENABLE([optional],
[  --enable-optional Turn on the optional function call in mainOptional],
[case "${enableval}" in
  yes) optional=true
  DEFINE_DIRECTIVE="#define USE_DYNAMIC_LIBRARY";;
  no)  optional=false ;;
  *) AC_MSG_ERROR([bad value ${enableval} for --enable-optional]) ;;
esac],[optional=false])
AM_CONDITIONAL(OPTIONAL, [test x$optional = xtrue])

CFLAGS=""

AC_CONFIG_HEADERS([config.h])
AC_SUBST([DEFINE_DIRECTIVE])
AC_CONFIG_FILES([Makefile srcsl/Makefile srcsa/Makefile srcdl/Makefile srcda/Makefile
    srcsda/Makefile srcoa/Makefile srcoa/config.h])
AC_OUTPUT
```

6.3.21. forráskód: common.mk

```
if DEBUG
MYCFLAGS = -O0 -g
else
MYCFLAGS = -O3
```

```
endif
if WARNINGS
MYCFLAGS += -Wall -Wextra
endif
```

6.3.22. forráskód: srcoa/config.h.in

```
@DEFINE_DIRECTIVE@
```

Az **configure** fájl **enable−optional** opcióval való futtatása után létrejön az **srcoa/config.h** állomány, amely éppen a megfelelő ***#define*** direktívát tartalmazza.

Összefoglalva tehát az autotools professzionális eszközöket biztosít források menedzselésére, azonban nagyobb projektek esetén a makrók használata miatt nehézkes lehet. Bár nagyon sok disztribúcióban használják, javaslom a későbbiekben ismertetendő **cmake** vagy **qmake** eszközök használatát, amelyek tovább egyszerűsítik a projektek menedzselését. Figyeljük meg, hogy ez utóbbi megoldásnál a **−DUSE_DYNAMIC_LIBRARY** kapcsoló bekerül minden fájl fordításának parancsába. Ez nem jelent problémát a fordítás során.

6.3.3. Külső könyvtárak használata

Külső könyvtárak használatának demonstrálásához adjuk hozzá a korábbi **mainPNG.c** forrásfájlt az aktuális környezetünkhöz, például az **srcpng** könyvtárba, és hozzunk létre hozzá egy megfelelő, egyszerű **Makefile.am**-et.

Egy csomag meglétét ezt követően a **configure.ac**-ba elhelyezett **PKG_CHECK_MODULES** utasítással ellenőrizhetjük. Működését tekintve két kötelező paramétert vár, az első paramétere egy sztring, a második paramétere pedig egy könyvtár (csomag) neve. Esetünkben a sztring **LIBPNG**, a csomagnév **libpng** lesz. Megkeresi a rendszeren (az alapértelmezett, és a **PKG_CONFIG_PATH** változóban megadott elérési utakon) a **libpng.pc** állományt. Amennyiben van ilyen, definiál egy **LIBPNG_CFLAGS** változót, amely a fordítási flag-eket tartalmazza, valamint egy **LIBPNG_LIBS** változót, ami a linker flageket fogja tartalmazni. Ezt követően a megfelelő **Makefile.am**-ekben csak ezen változók értékére kell hivatkoznunk a megfelelő fordítási és linkelési paranacsok összerakásához. A **configure.ac**-ba kerülő utasítás tehát:

```
PKG_CHECK_MODULES([LIBPNG],[libpng])
```

míg a **mainPNG.c** forrás **Makefile.am**-e az alábbi módon alakul:

6.3.23. forráskód: srcpng/Makefile.am

```
include ../common.mk

bin_PROGRAMS = mainPNG
mainPNG_SOURCES = mainPNG.c

AM_CFLAGS = ${MYCFLAGS} -I. -I$(srcdir)/../srcdl -I$(srcdir)/../srcsl ${LIBPNG_CFLAGS}

mainPNG_LDADD = ../srcdl/libtestDynamicLibrary.la \
  ../srcsl/libtestStaticLibrary.a \
  ${LIBPNG_LIBS}
```

6.3.4. Disztribúciók létrehozása

Az **automake** rendszer a **.tar.gz** jellegű, forráskódot és konfigurációs szkriptet tartalmazó disztribúciók létrehozását támogatja. Ahhoz azonban, hogy könyvtárunkat más könyvtárak is használhassák, szükség van arra, hogy a korábban már említett package-config fájlok is előálljanak a könyvtárunkhoz, hiszen más szoftverek ezen .pc fájlokon keresztül találják meg könyvtárunkat a rendszeren.

A package-config fájlok azonban konkrét elérési utakat tartalmaznak, amelyek csak egy adott gép esetén érvényesek, így nem készíthetünk egzakt .pc fájlokat előre. A megoldás hasonló a **config.h** header létrehozásához: készítünk egy .pc template-et, és azt felvesszük a **configure.ac** fájlban a konfiguráció során létrehozandó fájlok listájába. A package-config fájl template például az alábbi lehet teszcsomagjaink esetén:

6.3.24. forráskód: testlibs.pc.in

```
prefix=@prefix@
exec_prefix=@prefix@
libdir=@prefix@/lib
includedir=@prefix@/include

Name: testlibs
Description: Test libraries
Version: 0.0.1
Libs: -L@prefix@/lib -ltestStaticLibrary -ltestDynamicLibrary
Cflags: -I@prefix@/include
```

6.4. CMake

6.4.1. Egyszerű CMake környezet kialakítása

A **cmake** eszközrendszer segítségével magas szintű konfigurációs állományokból kiindulva szinte tetszőleges IDE-hez generálhatunk projektfájlokat, beleértve a GNU Makefile-okat is. A **cmake** használatát is az eddigi egyszerű statikus és dinamikus könyvtárakon és alkalmazásokon keresztül szemléltetjük.

Hozzuk létre az alábbi könyvtárszerkezetet, üres **CMakeLists.txt** állományokkal:

```
cmaketest
     srcda
         CMakeLists.txt
         mainDynamic.c
     srcdl
         CMakeLists.txt
         testDynamicLibrary.c
         testDynamicLibrary.h
     srcsa
         CMakeLists.txt
         mainStatic.c
     srcsda
         CMakeLists.txt
         mainStaticAndDynamic.c
     srcsl
         CMakeLists.txt
         testStaticLibrary.c
         testStaticLibrary.h
     CMakeLists.txt
```

Az egyes **CMakeLists.txt** állományok tartalma a következő:

6.4.25. forráskód: CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(CMAKE_C_COMPILER gcc)

PROJECT(cmaketest C)

SET(PACKAGE_NAME "cmaketest")
SET(MAJOR_VERSION "0")
SET(MINOR_VERSION "0")
SET(PATCH_VERSION "1")

SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SUBDIRS(srcsl srcdl srcsa srcda srcsda)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A gyökérkönyvtárban található **CMakeLists.txt** állomány szerepe hasonló az autotools **Makefile.am**-jéhez, azaz elsősorban azokat az alkönyvtárakat sorolja fel, amelyekben további **CMakeLists.txt** állományok találhatók, és amelyeket a fordítás során fel kell dolgozni.

Az első utasításban a minimális cmake verziót adjuk meg, amellyel a konfigurációs állomány használandó. A cmake 2.6 és 2.8 között számos különbség van, esetünkben ragaszkodunk a legalább 2.8-as verziószámú cmake-hez. A második utasítással a C fordítót adjuk meg explicit módon, ugyanis több különböző C fordító is lehet installálva. A haramdik utasítás a projekt definiálása, a **PROJECT** függvény első paramétere a projekt neve, második paramétere a projekt nyelve. C++ esetén a projekt nyelvét **CXX**-ként kell megadni. Ezt követően négy változó beállítása következik. Cmake-ben a változókhoz értéket a **SET** függvénnyel rendelhetünk, amelynek első paramétere a változónév, második paramétere pedig a változó értéke. A négy változó a csomag nevét és verziószámát hordozza. A következő utasításban a **PACKAGE_VERSION** változó értékeként összerakjuk a fő-, al- és patch-verziószámokat. Ezt követően a **SUBDIRS** függvény paramétereiként felsoroljuk a további feldolgozandó könyvtárakat. Utolsó utasításként bekapcsoljuk a beszédes Makefile-ok funkciót, azaz a **CMAKE_lstinlineOSE_MAKEFILE** változó értékét **on**-ra állítjuk.

6.4.26. forráskód: srcsl/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME testStaticLibrary)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 2)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_COMPILER gcc)

PROJECT(testStaticLibrary C)

AUX_SOURCE_DIRECTORY(. SRCVAR)

ADD_LIBRARY(${PACKAGE_NAME} ${SRCVAR})

INCLUDE_DIRECTORIES(.)

SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A statikus könyvtár **CMakeLists.txt** állománya sokban hasonlít a gyökérkönyvtárban létrehozott konfigurációs állomány tartalmához. A cmake minimális verziószámának megadásával kezdődik, majd definálunk a c nyelvű projektet, ezt követően beállítjuk a csomag nevét és verzióját tartalmazó változókat, és a fordítót. Az **AUX_SOURCE_DIRECTOR** függvény két paraméterrel rendelkezik. Az első paramétere egy könyvtárnév, a második egy változó neve. Összegyűjti az első paraméterként megadott könyvtárban található a projekt nyelvének megfelelő forrásfájlok neveit (esetünkben .c forrásfájlokat), és berakja nevüket az **SRCVAR** változóba. Az **SRCVAR** változó tartalmazza tehát azokat a forrásfájlokat, amelyeket le akarunk fordítani. Az **ADD_LIBRARY** függvénnyel adhatunk hozzá library target-et a projekthez. A függvény első paramétere a könyvtár neve, ami esetünkben a csomagnév, további paraméterei pedig a könyvtárhoz lefordítandó forrásfájlok nevei, amelyet az **SRC_VAR** változóba már összegyűjtöttünk. Az **INCLUDE_DIRECTORIES** függvényhívás paraméterei azok a könyvtárak, amelyek -I kapcsolóval bekerülnek a fordítási sorba. Ezt követően tulajdonságokat beállítjuk a **PUBLIC_HEADER** tulajdonságot a **PACKAGE_NAME** változó értéke targethez. A **SET_PROPERTY** paraméterezése a következő: az első paramétere annak az eszköznek a típusa, amely valamely tulajdonságát be szeretnénk állítani. Második paramétere az eszköz konkrét neve, ezt követi a **PROPERTY** kulcsszó, majd a beállítandó tulajdonság neve, és ezt követően a tulajdonság értéke. Esetünkben a **PUBLIC_HEADER** tulajdonság kapja meg a **testStaticLibrary.h** értéket, azaz a könyvtárhoz tartozó publikus header fájlt. Ez a fájl bele kell, hogy kerüljön az elkészülő csomagba ahhoz, hogy a könyvtárat használni lehessen. Az **INSTALL** függvénnyel állítjuk be az installálás paramétereit. Első paramétere az eszközök neve, amelyek installálását be szeretnénk állítani, esetünkben **TARGETS**, ezt követik a beállítandó target-ek nevei, esetünkben ez a könyvtárnév, amely neve a csomag név változóban érhető el. Ezt követően az egyes target típusokhoz állíthatjuk be **TARGET_TIPUS DESTINATION konyvtarnev** módon, hogy az installálás fő könyvtárához relatívan hová kerüljenek. A Linux rendszerek konvenciójának megfelelően a könyvtárak az installálás **lib** alkönyvtárába, a header-ök az **include** alkönyvtárba, míg az archívumok a szintén a **lib** alkönyvtárba kerülnek. Utolsó utasításként a **Makefile**-okat beszédesre állítjuk.

6.4.27. forráskód: srcdl/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME testDynamicLibrary)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 1)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_COMPILER gcc)

PROJECT(testDynamicLibrary C)

AUX_SOURCE_DIRECTORY(. SRCVAR)

ADD_LIBRARY(${PACKAGE_NAME} ${SRCVAR})

INCLUDE_DIRECTORIES(.)

SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A dinamikus könyvtárhoz tartozó konfigurációs állomány minimálisan tér el a statikus könyvtárétól. A "statikus" szó értelemszerű "dinamikusra" történő átírása mellett az **ADD_LIBRARY** függvényben a **SHARED** kulcsszóval specifikáljuk, hogy dinamikus könyvtárat hozunk létre, minden más utasítás és célja megegyezik a statikus könyvtárral.

6.4.28. forráskód: srcsa/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME mainStatic)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 1)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

#SET(CMAKE_C_COMPILER gcc)

PROJECT(${PACKAGE_NAME} C)

LINK_DIRECTORIES(../srcsl)

INCLUDE_DIRECTORIES(. ../srcsl)

AUX_SOURCE_DIRECTORY(. SRCVAR)
ADD_EXECUTABLE(${PACKAGE_NAME} ${SRCVAR})

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} testStaticLibrary)

SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A statikusan linkelendő alkalmazás **CMakeLists.txt** állománya is hasonlóan épül fel a könyvtárakéhoz. A különbség a **LINK_DIRECTORIES** függvény, amely paraméterei a fordítási sorba -L kapcsolóval kerülnek be, azaz itt keresi a fordító a -l kapcsolóval megadott könyvtárakat. Az **INCLUDE_DIRECTORIES** függvény parméterei közé bekerül most a statikus könyvtár elérési útja is, hiszen itt találjuk azt a header-t, amellyel a statikus könyvtárat használni tudjuk. A futtatható állomány target-eket az **ADD_EXECUTABLE** függvénnyel adhatjuk meg, amely paraméterezése azonos az **ADD_LIBRARY** függvény paraméterezésével. Ezt követően a **TARGET_LINK_LIBRARIES** függvénnyel adhatjuk meg azokat a könyvtárakat, amelyek az első paraméterként megadott target linkelési utasításába a -l kapcsolóval kerülnek be. Mivel a target most csak egy futtatható állományból áll, az **INSTALL** függvényben csak a futtatható azaz **RUNTIME** típusú target-ekhez kell megadnunk az installálás célkönyvtárát.

6.4.29. forráskód: srcda/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME mainDynamic)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 2)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_FLAGS "-fopenmp -Wall -Wextra -fPIC")
SET(CMAKE_C_FLAGS_DEBUG "-g -O0")
SET(CMAKE_C_FLAGS_RELEASE "-O2")

PROJECT(${PACKAGE_NAME} C)

LINK_DIRECTORIES(../srcdl)

INCLUDE_DIRECTORIES(. ../srcdl)

AUX_SOURCE_DIRECTORY(. SRCVAR)
ADD_EXECUTABLE(${PACKAGE_NAME} ${SRCVAR})

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} testDynamicLibrary)
```

```
SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A dinamikus alkalmazás szerkezete teljesen azonos, a "statikus" és "dinamikus" szavak értelemszerű cseréjével, a statikus és dinamikus linkelés közötti különbséget a cmake eszközök teljesen elfedik.

6.4.30. forráskód: srcsda/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME mainStaticAndDynamic)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 2)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_COMPILER gcc)

PROJECT(${PACKAGE_NAME} C)

LINK_DIRECTORIES(../srcsl ../srcdl)

INCLUDE_DIRECTORIES(. ../srcsl ../srcdl)

AUX_SOURCE_DIRECTORY(. SRCVAR)
ADD_EXECUTABLE(${PACKAGE_NAME} ${SRCVAR})

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} testStaticLibrary testDynamicLibrary)

SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A statikusan és dinamikusan linkelt alkalmazás konfigkurációs állományának szerkezete ugyanaz, mint az előző két alkalmazásé, a különbség, hogy a linkelendő library-k könyvtárai és az include könyvtárak között felsoroljuk mind a statikus, mind a dinamikus könyvtár elérési útját, valamint a linkelendő library-k listája most mind a statikus, mind a dinamikus könyvtárat tartalmazza.

Lássuk, hogyan használhatjuk a konfigurációs állományainkat.

A **cmake** program első parancssori argumentuma azon könyvtár elérési útja, amely a konfigurálandó **CMakeLists.txt** állományt tartalmazza. A **cmaketest** gyökérkönyvtárban állva tehát ha kiadjuk a **cmake . ** utasítást, a gyökérkönyvtárban található **CMakeLists.txt** alapján elkészül a GNU szabványú **Makefile**, valamint mivel a gyökérkönyvtár konfigurációs állományában további könyvtárakat adtunk meg a **SUBDIRS** függvény paramétereként, azokban is végbemegy a konfiguráció, azaz minden könyvtárban előáll egy **Makefile** állomány. A gyökérkönyvtárban kiadva most a **make** utasítást, a könyvtáraink és alkalmazásaink lefordulnak és linkelődnek.

A **cmake** −**h** utasítást kiadva áttekinthetjük, milyen opciói vannak a **cmake** alkalmazásnak. Az áttekintés utolsó szekciója azt sorolja fel, milyen IDE-khez készíthetünk projekt fájlokat. Linux rendszeren a

