



EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK TANSZÉK

Bináris kompatibilitás vizsgálata C++ nyelvben

Témavezető:

Dr. Porkoláb Zoltán

Programozási Nyelvek és Fordítóprogramok Tanszék
Egyetemi Docens, PhD.

Szerző:

Ispánovics Gábor Alex

Programtervező Informatikus szak
Nappali tagozat

Készült az Ericsson Magyarország Kft. támogatásával.
Budapest, 2014

Tartalomjegyzék

1. Bevezetés	3
2. Motiváció	5
3. Technikai alapok	7
3.1. Szimbólumok	7
3.2. A bináris kompatibilitás formálisan	8
3.2.1. Fordítás	8
3.2.2. Szerkesztés	8
3.2.2.1. Dinamikus programkönyvtárra szerkesztés	8
3.2.2.2. Futtatható programra szerkesztés	9
3.2.3. Futtatás	9
3.2.3.1. Specifikáció	10
3.2.4. Dinamikus programkönyvtárak	10
3.2.5. A bináris kompatibilitás definíciója	10
3.2.5.1. Forrás kompatibilitás	11
3.2.6. Kliens kód	11
3.3. A bináris inkompatibilitás okai	11
4. Tanulmány a bináris inkompatibilitásról	13
4.1. Implementáció változás	13
4.1.1. Változások a kivételkezelésben	14
4.2. Interfész változás	15
4.2.1. Adatszerkezet változás	15
4.2.1.1. A programkönyvtár memóriájának használata	15
4.2.1.2. Kliens által foglalt memória	16
4.2.1.3. Adattag jogok	17
4.2.2. Szimbólumváltozások	18
4.2.2.1. Globális változók	19
4.2.2.2. Függvények	19
4.2.3. A polimorfizmus bináris kompatibilitási vonzatai	20
4.2.3.1. Virtuális metódusok	21
4.2.3.2. Virtuális metódusok felüldefiniálása	21
4.2.4. A sablonok szerepe a bináris kompatibilitásban	23

5. A bináris kompatibilitás ellenőrzése	25
5.1. A bináris kód elemzése	25
5.2. Statikus kódelemzés	27
5.2.1. A forráskód értelmezése	27
5.2.2. Modell építés	28
6. Kapcsolódó munkák	29
6.1. abi-compliance-checker	29
6.2. shlib-compat	30
7. Tanácsok, technikák	31
7.1. Verziózás	31
7.1.1. Szimbólum verziózás bináris szinten	31
7.1.1.1. Esettanulmány	31
7.1.1.2. Limitációk	32
7.1.1.3. Konklúzió	32
7.1.2. Verziózás névterekkel	33
7.1.2.1. Esettanulmány C++03 szabványú kóddal	33
7.1.2.2. Esettanulmány C++11 szabványú kóddal	33
7.1.2.3. Konklúzió	34
7.1.3. Az új viselkedés elérhetőségének szabályozása	34
7.2. Híd programtervezési minta használata	34
8. Eredmények	36
9. Konklúzió	38

1. fejezet

Bevezetés

A C++ programozási nyelv napjaink egyik legnépszerűbb programozási nyelve [11]. A nyelv egyik vonzereje abban rejlik, hogy kombinálja az alacsony szintű kód hatékonyságát a magas szintű objektum-orientált, valamint a generikus programozás szemléletével.

Mint más modern programozási nyelvek esetén, a C++ esetén is lehetőség van programkönyvtárakat létrehozni. A programkönyvtárak viselkedések implementációját tartalmazzák, amelyet egy jól definiált interfészen keresztül érhetünk el. Egy programkönyvtárat több alkalmazásban, illetve más programkönyvtárakban újra felhasználhatunk. A programkönyvtárak egy fajtája az úgynevezett dinamikus programkönyvtár¹. Azok az alkalmazások amelyek dinamikus programkönyvtárakat használnak a programkönyvtárat futás időben töltik be, így a dinamikus programkönyvtárak kicserélhetőek, majd az alkalmazás újbóli futtatásakor már a kicserélt változat kerül betöltésre.

Akkor merül fel a bináris kompatibilitás problémája, mikor egy rendszerre egy dinamikus programkönyvtár egy új verzióját telepítik fel anélkül, hogy az azt használó programokat újrafordították volna. Bármilyen inkompatibilitás hibákat eredményezhet a könyvtárat használó alkalmazás elindításakor vagy később a futtatás közben amely az alkalmazás hibás működéséhez vagy teljes leállításához vezethet.

A bináris kompatibilitás vizsgálatában az egyik nagy nehézség, hogy a C++ programozási nyelv szabvány [7] nem definiálja a (dinamikus) programkönyvtárak fogalmát. Bár a szabvány nem ismeri a dinamikus programkönyvtárak fogalmát, az már a kezdetektől használt technika az informatikában, így szinte minden rendszeren elérhető valamilyen formában. A nyelvi szintű szabványosítás hiányának egy másik kellemetlen következménye, hogy a fordítóprogram készítőkre bízta az implementációs kérdések megoldását, így az egyes fordítók más-más bináris kódot generálnak ugyanazon kód fordításakor.

Látható tehát, hogy ahhoz, hogy megtudjuk mely változtatások okozhatnak bináris kompatibilitási problémákat, a fordítók belső implementációiba kell betekinteni. Számos weboldal és cikk [5, 1, 9, 6] foglalkozott már a bináris kompatibilitás kérdésével, de ezek többsége csak tanácsokat ad, hogy melyek azok a módosítások amelyek kerülendőek egy adott fordítóprogram esetén, az ellenőrzésről többnyire nem esik szó, összefoglaló munkát nem találtam ebben a

¹Angolul shared library

témában.

A diplomamunka bemutatja a bináris kompatibilitás feltételeit a C++ nyelvben (figyelembe véve a C++11 szabvány újdonságait is), tanácsokat ad és technikákat mutat be melyekkel megőrizhető a bináris kompatibilitás, valamint a diplomamunka célja, hogy megvizsgálja mely módszerek alkalmasak a bináris kompatibilitás automatikus ellenőrzésére.

A következő fejezetben példákon keresztül mutatom meg a probléma fontosságát. A diplomamunka harmadik fejezetében ismertetem a technikai alapokat, amelyek a bináris kompatibilitás ellenőrzését elengedhetetlenek, valamint formálisan definiálom a bináris kompatibilitás fogalmát. A negyedik fejezetben áttekintem, hogy mely tényezőktől függhet egy dinamikus programkönyvtár bináris kompatibilitása, figyelembe véve az elterjedt fordítóprogramok működését. Az ötödik fejezetben bemutatok két megközelítést amellyel ellenőrizni lehet a bináris kompatibilitást egy dinamikus programkönyvtár két verziója között. A hatodik fejezetben áttekintem azokat az eszközöket amelyek jelenleg a rendelkezésünkre állnak a bináris kompatibilitás ellenőrzésére. A hetedik fejezetben különböző megközelítéseket, technikákat ismertetek amelyek révén könnyebben készíthetők binárisan kompatibilis új verziók. A nyolcadik fejezetben ismertetem a statikus kódelemzésen alapuló megközelítés demonstrálására készített program eredményeit, összehasonlítva egy másik eszközzel. A kilencedik fejezetben levonom a munkám során gyűjtött tanulságokat.

2. fejezet

Motiváció

Tekintsünk egy dinamikus programkönyvtárat amelyet több különböző fejlesztő használ a saját alkalmazásában vagy könyvtárában. Továbbá tegyük fel, hogy a kliensek különböznek a dinamikus programkönyvtár fejlesztő csapattól (vagy társaságtól). Esetleg az is előfordulhat, hogy a könyvtár fejlesztői nem is ismerik a klienseiket.

Egy ilyen környezetben bármilyen olyan változtatás amely megköveteli a kliens kódjának újrafordítását a könyvtár interfészének megváltozása miatt¹, sok feladatot jelent mind a kliensek, mind a könyvtár fejlesztői számára: értesíteni kell a klienseket a változásról (amennyiben ez egyáltalán lehetséges), a klienseknek szinkronizálniuk kell a saját kiadási ciklusaikat a könyvtáréhoz, valamint azon kliensek számára akik nem tudnak valamilyen okból frissíteni, továbbra is támogatni kell a régi verziót. Ez a könyvtár fejlesztőinek és a klienseknek is megterhelő lehet. A probléma tehát a függőségek kezelése: ha a klienseknek nem kell újrafordítaniuk a terméküket az új könyvtár verzió használatához akkor nem jelent problémát számukra a frissítés.

Ha tehát megköveteljük a programkönyvtárak fejlesztőitől, hogy az interfész ne változzon az előző verziókhöz képest, akkor a probléma megoldódik. Egy ilyen megkötés azonban elfogadhatatlan lenne egy könyvtár fejlesztőjének. A valóságban a programkönyvtárak interfésze időről-időre megváltozik ahogy újabb képességekkel, szolgáltatásokkal bővülnek. A fentebbi megkötést azonban enyhíthetjük, hogy csak bizonyos változtatásokat, bővítéseket engedünk meg egy programkönyvtár új verziójának interfészében, így őrizve a bináris kompatibilitást.

Az egyik problémát az jelenti, hogy C++ programozási nyelv szabványa [7] nem definiálja a programkönyvtárak fogalmát, azaz nincs szabványos leírása, hogy hogyan kell kinéznie egy programkönyvtárnak bináris szinten, ez fordítótól, operációs rendszertől és processzor típustól függően eltérhet. A helyzetet tovább bonyolítja, hogy egyes fordítóprogramok esetén (mint például a Microsoft Visual C++ fordító) nem is ismert pontosan, hogy milyen szabályok, konvenciók alapján készülnek a binárisok. Egy programkönyvtár készítőnek tehát igen nagy feladatot jelent a bináris kompatibilitás ellenőrzése verzióról-verzióra.