```
Unix Makefiles               = Generates standard UNIX makefiles.
CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.
Eclipse CDT4 - Unix Makefiles
                             = Generates Eclipse CDT 4.0 project files.
KDevelop3                    = Generates KDevelop 3 project files.
KDevelop3 - Unix Makefiles   = Generates KDevelop 3 project files.
```

generátorok közül válogathatunk, azaz például KDevelop3 fejlesztőkörnyezethez létrehozhatunk projekt fájlokat a

```
cmake -G "KDevelop3" .
```

utasítással.

6.4.2. *Fordítási opciók*

A **CMakeLists.txt** fájlokban számos utasítás és vezérlési szerkezet áll rendelkezésünkre, hogy a konfigurációt megfelelően végrehajthassuk. A cmake paraméterezése parancssoron keresztül történik, a **cmake** utasítást követően **−D** kapcsolóval egybeírva tetszőleges változónevet megadhatunk és annak az **=** operátorral tetszőleges értéket be is állíthatunk. Ezt követően a **CMakeLists.txt** állományok úgy tekinthetők, mint egyfajta imperatív szkript, amelyekben a parancssorban megadott változók használhatóak. Ha nem adunk meg egy változónevet és mégis használjuk a szkriptben, akkor annak értéke üres lesz.

Egészítsük ki most konfigurációs állományainkat úgy, hogy a debug/release konfigurációt a **cmake** utasítás kapcsolóival szabályozni tudjuk. Ehhez minden forrást tartalmazó könyvtár **CMakeLists.txt** állományát egészítsük ki a következő sorokkal:

```
SET(CMAKE_C_FLAGS_DEBUG "-g -O0")
SET(CMAKE_C_FLAGS_RELEASE "-O3")
```

A **CMAKE_C_FLAGS_DEBUG** egy speciális változó, amely a debug fordítás kapcsolóit tartalmazza, míg a **CMAKE_C_FLAGS_RELEASE** változó a release fordítás kapcsolóit. Az egyes fordítási módokat a **cmake** parancsban definálható **CMAKE_BUILD_TYPE** változó **debug** vagy **release** értékével állíthatjuk be. Ha a konfigurációs parancsban definálunk változókat, azokat értékükkel együtt a -D kapcsolóval egybeírva kell megadni, azaz debug fordításhoz az alábbi módon kell konfigurálnunk a projektünket:

```
cmake -DCMAKE_BUILD_TYPE=debug .
```

További, bonyolultabb opciók is megadhatók parancssori változók definiálásával és a **CMakeLists.txt** állományokban elhelyezhető **if** szerkezetek segítségével: CMake-ben az elágaztató utasítások szerkezete

```
if(kifejezes)
  [utasitasok]...
[elseif(kifejezes2)
  [utasitasok]...]...
[else(kifejezes)
  [utasitasok]...]
endif(kifejezes)
```

A fenti szintaktikában a **kifejezes** mindhárom előfordulása ugyanazt a kifejezést jelenti. A kifejezés lehet egy egyszerű változó is. Ebben a kifejezés hamis, ha a változó értéke *üres*, **0**, **N**, **NO**, **OFF**, **FALSE** , **NOTFOUND** vagy <valtozonev>−**NOTFOUND**. Minden más esetben a kifejezés igaz lesz. Lássuk, hogyan állíthatjuk be parancssorból a debug/release fordítást a beépített **CMAKE_C_FLAGS_DEBUG** és **CMAKE_C_FLAGS_RELEASE** változók nélkül. Legyen a parancssori argumentum neve **ENABLE_DEBUG** . Ekkor az alábbi feltételes szerekezetet elhelyezve a **CMakeLists.txt** állományokban, éppen a kívánt működést kapjuk:

```
IF ( ENABLE_DEBUG )
  SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -g -O0")
ELSE (ENABLE_DEBUG )
  SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O3")
ENDIF( ENABLE_DEBUG )
```

A kódot elhelyezve azokban a **CMakeLists.txt** állományokban, amelyekben opcionálissá szeretnénk tenni a debug fordítást, a **cmake** parancsban a **−DENABLE_DEBUG=on** utasítással kapcsolhatjuk be azt.

6.4. Feladat: Adjon hozzá a **CMakeLists.txt** fájlokhoz olyan opciókat, amelyek a warningok és az opcionálisan linkelendő dinamikus könyvtár használatát kapcsolják be!

6.4. Megoldás: A warning-ok bekapcsolása a debug/release fordításhoz hasonlóan történik:

```
IF ( ENABLE_WARNINGS )
  SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall -Wextra")
ENDIF( ENABLE_WARNINGS )
```

A **mainOptional** alkalmazásba opcionálisan belefordítandó dinamikus könyvtárbeli függvényhívást az **ENABLE_OPTIONAL** változóval kapcsolhatjuk be:

```
IF ( ENABLE_OPTIONAL )
  SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -DUSE_DYNAMIC_LIBRARY")
ENDIF ( ENABLE_OPTIONAL )
```

A fenti feltételes szerkezeteket természetesen azon **CMakeLists.txt** állományokba kell beírnunk, amelyekben szeretnénk, hogy a kapcsolók hatása érvényesüljön. Ha a gyökérkönyvtárban lévő konfigurációs állományba írjuk be, és nem szerepel az alkönyvtárak konfigurációs állományaiban a **CMAKE_C_FLAGS** teljes felülírása értékadással, akkor a megfelelő kapcsolók minden alkönyvtár forrásfájljainak fordítási parancsába bekerülnek. Lássuk most, hogyan kapcsolható be a dinamikus könyvtár használata a **config.h** header-ön keresztül! Az **automake**-es megoldáshoz hasonlóan, itt is létre kell hoznunk egy template-et a konfigurációs állományhoz, azaz az alábbi **config.h.in** állományt, és include-oljuk a **config.h**-t a **mainOptional.c** forráskódjának első sorában.

6.4.31. forráskód: config.h.in

```
@DEFINE_DIRECTIVE@
```

A konfigurációs állományok konkretizálását a **CONFIGURE_FILE** utasítással végezhetjük el. Első kötelező paramétere egy template fájlnév (esetünkben a fenti **configure.h.in** állomány), második paramétere pedig a template konfigurációs állományból előálló állomány neve. A megfelelő **CMakeLists.txt** fájlban (például abban, amely az **srcoa** könyvtárban van), az alábbi kódrészletet kell elhelyezni:

```
CONFIGURE_FILE(config.h.in config.h)
```

Ezt parancssorban a

```
cmake -DUSE_DYNAMIC_LIBRARY=yes .
```

utasítással kapcsolhatjuk be a konfigurációs állomány megfelelő tartalommal való feltöltését.

6.4.32. forráskód: srcoa/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME mainOptional)
SET(MAJOR_VERSION 0)
```

```
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 2)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_FLAGS "-fPIC")

PROJECT(${PACKAGE_NAME} C)

LINK_DIRECTORIES(../srcdl)

INCLUDE_DIRECTORIES(. ../srcdl)

IF ( USE_DYNAMIC_LIBRARY )
    SET(DEFINE_DIRECTIVE "#define USE_DYNAMIC_LIBRARY")
ENDIF ( USE_DYNAMIC_LIBRARY )

CONFIGURE_FILE(config.h.in config.h)

AUX_SOURCE_DIRECTORY(. SRCVAR)
ADD_EXECUTABLE(${PACKAGE_NAME} ${SRCVAR})

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} testDynamicLibrary)

SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A kapcsoló egy **IF**-ben van lekezelve, amennyiben parancssori kapcsoló igaz értéket hordoz, akkor a **DEFINE_DIRECTIVE** értékül kapja a tényleges megfelelő ***#define*** direktívát. Ezt követően a **CONFIGURE_FILE** utasítás megnyitja az első paraméterként kapott fájlt és abban minden változó helyére beírja a Cmake változó nevét, esetünkben a **DEFINE_DIRECTIVE** változó tartalmát, és a megváltozott fájlt kiírja a második paraméterként.

6.4.3. Külső könyvtárak használata

Adjuk hozzá build környezetünkhoz a korábban is használt **mainPNG.c** forrásfájlt és készítsünk hozzá egy egyszerű **CMakeLists.txt** állományt.

A CMake egyik nagy előnye, hogy a külső csomagok használatával járó gondokat (installált csomag megtalálása, flag-ek beállítása, stb.), magas szinten, interoperábilis módon oldja meg. A koncepció hasonló a **pkg−config** működési elvéhez, azonban cmake esetén a leíró állományokról nem a forráskód csomagok gondoskodnak, hanem a CMake készítői.

A makró, amelyet használnunk kell, a **FIND_PACKAGE**. Első paramétere a csomag neve, amelyet szeretnénk megtalálni, második paramétere pedig egy opcionális **REQUIRED** kulcsszó. Működését tekintve ha a CMake megtalálja a <PACKAGE> nevű csomagot, akkor az alábbi változókat definiálja:

- **<PACKAGE>_FOUND** - azaz csomag installálva van a rendszerre,
- **<PACKAGE>_INCLUDE_DIRS** vagy **<PACKAGE>_INCLUDES** - tartalma a csomag használatához szükséges header fájlok elérési útja,
- **<PACKAGE>_LIBRARIES** vagy **<PACKAGE>_LIBS** - tartalma a csomag library-k elérési útja és neve,
- **<PACKAGE>_DEFINITIONS** - esetleges további definíciók.

Egy megtalált csomag esetén tehát létrejönnek a fenti változók, amelyeket felhasználhatunk a csomag jelenlétének ellenőrzésére, és arra, hogy beállítsuk a megfelelő fordítási/linkelési kapcsolókat.

A csomagok megtalálásának mechanizmusa a package-config-gal ellentétben itt nem .pc fájlokkal,

hanem úgynevezett modulokkal operál. Minden modul egy-egy csomaghoz tartozik, (például a **libpng** csomaghoz tartozó modul neve **PNG**, és minden modulhoz tartozik egy fájl, mely nevének szerkezete: **Find<PACKAGE>.cmake**. Ezen fájlban lévő utasítások találják meg a tényleges header fájlokat és könyvtárakat a rendszeren, és végzik a korábban említett négy változó értékének beállítását. A leggyakrabban használt külső csomagokhoz a cmake fejlesztői előre megírták ezeket a modulokat, és azokat a **CMAKE_MODULE_PATH** környezeti változóban felsorolt elérési utakon keresi a cmake.

A **libpng** csomag használatához tehát a **mainPNG.c** alkalmazás konfigurációs fájljában az alábbi utasítást kell elhelyeznünk:

```
FIND_PACKAGE(PNG)
INCLUDE_DIRECTORIES(${PNG_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(${PACKAGE_NAME} ${PNG_LIBRARIES})
```

konfigurálva a **cmake** paranccsal, látható, hogy a cmake keresi és meg is találja a **libpng** könyvtárat, valamint annak előfeltételét, a **zlib** könyvtárat is, majd a **make** parancs kiadása után a fordítási és linkelési parancsban jól látható a megfelelő **−I** és **−l** kapcsolók megjelenése. Jegyezzük meg, hogy itt a linkelendő könyvtárak teljes elérési útja jelenik meg a **−l** kapcsolók mögött, így a **−L** kapcsolókra nincs szükség.

6.4.4. Disztribúció létrehozása

Lássuk, hogyan hozhatunk létre disztribútolható csomagokat a **cmake** segítségével! Több különböző típusú csomag létrehozására van lehetőségünk a **cpack** programmal, azonban előbb néhány, a disztribúcióval kapcsolatos adatot meg kell adnunk változókban a **CMakeLists.txt** állományokban. Helyezzük el a következő sorokat minden forrásfájlt tartalmazó könyvtár **CMakeLists.txt** állományának végén, megfelelő változó értékekkel:

```
SET(CPACK_PACKAGE_DESCRIPTION_SUMMARY "testStaticLibrary")
SET(CPACK_PACKAGE_VENDOR "Gyorgy Kovacs")
SET(CPACK_PACKAGE_CONTACT "gyuriofkovacs@gmail.com")
SET(CPACK_DEBIAN_PACKAGE_MAINTAINER "gyuriofkovacs@gmail.com")
SET(CPACK_PACKAGE_DESCRIPTION_SUMMARY "cmaketest$")
SET(CPACK_PACKAGE_VERSION_MAJOR "${MAJOR_VERSION}")
SET(CPACK_PACKAGE_VERSION_MINOR "${MINOR_VERSION}")
SET(CPACK_PACKAGE_VERSION_PATCH "${PATCH_VERSION}")

INCLUDE(CPack)
```

Mindemellett azt is meg kell határoznunk, hogy az egyes fájlok a tepelítés során hová kerüljenek. Linux környezetben a header fájlok általában egy **include** nevű könyvtárba kerülnek, a futtatható állományok valamely **bin** könyvtárba, míg a library-k egy **lib** nevű könyvtárba. Windows-on már más a helyzet, ugyanis a futtatható állományokat és a dinamikus könyvtárakat célszerű egymás mellé telepíteni. Ezen különbségek elfedésére és maguknak a célkönyvtáraknak a megadására használható az **INSTALL** makró:

```
INSTALL(TARGETS ${PACKAGE_NAME} RUNTIME DESTINATION bin LIBRARY DESTINATION lib ARCHIVE
    DESTINATION lib)
FILE(GLOB HEADERS "*.h")
INSTALL(FILES ${HEADERS} DESTINATION include)
```

Az első sor jelentése egyrészt, hogy a **${PACKAGE_NAME}** nevű target-et installálja **make install** hatására, másrészt ugyanazon makró hívással specifikáljuk, hogy a futásidőben használandó eszközök (alkalmazások/dinamikus könyvtárak) a **bin** könyvtárba kerüljenek, a library-k (dinamikus könyvtárak) a **lib** könyvtárba és az archivumok is a **lib** könyvtárba. Kicsit zavaró ugyan, de a fenti utasítás

elfedi a Windows/Linux környezetek különbözőségét, ugyanis linux-on csak az alkalmazások kerülnek a **RUNTIME** eszközök osztályába, a dinamikus könyvtárak pedig a **LIBRARY** kulcsszó után kijelölt helyre települnek, míg Windows-on mind az alkalmazások, mind a dinamikus könyvtárak a **RUNTIME** kulcsszó után szereplő célkönyvtárba kerülnek. A **FILE** makró első paramétere (**GLOB**) azt jelzi, hogy a forráskönyvtárban lévő összes alkönyvtárral is dolgozzon, miközben a második paraméterként megadott **HEADERS** változóba kigyűjti a harmadik paraméterként kapott reguláris kifejezésre illeszkedő fájlneveket. A harmadik sorban egy újabb **INSTALL** makró hívással a **FILES** kulcsszó használatával tetszőleges fájlokra megadhatjuk, hogy hová kerüljenek installás után. A header fájlok azért tartoznak az egyéb kategóriába, mert csak a fejlesztői csomagokhoz van rájuk szükség. Az **INSTALL** makrókban megadott könyvtárak relatív könyvtárak a **CMAKE_INSTALL_PREFIX** beépített változó értékéhez képest, amely alapértelmezésben Linux rendszereken a **/usr** értéket tartalmazza. Windows rendszeren létrehozott grafikus NSIS installer esetén az felhasználó által kiválasztott telepítési célkönyvtárhoz képest lesznek relatívak.

A statikusan és dinamikusan linkelt alkalmazás **CMakeLists.txt** állománya tehát a fenti bővítések után az alábbi módon fog kinézni:

6.4.33. forráskód: srcsda/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SET(PACKAGE_NAME mainStaticAndDynamic)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 2)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_COMPILER gcc)
SET(CMAKE_C_FLAGS_DEBUG "-g -O0")
SET(CMAKE_C_FLAGS_RELEASE "-O2")

PROJECT(${PACKAGE_NAME} C)

LINK_DIRECTORIES(../srcsl ../srcdl)

INCLUDE_DIRECTORIES(. ../srcsl ../srcdl)

AUX_SOURCE_DIRECTORY(. SRCVAR)
ADD_EXECUTABLE(${PACKAGE_NAME} ${SRCVAR})

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} testStaticLibrary testDynamicLibrary)

SET_TARGET_PROPERTIES(${PACKAGE_NAME} PROPERTIES LINKER_LANGUAGE C)

INSTALL(TARGETS ${PACKAGE_NAME} RUNTIME DESTINATION bin LIBRARY DESTINATION lib ARCHIVE
    DESTINATION lib)
FILE(GLOB HEADERS "*.h")
INSTALL(FILES ${HEADERS} DESTINATION include)

SET(CPACK_PACKAGE_DESCRIPTION_SUMMARY "testApp")
SET(CPACK_PACKAGE_VENDOR "Gyorgy Kovacs")
SET(CPACK_PACKAGE_CONTACT "gyuriofkovacs@gmail.com")
SET(CPACK_DEBIAN_PACKAGE_MAINTAINER "gyuriofkovacs@gmail.com")
SET(CPACK_PACKAGE_DESCRIPTION_SUMMARYY "openip drscreen application")
SET(CPACK_PACKAGE_VERSION_MAJOR "${MAJOR_VERSION}")
SET(CPACK_PACKAGE_VERSION_MINOR "${MINOR_VERSION}")
SET(CPACK_PACKAGE_VERSION_PATCH "${PATCH_VERSION}")

INCLUDE(CPack)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

A fenti sorok **CMakeLists.txt** fájlokba történő beírása után tetszőleges alprojekthez, vagy akár az egész projekthez készíthetünk disztribúcióra kész csomagokat az alábbi módon. A **cpack −−help** utasítással nézhetjük meg, milyen disztribúció generátorok érhetőek el:

```
DEB                       = Debian packages
NSIS                      = Null Soft Installer
RPM                       = RPM packages
STGZ                      = Self extracting Tar GZip compression
TBZ2                      = Tar BZip2 compression
TGZ                       = Tar GZip compression
TZ                        = Tar Compress compression
ZIP                       = ZIP file format
```

Disztribúciót ezt követően a **cpack −G generator_azonosito .** paranccsal hozhatunk létre. A **STGZ**, **TBZ2**, **TGZ**, **TZ**, **ZIP** disztribúciók egyszerűen tömörítik a könyvtárakat, a **DEB** szabványos Debian csomagot, a **RPM** Red Hat Linux-ra épülő disztribúciók által használt csomagot, az **NSIS** pedig grafikus Windows telepítőt hoz létre. Természetesen az **NSIS** generátort csak windows rendszeren használhatjuk, használata előtt fel kell telepítenünk az ingyenes **NSIS** szoftvert, amely a következő linkről tölthető le: http://nsis.sourceforge.net/Main_Page.

Jegyezzük meg, hogy a CMake beépített módon csak a bináris (alkalmazások + dinamikus könyvtárak) és bináris fejlesztői (alkalmazások + dinamikus, statikus könyvtárak + header-ök) csomagok létrehozását támogatja. Szemben az **automake** eszközökkel, amelyek forráscsomag disztribúciót hoztak létre build script-tel. Ahhoz, hogy CMake-kel készítsünk forrás disztribúciót, a forráskódokat és a **CMakeLists.txt** fájlokat is meg kell adnunk megfelelő **INSTALL** makró hívásban.

A teljesség kedvéért lássuk, hogyan hozhatjuk létre a **FindTESTLIBS.cmake** modult, amelyet disztribútolva segíthetjük könyvtáraink megtalálását és felhasználását más rendszereken. A modul neve tehát rögzített, ha a csomagunkat testlibs-nek hívjuk, akkor a korábban említett nevű modult kell létrehoznunk, tartalmát tekintve pedig négy változót kell definiálnunk:

- **TESTLIBS_FOUND** abban az esetben, ha a rendszeren fenn van a könyvtárunk,
- **TESTLIBS_INCLUDE_DIR** ha a rendszerre telepítve van a könyvtárunk, akkor a header fájlok elérési útjai,
- **TESTLIBS_LIBRARIES** ha a rendszerre telepítve van a könyvtárunk, akkor a library-k nevei.

A megfelelő modul az alábbi lesz:

6.4.34. forráskód: FindTESTLIBS.cmake

```
# - Find TESTLIBS library
# Find the native TESTLIBS includes and library
# This module defines
#  TESTLIBS_INCLUDE_DIR, where to find testStaticLibrary.h, etc.
#  TESTLIBS_LIBRARIES, libraries to link against to use TESTLIBS.
#  TESTLIBS_FOUND, If false, do not try to use TESTLIBS.

#=============================================================================
# Copyright 2002-2009 Kitware, Inc.
#
# Distributed under the OSI-approved BSD License (the "License");
# see accompanying file Copyright.txt for details.
#
# This software is distributed WITHOUT ANY WARRANTY; without even the
# implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
# See the License for more information.
#=============================================================================
# (To distributed this file outside of CMake, substitute the full
#  License text for the above reference.)
```

```
FIND_PATH(TESTLIBS_INCLUDE_DIR testStaticLibrary.h)

SET(TESTLIBS_NAMES ${TESTLIBS_NAMES} testStaticLibrary testDynamicLibrary)
FIND_LIBRARY(TESTLIBS_LIBRARIES NAMES ${TESTLIBS_NAMES} )

# handle the QUIETLY and REQUIRED arguments and set TESTLIBS_FOUND to TRUE if
# all listed variables are TRUE
INCLUDE(FindPackageHandleStandardArgs)
FIND_PACKAGE_HANDLE_STANDARD_ARGS(TESTLIBS  DEFAULT_MSG  TESTLIBS_LIBRARIES
    TESTLIBS_INCLUDE_DIR)
```

Az első makró hívással a **testStaticLibrary.h** header elérési útját keressük meg, a második makróval beállítjuk a **TESTLIBS_NAMES** temporális változó értékét azon library-ke nevére, amelyeket szükségesek ahhoz, hogy a csomaggal fordítani tudjunk, majd megkeressük ezeket a library-ket a **FIND_LIBRARY** makró felhasználásával. Ezt követően a beépített **FindPackageHandleStandardArgs** csomag includeolása után a hasonló nevű makró létrehozza és beállítja az első paramétereként kapott csomagnévhez tartozó **TESTLIBS_FOUND** változó értékét annak megfelelően, hogy a harmadik és negyedik paraméterei értelmes library-ket és include könyvtárakat határoznak meg. Második paraméter a konfiguráció során a konzolon megjelenő, beépített, standard üzenet használatára vonatkozik.

6.5. qmake

6.5.1. Egyszerű qmake környezet kialakítása

A **qmake** a cmake-hez hasonlóan magas szintű eszközrendszer források menedzselésére. A **qmake** a **Qt SDK** része, amelyet a Trolltech cég fejleszt a NOKIA részeként. A **qmake** használatához installálnunk kell a **Qt SDK** csomagot. A következőkben bemutatott konfigurációs fájlok a **qmake** 2.0-ás verziójától használhatók.

A qmake fejlesztői észrevették, hogy a konfigurációs fájlok néhány fő típusba sorolhatók a legtöbb esetben: rekurzívan további könyvtárakat adunk meg, amelyek konfigurációs állományokat tartalmaznak, statikus vagy dinamikus könyvtárakat vagy alkalmazásokat specifikálnak. Ennek megfelelően a qmake fő erőssége, hogy az egyes targetek általános beállításait elrejtik, és egy speciális **TEMPLATE** változóval határozhatjuk meg, hogy a konfigurációs állomány egyes utasításait melyik lehetséges minta konkreiztálásaként értelmezze.

Hozzunk létre ismét egy, az előzőekhez hasonló könyvtárszerkezetet:

6.5.35. forráskód: Könyvtárszerkezet és konfigurációs állományok qmake esetén

```
qmaketest
     srcda
         srcda.pro
         mainDynamic.c
     srcdl
         srcdl.pro
         testDynamicLibrary.c
         testDynamicLibrary.h
     srcsa
         srcsa.pro
         mainStatic.c
     srcsda
         srcsda.pro
         mainStaticAndDynamic.c
     srcsl
         srcsl.pro
         testStaticLibrary.c
         testStaticLibrary.h
     qmaketest.pro
```

Lássuk az egyes projekt fájlok (.pro) tartalmát:

6.5.36. forráskód: qmaketest.pro

```
TEMPLATE = subdirs

SUBDIRS += srcsl
SUBDIRS += srcdl
SUBDIRS += srcsa
SUBDIRS += srcda
SUBDIRS += srcsda
```

A gyökérkönyvtárban található projekt fájl a **subdirs** mintát követi, azaz további könyvtárakat ad csak meg, melyek konfigurációs állományokat tartalmaznak. Ezeket a további könyvtárakat a speciális **SUBDIRS** változóhoz kell hozzáadnunk a **+=** operátorral.

6.5.37. forráskód: srcsl/srcsl.pro

```
TEMPLATE = lib
TARGET = testStaticLibrary
QT -= core
QT -= gui
CONFIG -= qt
CONFIG += static

HEADERS += testStaticLibrary.h

SOURCES += testStaticLibrary.c
```

Az első értékadással a library template-et, azaz mintát adjuk meg, majd target-ként a **testStaticLibrary** nevet specifikáljuk. Mivel a **qmake** alapjában véve Qt-t használó projektek konfigurálására készült, néhány Qt könyvtár linkelését alapértelmezetten hozzáadja a projekthez. Ezek eltávolítására a **−=** operátorral kivesszük a speciális **QT** változóból a **core** és **qui** modulokat, valamint a szintén speciális **CONFIG** változóból a **qt** sztringet, ami további Qt specifikus eszközöket ad a fordítási és linkelési parancsokhoz. A **CONFIG** változóhoz hozzáadott **static** sztring definiálja, hogy statikus könyvtárat szeretnénk létrehozni. Ezt követően a könyvtárhoz tartozó header fájlokat hozzáadjuk a **HEADERS** változóhoz, a forrásfájlokat pedig a **SOURCES** változóhoz.

6.5.38. forráskód: srcdl/srcdl.pro

```
TEMPLATE = lib
TARGET = testDynamicLibrary
QT -= core
QT -= gui
CONFIG -= qt
CONFIG += shared

HEADERS += testDynamicLibrary.h

SOURCES += testDynamicLibrary.c
```

A dinamikus könyvtárhoz tartozó konfigurációs állomány értelemszerűen a target és fájlnevekben tér el, valamint a **CONFIG** változóhoz a **static** helyett hozzáadott **shared** sztringben, amely shared object fájl, azaz dinamikus könyvtár létrehozását specifikálja.

6.5.39. forráskód: srcsa/srcsa.pro

```
TEMPLATE = app
TARGET = mainStatic
QT -= core
QT -= gui
CONFIG -= qt
```

```
INCLUDEPATH += ../srcsl

SOURCES += mainStatic.c

LIBS += ../srcsl/libtestStaticLibrary.a
```

A statikusan linkelendő alkalmazás projekt fájlját az **app** template specifikálásával kezdjük. **TARGET** névként beállítjuk a **mainStatic** sztringet, ami az alkalmazás neve lesz, majd a megfelelő Qt specifikus konfigurációs paraméterek eltávolítása után hozzáadjuk az **INCLUDEPATH** változóhoz a statikus könyvtár elérési útját, a **SOURCES** változóhoz a lefordítandó forrásfájlokat, és a **LIBS** változóhoz a linkelendő statikus könyvtárakat.

6.5.40. forráskód: srcda/srcda.pro

```
TEMPLATE = app
TARGET = mainDynamic
QT -= core
QT -= gui
CONFIG -= qt

INCLUDEPATH += ../srcdl

SOURCES += mainDynamic.c

SHARED_LIBS += ../srcdl/libtestDynamicLibrary.so

LIBS += ../srcdl/libtestDynamicLibrary.so
```

A dinamikusan linkelendő alkalmazás konfigurációs állománya hasonló a statikusan linkelendő alkalmazáséhoz, a különbség, hogy a dinamikus könyvtár elérési útját adjuk hozzá az **INCLUDEPATH** változóhoz, és magát a könyvtárat a **LIBS** változóhoz.

6.5.41. forráskód: /srcsda/srcsda.pro

```
TEMPLATE = app
TARGET = mainStaticAndDynamic
QT -= core
QT -= gui
CONFIG -= qt

INCLUDEPATH += ../srcdl
INCLUDEPATH += ../srcsl

SOURCES += mainStaticAndDynamic.c

SHARED_LIBS += ../srcdl/libtestDynamicLibrary.so

LIBS += ../srcsl/libtestStaticLibrary.a
LIBS += ../srcdl/libtestDynamicLibrary.so
```

A statikusan és dinamikusan is linkelendő alkalmazás projektfájlja szintén hasonlóan alakul, a fő különbség, hogy a statikus és dinamikus könyvtárak header-jeinek elérési útját és magukat a statikus és dinamikus könyvtárakat is hozzáadjuk a megfelelő változókhoz.

A konfigurációs állományok alapján gyökérkönyvtárban kiadott

```
qmake -r .
```

paranccsal állíthatjuk elő a GNU Makefile-okat, amelyeket a fordítás során felhasználhatunk. A . a feldolgozandó könyvtár elérési útja, a -r a rekurzivitásra utal, azaz ha az elérési úton található projekt

fájlok valamelyike a **subdirs** template-et valósítja meg, akkor az abban megadott alkönyvtárakra is végrehajtja a qmake a konfigurációt. Az előállt **Makefile**-okat felhasználva a **make** utasítással fordíthatjuk le és linkelhetjük könyvtárainkat és alkalmazásainkat.

6.5.2. Fordítási opciók

Lássuk, hogyan bővíthetjük ki a konfigurációt úgy, hogy a debug/release fordítást parancssorból szabályozhassuk! Hozzunk létre a gyökérkönyvtrában egy **conf.pri** nevű állományt az alábbi tartalommal:

6.5.42. forráskód: conf.pri

```
QMAKE_CFLAGS_RELEASE -= -O2
QMAKE_CFLAGS_RELEASE += -O3
QMAKE_CFLAGS_DEBUG += -O0
QMAKE_CFLAGS_DEBUG += -g
```

Az első sorban kivesszük a speciális **QMAKE_CFLAGS_RELEASE** változóból az **−O2** kapcsolót, majd hozzáadjuk a **−O3** kapcsolót. A **QMAKE_CFLAGS_DEBUG** változóhoz hozzáadjuk a **−O0** és a **−g** kapcsolókat. Ezt a kódrészletet include-oljuk minden forrásfájlt tartalmazó könyvtár projekt fájljába a projekt fájlban elhelyezett alábbi utasítással:

```
include(../conf.pri)
```

Azaz például a statikusan és dinamikusan linkelt alkalmazás konfigurációs állománya a következőre módosul:

6.5.43. forráskód: srcsda/srcsda.pro

```
TEMPLATE = app
TARGET = mainStaticAndDynamic
QT -= core
QT -= gui
CONFIG -= qt

include(../conf.pri)

INCLUDEPATH += ../srcdl
INCLUDEPATH += ../srcsl

SOURCES += mainStaticAndDynamic.c

LIBS += ../srcsl/libtestStaticLibrary.a
LIBS += ../srcdl/libtestDynamicLibrary.so
```

Ezt követően a konfigurációs parancsban a **CONFIG** változóhoz hozzáadott **debug** vagy **release** sztringgel specifikálhatjuk a debug vagy release fordítást:

```
qmake -r CONFIG+=debug .
```

A fenti példában a beépített **QMAKE_CFLAGS_RELEASE** és **QMAKE_CFLAGS_DEBUG** változókat használtuk. Emlékezzünk vissza, hogy a CMake setén is rendelkezésre álltak hasonló, beépített változók ugyenezen célra. Lássuk most, hogyan oldhatjuk meg a debug/release fordítást általánosabb eszközökkel!

A qmake legfontosabb konfigurációs eszközeit a korábbi eszközrendszerekhez hasonlóan a változók és feltételes szerkezetek adják. A koncepció a következő: változókhoz opciókat, mint sztringeket

adunk hozzá és távolítuk el a **+=** és **−=** operátorok felhasználásával, és ezt követően a feltételes szerkezetek ezen sztringek jelenlétének alapján állítják be a megfelelő kapcsolókat, flag-eket. Az általánosan használható konfigurációs változó a **CONFIG**, amelyet már korábban is használtunk. A feltételes szerkezetek az alábbi szintaktikát használják:

```
feltétel{
}
```

Ezen szintaktika miatt a feltételes szerkezetet hatáskörnek is nevezik a qmake terminológiájában. Ezen feltételes szerkezetek tetszőleges mélységben egymásba ágyazhatóak. A feltételek lehetnek a **CONFIG** változóban szereplő sztringek, ekkor csak a megfelelő sztringet használhatjuk önmagában feltételként. A feltétel igaz, ha a sztring jelen van a **CONFIG** változóban. Ha más változókat szeretnénk használni, néhány előre megírt tesztfüggvény közül választhatunk, amelyek a változók tartalmát tesztelik:

- **contains(valtozonev, ertek)** - a feltétel igaz, ha a változó tartalmazza az értéket,
- **count(valtozonev, ertek)** - a feltétel igaz, ha a változó éppen **ertek** darab sztringet tartalmaz,
- **exists (fajlnev)** - a feltétel igaz, ha a fájl létezik,
- **isEmpty(valtozonev)** - a feltétel igaz, ha a változó üres,
- **equals(valtozonev, ertek)** - a feltétel igaz, ha a változó értéke éppen **ertek**.

A fordító és linker flag-eket a **QMAKE_CFLAGS** és **QMAKE_LFLAGS** változókban állíthatjuk be. Az előbbi feltételes szerkezet alapján, így a **config.pri** fájlban elhelyezve a

```
debug{
  QMAKE_CFLAGS+= -O0 -g
}
release{
  QMAKE_CFLAGS+= -O3
}
```

feltételes szerkezetekkel éppen úgy bekerülnek a debug/release fordításhoz szükséges kapcsolók a fordítási parancsba, mint a speciális **QMAKE_CFLAGS_DEBUG** és **QMAKE_CFLAGS_RELEASE** változók használata esetén. Ügyeljünk azonban arra, hogy a **debug** és **release** opciók alapértelmezésként szerepelnek a **CONFIG** változóban.

6.5. Feladat: Adjon hozzá konfigurációs lehetőségeket a qmake környezethez úgy, hogy a **warnings** sztring esetén a **−Wall −Wextra** kapcsolók szerepeljenek a fordítási sorban, míg a **usedynlib** sztring esetén a dinamikus könyvtár használatát bekapcsoló **−DUSE_DYNAMIC_LIBRARY** kapcsoló legyen benne a fordítási sorban!

6.5. Megoldás: A **conf.pri** fájlban az alábbi feltételes szerkezeteket kell elhelyeznünk:

```
warnings{
  QMAKE_CFLAGS+= -Wall -Wextra
}
usedynlib{
  QMAKE_CFLAGS+= -DUSE_DYNAMIC_LIBRARY
}
```

A **mainOptional** alkalmazásban a dinamikus könyvtár használatának **config.h** header fájlon keresztül történő bekapcsolása már problémás kérdés, ugyanis a qmake nem tartalmaz eszközöket változó fájlokba történő helyettesítésére, úgy, mint az automake vagy a cmake. A megoldás egy egyszerű **system** függvényhívás lehet, amelyben az operációs rendszernek kiadott **echo** parancsot a megfelelő fájlba irányítva létrehozhatunk akár header fájlokat is:

```
usedynlib{
  system(echo "#define USE_DYNAMIC_LIBRARY" > config.h)
}
```

6.5.3. Külső könyvtárak használata

A külső könyvtárak használatára qmake-ben nem állnak rendelkezésünkre olyan kifinomult eszközök, mint automake-ben vagy cmake-ben. Legkevésbé hordozható megoldás a felhasználandó könyvtárak fordító és linker flag-jeinek rögzítése a konfigurációs állományokban. Ez linux rendszerek esetén többé-kevésbé működőképes, hiszen a különböző forráskód csomagok mind a **/usr** könyvtár különböző alkönyvtáraiba installálják a header-öket és a library-kat is. Windows környezetben ajánlott a külső csomagokat is elhelyezni a fordítási környeztünkben, annak részeként, és ekkor a megfelelő elérési utak rögzítése nem jelent problémát a fordítás és linkelés szempontjából.

Mindazonáltal az **exist**, **for** és **defineTest** makrók felhasználásával bizonyos rugalmasságot vihetünk a build rendszerbe, a leggyakoribb helyeket ellenőrizve, ahol a szükséges csomag előfordulhat.

Beillesztve a **mainPNG.c** forrást jelenlegi környezetünkbe, Linux rendszeren az alábbi sorokkal érhetjük el, hogy a megfelelő flag-ek bekerüljenek a fordítási és linkelési parancsba:

```
exists(/usr/include/png.h){
  INCLUDEPATH+= /usr/include
}
exists(/usr/local/include/png.h){
  INCLUDEPATH+= /usr/local/include
}
exists(/usr/lib/libpng.so){
  LIBS+= -L/usr/lib -lpng
}
exists(/usr/local/lib/libpng.so){
  LIBS+= -L/usr/local/lib -lpng
}
```

6.5.4. Disztribúció létrehozása

Az eddig létrehozott **Makefile**-ok mind tartalmaznak **install** target-et. Nincs ez másképp a qmake által létrehozott **Makefile**-lal sem, azonban ez alapértelmezésben üres, azaz nincs hatása. Ahhoz, hogy környezetünk bizonyos részeit (header-ök, targetek, stb.) installálni tudjuk a **make install** paranccsal, hozzá kell adnunk a megfelelő sztringeket az **INSTALLS** változóhoz. Alkalmazások és könyvtárak esetén az alábbi formában specifikálhatjuk az installálás helyét:

```
target.path=konyvtarnev
INSTALLS += target
```

ahol a **target** nem az aktuális target neve, hanem maga a **target** sztring. Tehát ha például a statikus könyvtárat szeretnénk insallálni a **/usr/lib** könyvtárba, akkor az alábbi sorokat kell elhelyeznünk a statikus könyvtárhoz tartozó .pro fájlban:

```
target.path=/usr/lib
INSTALLS += target
```

Újrakonfigurálás után már valóban installál a **make install** parancs.

A qmake forráscsomagok disztribúcióját támogatja, olyan .tar.gz csomagokat hoz létre, amelyek tartalmazzák a forrásfájljainkat valamint a konfigurációs .pro fájlt. Forráscsomag létrehozásánál a qmake a csomagba elhelyezi még a .pro fájlunkban használt beépített funkciók definíciós fájljait, tehát sok néhány soros qmake .pro fájl is bekerül a csomagba, amely függetlenné teszi azt a qmake célgépen használt verziójától. A csomagba bekerülnek a **SOURCES** változóban szereplő fájlok, valamint a **DISTFILES** változónak értékül adott fájlok. Ahhoz tehát, hogy a header fájljok is bekerüljenek a készülő forráscsomagba, hozzá kell adnunk őket a **DISTFILES** változóhoz, azaz a statikus könyvtár esetén például az alábbi módon néz ki a végső projekt fájl:

6.5.44. forráskód: srcsl/srcsl.pro

```
TEMPLATE = lib
TARGET = testStaticLibrary
QT -= core
QT -= gui
CONFIG -= qt
CONFIG += static

include(../conf.pri)