Fontos tehát elemezni, hogy melyek azok a tényezők amelyek befolyásolják a bináris kompatibilitást, valamint ki kell dolgozni egy olyan eljárást amellyel jól

¹Az interfészváltozást kiváltó módosításokról bővebben a 3.3. részben olvashat.

lehet ellenőrizni egy programkönyvtár két verziója közötti kompatibilitást. Egy programkönyvtár tesztelési folyamatába célszerű beépíteni a bináris kompatibilitás ellenőrzését is, hogy elkerülhetőek legyenek a fentebb leírt problémák.

3. fejezet

Technikai alapok

3.1. Szimbólumok

A C++ programozási nyelv egyik tervezési szempontja volt, hogy forrásszintű kompatibilitást tartson fenn a C programozási nyelvvel [10], valamint a C nyelvű dinamikus programkönyvtárakkal. Ez a döntés főként annak is köszönhető, hogy a C++ tervezésének idejében a C nyelv már széles körben elterjedt programozási nyelv volt és a nyelv tervezői egy már kipróbált alacsony szintű nyelvre akarták építeni a C++ programozási nyelvet. Amellett, hogy azóta is törekednek a minél nagyobb kompatibilitás megőrzésére, a teljes kompatibilitás sosem volt a tervezők célja.

Ennek a tervezési döntésnek az egyik következménye, hogy a C nyelvhez hasonlóan a C++ is szimbólumokkal hivatkozik a bináris fájlokban található adatokra. A szimbólumok tulajdonképpen egyszerű karakterláncok melyek speciális jelentéssel bírnak. A szimbólumneveket a hivatkozott entitás nevéből generálják, ezért a szimbólumok szemantikáját nem találjuk meg a bináris kódokban, ezeket az információkat a fordítók a mellékelt fejálmányókból és a fordított forrásfájlból állapítják meg. Annak ellenére, hogy maga a koncepció a C nyelvből származik a C++ nyelv komplexitása miatt a szimbólumok által hivatkozott adatok szemantikája is sokkal bonyolultabb a C nyelvéhez képest.

Amikor tehát egy alkalmazás egy dinamikus programkönyvtárban található entitást (például globális változót vagy függvényt) szeretne elérni, akkor ezt egy szimbólumnévvel keresztül teszi meg. A bináris kompatibilitás ellenőrzésének egyik pontja, hogy ellenőrizzük, hogy a szimbólumok amelyekre egy alkalmazás korábban hivatkozott, az új verzióban továbbra is elérhetőek legyenek.

A szimbólumokkal való hivatkozás egy problémája, hogy a C++ programozási nyelv szabványa [7] nem mond semmit a szimbólumnevek generálási szabályairól. A szimbólumnevek tehát nincsenek a C++ szabvány által szabványosítva, így a különböző C++ fordító készítőik számára megengedett, hogy ugyanahhoz az entitáshoz más szimbólumnevet generáljanak mint egy másik fordító.

A múltban e probléma kiküszöbölésére létrehoztak egy egyezményes bináris interfész konvenciót[1] amelynek részét képezi a szimbólum névgenerálás szabálya is. A kezdeményezés azóta egyfajta ipari szabvánnyá vált, de még továbbra

is vannak olyan fordítók amelyek más névgenerálási szabályokat használnak¹, ebből adódóan csak akkor értelmes a bináris kompatibilitást ellenőrizni, ha a szóban forgó programkönyvtár mind két vizsgált verzióját ugyanazzal a fordítóval vagy legalább ugyanazzal a bináris interfész konvencióval fordították.

3.2. A bináris kompatibilitás formálisan

Ebben a szakaszban definiálásra kerül, hogy mit is értünk bináris kompatibilitás alatt, valamint ehhez kapcsolódóan a fordítás és szerkesztés folyamata is ismertetésre kerül. A [9] cikkhez hasonlóan én is célszerűnek tartottam egy formális jelölés bevezetését a bináris kompatibilitás definiálására.

3.2.1. Fordítás

Egy szoftver legalább egy, de jellemzően több forrásfájlból áll. A C++ forrásfájloknak nincs megszabva a kiterjesztése, de az egyik legelterjedtebb konvenció szerint a „.cpp”-t szokták használni, ezért a továbbiakban én is „.cpp” kiterjesztéssel fogok hivatkozni a forrásfájlokra. Ezek a forrásfájlok használhatnak fejállományokat a szóban forgó alkalmazásból illetve külső könyvtárakból. Néhány forrásfájlhoz tartoznak olyan releváns fejállományok, amelyek a forrásfájl által definiált fordítási egység interfészét definiálják. A fejállományok kiterjesztésének a „.h” kiterjesztést fogom használni a továbbiakban. Egy forrásfájl a fordításkor előfeldolgozásra kerül, majd a fordító egy tárgykódot generál. Az ilyen a tárgykódú fájlokat a UNIX (szerű) rendszerek konvencióit követve „.o” kiterjesztéssel fogom jelölni.

Formálisan a fordítást jelöljük a következő módon:

$$f.o = \text{compile}(f.cpp, f_1.h, \dots, f_n.h) \quad (n \in \mathbb{N})$$

A compile függvény első paramétere a fordítandó C++ forrásfájl, a többi paramétere pedig a forrásfájl által használt fejállományok, az eredménye pedig a tárgykódú lefordított fájl.

3.2.2. Szerkesztés

A forrásfájlok lefordításával még nem értünk a szoftverkészítési folyamat végére. Ahhoz, hogy használható szoftvert kapjunk ezeket még össze kell szerkeszteni. Attól függően, hogy mit szeretnénk készíteni, több fajta szerkesztési mód közül választhatunk: a tárgykódokat összeszerkeszthetjük egy futtatható bináris (programmá) vagy összeszerkeszthetjük dinamikus programkönyvtárrá.

3.2.2.1. Dinamikus programkönyvtárra szerkesztés

A dinamikus programkönyvtárakat „.so”² kiterjesztéssel fogom jelölni, a UNIX-os konvenció szerint³. Formálisan a dinamikus programkönyvtárra szerkesztést jelöljük a következő módon:

¹A népszerű fordítók (mint például a GCC, az Intel compiler vagy a clang) többsége használja ezt a konvenciót, de például a Microsoft C++ fordítója egy saját megoldást használ.

²Az „.so” az angol „shared object” kifejezésre utal.

³Windows rendszereken a „.dll” kiterjesztést használják.

$$f.so = \text{linkdyn}(f_1.o, \dots, f_n.o, lib_1.so, \dots, lib_m.so) \quad (n, m \in \mathbb{N})$$

A linkdyn függvény paraméterül kapja az előzetesen lefordított tárgykódú fájlokat valamint opcionálisan más, korábban már lefordított dinamikus programkönyvtárakat, eredményként pedig visszaadja a betöltésre kész dinamikus programkönyvtárat.

3.2.2.2. Futtatható programma szerkesztés

Jelöljük a futtatható programokat (alkalmazásokat) „.exe” kiterjesztéssel. Egy futtatható program szerkesztését a következő módon jelöljük:

$$f.exe = \text{link}(f_1.o, \dots, f_n.o, lib_1.so, \dots, lib_m.so) \quad (n, m \in \mathbb{N})$$

A paraméterek a linkdyn függvénnyel analóg módon adhatóak meg, az eredményként pedig a (majdnem) futtatható programot kapjuk. Ha a program dinamikus programkönyvtárakat használ, akkor ahhoz, hogy a programot ténylegesen futtatni lehessen még a könyvtárakat is be kell tölteni (erről bővebben a 3.2.3 szakaszban lesz szó).

3.2.3. Futtatás

A dinamikus programkönyvtárakat használó programoknak futtatás előtt még be kell töltenie azokat, hogy használni tudja a könyvtárakban implementált logikát. Ezt a lépést szokták dinamikus szerkesztésnek vagy dinamikus betöltésnek is hívni. Jelöljük ezt a következő módon:

$$\text{dynload}(prog.exe, lib_1.so, \dots, lib_n.so) \quad (n \in \mathbb{N})$$

A dynload függvény első paramétere a program, a további paraméterei pedig a dinamikus programkönyvtárak amelyeket használni szeretne.

Egy futtatható programot tekinthetünk úgy mint egy függvényt amely az adott bemenethez egy valamilyen jól specifikált kimenetet állít elő. Jelöljük a futtatást a következő módon:

$$prog.exe(input) = output \quad (3.1)$$

$$\text{dynload}(prog.exe, lib.so)(input) = output \quad (3.2)$$

Mind két esetben az input jelöli a bemenetet és az output a kimenetet. Az első esetben a prog.exe nem használ egyetlen dinamikus programkönyvtárat sem, ezért nem szükséges a dynload függvény használata. A második esetben viszont be kell tölteni a lib.so fájlt annak érdekében, hogy a programot futtatni lehessen.

3.2.3.1. Specifikáció

Amikor egy programot futtatunk akkor elvárjuk, hogy „helyesen” működjön. Egy program vagy programkönyvtár helyes működését annak specifikációja írja le. Egy specifikáció egy leképezés amely a korrekt bemenetekről a korrekt kimenetekre képez. Egy programkönyvtár esetén egy korrekt bemenet nem sérti meg a programkönyvtár által definiált függvények előfeltételeit⁴. Egy program vagy programkönyvtár megfelel a spec specifikációnak ha az adott környezettel a korrekt bemenetekre megegyeznek:

$$\text{dynload}(\text{prog.exe}, \text{lib.so}) \sqsubseteq \text{spec}$$

A továbbiakban feltételezem, hogy a programok és a programkönyvtárak helyesen vannak megírva, azaz megfelelnek a specifikációnak. A továbbiakban jelöljük a helyesen megírt programok halmazát $\mathbb{P}$ -vel.