HEADERS += testStaticLibrary.h
DISTFILES += testStaticLibrary.h

SOURCES += testStaticLibrary.c

target.path=/usr/lib
INSTALLS += target
```

Kiadva a **make dist** parancsot, előáll a forráscsomag.

6.6. Összefoglalás

Összefoglalva tehát a C/C++ forráskódok menedzselése számos kérdést vet fel: hogyan állíthatunk be fordítási opciókat? hordozható-e a projekt fájl? támogatja-e linux rendszereken az installációt? támogatja-e disztribúciók készítését? ha igen, milyen disztribúciókat készíthetünk?

Ezen kérdésekre a korábban tárgyalt eszközrendszerek az alábbi válaszokat adják:

GNU Make. A GNU Makefile-ok egy rendkívül rugalmas eszközrendszert biztosítanak, amelyekkel teljesen személyre szabhatjuk az egyes targetekhez rendelt fordítási parancsot. Néhány fájlból álló projekt esetén tökéletes, azonban sok fájlból álló projekt esetén a **Makefile** megírása nyűgös. A **Makefile** maga csak addig hordozható, amíg nem használunk benne a parancsértelmezőtől és így az operációs rendszertől függő utasításokat, programhívásokat. Opciókat környezeti változókon keresztül adhatunk meg, az installációt és disztribúciók készítését vezérlő targeteket a fejlesztőnek kell megírnia, ami kényelmetlen.

Automake. Az automake eszközrendszer egy magasabb szintű makrógyűjtemény, amellyel GNU Makefile-okat generálhatunk. Az automake egy kiforrt, professzionális eszköz, nagy szabadsági fokkal, Linux környezetben *de facto* szabványnak tekinthető. Opciókat magas szinten adhatunk meg, az installációt is magas szinten támogatja, és forráscsomagok is készíthetőek a segítségével. Hátránya, hogy a konfigurációs fájlok szintaktikája igen kényes és sokszor nem az intuitív megoldások nem működnek, használatához viszonylag nagy gyakorlat kell. További hátránya, hogy Windows rendszeren csak a Linux parancsértelmezők emulátoraival használható, mert nagyon szoros kapcsolatban van a shell-el.

CMake. A CMake szintén egy magasszintű eszközrendszer, amely GNU Makefile-ok generálására használható. A CMake rendkívül kényelmes, a build script-ek szinte imperatív programként olvashatóak. Könnyen adhatjunk meg fordítási opciókat parancssoron keresztül, támogatja az installációt, a legkülönbözőbb projektfájlokat és disztribúciós csomagokat hozhatjuk létre vele, emellett teljesen interoperábilis és a külső csomagok használatát is interoperábilis módon támogatja. A vizsgált eszközrendszerek közül kezdők számára mindenképpen a CMake használata javasolt!

qmake. A qmake nagyon kényelmes eszközrendszer kis projektek menedzselésére, a template projektfájloknak köszönhetően rövidebb scriptekkel érhetjük el ugyanazt az eredményt, amit CMake vagy Automake segítségével. A fordítási opciók megadása nagyon kényelmes. Beépített eszközökkel azonban csak korlátozottan támogatja az installációt és disztribúciók létrehozását. Természetesen számos eszközt nyújt arra, hogy saját, összetettebb megoldásokat készítsünk például külső könyvtárak keresésére, vagy akár disztribúcióra, használatuk gyakorlatot igényel.

7. Párhuzamos programozás

7.1. Folyamat/Process vs. Szál/Thread.

A folyamat, mint futási egység, az alábbi elemekből épül fel:

- egy programhoz tartozó futtatható gépi kód képe a memóriában,
- memóriaterület, amely a futtatható kódon kívül tartalmazza a folyamathoz tartozó be- és kimenő adatokat, a hívási vermet (stack) és a kupacot (heap), amelyben a köztes adatok tárolja a rendszer,
- operációs rendszerhez kapcsolódó deszkriptorok, amelyek a folyamathoz rendelt erőforrásokat tartják nyilván,
- jogosultségi információk,
- processzor állapot, azaz a regiszterek tartalma, fizikai memória címzés, stb.

az operációs rendszer a fenti információkat egy úgynevezett process control block nevű adatszerkezetben tartja nyilván. A folyamatokat az operációs rendszer vagy felhasználók is indíthatják, a processzorütemezésnek köszönhetően egy processzoron is futhat "egyszerre" több folyamat látszólagos párhuzamossággal.

A szál egyik elterjedt difiníciója a *könnyűsúlyú folyamat*. Az elnevezés többek között arra utal, hogy egy szál többnyire a legkisebb futási egység, amelyet az operációs rendszer ütemezni tud. Leggyakrabban egy függvényhívásban jelennek meg azok az utasítások, amelyek egy szálban végrehajthatóak.

A folyamatok többnyire egymástól függetlenek, míg a szálak létező folyamatok részeiként jönnek létre. A folyamatokat lejró adatszerkezet (PCB) minden olyan információt tartalmaz, ami a folyamat menedzseléséhez szükséges, ezzel szemben a szálakat ezen információknak csak egy részhalmaza jellemzi, a szálak ugyanis megosztják egymás között a memóriát és más erőforrásokat. A szálak nem rendelkeznek dedikált címtérrel (memóriaterülettel). A folyamatok csak az operációs rendszer által biztosított inter-process kommunikációs mechanizmusokkal kommunikálnak, míg a szálak a közös memóriaterületen keresztül kommunikálhatnak. A gyakorlati szempontból legfontosabb különbség az, hogy mivel a szálak leírásához kevesebb információ kell, a szálak futási környezetének váltása (context switching) a processzorban sokkal gyorsabb, mint a folyamatok váltása.

7.2. Adat- és funcionális párhuzamosítás.

A párhuzamosítási probléma dekompozíciója során a problémák két fő csoportra bomlanak. *Adatpárhuzamosítás*ról akkor beszélünk, ha ugyanazon műveletsorozatot szeretnénk végrehajtani nagy mennyiségű adaton. Ebben az esetben a párhuzamos megvalósítás, tehát az adatok egymástól független

feldolgozása csak akkor jelent előnyt, ha ténylegesen több processzor áll rendelkezésünkre. *Funkcionális párhuzamosítás*ról beszélünk akkor, ha a párhuzamosan futó szálak/processzek nem használják 100%-osan a processzort, a cél nem nagy mennyiségű adat feldolgozása, hanem az, hogy amíg bizonyos műveletek végrehajtására várnia kell egy szálnak/folyamatnak (például IO műveletek), addig a programunk más részei továbbra is elérhetőek legyenek a felhasználó számára. Ez utóbbi funkcionális párhuzamosításra például szolgálnak olyan grafikus interfésszel rendelkező alkalmazások, amelyekben ha megnyitunk egy párbeszédablakot, akkor az alkalmazás főablaka nem válik inaktívvá, amíg a párbeszédablakot be nem zárjuk, hanem a főablak továbbra is elérhető és használható.

7.3. Párhuzamos architektúrák.

A későbbiekben bemutatásra kerülő párhuzamos programozási eszközrendszerek alapvetően különböznek abban, hogy milyen architektúrán használhatók, hiszen a többmagos gépekben végzett párhuzamosítás és egy klaszteren végzett párhuzamosítás alapjában véve különbözik. Az architektúrák közötti fő különbség az, hogy az egyes számítási egységek (CPU-k) milyen módon kapcsolódnak a számítógép más egységeihez, a fő kérdés, hogy a memóriát hogyan érhetik el.

A *megosztott memórát* (shared memory model) használó architektúrák esetén az egyes számítási egységek egy, közös memóriához férhetnek hozzá, így azon keresztül a kommunikáció is megvalósítható az egyes folyamatok/szálak között. Ezzel szemben az *elosztott memória* modellek (distributed memory model) esetén az egyes processzorok fizikailag különböző memóriában dolgoznak, így a szálak/-folyamatok közötti kommunikáció nem valósítható meg a memórián keresztül, a kommunikációhoz többnyire hálózati interfészre és nyitott TCP/UDP csatornákra van szükség. A kommunikáció ezen utóbbi esetben úgynevezett *Message Passing*, azaz üzenetküldő protokollokon keresztül zajlik.

7.4. Flynn osztályok

Michael J. Flynn 1966-ban definiálta számítógép architektúrák négy osztályát, amelyek az architektúrák adatfeldolgozó mechanizmusának fő jellemzőit fogják össze.

- SSID (Single Instruction, Single Data stream): Az SSID osztályba olyan számítógépek tartoznak, amelyek nem teszik lehetővé a párhuzamosítást, ide tartoznak a klasszikus egyprocesszoros PC-k.
- SIMD (Single Instruction, Multiple Data streams): Az SIMD számítógépek egyetlen utasítássorozatot hajtanak végre több adatsorozaton párhuzamosan. Ebbe a csoportba sorolhatóak a vektorprocesszorok és például a GPU. A leggyakrabban használt asztali PC hardware korlátozott módon biztosít lehetőséget annak vektorprocesszorként történő használatára.
- MISD (Multiple Instruction, Single Data stream): Több különböző utasítássorozatot alkalmaz ugyanazon egyetlen bemenő adatsorozatra. Megvalósításai ritkák.
- MIMD (Multiple Instruction, Multiple Data stream): Több önálló processzor párhuzamosan futtat különböző utasítássorozatokat különböző adatokon. Ebbe a kategóriába sorolhatóak a mai többmagos számítógépek, de a klaszterek architektúráját is szokás MIMD jellegű architektúrának nevezni.

Az MIMD kategóriát szokás még két kisebb kategárióra osztani:

- SPMD (Single Program, Multiple Data): A különböző processzorok különböző adatokon ugyanazon programot futtatják, azonban a program különböző futásai különböző vezérlési útvonalakat járnak be.
- MPMD (Multiple Program, Multiple Data): A különböző processzorok legalább két különböző programot futtatnak, amelyek közül az egyik egyedülálló *host/master/vezérlő* program, míg a többi *slave/dolgozó* program. Jellemzően a vezérlő program osztja ki a feladatokat a dolgozó folyamatoknak, amelyek a feldolgozás befejezését követően befejezik működésüket.

7.5. Amdahl törvénye

Amdahl törvénye nevét Gene Amdahlról kapta és egy szoftver teljesítményjavulása várható mértékének becslésére szolgál. Gyakran használják a párhuzamos rendszerekben annak meghatározására, hogy újabb processzorok bevezetésével maximálisan mekkora gyorsulás érhető el.

A törvény általánosan a következő módon fogalmazható meg: Legyen P a program számításainak azon hányada, amelyet hatékonyságát növelni tudjuk, tehát például ha a program által végzett összes művelet 30%-át tudjuk gyorsabbá tenni, akkor $P = 0.3$. Legyen S a hatékonyság növelésének mértéke, azaz az, hogy mennyivel válik gyorsabbá a P kódrészlet. Például ha a P számításokat kétszer olyan gyorsan tudjuk elvégezni, akkor $S = 2$. Ezen jelölésekkel a szoftver maximális gyorsulása

$$SU = \frac{1}{(1-P) + \frac{P}{S}}. \tag{1}$$

Párhuzamosításra konkretizálva a törvényt: ha a számítások P hányada végezhető párhuzamosan, és a párhuzamosításhoz N processzort használhatunk, a maximális gyorsulás

$$SU = \frac{1}{(1-P) + \frac{P}{N}}. \tag{2}$$

A P érték már működő párhuzamosítás esetén az alábbi formulával becsülhető:

$$P_{becsult} = \frac{\frac{1}{SU} - 1}{\frac{1}{NP} - 1}, \tag{3}$$

ahol SU az NP számú processzorral elért gyorsulás. Ezen becsült P értékkel aztán kiszámítható, hogy a processzorok számot növelve mennyivel csökkenhet a program végrehajtásának ideje.

Feltéve, hogy szekvenciális programunk futási idejének p-ad részét az A kódrészben tölti, az A rész N szálra történő párhuzamosításával elérhető maximális gyorsulása az alábbi formulával becsülhető:

$$MSU \leq \frac{N}{1 + p(N-1)}, \tag{4}$$

ahol MSU a gyorsulás mértékére utal (Maximum Speed-Up). Ha adott tehát egy program, amely futási idejének $\frac{2}{3}$-ad részét tölti az A kódrészletben, míg $\frac{1}{3}$-ad részét a B kódrészletben, az A kódrészletet párhuzamosítva 3 szálra az elérhető legjobb sebességjavulás:

$$MSU_A \leq \frac{3}{1 + 0.66(3-1)} = 1.29. \tag{5}$$

A B kódrészletet párhuzamosítva 2 szálra, az elérhető legjobb sebességjavulás:

$$MSU_B \leq \frac{2}{1 + 0.33(2-1)} = 1.50, \tag{6}$$

azaz az A kódrészlet 3 szálra történő párhuzamosítása kisebb sebességnövekedéssel járhat, mint a B kódrészlet 2 szálra történő párhuzamosítása.

7.6. Gustafson törvénye

Amdahl törvénye sok kritikát kapott, ugyanis meglehetősen pesszimista, túl nagy szekvenciális futási időket feltételez (például a becslésre szolgáló formulával). Amdahl törvényének fő gyengesége, hogy

rögzített méretű probléma esetén használható, tehát a törvény nem ad lehetőséget a probléma méretével történő paraméterezésre. Gustafson a következő észrevételt tette: a legtöbb fejlesztő nem érdekelt rögzített méretű problémák legrövidebb idő alatt történő megoldásában, sokkal fontosabb, hogy a lehető legnagyobb méretű problémákat oldják meg elfogadható időn belül. Ha tehát a nem párhuzamosítható kódrészlet rögzített, vagy csak nagyon lassan nő, szemben Amdahl törvényével, a hatékonyság újabb processzorok bevezetésével szinte korlátlanul növelhető.

A törvény alakja a következő: Legyen P a processzorok száma, α pedig folyamat nem párhuzamosítható hányada. Ekkor a várható sebességnövekedés a processzorok számának függvényében:

$$S(P) = P - \alpha(P - 1). \tag{7}$$

A fenti formula segítségével a bemenet méretének függvényében az alábbi módon írható fel a várható sebességnövekedés. Legyen n a probléma mérete. A program futása az alábbi módon osztható két részre:

$$a(n) + b(n) = 1, \tag{8}$$

ahol a a szekvenciálisan futó rész, b a párhuzamosan futó kódrészlet (a párhuzamosításból eredő overhead-től most eltekintünk). Ekkor egy processzor esetén a szekvenciálisan elvégzett műveletek száma

$$a(n) + pb(n). \tag{9}$$

A sebességnövekedés tehát a szekvenciális végrehajtá és a párhuzamos végrehajtás idejének hányados, azaz

$$SU = \frac{a(n) + pb(n)}{a(n) + b(n)} = \frac{a(n) + pb(n)}{1} = a(n) + p(1 - a(n)). \tag{10}$$

Könnyű látni, hogy ha a szekvenciálisan végrehajtott kódrészlet mértéke arányaiban 0-hoz tart, míg a processzorok száma a végtelenhez, tetszőleges gyorsulás érhető el.

8. Esettanulmány

A továbbiakban a párhuzamosítási eszközöket egy a képfeldolgozás területéről származó problémán fogjuk szemléltetni, nevezetesen a képszűrésen, vagy másnéven filterezésen. Az alábbiakban bemutatok egy egyszerű tesztalkalmazást, amely CPU-n végzi el az átlagoló-filter alkalmazását egy beolvasott képre. A forráskódok, amelyeket itt bemutatok, magukban foglalják a kép beolvasást is, amelyet esetünkben PNG képekre írunk meg a libpng csomag felhasználásával. A későbbiekben ezt az alkalmazást és könyvtárat fogjuk bővíteni a különböző párhuzamosítási eszközökkel. A példakönyvtárat és alkalmazást a cmake eszközök segítségével menedzseljük. A forrásfájlok a A. kiegészítésben találhatóak meg. A cmake konfigurációs állományok a cmake eszközökkel foglalkozó fejezet alapján könnyen értelmezhetők. Lássuk most a forrásfájlokat: Az **Image** osztály specifikációja, amely lényegében egy kétdimenziós tömböt reprezentál, amely **unsigned char** típusú intenzitásértékeket tartalmaz. Az **Image** osztály a C++ standard könyvtár **vector** osztályából származik. Az, hogy kétdimenziós tömböt reprezentál, csak a felüldefiniált két paraméteres zárójel operátorban testesül meg, amely a sor és oszlopkoordinátákának megfelelően a sorfolytonos tárolás megfelelő elemét adja vissza. Az IO.h header specifikálja a png képek beolvasására és kiírására szolgáló függvényeket. A kiíró és beolvasó függvényekhez felhasználjuk a png.h header-t, amely része a libpng-dev csomagnak. A png.h headerben található függvények segítségével hozzuk létre azokat a függvényeket, amelyek képesek három csatornás RGB képek beolvasására és kiírására. Az IO.cc a függvények implementációját tartalmazza. Az OptionTable.h header egy olyan osztályt tartalmaz, amely a parancssori kapcsolók feldolgozását

segíti, az OptionTable.cc ezt az osztályt implementálja. A Stopper.h egy időmérésre szolgáló stopper ősosztályt tartalmaz, a Stopper.cc az implementációját. Az futási idő mérése az egyes práhuzamos eszközök esetén problémás lehet, ezért minden párhuzamosítási eszközhöz létrehozunk egy időmérésre alkalmas osztályt is. A StopperCPU.h és StopperCPU.cc fájlok a szekvenciális futtatás idejének mérésére alkalmas eszközöket tartalmaz. A Filter.h és Filter.cc a Stopper.h és .cc-hez hasonlóan egy általános Filter ősosztályt tartalmaz. Alkalmazásunkat a main.cc-ben valósítjuk meg. Lássuk most az tesztkörnyezetünk lelkét, azaz magát a filterezést megvalósító osztályt:

8.0.1. forráskód: lib/MeanFilter.h

```
#ifndef _MEAN_FILTER_H_
#define _MEAN_FILTER_H_

#include <Filter.h>

class MeanFilter: public Filter
{
public:
  MeanFilter(int w);

  MeanFilter(const MeanFilter& m);

  ~MeanFilter();

  virtual void apply(Image& input, Image& output);

  virtual unsigned char apply(Image& input, int r, int c);

  int w;
};

#endif
```

8.0.2. forráskód: lib/MeanFilter.cc

```
#include <MeanFilter.h>
#include <stdio.h>

MeanFilter::MeanFilter(int w)
: Filter()
{
  this->w= w;
}

MeanFilter::MeanFilter(const MeanFilter& m)
: Filter(m), w(m.w)
{
}

MeanFilter::~MeanFilter()
{
}

void MeanFilter::apply(Image& input, Image& output)
{
  for ( int i= 0; i < input.rows; ++i )
    for ( int j= 0; j < input.columns; ++j )
      output(i,j)= apply(input, i, j);
}

unsigned char MeanFilter::apply(Image& input, int r, int c)
{
```

```
  int w2= w/2;

  int sum= 0;
  int n= 0;
  for ( int i= -w2; i <= w2; ++i )
    for ( int j= -w2; j <= w2; ++j )
      if ( r + i >= 0 && r + i < input.rows && c + j >= 0 && c + j <= input.columns )
      {
  sum+= input(r + i, c + j);
  ++n;
      }

  sum= (float)sum/(float)n;

  return (unsigned char)sum;
}
```

MeanFilter osztályunk lényegében egy funktor, azaz olyan osztály, amely egy függvényszerű műveletet csomagol be (wrapperel). A későbbiekben ennek leszármazott osztályait hozzuk létre, amelyek egy-egy párhuzamosítási eszközrendszerrel valósítják meg ugyanezt a műveletet, s így a párhuzamosított objektumok paraméterül adhatóak minden olyan függvénynek, amely MeanFilter objektum mutatóját várja paraméterül. Ez a megközelítés nagyon általános a C++ programozásban. Osztályunk a konstruktora egy paramétert kap, ami a négyszög alakú átlagoló filter mérete. A másoló konstruktor és a destruktor megvalósítása a szokásos. Két függvényünk van még, mindkettőnek **apply** a neve, csak paraméterezésben különböznek. Az átlagoló filter egymásba ágyazott for-ciklusokkal valósítható meg. A kép minden pixelére ki kell számolnunk annak valamely környezetében az átlagos intenzitást. Az egymásba ágyazott ciklusokat szedtem szét két függvénybe. A külső ciklusok, amelyek a kép pixelein iterálnak, a "külső" **apply**-ba kerülnek, az átlagot számító ciklusok pedig a "belső" **apply**-ba. A "külső" **apply** paraméterként kap egy input és output kép objektumot, a "belső" **apply** paraméterként kapja az input képet, és azon pixelnek a sor és oszlop koordinátáját, amely **w** méretű környezetében az átlagszámítást el kell végeznie. Hatékonyság szempontjából nem lassít a programon az, hogy két függvénybe szedtük szét a ciklusokat, ugyanis **−O3** optimaliázással (release fordításnál) a fordító helyettesi a függvényhívást annak implementációjával. A "belső" **apply** visszaadja az átlagos intenzitásértéket visszatérési értékként. tekintsük most a főprogramot:

8.0.3. forráskód: app/main.cc

```
#include <OptionTable.h>
#include <IO.h>
#include <MeanFilter.h>
#include <StopperCPU.h>

int filter(char* input, char* output, MeanFilter* mf, Stopper* st)
{
  Image r, g, b;

  readPNG(input, r, g, b);

  Image r2, g2, b2;

  r2.resizeImage(r.rows, r.columns);
  g2.resizeImage(r.rows, r.columns);
  b2.resizeImage(r.rows, r.columns);

  st->start();
  mf->apply(r, r2);
  mf->apply(g, g2);
  mf->apply(b, b2);

  st->stop();
  std::cout << "execution time: " << st->getElapsedTime() << std::endl;
```

```
  writePNG(output, r2, g2, b2);

  return 0;
}

int main(int argc, char** argv)
{
  bool cpu;

  int kernelSize;
  int numberOfThreads= 1;

  OptionTable ot;

  ot.setPrefix("\nDemonstrating application for parallel image filtering.");
  ot.setPostfix("usage: filterApp [options] {input_image}.png {output_image}.png\n");
  ot.addOption("--cpu", OPTION_BOOL, (char*)&cpu, 0, "mean filter on cpu");
  ot.addOption("--kernel", OPTION_INT, (char*)&kernelSize, 1, "mean filter kernel size")
      ;

  ot.processArgs(&argc, argv);

  if ( cpu )
    return filter(argv[1],argv[2], new MeanFilter(kernelSize), new StopperCPU());

  return 0;
}
```

Főprogramunkban létrehoztunk egy logikai változót **cpu**, amelynek értéke igazzá válik, ha a futtatási parancs tartalmazza a "–cpu" kapcsolót. A **kernelSize** változó a −−**kernel** kapcsoló utáni számot veszi fel értékül, azaz az átlagszámításnál használt ablakméretet. A **numberOfThreads** változóra egyelőre nincs szükségünk. Ezt követőn létrehozunk egy **OptionTable** objektumot, amelyhez hozzáadjuk a "–cpu" és "–kernel" kapcsolót, majd feldolgozzuk a parancssori argumentumokat. A ot.processArgs függvényhívás után a parancssori argumentumok között továbbra is ott maradnak azok, amelyeket nem sikerült kapcsolóként, vagy kapcsolót követő paraméterként felismerni. Mindezek után ha a "–cpu" kapcsoló be van kapcsolva, meghívjuk a filter függvényt az első és második parancssori argumentumra, amelyek az input és output képfájlok nevét tartalmazzák. További paramétere a filter függvénynek egy MeanFilter objektum, azaz a funktor, amely a filterzést végzi, valamint egy időmérő objektum, azaz a Stopper osztály leszármazottja. Lássuk most a filter függvényt: a függvény létrehoz egy r, g és b Image objektumot, majd beolvassa az input kép csatornáit. Ezt követően létrehoz három újabb kép objektumot az eredmény képeknek: r2, g2, b2, majd beindítja a stoppert és alkalmazza a MeanFilter funktort a képekre. Ezt követően kiírjuk a futási időt és fájlba írjuk a képeket.

Alkalmazásunkat konfigurálás és fordítás után a következő módon próbálhatjuk ki: a

```
./filterApp --help
```

utasítással megnézhetjük az **OptionTable** objektumban specifikált kapcsolóinkat. Egyelőre csak a szekvenciálisan végrehajtott átlagoló filterezést próbálhatjuk ki az alábbi utasítással:

```
./filterApp --cpu --kernel lena.png lena-filtered.png
```

A program kimenete:

```
execution time: 0:0:2.340 (h:m:s)
```

azaz 2.3 másodperc alatt hajtotta végre a filterezést. Az input és output képeket az 1. ábrán tekinthetjük meg.

Fig 1: Az átlagoló filterezés input és output képe

9. Automatikus vektorizálás

Az automatikus vektorizálás a legegyszerűbb, fordító által is támogatott párhuzamosítási eszközrendszer. Működése és elnevezése a vektor processzorok területéről eredeztethető (SIMD architektúra), ahol ugyanazon műveletsorozat végrehajtása a cél nagy számú adaton. Ezek az adatok úgynevezett *vektor regiszter*ekbe kerülnek. A processzor ezt követően a viszonylag egyszerű műveletsorozatot a vektorregiszterek elemeire párhuzamosan képes végrehajtani. A vektor regiszterek mérete meglehetősen kicsi. A vektorprocesszorokra jellemző architektúrális megoldások bekerültek a mai asztali számítógépekbe, így a vektorizálás alapú párhuzamosítás bárki számára elérhető. A párhuzamosan végrehajtható kódok felderítését és a vektorizálásnak megfelelő átszervezését a fordítóprogram végzi.

A GCC csomag fordítóprogramjai képesek automatikus vektorizálásra, megfelelő flag bekapcsolását követően. Fontos azonban, hogy a gyakorlatban gcc-vel csak néhány nagyon egyszerű programozási szerkezet vektorizálható. Az automatikus vektorizálást a **−ftree−vectorize** fordító flag-gel kapcsolhatjuk be. A flag automatikusan bekapcsol az **−O3** optimalizálás használatakor. A fordító vektorizáláshoz kapcsolódó kimenetei alapértlemezésként nem jelennek meg a kimeneten. A vektorizálás sikerességének ellenőrzése érdekében bekapcsolhatjuk az automatikus vektorizálás beszédes módját a **−free−vectorizer−lstinlineose=<number>** parancssori opcióval. A **<number>** a következő beszédességi szinteket jelöli:

- 0: nincs kimenet (alapértelmezett),
- 1: beszámol a vektorizált ciklusokról,
- 2: beszámol a nem-vektorizált, de "jól-formázott" ciklusokról, és a vektorizálás kihagyásának okairól,
- 3: beszámol elhelyezési információkról,
- 4: megegyezik a 3-as szinttel, és a nem "jól-formázott" belső ciklusokról is beszámol,
- 5: megegyezik a 3-as szinttel, és minden ciklusról beszámol,
- 6: a vektorizálás során előálló minden információt megjelenít.

Lássuk, milyen szerkezetek vektorizálására képes a fordító automatikusan! Rögzített méretű tömböket befutó ciklusok, melyek magjában az indexváltozó módosítás nélkül indexeli a tömböket és a tömbelemeken aritmetikai műveleteket hajtunk végre:

```
int a[256], b[256], c[256];
void foo ()
{
  int i;

  for (i=0; i<256; i++)
    a[i] = b[i] + c[i];
}
```

Ciklus változó indexhatárral és változót tartalmazó kifejezéssel:

```
int b[256];
void foo (int n, int x)
{
   int i;

   for (i=0; i<n; i++)
      b[i] = x;
}
```

Feltételes ciklus, a ciklusmagban bitműveletekkel:

```
int a[256], b[256], c[256];
void foo (int n)
{
   while (n--)
   {
      a[i] = b[i]&c[i];
      i++;
   }
}
```

Igazított mutatók használata:

```
typedef int aint __attribute__ ((__aligned__(16)));
foo (int n, aint * __restrict__ p, aint * __restrict q)
{
   while (n--)
      *p++= *q++;
}
```

Igazított mutatók használata konstasokkal:

```
typedef int aint __attribute__ ((__aligned__(16)));
int a[256], b[256], c[256];
foo (int n, aint * __restrict__ p, aint * __restrict__ q)
{
   int i;

   while (n--)
      *p++ = *q++ + 5;
}
```

Mutatók dereferenciája olvasásra fordítási időben ismert eltolással:

```
typedef int aint __attribute__ ((__aligned__(16)));
int a[256], b[256], c[256];
foo (int n, aint * __restrict__ p, aint * __restrict__ q)
{
   /* feature: support for read accesses with a compile time known misalignment  */
   for (i=0; i<n; i++)
      a[i] = b[i+1] + c[i+3];
}
```

A háromoperandusú operátor használata:

```
typedef int aint __attribute__ ((__aligned__(16)));
int a[256], b[256], c[256];
foo (int n, aint * __restrict__ p, aint * __restrict__ q)
{
   /* feature: support for if-conversion  */
   for (i=0; i<n; i++)
   {
      j = a[i];
      b[i] = (j > MAX ? MAX : 0);
   }
}
```

Igazítható struktúra elérése:

```
struct a
{
  int ca[N];
} s;

for (i = 0; i < N; i++)
    s.ca[i] = 5;
```

Mutató derefernciája olvasásra, változó méretű eltolással:

```
int a[256], b[256];
foo (int x)
{
   int i;

   for (i=0; i<N; i++)
      a[i] = b[i+x];
}
```

Többdimenziós tömbök használata:

```
int a[M][N];
foo (int x)
{
   int i,j;

   for (i=0; i<M; i++)
     for (j=0; j<N; j++)
       a[i][j] = x;
}
```

Összegzés egész típusú tömbök esetén:

```
unsigned int ub[N], uc[N];
foo ()
{
  int i;
  unsigned int diff = 0;
  for (i = 0; i < N; i++)
    udiff += (ub[i] - uc[i]);
}
```

Különböző típusú adatokkal végzett műveletek egy ciklusban. A műveletek azonos típusú adatokon operálnak!

```
short *sa, *sb, *sc;
int *ia, *ib, *ic;
for (i = 0; i < N; i++)
{
  ia[i] = ib[i] + ic[i];
  sa[i] = sb[i] + sc[i];
}
```

Explicit konverzió:

```
for (i = 0; i < N; i++)
  ia[i] = (int) sb[i];
```

```
Sávozott feldolgozás, a ciklusl´épésenként elért adatelemek nem fednek át!
\begin{ccode}
for (i = 0; i < N/2; i++)
{
  a[i] = b[2*i+1] * c[2*i+1] - b[2*i] * c[2*i];
  d[i] = b[2*i] * c[2*i+1] + b[2*i+1] * c[2*i];
}
```

Indukció:

```
for (i = 0; i < N; i++)
  a[i] = i;
```

Külső ciklus (a belső ciklus lesz vektorizálva):

```
for (i = 0; i < M; i++)
{
    diff = 0;
    for (j = 0; j < N; j+=8)
      diff += (a[i][j] - b[i][j]);
    out[i] = diff;
}
```

Dupla redukció:

```
for (k = 0; k < K; k++)
{
    sum = 0;
    for (j = 0; j < M; j++)
      for (i = 0; i < N; i++)
          sum += in[i+k][j] * coeff[i][j];

    out[k] = sum;
}
```

Belső ciklusban használt feltétel:

```
for (j = 0; j < M; j++)
{
   x = x_in[j];
   curr_a = a[0];

   for (i = 0; i < N; i++)
   {
      next_a = a[i+1];
      curr_a = x > c[i] ? curr_a : next_a;
   }

   x_out[j] = curr_a;
}
```

Ciklustól független permutáció:

```
for (i = 0; i < N; i++)
{
   a = *pInput++;
   b = *pInput++;
   c = *pInput++;

   *pOutput++ = M00 * a + M01 * b + M02 * c;
   *pOutput++ = M10 * a + M11 * b + M12 * c;
   *pOutput++ = M20 * a + M21 * b + M22 * c;
}
```

Blokkonkénti feldolgozás:

```
void foo ()
{
  unsigned int *pin = &in[0];
  unsigned int *pout = &out[0];

  *pout++ = *pin++;
  *pout++ = *pin++;
  *pout++ = *pin++;
  *pout++ = *pin++;
}
```

Az alábbi kód azonban például nem vektorizálható:

```
while (*p != NULL) {
  *q++ = *p++;
}
```

10. OpenMP

Az OpenMP a legegyszerűbb párhuzamosítási eszközrendszer. Elsősorban adatpárhuzamosítást tesz lehetővé. A gcc fordítók a 4.0-s verziótól támogatják. Az OpenMP használatához include-olnunk kell az omp.h header-t, amely része a standard C,C++ környezetnek (akárcsak az stdio.h vagy az iostream), és be kell kapcsolnunk egy fordítási flag-et, az **−fopenmp**-t.

Az OpenMP pragmákból és könyvtári függvényekből áll.

10.1. Pragmák

Az OpenMP pragmák általános formája:

```
#pragma omp [opciok]
```

Egy blokk elé elhelyezett **parallel** pragma hatására a blokk több szálon fog elindulni.