3.2.4. Dinamikus programkönyvtárak

Eddig már sok szó esett a dinamikus programkönyvtárakról, így éppen ideje, hogy definiáljam, hogy formálisan hogyan is néz ki. Legyen egy dinamikus programkönyvtár az alábbi struktúra:

$$\text{Lib} = \langle H = \{h_i\}, S = \{cpp_j\}, \text{lib}, \text{spec} \rangle \quad (3.3)$$

, ahol

- A H halmaz tartalmazza a programkönyvtár releváns fejállományait.⁵
- Az S halmaz tartalmazza a programkönyvtárt alkotó forrásfájlokat, melyek a fejállományokban deklarált entitásokat definiálják.
- $\text{lib} = \text{linkdyn}(\dots, \text{compile}(cpp_j, h_{j_1}, h_{j_2}, \dots), \dots)$, azaz a lib maga a programkönyvtárt binárisa.
- A spec a 3.2.3.1 résznek megfelelő specifikáció.

Mint látható, a definíció megköveteli, hogy a forrásfájlok hiba nélkül leforduljanak a fejállományok deklarációival.

3.2.5. A bináris kompatibilitás definíciója

Most már minden adott, hogy definiálni lehessen a bináris kompatibilitást. Annak érdekében, hogy egyszerűbben lehessen definiálni a bináris kompatibilitást, valamint később is könnyebben hivatkozzunk formálisan a programokra és a programkönyvtárak különböző verzióira, mostantól tegyük fel, hogy az L dinamikus programkönyvtár csak egy forrásfájlból és egy fejállományból áll, valamint a p program csak a $p.cpp$ forrásfájlból áll amely csak az L fejállományát használja⁶.

Tegyük fel, hogy az L könyvtárnak megjelent egy új verziója L' . Az L' könyvtár binárisan kompatibilis az L -vel, amennyiben:

⁴Gyakran a dokumentációk nem az előfeltétel kifejezést használják, hanem csak a dokumentáció szövege tartalmazza, hogy bizonyos értékek használata nem megengedett.

⁵Ezeket tudják az alkalmazások használni.

⁶Kivéve amikor a jelölésből ez másként következik.

$$\begin{aligned} \forall p \in \mathbb{P} \quad : \quad & \text{dynload}(\text{link}(\text{compile}(p.cpp, L.h), L.lib), L.lib) \stackrel{\sqsubseteq}{=} \\ & \stackrel{\sqsubseteq}{=} \text{dynload}(\text{link}(\text{compile}(p.cpp, L.h), L.lib), L'.lib) \end{aligned} \quad (3.4)$$

Ebból látszik, hogy a p program még a korábbi verzió fejállományával lett lefordítva, de ennek ellenére azt várjuk el, hogy a korábban jó bemenetekre az L' könyvtár használatával is jó (azaz a specifikációnak megfelelő) eredményeket kapjunk. Ha a programot újrafordítottuk volna, akkor az alábbi kifejezést kellett volna használni:

$$\text{dynload}(\text{link}(\text{compile}(p.cpp, L'.h), L'.lib), L'.lib)$$

Tehát azt szeretnénk elérni, hogy az új környezettel a viselkedés megmaradjon. Ez egy elég erős megkötés: a korábbi funkcionalitást nem szabad megváltoztatni esetleg csak bővíteni szabad, azaz egy korábban értelmes bemenetre az új verzióknak is ugyanazt az eredményt kell adnia mint a korábbi verzióknak, de új inputokat is elfogadhat (az értelmezési tartományt és az értékkészletet bővíthetjük). Ez azt is jelenti, hogy egy hiba kijavítása is eredményezheti a bináris kompatibilitás elvesztését mivel egy hibajavítás célja pontosan az, hogy a viselkedést megváltoztassa.

3.2.5.1. Forrás kompatibilitás

Forrás kompatibilitás alatt a következőt értjük:

$$\forall p \in \mathbb{P} : (\exists p.o = \text{compile}(p.cpp, L.h)) \Rightarrow (\exists p'.o = \text{compile}(p.cpp, L'.h)) \quad (3.5)$$

Azaz ha a program lefordult a régi verzió fejállományával, akkor az újjal is lefordul. Ezentúl csak mindig feltesszük, hogy a vizsgált programkönyvtárak különböző verziói forrás kompatibilisek.

3.2.6. Kliens kód

A könnyebb tárgyalás érdekében vezessük be a kliens kód fogalmát a következő módon: nevezzük mostantól *kliens kódnak* azt a kódot amelyet a kliens szoftver fordításkor elérhet. Ez azt jelenti, hogy kliens kódnak számít a kliens forráskódján kívül a programkönyvtár fejállománya is. Esetünkben a kliens szoftver a $p = \text{compile}(p.cpp, L.h)$ program, így a kliens kód a $p.cpp$ és az $L.h$ fájlokban található kód.

A továbbiakban a programkönyvtár azon részének elemzésével foglalkozunk amely a kliens számára elérhető. Ez azt jelenti, hogy amikor a bináris kompatibilitásról beszélünk, akkor a programkönyvtár írójának az **inline** függvények és metódusok esetén is be kell tartania a bináris kompatibilitásra vonatkozó szabályokat, mivel azok is kliens kódok.

3.3. A bináris inkompatibilitás okai

A 3.2.5 részben megismerhettük, a bináris kompatibilitás definícióját, most nézzük meg részletesebben, hogy melyek azok az okok amelyek a bináris kompatibilitás elvesztését okozzák.

Ez egyszerűbb vizsgálat érdekében éljünk ugyanazokkal a feltevésekkel mint amelyekkel a 3.2.5 részben is éltünk. Ekkor a p programot programot az alábbi értelmes módokon futtathatunk:

$$\text{dynload}(\text{link}(\text{compile}(p.\text{cpp}, L.h), L.lib), L.lib)) \quad (3.6)$$

$$\text{dynload}(\text{link}(\text{compile}(p.\text{cpp}, L.h), L.lib), L'.lib)) \quad (3.7)$$

$$\text{dynload}(\text{link}(\text{compile}(p.\text{cpp}, L'.h), L'.lib), L'.lib)) \quad (3.8)$$

A (3.6) és a (3.8) esetekben minden fázisban a programkönyvtár azonos verzióját használjuk, így ezekben az esetekben nem kell a bináris kompatibilitás problémájával foglalkozni. A (3.6) valamint a (3.7) eset között viszont csak a dinamikus betöltésben van különbség. A bináris kompatibilitást a (3.7) esetben már vizsgálni kell, mivel a két programkönyvtár által definiált szimbólumok halmaza és ezek jelentése eltérhet.

A $p.\text{cpp}$ forrásfájlban található definíciók alapján a fordító a fordításkor ($\text{compile}(p.\text{cpp}, L.h)$) már el tudja dönteni, hogy melyek azok a szimbólumok amelyre szükség lehet az L dinamikus programkönyvtárból. Mivel a feltételezéseinknek megfelelően az egyetlen információforrás a programkönyvtárról a fordítás során az $L.h$ fejállomány, ezért elmondhatjuk, hogy a fejállomány tartalmazza a programkönyvtár által publikált szimbólumok halmazát, valamint ezek belső struktúráját. Az $L.\text{cpp}$ csupán definiálja a fejállományban publikált szimbólumokat.

Amikor a link függvénnyel egy dinamikus programkönyvtárhoz szerkesztünk egy programot, akkor a programkönyvtárból a szerkesztő nem használ közvetlenül programkódot, csupán megjelöli a felhasznált szimbólumokat dinamikus betöltésre. A (3.7) esetben p program futtatása megköveteli, hogy a dinamikus betöltés során megtalálja az $L.h$ fejállomány által deklarált szimbólumokat, azonban ebben az esetben a dinamikus betöltő az L' programkönyvtárban fogja keresni a szimbólumokat amely viszont az $L'.h$ fejállomáynak megfelelő szimbólumokat definiálja.

Egy L' programkönyvtárban implementált függvény vagy metódus feltételezi, hogy az argumentumok amelyekkel meghívták az $L'.h$ fejállomáynak megfelelőek, viszont a hívó egy szimbólumon keresztül hívja meg az implementációt az $L.h$ fejállomáynak megfelelően megadott argumentumokkal. Ezt azért fontos kiemelni, mert előfordulhat, hogy a szemantika forrás szinten nem változik meg, de bináris szinten mégis megváltozik.

4. fejezet

Tanulmány a bináris inkompatibilitásról

Ebben a fejezetben részletesen áttekintem a 3.3 részben említett okokat amelyek a bináris kompatibilitás elvesztéséhez vezethetnek.

4.1. Implementáció változás

Egy programkönyvtárból számos okból jelenhet meg új változat: új funkciókat, szolgáltatásokat implementál, elavult szolgáltatásokat távolítanak el, hibákat javítanak vagy korábbi algoritmusok implementációit fejlesztenek tovább. Ebben a szakaszban azokkal a változásokkal foglalkozunk, amelyek nem okoznak interfész változást, azonban ennek ellenére okozhatnak problémákat a programkönyvtár felhasználóinak. Mivel az interfész nem módosul, ezért csupán a programkönyvtár fejállományának ellenőrzésével nem lehet ellenőrizni a bináris kompatibilitást. Bizonyos esetekben ha hozzáférünk a programkönyvtár teljes implementációjához, akkor ellenőrizhetünk néhány feltételt (lásd 4.1.1).

Első ránézésre a hibajavítások ártalmatlannak tűnnek, mivel a kód csak kis mértékben változik, ráadásul a programkönyvtár publikus szimbólumneveit egyáltalán nem is változtatják. Amennyiben viszont formálisan tekintjük, akkor egy specifikáció változtatás történt, pontosabban kiderült, hogy a korábbi implementáció nem a korábban dokumentált specifikációt valósította meg, hanem az csak az új változatra lesz érvényes. Amennyiben a javított implementációt a kliensek szimbólumokon keresztül érik el, akkor a kliensek a frissített környezettel az új implementációt fogják meghívni ami a megváltozott viselkedés miatt bináris kompatibilitás veszteséhez vezethet.

Tekintsük példaként azt az esetet amikor egy L programkönyvtárról kiderül, hogy az egyik funkciója hibás (azaz nem a leírt specifikációt valósítja meg). Amennyiben a hiba kliens oldalról javítható a ténylegesen implementált specifikáció tudatában és ezt a p program meg is tesztli (például azért, mert nem tudják megvárni a programkönyvtár hibajavítását) akkor előfordulhat, hogy az új verzióban (L') javított változattal ez a kliens oldali módosítás már nem fog jól működni, így sérült a bináris kompatibilitás.