```
#pragma omp parallel [kloz[, kloz]...]...
  blokk
```

Az alábbi klózok használhatók a **parallel** pragmában:

- **if(konstans_kifejezes)**,
- **num_threads(egesz_kifejezes)**,

- **default** \{**shared**none}|,
- **private (valtozo_lista)**,
- **firstprivate (valtozo_lista)**,
- **copyin (valtozo_lista)**,
- **reduction (operator: valtozo_lista)**.

Az alábbi példakód 4 szálon fogja elindítani a "Hello vilag!" sztringet kiíró blokkot, s ennek megfelelően négyszer fog az megjelenni a konzolon:

```
#pragma omp parallel num_threads(4)
{
  printf("Hello vilag!\n");
}
```

A **for** pragma hatására az őt követő **for**-ciklust a futtató rendszer fogja elosztani a szálak között, ciklusléptenként.

```
#pragma omp for [kloz[, kloz]...]...
  for_ciklus
```

A felhasználható klózok:

- **private(valtozo_lista)**,
- **firstprivate (valtozo_lista)**,
- **lastprivate(valtozo_lista)**,
- **reduction(operator: valtozo_lista)**,
- **schedule(kind[, chunk_meret])**,
- **collapse(n)**,
- **ordered**,
- **nowait**.

A **sections** pragma hatására az őt követő blokkban lévő blokkokat elosztja a futtató rendszer a szálak között.

```
#pragma omp sections [kloz[, kloz]...]...
{
  [pragma omp section
  blokk]
  [pragma omp section
  blokk]...
}
```

A felhasználható klózok listája:

- **private(valtozo_lista)**,
- **firstprivate (valtozo_lista)**,
- **lastprivate(valtozo_lista)**,
- **reduction(operator: valtozo_lista)**,
- **nowait**.

A **single** pragma hatására az őt követő blokkot csak egy szál fogja lefuttatni (ha a konstrukció például **sections** pragmát követő blokkban szerepel):

```
#pragma omp single [kloz[, kloz]...]...
  blokk
```

A felhasználható klózok:

- **private(valtozo_lista)**,

- **firstprivate (valtozo_lista)**,
- **copyprivate(valtozo_lista)**,
- **nowait**.

A korábbi konstrukciók összevonhatóak egy pragmává, az alkalmazható klózok uniójával:

```
#pragma omp parallel for [kloz[, kloz]...]...
  for_ciklus
#pragma omp parallel sections [kloz[, kloz]...]...
{
  [#pragma omp section
    blokk]
  [#pragma omp section
    blokk]...
}
```

A **task** konstrukcióval explicit módon definiálhatunk taszkokat. A taszk adatkörnyezetát a megfelelő klózok definiálják.

```
#pragma omp task [kloz[, kloz]...]...
  blokk
```

Az OpenMP párhuzamosítás legegyszerűbb módja a for-ciklus előtt elhelyezet pragma:

```
#pragma omp parallel for
for ( int i= ...)
```

Ezt követően a for-ciklus annyi processzoron fog futni, amennyi a gépünkben található. A párhuzamos kódot a fordító állítja elő. A felhasználható párhuzamos szálak beállítására az omp.h header-ben specifikált **omp_set_num_threads** függvényt használhatjuk, melynek egyetlen paramétere a párhuzamos szálak száma. A függvényhívást követő minden pragma utáni ciklus a függvényhívásban beállított számú szálon fog futni.

Példaként tekintsük a tesztkörnyezetünkben megvalósított átlagoló szűrő OpenMP alapú párhuzamosítását! Cmake környezetünket tehát úgy készíthetjük fel OpenMP párhuzamosítás fordítására, hogy a fordítási kapcsolók közzé felvesszük az **−fopenmp**-t, azaz a **lib/CMakeLists.txt** állományban a következőre módosítjuk a fordítási kapcsolókat taratalmazó sort:

```
SET(CMAKE_CXX_FLAGS "-fopenmp -O3 -Wall -Wextra")
```

Ezt követően hozzuk létre a **MeanFilterOpenMP.h** és **MeanFilterOpenMP.cc** állományokat a következő tartalommal:

10.1.1. forráskód: lib/MeanFilterOpenMP.h

```
#ifndef _MEAN_FILTER_OPENMP_
#define _MEAN_FILTER_OPENMP_

#include <omp.h>

#include <MeanFilter.h>

class MeanFilterOpenMP: public MeanFilter
{
public:
  using MeanFilter::apply;

  MeanFilterOpenMP(int w, int numberOfThreads);

  MeanFilterOpenMP(const MeanFilterOpenMP& mf);
```

```
  ~MeanFilterOpenMP();

  virtual void apply(Image& input, Image& output);

  int numberOfThreads;
};

#endif
```

10.1.2. forráskód: lib/MeanFilterOpenMP.cc

```
#include <MeanFilterOpenMP.h>

MeanFilterOpenMP::MeanFilterOpenMP(int w, int numberOfThreads)
: MeanFilter(w)
{
  this->numberOfThreads= numberOfThreads;
}

MeanFilterOpenMP::MeanFilterOpenMP(const MeanFilterOpenMP& mf)
: MeanFilter(mf), numberOfThreads(mf.numberOfThreads)
{
}

MeanFilterOpenMP::~MeanFilterOpenMP()
{
}

void MeanFilterOpenMP::apply(Image& input, Image& output)
{
  int w2= w/2;

  omp_set_num_threads(this->numberOfThreads);

  #pragma omp parallel for
  for ( int i= 0; i < input.rows; ++i )
    for ( int j= 0; j < input.columns; ++j )
      output(i,j)= apply(input, i, j);
}
```

A MeanFilterOpenMP.h a MeanFilter-ből származó MeanFilterOpenMP funktor specifikációját tartalmazza. Felhasználjuk a MeanFilter-ben definiált "belső" **apply** függvényt, a párhuzamosítást a felüldefiniált "külső" **apply** függvényben végezzük. Az örökölt **w** kernel méreten kívül létrehozunk egy **numberOfThreads** adattagot, amely azt tartalmazza, hogy hány szálon végezzük a párhuzamosítást. Ezt a paramétert osztályunk a konstruktorán keresztül veszi át. Lássuk magát a párhuzamosítást: a "külső" **apply** függvényben elhelyezzük a párhuzamosításnál használt szálak számának beállítására szolgáló függvényt, majd a külső for-ciklus előtt a pragma-t. Ezzel a párhuzamosítást kódolását be is fejeztük. A futási idő mérésére létre kell hoznunk egy új időmérő eszközt, ugyanis a **time.h** felhasználásával működő **StopperCPU.h** hibás időt mér párhuzamos futtatás esetén.

10.1.3. forráskód: lib/StopperOpenMP.h

```
#ifndef _STOPPER_OPENMP_H_
#define _STOPPER_OPENMP_H_

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <time.h>
```

```
#include <Stopper.h>

/**
  * StopperOpenMP is a simple tool to measure time intervals
  */
class StopperOpenMP: public Stopper
{
public:
    /**
      * starts the stopper
      */
    virtual void start();

    /**
      * stops the stopper
      */
    virtual void stop();

    /**
      * converts to time
      */
    virtual void convertToTime();

    double begin;

    double end;
};

#endif
```

10.1.4. forráskód: lib/StopperOpenMP.cc

```
#include <StopperOpenMP.h>

#include <sstream>
#include <iostream>
#include <omp.h>

#define _CRT_SECURE_NO_WARNINGS

void StopperOpenMP::start()
{
  begin= omp_get_wtime();
}
void StopperOpenMP::stop()
{
  end= omp_get_wtime();
}
void StopperOpenMP::convertToTime()
{
  double d= end - begin;

  d*=1000;

  ms= d;
  s= ms / 1000;
  m= s / 60;
  h= m / 60;

  ms= ms % 1000;
  s= s % 60;
  m= m % 60;
}
```

Időmérő eszközünk nagyon egyszerű időmérést valósít meg, az **omp.h** header-ben definiált **omp_get_wtime ()** függvény segítségével, amely az aktuális időt adja vissza a program indítása óta, lebegőpontos számként, másodpercekben. A **start** és **stop** függvények által rögzített **begin** és **end** adattagokban tárolt értékeket a **convertToTime()** függvény alakítja óra, perc, másodperc és ezredmásodperc értékekké, amelyeket a megfelelő adattagokban tárol. Ezekből az ősosztály **Stopper.h**-ban definiált **getElapsedTime ()** alakítja sztringgé, amit visszatérési értékként ad vissza. Lássuk, hogyan kell módosítanunk az alkalmazásunkat ahhoz, hogy egyszerűen kipróbálhassuk az openmp alapú párhuzamosítást: includeoljuk alkalmazásunkba a **MeanFilterOpenMP.h** és **StopperOpenMP.h** header-öket. Hozzunk létre a főprogramunkban egy új logikai változót **openmp** néven:

```
bool openmp;
```

Adjunk hozzá egy új kapcsolót parancssori argumentumokat feldolgozó OptionTable objektumhoz, amely beállítja az **openmp** változó értékét! Valamint adjunk hozzá egy olyan kapcsolót ("–threads"), amely az kapcsolót követő szám paraméterrel beállítja a párhuzamos futtatás szálainak számát!

```
ot.addOption("--openmp", OPTION_BOOL, (char*)&openmp, 0, "mean filter using openmp");
ot.addOption("--threads", OPTION_INT, (char*)&numberOfThreads, 1, "number of parallel
    threads");
```

Adjunk hozzá egy új ágat a már létező **if** szerkezethez, amely az openmp kapcsoló jelenléte esetén a MeanFilterOpenMP és StopperOpenMP objektumokat adja át a filter függvénynek:

```
else if
  return filter(argv[1], argv[2], new MeanFilterOpenMP(kernelSize, numberOfThreads, new
      StopperOpenMP());
```

Párhuzamosan futó kódunkat most a $--$**openmp** kapcsolóval próbálhatjuk ki, például két szálon:

```
./filterApp --openmp --kernel 21 --threads 2 lena.png lena-filtered.png
```

A program kimenete:

```
execution time: 0:0:0.984 (h:m:s)
```

Azaz ebben az esetben a filterezés kevesebb, mint 1 másodperc alatt lefutott. A futási idő természetesen függ a háttérben futó egyéb alkalmazásoktól, de a gyorsulás nagyságrendileg kétszeres, a két szálon történő futtatásnak megfelelően. A több szálon történő futásról egy új konzol ablakban kiadott **top** utasítással győződhetünk meg. A **top** utasítás a futó alkalmazások erőforrásfelhasználását monitorozza. Miközben párhuzamosan, például 2 szálon fut a **filterApp** alkalmazás (nagyobb kernel mérettel, pl. 51, több másodpercig is tarthat), a **top** felsorolásában a **filterApp** mellett 200%-os processzor használat jelenik meg, ami a két mag 100-100%-os használatára utal.

Appendix A: Tesztkörnyezet

1.0.1. forráskód: A tesztkörnyezet szerkezete

```
parallelismTests
   app
      CMakeLists.txt
      main.cc
   lib
      CMakeLists.txt
      Filter.cc
```

```
        Filter.h
        IO.cc
        IO.h
        Image.cc
        Image.h
        MeanFilter.cc
        MeanFilter.h
        OptionTable.cc
        OptionTable.h
        Stopper.cc
        Stopper.h
        StopperCPU.cc
        StopperCPU.h
    CMakeLists.txt
```

1.0.2. forráskód: CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

SUBDIRS(lib app)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

1.0.3. forráskód: lib/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

PROJECT(filter C CXX)

SET(PACKAGE_NAME filter)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
SET(PATCH_VERSION 0)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_COMPILER gcc)
SET(CMAKE_CXX_COMPILER g++)

SET(CMAKE_CXX_FLAGS "-O3 -Wall -Wextra")

AUX_SOURCE_DIRECTORY(. SRCVAR)

ADD_LIBRARY(${PACKAGE_NAME} SHARED ${SRCVAR})

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} png)

INCLUDE_DIRECTORIES(. /usr/include)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

1.0.4. forráskód: app/CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

PROJECT(filterApp C CXX)

SET(PACKAGE_NAME filterApp)
SET(MAJOR_VERSION 0)
SET(MINOR_VERSION 0)
```

```
SET(PATCH_VERSION 0)
SET(PACKAGE_VERSION ${MAJOR_VERSION}.${MINOR_VERSION}.${PATCH_VERSION})

SET(CMAKE_C_COMPILER gcc)
SET(CMAKE_CXX_COMPILER g++)

SET(CMAKE_CXX_FLAGS "-fopenmp -O3 -Wall -Wextra")

AUX_SOURCE_DIRECTORY(. SRCVAR)

ADD_EXECUTABLE(${PACKAGE_NAME} ${SRCVAR})

LINK_DIRECTORIES(../lib /usr/lib)

TARGET_LINK_LIBRARIES(${PACKAGE_NAME} filter png pthread)

INCLUDE_DIRECTORIES(. /usr/include ../lib)

SET(CMAKE_VERBOSE_MAKEFILE on)
```

1.0.5. forráskód: lib/Image.h

```
#ifndef _IMAGE_H_
#define _IMAGE_H_

#include <vector>

class Image: public std::vector<unsigned char>
{
public:
  using std::vector<unsigned char>::resize;
  using std::vector<unsigned char>::operator[];

  Image();

  Image(const Image& im);

  Image(int rows, int columns);

  ~Image();

  void resizeImage(int rows, int columns);

  unsigned char& operator()(int n);

  unsigned char operator()(int n) const;

  unsigned char& operator()(int row, int column);

  unsigned char operator()(int row, int column) const;

  int rows;
  int columns;
};

#endif
```