Némely esetekben szokták a bináris kompatibilitást úgy is definiálni, hogy egy program az új környezettel működhet másként is mint az eredeti környezet-

tel. Ez egy nézetbeli különbség és ha így tekintjük, akkor az előbb említett példa nem helytálló, mindazonáltal ebben a munkában a 3.2.5 részben található definícióval definiáltam a bináris kompatibilitást, mivel szerintem nem lehet elvárni egy kliensről, hogy alkalmazkodni tudjon a későbbi specifikáció változásokhoz.

Egy már meglévő algoritmus módosíthatunk, fejleszthetünk olyan módon, hogy az binárisan kompatibilis maradjon amennyiben a megváltoztatott viselkedést a kliensnek külön igényelni kell. Nézzünk erre egy példát¹: vegyünk egy olyan L könyvtárat amelyben a `bool streq(const char*, const char*)` függvény két karakterlánc egyenlőségét vizsgálja. Az új L' verzióba már lehetőség van arra is, hogy a kis- és nagybetűk közötti különbségek figyelembevétele nélkül is össze lehessen hasonlítani a karakterláncokat az `streq` függvénnyel, de csak abban az esetben, ha az újonnan bevezetett `bool case_insen` globális változó értéke igaz. Az L' -ben viszont alapértelmezés szerint hamisra inicializálják a globális változó értékét, így alapértelmezés szerint az `streq` függvény az új verzióban is a korábbiak szerint működik, a régebbi verzióval fordított p program nem is éri el az új funkciót.

Összefoglalva tehát mondhatjuk a következőt:

Egy függvény vagy metódus implementáció megváltoztatása akkor és csak akkor nem vezet bináris inkompatibilitáshoz ha a megváltozott funkcionalitást nem lehet elérni olyan kliensből amely a könyvtár előző verziójával lett lefordítva.

4.1.1. Változások a kivételkezelésben

A kivételkezelés vizsgálata a bináris kompatibilitás vizsgálatának egy határterületét képezi, mivel a kivételek típusa alapján lehet valamelyest ellenőrizni a programkönyvtár verziók közötti változásokat, de nem minden esetben lehet automatikusan ellenőrizni, hogy sikerült-e megőrizni a bináris kompatibilitást.

A C++ programozási nyelv támogatja a kivételkezelést azonban egy adott függvény vagy metódus által dobható kivételek halmaza nem képezi a függvény vagy metódus specifikációjának részét. Továbbá a C++11 szabvány [7] elavultnak tekinti az eddigi kivétel specifikációs szintaxist és csupán azt adhatjuk meg, hogy egy függvény nem dobhat kivételt (`noexcept` kulcsszó). A korábbi szabványban a függvény paraméterlistája után a `throw` kulcsszó után zárójelek között fel lehetett sorolni a függvény által dobható kivételek listáját, azonban legtöbb fordító nem foglalkozott ezekkel az információkkal, esetleg csak a `throw()` kifejezésnek volt optimalizációt befolyásoló hatása².

A kivételkezelésre is alkalmazhatjuk a 4.1 részben a(z) 13. oldalon található szabályt abban az értelemben, hogy a módosított implementáció nem dobhat újabb típusú kivételt kivéve ha a kliens jelezte, hogy fel van erre készülve, azaz már az új verzióval lett lefordítva. Nevezzük ezt *Erős kivétel szabálynak*³. Ez az elvárás abból adódik, hogy ha a programkönyvtár új verziójában váltódhat ki olyan kivétel amelyet a kliens kód nem képes elkapni (mivel a létezéséről sem tud), akkor a program futása leállhat, de még ha sikerül is elkapni a kivételt a kliens kódban, a viselkedése akkor is megváltozhatott, ami a bináris kompatibilitás definíciója szerint (3.2.5) inkompatibilitásnak tekintendő.

¹A példa a [9] cikkből való.

²Ezzel magyarázható a `noexcept` megjelenése.

³Ez igazából nem más mint a 4.1 részben leírt szabály értelmezés kivételekre.

Esetleg lehetne úgy gyengíteni az előbbi szabályt, hogy megengedjük, hogy az új implementáció dobhasson olyan típusú kivételt amely altípusa valamely korábban már dobott kivételnek. Nevezzük ezt a szabályt *Gyengített kivétel szabálynak*. Ebben az esetben a kliens kód le tudná kezelni a kivételt mivel a szülője alapján el tudja azt kapni. Ezt csak abban az esetben tehetjük meg, ha garantálni tudjuk (a programkönyvtár tervezése és felépítése által), hogy ha az újonnan dobott kivétel típust valamely szülője szerint kezelik le, akkor az nem okoz viselkedés változást a kliens kódban. Ekkor nem sértjük meg a bináris kompatibilitást (3.2.5).

Bármely szabályt is alkalmazzuk, ezeket a szabályokat csak részben tudjuk statikusan ellenőrizni. A [8] cikkben találunk egy módszert amely segítségével elvégezhető egyfajta kivétel analízis, de teljes módszer ebben az anyagban sem található, mivel a virtuális metódusok valamint a függvényekre illetve metódusokra mutató mutatók miatt szükség lenne mutató analízisre is, de ezt nem minden esetben lehet statikusan elvégezni.

Mindenesetre az látható, hogy némely esetben jelezhető statikus ellenőrzéssel ha az új programkönyvtár verzió megszegi a kivételkezelésre vonatkozó szabályokat, de ehhez már nem elegendő a programkönyvtár interfészét leíró fejláblományok vizsgálata, hanem a programkönyvtár teljes forráskódját is vizsgálni kell.

4.2. Interfész változás

Ebben a szakaszban áttekintem azokat az interfész változásokat amelyek bináris inkompatibilitást idézhetnek elő. Mivel a programkönyvtár új és régi változatának interfészeit azok fejláblományaik írják le, azért elegendő ezeket megvizsgálni, hogy eldöntsük a bináris kompatibilitást vagy inkompatibilitást.

4.2.1. Adatszerkezet változás

Ebben a részben azokat a fejláblomány változásokat tekintem át amelyek hatására az adatok belső struktúrája inkompatibilis módon változik meg. A probléma fő oka, hogy az *L'.cpp*-ben található implementáció az *L'.h* fejláblomány jelöléseinek megfelelően éri el az adatokat, míg a kliens az *L.h* fejláblománynak megfelelő adatszerkezetekkel dolgozik.

Az ebben a részben leírt változtatások csak az adatok memóriában való elrendezésének szemantikáját változtatják meg, így ezeket a változásokat a dinamikus betöltő nem képes jelezni.

4.2.1.1. A programkönyvtár memóriájának használata

Elsőként tekintsük át, hogy a programkönyvtár által definiált változókra milyen megkötéseket kell tennünk amennyiben tartani szeretnénk a bináris kompatibilitást. Változó alatt ebben az esetben érthetünk globális változókat valamint egy függvény hívás esetén a paraméter- illetve a visszatérési értékátadásra használt memóriaterületet is.

Az az eset még nem okoz önmagában inkompatibilitást ha a kliens kód csak közvetetten, interfész hívásokkal fér hozzá a programkönyvtár memóriájához, akár egy osztály statikus metódus hívásán, akár globális változó metódus hívásán

keresztül. Amennyiben viszont a kliens közvetlenül olvashat vagy írhat egy változót akkor már ügyelni kell arra, hogy a változó struktúrájának kliens által látható részének szemantikája ne változzon meg.

Használjuk a következő jelöléseket:

$$\begin{aligned} V &\doteq \{(T, T'), \mid T \text{ az } L \text{ egy változójának típusa, } T' \text{ pedig ennek } L' \text{ belüli megfelelője}\} \\ T|_x &\doteq \text{A } T \text{ struktúra első } x \text{ byte-ja.} \end{aligned}$$

Egy binárisan kompatibilis új verzióban a következő egyenletnek teljesülnie kell:

$$\forall v \in V : v.T \equiv v.T'_{\text{sizeof}(v.T)} \quad (4.1)$$

Azaz csak akkor akkor férhetünk hozzá egy változóhoz binárisan kompatibilis módon, amennyiben a változó kliens kód által látható része - azaz az első sizeof(T)⁴ byte-ja - nem változott meg. Ez azt is jelenti, hogy újabb mezőket, adattagokat vehetünk fel a változókhoz az új verzióban, de arra ügyelni kell, hogy a korábbiak pozíciója (és szemantikája) ne változzon.

4.2.1.2. Kliens által foglalt memória

A memóriakezelés másik iránya amikor a programkönyvtár újabb változatából kell elérni a kliens kód által foglalt memóriastruktúrákat. Az egyszerűbb tárgyalás érdekében tegyük fel, hogy minden függvénynek egyetlen paramétere van amely T típusú vagy T típusú referencia típus (mutató, referencia, jobb oldali referencia) az L programkönyvtárban. Az egyszerűség kedvéért tekintsük a metódusokat is egyszerű függvényeknek amelyek első paraméterükként a **this** mutatót kapják. Jelöljük T' -vel azt a típust amely a T -ből lett az L' programkönyvtárban.

A C++ programozási nyelv szabványa [7] nem definiálja a függvények hívásának konvencióját, ezért tegyük fel, hogy a hívónak kell lefoglalnia az argumentumok, valamint a visszatérési értéknek a memóriaterületet, valamint a hívó is szabadítja ezeket fel.