1.0.6. forráskód: lib/Image.cc

```
#include <Image.h>
#include <stdio.h>
```

```
Image::Image()
: std::vector<unsigned char>()
{
  this->rows= 0;
  this->columns= 0;
}

Image::Image(const Image& im)
: std::vector<unsigned char>(im)
{
  this->rows= im.rows;
  this->columns= im.columns;
}

Image::Image(int rows, int columns)
: std::vector<unsigned char>(rows*columns)
{
  this->rows= rows;
  this->columns= columns;
}

Image::~Image()
{
}

void Image::resizeImage(int rows, int columns)
{
  this->resize(rows*columns);
  this->rows= rows;
  this->columns= columns;
}

unsigned char& Image::operator()(int row, int column)
{
  return this->operator[](row * this->columns + column);
}

unsigned char Image::operator()(int row, int column) const
{
  return this->operator[](row * this->columns + column);
}

unsigned char& Image::operator()(int n)
{
  return this->operator[](n);
}

unsigned char Image::operator()(int n) const
{
  return this->operator[](n);
}
```

1.0.7. forráskód: lib/IO.h

```
#ifndef _IO_H_
#define _IO_H_

#include <png.h>
#include <stdio.h>

#include <Image.h>

int readPNG(char* filename, Image& r, Image& g, Image& b);
```

```
int writePNG(char* filename, Image& r, Image& g, Image& b);

#endif
```

1.0.8. forráskód: lib/IO.cc

```
#include <stdlib.h>

#include <IO.h>

int writePNG(char* filename, Image& r, Image& g, Image& b)
{
  #ifdef DEBUG
    printf("writing image..."); fflush(stdout);
  #endif

  FILE* file;
  file= fopen(filename, "wb");

  png_structp png_ptr= png_create_write_struct(PNG_LIBPNG_VER_STRING, 0, 0, 0);

  png_infop info_ptr= png_create_info_struct(png_ptr);

  png_init_io(png_ptr, file);

  png_set_IHDR(png_ptr, info_ptr, r.columns, r.rows, 8, PNG_COLOR_TYPE_RGB,
      PNG_INTERLACE_NONE, PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);

  png_write_info(png_ptr, info_ptr);

  png_bytepp row_pointers;

  png_uint_32 rowbytes;

  row_pointers= (png_byte**)png_malloc(png_ptr, r.rows*sizeof(png_bytep));
  for ( int i= 0; i < r.rows; ++i )
    row_pointers[i]= (png_byte*)png_malloc(png_ptr, r.columns*3*sizeof(png_byte));

  rowbytes= 3 * r.columns;

  int cc;
  int rr= 0;

  for ( int i= 0; i < r.rows; ++i )
  {
    cc= 0;
    for ( unsigned int j= 0; j < rowbytes; j+= 3 )
    {
      row_pointers[i][j]= r(rr + cc);
      row_pointers[i][j+1]= g(rr + cc);
      row_pointers[i][j+2]= b(rr + cc);
      cc+= 1;
    }
    rr+= r.columns;
  }

  png_write_rows(png_ptr, row_pointers, r.rows);

  png_write_end(png_ptr, info_ptr);

  png_destroy_write_struct(&png_ptr, &info_ptr);

  #ifdef DEBUG
    printf("ok\n"); fflush(stdout);
```

```
  #endif

  return 0;
}

int readPNG(char* filename, Image& r, Image& g, Image& b)
{
  FILE* file;
  file= fopen(filename, "r");

  png_uint_32 rows;
  png_uint_32 columns;
  int bit_depth;
  int color_type;

  unsigned char sig[8];

  fread(sig, 1, 8, file);
  if ( !png_check_sig(sig, 8) )
    return 1;

  png_structp png_ptr;
  png_infop info_ptr;

  png_ptr= png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
  if ( !png_ptr )
    return 4;

  info_ptr= png_create_info_struct(png_ptr);
  if ( !info_ptr )
  {
    png_destroy_read_struct(&png_ptr, NULL, NULL);
    return 4;
  }

  png_init_io(png_ptr, file);

  png_set_sig_bytes(png_ptr, 8);

  png_read_info(png_ptr, info_ptr);

  png_get_IHDR(png_ptr, info_ptr, &columns, &rows, &bit_depth, &color_type, NULL, NULL,
      NULL);

  #ifdef DEBUG
    printf("reading png image - rows: %d, columns: %d, bit depth: %d, color type: %d\n",
        rows, columns, bit_depth, color_type); fflush(stdout);
  #endif

  if ( color_type == PNG_COLOR_TYPE_RGB )
  {
    #ifdef DEBUG
      printf("reading image data..."); fflush(stdout);
    #endif

    png_uint_32 rowbytes;
    png_bytep row_pointers[rows];

    rowbytes= png_get_rowbytes(png_ptr, info_ptr);

    r.resizeImage(rows, columns);
    g.resizeImage(rows, columns);
    b.resizeImage(rows, columns);

    for ( unsigned int i= 0; i < rows; ++i )
```

```
      row_pointers[i]= (png_bytep)(malloc(rowbytes));

    png_read_image(png_ptr, row_pointers);

    int cc;
    for ( unsigned int i= 0; i < rows; ++i )
    {
      cc= 0;
      for ( unsigned int j= 0; j < columns * 3; j+= 3 )
      {
    r(i,cc)= row_pointers[i][j];
    g(i,cc)= row_pointers[i][j+1];
    b(i,cc)= row_pointers[i][j+2];
    cc+= 1;
      }
    }

    png_read_end(png_ptr, NULL);

    for ( unsigned int i= 0; i < rows; ++i )
      free(row_pointers[i]);

    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);

    #ifdef DEBUG
      printf("ok\n"); fflush(stdout);
    #endif
  }

  return 0;
}
```

1.0.9. forráskód: lib/Stopper.h

```
#ifndef _STOPPER_H_
#define _STOPPER_H_

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>

/**
  * Stopper is a simple tool to measure time intervals
  */
class Stopper
{
public:
    /**
      * starts the stopper
      */
    virtual void start();

    /**
      * stops the stopper
      */
    virtual void stop();

    /**
      * converts to time
      */
    virtual void convertToTime();

    /**
```

```
        * prints out the difference
        * @param format the in which to print out the time difference
        */
      virtual std::string getElapsedTime();

      /**
        * hours
        */
      unsigned int h;

      /**
        * minutes
        */
      unsigned int m;

      /**
        * seconds
        */
      unsigned int s;

      /**
        * milliseconds
        */
      unsigned int ms;
};

#endif
```

1.0.10. forráskód: lib/Stopper.cc

```
#include <Stopper.h>

#include <sstream>
#include <iostream>

#define _CRT_SECURE_NO_WARNINGS

void Stopper::start()
{
}
void Stopper::stop()
{
}
void Stopper::convertToTime()
{
}

std::string Stopper::getElapsedTime()
{
  convertToTime();
  char buffer[256];

  sprintf(buffer,"%d:%d:%d.%d (h:m:s)", h, m, s, ms);

  return std::string(buffer);
}
```

1.0.11. forráskód: lib/StopperCPU.h

```
#ifndef _STOPPER_CPU_H_
#define _STOPPER_CPU_H_
```

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <time.h>
#include <Stopper.h>

/**
  * StopperCPU is a simple tool to measure time intervals
  */
class StopperCPU: public Stopper
{
public:
    /**
      * starts the stopper
      */
    virtual void start();

    /**
      * stops the stopper
      */
    virtual void stop();

    /**
      * converts to time
      */
    virtual void convertToTime();

    clock_t begin;

    clock_t end;
};

#endif
```

1.0.12. forráskód: lib/StopperCPU.cc

```
#include <StopperCPU.h>

#include <sstream>
#include <iostream>

#define _CRT_SECURE_NO_WARNINGS

void StopperCPU::start()
{
  begin= clock();
}
void StopperCPU::stop()
{
  end= clock();
}
void StopperCPU::convertToTime()
{
  float d= end - begin;

  d*=1000;
  d/=CLOCKS_PER_SEC;

  ms= d;
  s= ms / 1000;
  m= s / 60;
  h= m / 60;
```

```
  ms= ms % 1000;
  s= s % 60;
  m= m % 60;
}
```

1.0.13. forráskód: lib/OptionTable.h

```
#ifndef _OPTIONTABLE_H
#define _OPTIONTABLE_H

#include <iostream>
#include <string>
#include <vector>

/**
  * Enum for possible Option types
  */
typedef enum
{
  OPTION_INT,
  OPTION_FLOAT,
  OPTION_CHAR,
  OPTION_DOUBLE,
  OPTION_BOOL,
  OPTION_HELP,
} OptionType;

/**
  *Option class specify a possible option in the command line argument vector.
  */
class Option
{
public:
  /**
    * Constructor of the Option class.
    * @param nameL option name
    * @param optionL type of the option
    * @param pL pointer to the option
    * @param nL number of parameters after the switch
    * @param descriptionL description of the option
    */
  Option(std::string nameL, OptionType optionL, char* pL, unsigned int nL, std::string
      descriptionL);

  /**
    * toString method to describe Option object
    * @return string descriptor
    */
  std::string toString() const;

  /**
    * pointer to the argument
    */
  char* p;

  /**
    * number of parameters following the switch in the command line
    */
  unsigned int n;

  /**
    * name of the option
    */
  std::string name;
```

```
  /**
    * type of the option
    */
  OptionType optionType;

  /**
    * description of the option
    */
  std::string description;
};

/**
  * OptionTable class to cover a set of Options.
  */
class OptionTable: public std::vector<Option*>
{
public:

  /**
    * Constructor for the option table.
    */
  OptionTable();

  /**
    * Method for adding a new option to the table.
    * @param name option name
    * @param option option type
    * @param p option pointer
    * @param n number of parameters after the command line switch
    * @param description description of the option
    */
  void addOption(std::string name, OptionType option, char* p, unsigned int n, std::
      string description);

  /**
    * Method to process the command line arguments.
    * @param argc number of command line arguments
    * @param argv pointer to command line argument array
    */
  void processArgs(int* argc, char** argv);

  /**
    * dump method of the optionTable class.
    */
  void dump();

  /**
    * setting the prefix text.
    * @param prefixL table prefix text
    */
  void setPrefix(std::string prefixL);

  /**
    * setting the postfix text.
    * @param postfixL table postfix text
    */
  void setPostfix(std::string postfixL);

  /**
    * toString method to describe OptionTable object
    * @return descriptor string
    */
  std::string toString() const;
```

```
  /**
    * prefix text to print out with the option table
    */
  std::string prefix;

  /**
    * postfix text to print out with the option table
    */
  std::string postfix;
};

#endif  /* _OPTIONTABLE_H */
```

1.0.14. forráskód: lib/OptionTable.cc

```
#include <OptionTable.h>
#include <stdio.h>
#include <sstream>
#include <string>
#include <string.h>
#include <iostream>

Option::Option(std::string nameL, OptionType optionL, char* pL, unsigned int nL, std::
    string descriptionL)
{
  p= pL;
  n= nL;
  name= nameL;
  optionType= optionL;
  description= descriptionL;
  switch (optionType) {
      case OPTION_INT:
    *((int *) p) = 0;
    break;
      case OPTION_FLOAT:
    *((float *) p) = 0.0f;
    break;
      case OPTION_CHAR:
    *((char *) p) = 0;
    break;
      case OPTION_DOUBLE:
    *((double *) p) = 0.0;
    break;
      case OPTION_BOOL:
    *((bool *)p) = false;
    break;
      case OPTION_HELP:
    break;
  }
}

std::string Option::toString() const
{
  std::vector<char> tmp(name.size() + description.size() + 40);

  switch (optionType)
  {
    case OPTION_INT:
  sprintf(&(*(tmp.begin())), "\t%-25s%-4d%-8s%s", name.c_str(), n, "int", description.
      c_str());
  break;
    case OPTION_FLOAT:
  sprintf(&(*(tmp.begin())), "\t%-25s%-4d%-8s%s", name.c_str(), n, "float", description.
      c_str());
```

```
  break;
    case OPTION_DOUBLE:
  sprintf(&(*(tmp.begin())), "\t%-25s%-4d%-8s%s", name.c_str(), n, "double", description
      .c_str());
  break;
    case OPTION_BOOL:
  sprintf(&(*(tmp.begin())), "\t%-25s%-4d%-8s%s", name.c_str(), n, "bool", description.
      c_str());
  break;
    case OPTION_CHAR:
  sprintf(&(*(tmp.begin())), "\t%-25s%-4d%-8s%s", name.c_str(), n, "char", description.
      c_str());
  break;
    case OPTION_HELP:
  sprintf(&(*(tmp.begin())), "\n\t%s\n", description.c_str());
  break;
  }

  return std::string(&(*(tmp.begin())));
}

OptionTable::OptionTable()
: std::vector<Option*>()
{
  prefix= std::string("");
  postfix= std::string("");
}

void OptionTable::addOption(std::string name, OptionType option, char* p, unsigned int n
    , std::string description)
{
  this->push_back(new Option(name, option, p, n, description));
}

void OptionTable::processArgs(int* argc, char** argv)
{
  std::vector<Option*>::iterator it= this->begin();
  unsigned int i, j;

  for (i = 0; (int)i < *argc; ++i)
  {
      if (strcmp(argv[i], "--help") == 0)
      {
    dump();
    return;
      }
  }

  while (it != this->end())
  {
      Option *o = *it;
      for (i = 0; (int)i < *argc; ++i)
      {
    if (strcmp(argv[i], o->name.c_str()) == 0)
    {
        for (j = 0; j <= o->n; ++j)
      if (i + j <= (unsigned int)(*argc))
          switch (o->optionType)
          {
        case OPTION_INT:
            if (j != 0)
          sscanf(argv[i + j], "%d", ((int *) (o->p)) + j - 1);
            break;
        case OPTION_FLOAT:
            if (j != 0)
```

```
            sscanf(argv[i + j], "%f", ((float *) (o->p)) + j - 1);
              break;
          case OPTION_CHAR:
              if (j != 0)
            sscanf(argv[i + j], "%s", ((char *) (o->p)) + j - 1);
              break;
          case OPTION_BOOL:
              *((bool *)(o->p)) = true;
              break;
          case OPTION_DOUBLE:
              if (j != 0) {
            double d;
            sscanf(argv[i + j], "%lf", &d);
            *(((double *) (o->p) + j - 1)) = d;
              }
              break;
          case OPTION_HELP:
              break;
            };

          for (unsigned int k = i; k < *argc - j; ++k)
          {
        argv[k] = argv[k + j];
          }
          *argc -= j;
          break;
      }
        }
        ++it;
    }
}

std::string OptionTable::toString() const
{
  std::stringstream ss;

  ss << prefix;
  ss << std::endl;

  ss << std::endl;
  ss << "Parameters:";
  ss << std::endl;
  ss << std::endl;

  for (std::vector<Option*>::const_iterator it= this->begin(); it != this->end(); ++it )
  {
      ss << (**it).toString();
      ss << std::endl;
  }

  ss << std::endl;
  ss << postfix;
  ss << std::endl;

  return ss.str();
}

void OptionTable::dump()
{
  std::cout << this->toString();
}

void OptionTable::setPrefix(std::string prefixL)
{
  prefix= prefixL;
```

```
}


void OptionTable::setPostfix(std::string postfixL)
{
  postfix= postfixL;
}
```

References