Ha a függvény paraméterét érték szerint adjuk át akkor ezt egy binárisan kompatibilis alkalmazásban csak akkor tehetjük meg ha $T = T'$, mivel a híváskor a hívónak kell lefoglalnia a paraméter számára a helyet, viszont az implementáció az L' programkönyvtárban van, így a méretüknek és szemantikájuknak meg kell egyezniük (4.1). Ugyanez a megszorítás él a visszatérési értékre is hasonló megfontolások miatt. Látható tehát, hogy *csak abban az esetben módosíthatjuk egy típus szemantikáját, amennyiben az csak referencia típusokon keresztül kerül átadásra a programkönyvtár függvényeinek, illetve a visszatérési értéként is csak referencia szerint lesz átadva a programkönyvtár függvényeiből.*

Az előzőeken kívül további problémát jelent amennyiben egy L szerinti T (osztály vagy struktúra) típusú változót példányosítani lehet a kliens kód automatikus tárterületén⁵: mivel a kliens az $L.h$ szerinti memóriaterületet foglalja le, de az műveletek implementációi az $L'.cpp$ -ben találhatóak, így azok az $L'.h$ szerinti méretű és szerkezetű adatokkal dolgoznak, tehát akár már a létrehozáskor meghívott konstruktor is okozhat memória hozzáférési hibákat, illetve a később

⁴Itt sizeof a C++ programozási nyelvből ismert operátor.

⁵Például a program vermében, egy osztály tagjaként vagy akár mint ösztály.

használt műveletek bármelyike. Az előző bekezdés alapján ezt a fajta hozzáférést nem tekintenénk binárisan inkompatibilisnek, mivel egy osztály műveletei referencia szerint (a **this** mutatón keresztül) kapják meg az osztálypéldányt, ezért erre külön kikötést kell tennünk:

Egy olyan struktúrán (osztályon) amely példányosítható a kliens kód automatikus tárterületén bármely olyan változtatás amely megváltoztatja a struktúra (osztály) méretét vagy bármely elérhető adattagjának pozícióját, binárisan inkompatibilis módosításnak tekintendő.

Néhány esetben lehetőségünk van új adattagokat adni egy osztályhoz az osztályméret valamint a többi adattag befolyásolása nélkül bitmezők segítségével.

Egy osztály szerkezete nem csak az adattagok módosításával változhat, hanem az osztályhierarchia megváltoztatása is megváltoztathatja az adattagok pozícióját, mivel ha egy nem üres ősosztály sorrendje megváltozik, akkor az adattagok pozíciói is megváltoznak. Egy új üres ősosztály esetleg hozzáadható az osztályhoz, viszont nem garantálható, hogy egy látszólag üres ősosztály nem okoz méretnövekedést a leszármazottban, ezért ez is kerülendő. Egy nem üres ős hozzáadása viszont egyértelműen növeli egy osztály méretét, ezért ez egyértelműen bináris inkompatibilitáshoz vezet. További problémát jelenthet a polimorfizmus kezelése (erről bővebben a 4.2.3 részben olvashat).

Mivel a szabály a módosítások tiltását csak a kliens kódban példányosítható (illetve abban leszármaztatható) osztályokra írja elő, ezért léteznek technikák (például a 7.2 leírt megoldás) amellyel binárisan kompatibilis módon lehet egy osztályt bővíteni, de ezeknek a technikáknak mindig megvannak a maguk limitációi.

4.2.1.3. Adattag jogok

A C++ programozási nyelvben az objektum adattagokhoz jól meghatározott láthatósági szabályok alapján férhetünk hozzá. Egy adattag lehet public, protected vagy private láthatóságú. Bináris szinten viszont a láthatósági jogok eltűnnek, mivel itt már csak nyers memóriaterületekről beszélünk, az adat szemantikája már nincs jelen. Ebből kifolyólag egy tag láthatóságának megváltoztatása bináris szinten nem jelentkezik.

Mikor egy programkönyvtár új verziójától megköveteljük a bináris kompatibilitás teljesítését, akkor általában megköveteljük annak forráskompatibilitását is, hogy a programkönyvtárat használó alkalmazásokat le lehessen majd fordítani az új verzióval is. Ha ezt is figyelembe vesszük, akkor az adattagok hozzáférési jogait csak enyhíthetjük, mivel máskülönben előfordulhatna, hogy egy korábban megengedett hozzáférés az új verzióban nem fordulhatna le. Ebből adódik a következő állítás:

Egy objektum adattag láthatóságának gyengítése binárisan kompatibilis módosítás.

Mondhatnánk, hogy a láthatóság szigorítása is binárisan kompatibilis módosítás, a fentebb leírtak miatt. Valóban, ha eltekintünk a forráskompatibilitástól (ami nem követelménye a bináris kompatibilitásnak), akkor tekinthetnénk ezt binárisan kompatibilis módosításnak. Másfelől megvizsgálva, viszont általában a szigorításnak az az oka, hogy más objektumok ne „ronthassák el” az objektum belső állapotát. Az objektum műveletei feltételezhetnek olyan állításokat az

adattagról, amelyek az objektumon értelmezett műveletekre nézve invariánsak, de amennyiben az adattagot a régebbi kliens kód módosíthatja, úgy az objektum belső működésében zavar keletkezhet.

Látható tehát, hogy csak akkor szigoríthatjuk egy objektum adattag láthatóságát, ha feladjuk a forráskompatibilitást, valamint meggyőződünk arról, hogy az esetleges külső módosítások az adattagon nem zavarják meg az objektum helyes működését.

4.2.2. Szimbólumváltozások

Az interfészváltozások egy nagy csoportja a szimbólumváltozásokat előidéző változások. Egy alkalmazás amikor egy dinamikus programkönyvtárral szerkesztik össze, akkor számos a függvénykönyvtárban megtalálható szimbólumtól függhet, ezeket nevezzük külső függőségeknek. A dinamikus betöltő ezeket a külső függőségeket keresi a betöltött programkönyvtárakban. Egy ilyen külső függőség lehet globális változó szimbóluma, nem inline függvény vagy egy teljesen példányosított sablon függvény de ezeken kívül még számos egyéb szimbólum függőség is elképzelhető.

Az ebben a részben leírtak főként az Itanium C++ ABI [1] valamint a Microsoft névgenerálási szabályain alapul (bár utóbbiról nincs a gyártó által hivatalosan kiadott dokumentum, de Anger Fog munkája nyomán [5] mégis megismerhetjük ennek módját).

Mielőtt részletesen tárgyalnám, hogy mely kódváltoztatások okozhatnak bináris szinten is változást, álljon itt az egyik legfőbb szabály:

Egy olyan szimbólumot, amely a kliens kódból elérhető sosem szabad eltávolítani az új verzióból.

Ennek a szabálynak az a nyilvánvaló oka, hogy egy szimbólum eltűnése dinamikus betöltési hibákat okozhat a kliens oldalon.

A névgenerálástól

A szimbólumnevek megváltozása a névgenerálástól is erősen függ. Mielőtt részletesebben áttekinteném a potenciális binárisan inkompatibilitásokat, definiálok pár általános fogalmat amely segíti a téma tárgyalását.

Egy deklaráció teljes minősített neve a deklaráció neve valamint a befoglaló hatóköre (osztály vagy névtér) teljes minősített neve⁶. A deklarációkhoz tartozó szimbólumok nevéről elmondható, hogy függnek az adott deklaráció teljes minősített nevéől, így *egy deklaráció nevének módosítása a hozzá tartozó szimbólum megváltozását vonja maga után (egy régi szimbólum eltűnik, egy új megjelenik), így inkompatibilis módosításnak tekintendő.*

Amikor egy szimbólum neve függ egy típustól, akkor a típus alatt a kifejtett típust értjük, azaz összes **typedef** helyettesítve lesz a jelentésével, valamint az összetett típusok esetén a teljes minősített nevet használjuk. Ez természetesen azt jelenti, hogy ha egy típusként használt **typedef** jelentése megváltozik egy inkompatibilis típusra akkor az összes olyan szimbólum, amely nevének generálásához felhasználtuk az adott típus szinonimát, szintén megváltozik. Ezek a hibák igen nehezen észrevehetőek egy programozó számára, mivel a típusok nevein nem látszik a változás.

⁶Tekintsük ezt egy rekurzív definíciónak.

4.2.2.1. Globális változók

A globális változók (ideértve az osztályok statikus adattagjait is) szimbólumokon keresztül érhetőek el a kliens számára. Egy *globális változó szimbólumának neve a következő információktól függhet* :

1. A változó típusa.
2. A változó minősített neve.
3. A változó CV-minősítői⁷.
4. Statikus adattag esetén a változó látatósága.

A felsoroltak közül bármelyik megváltoztatása potenciálisan binárisan inkompatibilis módosítás. Természetesen a globális adatokra is be kell tartani a 4.2.1.1 részben leírtakat.

4.2.2.2. Függvények

A függvények és metódusok neveinek generálására többnyire ugyanazok a szabályok alkalmazandóak, ezért tekintsük a metódusokat egyszerű függvényeknek. A függvények szimbólumnevei sokszor igen bonyolult szabályok alapján generálhatóak aminek az egyik oka, hogy a C++ programozási nyelv támogatja a függvény túlterhelést. Annak érdekében, hogy a szabványban meghatározott túlterhelést támogatni lehessen a visszatérési típuson kívül szinte mindent be kell építeni a szimbólumnévbe ami a függvény szignatúrájában megtalálható. Ezek alapján *egy függvény szimbólum neve a következőktől függhet*:

1. A függvény visszatérési típusa.
2. Metódusok esetén a metódus láthatósága.
3. A függvény minősített neve.
4. A függvény argumentum típusainak vektora.
5. Sablon függvény példány esetén a függvény sablon paramétereinek vektora.
6. A függvény CV-minősítői.

Bár kódgenerálási szabálytól függően eltérhet, hogy a felsoroltak közül valójában melyektől függ pontosan a szimbólumnév, de ha fordítótól függetlenül szeretnénk vizsgálni a bináris kompatibilitást, akkor elmondható, hogy bármely ponton történő módosítás bináris inkompatibilitást vonhat maga után.

Egyes névgenerálási konvenciók (például az amelyet a Microsoft használ [5]) a visszatérési típust is hozzáadják a szimbólumnévhez. A visszatérési típus hozzáadása a sablon függvények esetén kiváltképp indokolt lehet, mivel egy sablon függvény visszatérési típusa függhet a sablon paraméterektől.

Metódusok esetén érdekes kérdés lehet, hogy egy metódus láthatóságát a metódus szignatúrájának tekintjük-e. Szigorúan technikai értelemben a szimbólumnévbe nem szükséges belevenni egy metódus láthatóságát, mivel egy osztályon belül nem lehet két azonos nevű és szignatúrájú, de különböző láthatóságú

⁷Angolul CV-qualifier: a const, volatile és mutable minősítők.

metódus. A Microsoft fordítója ennek ellenére ezt az információt is belekódolja a metódusnevekbe, mivel Microsoft névgenerálása azt a filozófiát követi, hogy a lehető legtöbb információt kódolja bele a szimbólumnévbe, hogy csupán a szimbólumnév alapján később pontosabb analízist, esetleg hibaüzeneteket lehessen kiírni.

A C++ programozási nyelv túlterhelési szabályainak megfelelően az argumentumok típusának azonosnak kell maradnia a két verzió között, mivel az argumentumok számának megváltoztatásával, illetve egy korábbi argumentum típusának megváltoztatásával tulajdonképpen egy új függvény jön létre és ennek megfelelően a régi függvény törlődik a kliens szempontjából. Az argumentumok típusának szigorúan nem szabad változnia, azaz még a CV-minősítők sem változhatnak, mivel egyes fordítók a típusok CV-minősítőit is belevehetik a szimbólumnévbe.

Egy sablon függvény példány kódja függhet a sablon paramétereiktől, illetve a sablon specializációs szabályok miatt a függvény sablon paramétereinek vektorát is bele kell kódolni a példány szimbólumnevébe. Ebből adódik egy újabb megkötés, amely a 4.2.4. részben kerül tárgyalásra.

Egy metódust túl lehet terhelni a CV-minősítők alapján, így annak mindenképp szerepelnie kell a metódushoz tartozó szimbólumnévben.

Inline függvények. Ha egy függvény **inline** függvényként definiálunk, akkor a fordítónak nem szükséges szimbólumot generálnia, mivel az implementáció törzse a felhasználás helyén rendelkezésre áll. Ebből következik a következő szabály:

*Egy korábban nem **inline** függvényhez nem szabad **inline** minősítőt adni, amennyiben az elérhető a kliens kódból.*

A kliens szempontjából az újonnan **inline** függvényként definiált függvény szimbóluma elveszhet, így a dinamikus betöltésnél hiba léphet fel.

4.2.3. A polimorfizmus bináris kompatibilitási vonzatai

A C++ programozási nyelv támogatja a polimorfizmust: az öröklődésnél lehetőségünk van virtuális öröklésre, létrehozhatunk dinamikus kötésű műveleteket virtuális metódusok segítségével, valamint futásidőben információkat kérhetünk le egy objektum dinamikus típusáról. A C++ nyelv szabványa ([7]) csak a dinamikus működést írja le, itt sem ad implementációbeli megkötéseket, tehát ebben az esetben is teljesen a C++ fordítók íróira bízta ennek megvalósítását.

A virtuális metódusok egyrészt tekinthetők hagyományos függvénynek, így vonatkozik minden megkötés ami a 4.2.2.2. részben volt olvasható, másrészt a dinamikus működés miatt további megkötések lesznek rájuk érvényesek. Amikor dinamikusán hívunk meg egy virtuális metódusok, akkor egy úgynevezett virtuális hívási mechanizmus (röviden virtuális hívás) megy végbe, amely során kiválasztásra kerül a konkrétan meghívandó függvény címe, majd megtörténik a függvényhívás.

Ezt a mechanizmust általában virtuális táblákon keresztül szokták implementálni. A virtuális táblák tartalma fordítófüggő (mint ahogy a virtuális táblák maguk is), de a módszer lényege a következő: minden polimorfikus osztályhoz tartozik egy táblázat, amely egy leképezés az implementálható virtuális metódusok, valamint azok valódi implementációi között, azaz függvény mutatókat

tartalmaz. Amikor egy osztályban felüldefiniálunk egy virtuális metódust, akkor az osztály virtuális táblájában a felüldefiniált virtuális metódushoz a saját implementációjára mutató mutató kerül. Az osztály objektumokban létrejön egy mutató amely az osztály virtuális táblájára mutat. A virtuális híváskor a virtuális táblán (azaz egy mutatón) keresztül kerül meghívásra a megfelelő függvény. A függvénymutató helyét a táblában a fordító már fordítási időben is képes lehet eldönteni.

Egyes fordítók a virtuális bázisosztályok implementációjához is külön táblázatot használnak, mások ugyan azt a táblázatot használják a virtuális metódusokhoz valamint a virtuális bázisokhoz is, de az is elképzelhető, hogy valamilyen teljesen más módon implementálják a dinamikus működést.

4.2.3.1. Virtuális metódusok

Az előzőekből látható volt, hogy nem tételezhetünk fel semmi konkrétat a virtuális működések implementációjáról, ezért induljunk ki az alábbi erős szabályból:

Egy új virtuális metódus hozzáadása egy osztályhoz bináris inkompatibilitáshoz vezethet.

A amennyiben feltesszük, hogy virtuális táblával valósították meg a virtuális hívásokat, úgy egy új virtuális metódus növelheti a táblázat méretét. A lezármazottak virtuális táblázata épülhet az ősz virtuális táblázatára, így annak mérete is növekszik. A virtuális táblázatok globális adatoknak is tekinthetők, így a 4.2.1.1. részben leírtak miatt ez inkompatibilis módosítás. Néhány speciális esetben virtuális táblákhoz is hozzá lehet adni új virtuális metódust, de ezek erősen függnének az implementációtól.

Egy másik fontos kérdés a virtuális táblák esetén, hogy a metódusok milyen sorrendbe kerüljenek a táblázatban. Általában az a szabály, hogy a virtuális metódusok a deklarációjuk sorrendjében kerülnek a táblázatba. Ez azt jelenti, hogy nem szabad megváltoztatni a virtuális metódusok sorrendjét (szintén a 4.2.1.1 részben leírtak miatt):

A virtuális metódusok sorrendjének megváltoztatása bináris inkompatibilitáshoz vezethet.

4.2.3.2. Virtuális metódusok felüldefiniálása⁸

Egy korábban nem felüldefiniált (ősből örökölt) virtuális metódus felüldefiniálása látszólag veszélytelennek tűnik, hiszen azt gondolhatnánk, hogy csupán a virtuális táblában egy bejegyzés frissítésre kerül frissítésre. Valójában az implementációtól függően igen változatos módokon okozhat bináris inkompatibilitást egy felüldefiniálás. A könnyebb tárgyalás érdekében tekintsük úgy, hogy az Itanium ABI [1] által definiált virtuális táblát használja a fordító.

Devirtualizálás. Mielőtt a felüldefiniálás okozta inkompatibilitásokat vizsgálnánk tekintsük át, hogy a fordítóprogram mikor kerülheti ki a virtuális metódushívást. A devirtualizáció egy egyszerű optimalizálás amelyet a fordítók általában végrehajtanak annak érdekében, hogy megspórolják egy adott virtuális metódus hívásánál a virtuális táblán keresztüli hívást.

⁸Angolul: override.

Az optimalizálás lényege, hogy mennyiben a fordító el tudja dönteni egy objektum dinamikus típusát fordítási időben, akkor egyszerű függvényhívással hívja meg az adott virtuális metódus megfelelő példányát (a legutolsó felüldefiniálást). Ez például akkor lehetséges ha az objektum amelynek a virtuális metódusát szeretnénk meghívni egy függvény lokális változója, mivel ekkor a fordító pontosan tudja, hogy az objektum statikus és dinamikus típusa megegyezik, így nem szükséges a virtuális hívás. Ez azt jelenti, hogy amennyiben a kliens kódban még nem volt a szóban forgó objektum osztályában fölüldefiniálva az adott virtuális metódus (amely az új verzióban már felül lett definiálva), akkor egészen a kliens újrafordításáig nem az új felüldefiniálás fog meghívódni. A devirtualizálás akkor is elvégezhető, ha egy olyan referencia típuson keresztül történik a metódushívás amely osztálya **final** módosítóval lett deklarálva vagy maga a virtuális metódus **final**.

A fentiek alapján tehát úgy kell megtervezni minden új felüldefiniálást, hogy akkor is jól működjön egy kliens, ha a korábbi implementációt hívja meg.

Egyszerű felüldefiniálás. Vegyük az alábbi osztályhierarchiát: adott egy A osztály, valamint egy ebből leszármazó osztály B . Az B osztály elsődleges bázisosztálya az A és az A egy virtuális metódusát (vagy annak egy ősének metódusát) definiáljuk felül a B osztályban. Ebben az esetben sem a virtuális tábla okozhatja a problémát, hanem a viselkedés: a kliens számára módosulhat a viselkedés mivel a virtuális híváskor az új metódus kerülhet meghívásra, ami a 4.1 szakaszban tárgyaltak alapján jelenthet problémát. Amennyiben a 4.1 szakaszban tárgyaltak szerint nem inkompatibilis az új működés, úgy megengedhető a felüldefiniálás (kivéve a kovariáns esetet, amelyet később tárgyalunk).

Nem elsődleges felüldefiniálás. Az előbbi esetben egy elsődleges bázisosztály egy virtuális metódusát definiáltuk felül. Láthattuk, hogy ez ilyen felüldefiniálások nem okoznak közvetlenül bináris inkompatibilitást. Most nézzünk egy másik esetet: adott egy C osztály, valamint egy ebből leszármazó osztály D . Az D osztály elsődleges bázisosztálya *nem* a C osztály. A D osztályban a C osztály egy virtuális metódusát (vagy annak egy ősének metódusát) definiáljuk felül.

A hierarchiából látszik, hogy hasonló esettel van dolgunk mint az előző példában, csak itt nem az elsődleges bázisosztály virtuális metódusát akarjuk felüldefiniálni. Bináris szempontból a két eset teljesen másképpen kezelendő, a különbség a metódusok természetéből adódik: a metódusokat tekinthetjük olyan függvényeknek amelyek első paramétere a **this** mutató. Amennyiben egy osztály több bázisosztályból örököl, akkor az tekinthető úgy mint ha a felsorolás sorrendjében konkatenálnánk az osztályokat. Ez azt jelenti, hogy ha egy osztály nem elsődleges (elsőként felsorolt) bázisosztálya egy leszármazottnak, akkor egy leszármazotton keresztüli metódushívásnál a **this** mutatót el kell tolni, hogy a leszármazott objektumnak a megfelelő részére mutasson. Ezeket az eltolásokat az egyszerű metódushívásoknál a fordítóprogram a statikus típusok alapján ki tudja számolni.

Tekintsük most azt az esetet amikor egy C referencia típusú változón keresztül szeretnénk meghívni egy virtuális metódust amelyet a D osztályban felüldefiniáltunk. Ebben az esetben egy virtuális hívás történik, csak hogy a **this** mutatót ekkor is el kell tolni valamennyivel, mivel a C nem elsődleges bázis-

osztálya a D osztálynak, így a **this** mutató nem az objektum „elejére” mutat. Ekkor azonban a fordító már nem feltétlenül tudja eldönteni, hogy a változónak mi a dinamikus típusa, így az eltolást sem tudja automatikusan elvégezni.

A probléma megoldására a következő történhet: amikor a D osztályban felüldefiniálunk egy nem elsődleges bázisbeli metódust, akkor két metódus jön létre. Az egyik metódus lesz az amelynek a törzse az implementációt tartalmazza, ez a D osztály egy új virtuális metódusa lesz, így bekerül egy új bejegyzés a virtuális táblába. A másik pedig a felüldefiniált változat lesz amely a **this** konverzióját végzi, majd (virtuálisan) továbbhív az új metódusba, ez utóbbi metódus kerül bejegyzésre a C -beli metódus felüldefiniálásaként. A D osztály leszármazottjai már a D -beli új virtuális metódust definiálhatják felül. Ezek alapján a következő mondható:

Egy nem elsődleges osztálybeli virtuális metódus felüldefiniálása binárisan inkompatibilis módosítás.

Kovariáns felüldefiniálás. A C++ programozási nyelv megengedi, hogy egy metódust kovariáns módon definiáljanak felül: ha adott egy X osztály, valamint egy ebből leszármazó osztály X' , akkor egy X referenciával, vagy egy X -re mutató mutatóval visszatérő virtuális metódus felüldefiniálható egy olyannal amely X' -re hivatkozó referencia típussal tér vissza. Abban az esetben viszont ha X' osztálynak nem az X az első őse, akkor a felüldefiniált változatban még szükség van egy típuskonverzióra (egy mutató eltolásra) annak érdekében, hogy a hívó X -ként tudja kezelni.

A Itanium ABI-t használó fordítók esetén ilyenkor valójában két metódus keletkezik. Az egyik metódus egy új virtuális metódus lesz amely X' -vel tér vissza, ez egy új bejegyzésként kerül a virtuális táblába. A másik pedig egy fordító által generált felüldefiniálás lesz, amely csupán a konverziót tartalmazza, ez kerül a virtuális táblába mint az eredeti metódus felüldefiniálása. Az eset hasonló mint a nem elsődleges bázisosztálybeli felüldefiniálás esetén, azaz egy új virtuális metódust adtunk az osztályhoz ami binárisan inkompatibilis:

Azok a kovariáns felüldefiniálások amelyek esetén a leszármazott típusnak nem a felüldefiniált visszatérési típus az elsődleges bázisa binárisan inkompatibilis módosításnak számít.

4.2.4. A sablonok szerepe a bináris kompatibilitásban

A C++ programozási nyelven a generikus programozást sablonokkal valósíthatjuk meg. A sablonokkal típusfüggetlenül írhatunk le algoritmusokat valamint adatszerkezeteket. A sablonok egyik érdekessége, hogy **inline** kerülnek implementálásra, azaz a programkönyvtár által publikált sablonok a programkönyvtár fejállományaiiban találhatók. Egy sablon függvény teljesen specializált változata hagyományos függvénynek tekintendő, így ezeket elég csak deklarálni, az implementáció kerülhet a programkönyvtár forrásfájljaiba.

Egy sablon osztály önmagában még nem tekinthető egy kész osztálynak, csak egy sablonnak amely alapján a példányosításakor egy osztály készíthető, mindazonáltal meghatározza a készítendő osztály számos tulajdonságát. Egy sablon osztály példány egy hagyományos osztálynak tekinthető, azzal a különbséggel, hogy a nevében szerepel a sablonparamétereinek aktuális értéke is. Ez

azt jelenti, hogy ha egy függvény egy sablon osztály példány típusú paraméterrel rendelkezik akkor a függvény nevében szerepelni fog az osztály teljes minősített neve mellett az összes sablon paramétere is. Mivel a sablon osztály példányok hagyományos osztályoknak tekinthetők, így vonatkozik rájuk minden korlátozás ami a 4.2.1 részben szerepelt, azaz az adattagjait csak binárisan kompatibilis módon lehet megváltoztatni⁹.

Az előbb leírtak a sablon függvény példányokra, valamint a specializációikra is igazak, tehát a 4.2.2.2 részben tárgyaltak ezekre is alkalmazhatóak.

További kérdéseket vet fel, hogy mivel a sablonok esetén az implementáció **inline**, így az új verzió megjelenésekor a kliens kódban csak újrafordításkor jelennek meg a változások. Ez egyrészt garantálja, hogy a kliens kódban példányosult sablon függvények és metódusok viselkedése nem változott meg ami némely esetben kifejezetten előnyös, másrészt ugyanezért veszélyes is lehet. Egy sablon módosításánál tehát figyelembe kell vennünk, hogy a kliens oldalon nem minden esetben a módosított viselkedés fog megjelenni.

Az eddigiek alapján tehát a sablon paramétereknek lehet közvetlen bináris kompatibilitásra gyakorolt hatásuk. Tekintsük például azt az esetet amikor egy sablon osztályhoz még egy sablon paramétert adunk, de annak érdekében, hogy a forráskompatibilitás megmaradjon a sablon paraméternek egy alapértelmezett értéket adunk. Ekkor a programkönyvtárban az összes olyan függvénynek amelynek az argumentumlistájában szerepel ez az osztály, továbbra is le fog fordulni, de a hozzá tartozó szimbólum módosulni fog, ami mint korábban láttuk bináris inkompatibilitást okoz. Hasonló probléma adódik ha egy sablonparaméter alapértelmezett értékét megváltoztatjuk. Az alábbi szabályt tehát be kell tartanunk:

Egy sablon (osztály vagy függvény) paramétereit valamint a sablonparaméterek alapértelmezett értékeit nem szabad megváltoztatni abban az esetben, ha a sablon egy teljesen példányosított változata szerepel a programkönyvtár fejállományaiban.

⁹A GCC 4.7.1 verziójában az `std::list` szabványos konténer osztály tartalmazza az elemek számát is (ellenben a korábbi változatokkal), amennyiben a C++11 standard szerint fordítunk. Ez azt jelenti, hogy egy nem C++11 standard szerint fordított programkönyvtár használatakor problémák adódhatnak ha a kliens a C++11 szabványt használja.

5. fejezet

A bináris kompatibilitás ellenőrzése

A 4. fejezetben láthattuk, hogy melyek azok a körülmények amelyek hatására binárisan inkompatibilissé válhat egy dinamikus programkönyvtár két verziója. A 4.1. részben láthattuk, hogy vannak olyan változások melyeket nem lehet a dinamikus programkönyvtár fejállományai alapján ellenőrizni, mivel az inkompatibilitást a módosított programlogika idézi elő. A 4.1.1. részben említettem, hogy némely esetben azonban ilyenkor is lehetőségünk van néhány lehetséges inkompatibilitás jelzésére, bár ehhez nem elegendő csupán a dinamikus programkönyvtár fejállományait vizsgálni, annak teljes forráskódjára is szükség van. A 4.2. részben olyan módosításokról volt szó amelyek a fejállományokból teljes egészében kikövetkeztethetők.

A bináris kompatibilitást egy új programkönyvtár kiadásakor ellenőrizhetik a programozók maguk is tesztekkel illetve a kód elemzésével. Ez természetesen nem triviális feladat, mivel igen sok tényezőre kell figyelnie a programozóknak és a tesztelőknak, így ez sok időt és pénzt emészt fel. A 4.2. részben szereplő inkompatibilitások ellenőrizhetőek programmal, így ezzel a teszteléskor segíthetjük a programozók munkáját (az implementáció változásokat továbbra is a fejlesztőknek kell ellenőrizniük).

Két fő irányt vizsgáltam meg a bináris kompatibilitás programmal való ellenőrzésére: a dinamikus programkönyvtár lefordított kódjának vizsgálatát, valamint a kompatibilitás ellenőrzését statikus kódelemzéssel.

5.1. A bináris kód elemzése

Egy dinamikus programkönyvtár binárisának elemzésével kijelezhető néhány inkompatibilitás. Alapvetően azokat az inkompatibilitásokat tudjuk kiszűrni amelyek közvetlen betöltési hibákhoz vezethetnek, mivel ugyanazok az információk állnak rendelkezésre számunkra mint amelyet a dinamikus programkönyvtár betöltéséhez használt betöltő is felhasználhat. Egy dinamikus programkönyvtárat a betöltő szempontjából a szimbólumai határozzák meg, amikor egy alkalmazás egy szimbólumra hivatkozik, a betöltő feladata, hogy a megfelelő binárist betöltse és megkeresse a szóban forgó szimbólumot. Ha a szimbólumot nem sikerül megtalálni, akkor betöltési hiba keletkezik. A fő célunk a bináris kód

ellenőrzésével, hogy a betöltési hibákat elkerüljük, de mint látható lesz néhány speciális esetben ennél is többet ellenőrizhetünk.

Szimbólumok összehasonlítása

Egy új könyvtárverzió ellenőrlésekor ellenőrizhetjük a benne megtalálható szimbólumok, valamint a régebbi verzióban megtalálható szimbólumok közötti különbségeket. *Elmondható, hogy a korábbi verzió szimbólumainak halmaza rész-halmaza kell hogy legyen az új verzió szimbólumainak halmazának, mivel bármely szimbólum eltávolítása potenciálisan inkompatibilis módosítás.* Mondhatnánk, hogy egy privát láthatóságú metódust esetleg el lehetne távolítani, de a bináris elemzésekor nem győződhetünk meg arról, hogy a programkönyvtár fejálmányban nem hivatkozik-e az adott metódusra egy kliens kódból elérhető **inline** függvény vagy metódus, így tehát ezeket a szimbólumokat sem távolíthatjuk el.

A hibák jelzését segíti, hogy a szimbólumokból a névgenerálási szabályokból adódóan általában visszaállíthatóak a globális változók nevei valamint a függvények illetve metódusok neve és szignatúrája¹, így a programozónak általában értelmes hibaüzenetek adhatóak.

Adatszerkezetek ellenőrzése

A lefordított binárisban önmagában már nem állnak rendelkezésre az adatok belső struktúráját leíró információk, csupán a szimbólumnevek alapján az adatok struktúrája nem ellenőrizhető. A binárisok azonban nem csak az szimbólumokból és az implementáció gépi kódjából állhatnak: amennyiben a binárist úgy fordították, hogy tartalmazzon nyomkövető információkat²³, úgy ennek részletességétől függően van esély az adatstruktúrák szerkezetének ellenőrzésére. Példaként a Unix és Linux rendszereken elterjedt nyomkövető formátum a DWARF[3, 4] tartalmazhatja ezeket az információkat, így ez segíthet a bináris kompatibilitás ellenőrzésében.

Implementáció változások ellenőrzése

Egy dinamikus programkönyvtár lefordított kódjából esetenként kinyerhető, hogy egy adott szimbólumnak mi gépi kódú vagy assembly nyelvű implementációja⁴, így a programkönyvtár két verziójában található implementációt ez alapján összehasonlíthatjuk. Amennyiben nem találunk különbséget, úgy nagy biztonsággal állíthatjuk, hogy a szimbólum implementációja (jelentése) nem változott. Abban az esetben amikor eltérést tapasztalunk, még további ellenőrzésre van szükség, hogy eldönthető legyen, hogy valóban történt-e változás és hogy ez a változás okozhat-e problémát.

Ahhoz, hogy ez a technika minél pontosabb legyen, lehetőleg ugyanazt a fordítót kell használnunk az új verzióhoz, ugyanazokkal a fordítási paraméterekkel futtatva mint a korábbi verzió esetén, de még ekkor is a fordító általi kód optimalizálás miatt előfordulhatnak nem várt különbségek.

¹A visszatérési típus általában nem állítható vissza, mivel sokszor az nem része a szimbólumnévnek.

²GCC esetén a -g opció család.

³Némely esetben előfordulhat, hogy a nyomkövető információk külön fájlba kerülnek és nem a binárisba, ekkor természetesen azokat a fájlokat is fel kell használni az ellenőrzéshez.

⁴Linux rendszereken például az objdump programot is használhatjuk.

Mint látható ezzel a technikával többnyire csak tájékoztató jellegű ellenőrzéseket végezhetünk, de speciális esetekben segíthetnek a nem várt változások felderítésében.

5.2. Statikus kódelemzés

Egy kliens program fordításakor a fordítóprogram a dinamikus programkönyvtár fejláblományaiából következteti ki a felhasználni kívánt szimbólumokat, valamint a fejláblományok alapján határozza meg az adatstruktúrák szerkezetét, ezért a fejláblomány elemzésével az interfészváltozásokat is elemezni lehet. A statikus kódelemzés lehetőségeinek vizsgálatára készítettem egy programot amely fejláblományok alapján képes elemezni egy programkönyvtár két verziója közötti interfész változásokat. A továbbiakban a program készítése során szerzett tapasztalatok alapján fogom bemutatni ezt a módszert.

5.2.1. A forráskód értelmezése

A statikus kódelemzés első lépése, hogy feldolgozzuk a C++ fejláblományokat amelyek meghatározzák a programkönyvtárat, hogy a számunkra lényeges információkat kigyűjthessük belőlük. Amennyiben a programkönyvtár implementációját is elemezni szeretnénk, úgy annak teljes forráskódját is elemeznünk kell a fejláblományok mellett.

A feldolgozás mind szintaktikai, mind szemantikai elemzést jelent, így ez nem triviális feladat, ezért erre célszerű egy már meglévő C++ fordítót használni. Erre a feladatra én a clang projektet választottam, amely egy ingyenes és nyílt forráskódú, LLVM fordítóra épülő frontend. A clang amellett, hogy egy egyre jobban elterjedt fordító amely támogatja a C++11 szabványt is, egy kényelmesen használható C++ keretrendszert is szolgáltat, amellyel egyszerűen elemezhetünk C++ forráskódokat. További érvként szólt a clang általi elemzés mellett, hogy így teljes mértékben egy fordító szemszögéből vizsgálható a forráskód.

A clang használatán kívül szóba jöhetett volna még a GCC használata is, de ez nem támogat olyan magas szintű keretrendszert a kódelemzésre mint amelyet a clang igen, ezért különböző - többnyire nem dokumentált - parancssori argumentummal kellett volna a forráskódok szemantikáját leíró fájlokat generálni, majd ezeket elemezni amely tovább bonyolítja a feldolgozás folyamatát, valamint a különböző GCC verziók között megváltozhat a generált fájlok formátuma is ami további problémát jelenthet.

Deklarációk szűrése

Egy dinamikus programkönyvtár felhasználhat más programkönyvtárakból is elemeket. Amikor egy programkönyvtár bináris kompatibilitását szeretnénk vizsgálni, akkor feltehetjük, hogy a függőségei binárisan kompatibilisek (vagy amennyiben erre igény van, akkor a függőségek bináris kompatibilitását külön kell ellenőrizni), így azokat az entitásokat amelyek a függőségek fejláblományai miatt láthatóak már nem kell ellenőrizni, ezáltal is gyorsítani lehet az ellenőrzést. A szűrés során arra mindenképp figyelni kell, hogy azokat a deklarációkat ne szűrjük ki amelyek valamilyen módon befolyásolják a programkönyvtár

adatszerkezeteit, például azokat az osztályokat amelyeket valamely vizsgálandó osztály használ mint ős vagy mint adattag.

Ezen elemek kiszűrésének egyik módja, ha leszűkítjük, hogy mely fájlokból származó deklarációkat vesszük figyelembe. Mivel egy programkönyvtár fejállományai a fájlrendszerben jellemzően egy (vagy több) jól meghatározható szülőkönyvtár alá tartoznak így ez a fájl útvonalak alapján nem nehéz feladat.

5.2.2. Modell építés

A forrásfájlok feldolgozásakor mind két programkönyvtár verzióból ki kell gyűjtenünk azokat az információkat amelyek bináris kompatibilitás szempontjából relevánsak, majd ezeket kell összehasonlítani. Ezeket az adatokat valamilyen modellben tárolnunk kell legalább addig amíg az összehasonlítás meg nem történt.

Modell építése adatbázisban

A modellépítés egyik módja, hogy a modellt egy adatbázisban építjük fel, majd később az összehasonlítási fázisban az adatbázisok tartalmát kell összehasonlítani. A módszer egyik előnye, hogy egy verzióhoz az adatbázist elegendő csak egyszer felépíteni, így egy új verzió megjelenése esetén elegendő csak az új verzió kódját elemezni, majd a két adatbázist összehasonlítani. Akkor érdemes ezt a módszert választani amikor a modellépítés jelentős időt vesz igénybe a programkönyvtár méretéből kifolyólag vagy esetleg mélyebb analízisek miatt mint például a kivételek analízise (4.1.1). Az előbbi megfontolások miatt a programom jelenlegi implementációjához egy relációs adatbázis alapú modellt használtam.

Modell építése memóriában

A modell építés egy másik módja amikor a modellt mind két vizsgált verzióhoz a memóriában építjük fel. Ellenben az adatbázisban tárolt modellel, ebben az esetben a régebbi programkönyvtár verzió kódjának is jelen kell lennie az összehasonlítás során. Ennek az előző verzióhoz képest nyilvánvaló előnye, hogy így megspórolhatjuk a fájlműveleteket, ezáltal gyakorlatilag csak az analízis sebessége határozza meg a program sebességét, viszont mind a két programkönyvtárat újra fel kell dolgozni. A módszernek korlátot szab a rendelkezésre álló memória, de amennyiben a két modell együttesen elfér a memóriában, ez a fajta megközelítés gyorsabb lehet mint az előző.

6. fejezet

Kapcsolódó munkák

A kutatómunka során megpróbáltam már meglévő, jól használható megoldásokat keresni a bináris kompatibilitás automatikus ellenőrzésére. A keresés során arra lettem figyelmes, hogy bár igény van ilyen szoftverekre, de elég kevés igazán használható megoldást találtam. Az összes találat közül az abi-compliance-checker messze kiemelkedett a többi közül. Mint az később látható lesz, igen jól támogatja a bináris kompatibilitás ellenőrzését.

6.1. abi-compliance-checker

Az abi-compliance-checker egy C illetve C++ programozási nyelvhez kifejlesztett ingyenes és nyílt forráskódú eszköz, amellyel ellenőrizhető a bináris, valamint a forrás kompatibilitás is. Az ellenőrzéshez szükséges információkat a programkönyvtár lefordított binárisából, valamint a fejállományaiból gyűjti ki, amelyek megadásához programkönyvtár verzióként egy XML fájlt kell készíteni. Az eszköz (amennyiben ezt külön bekapcsoljuk) képes az implementáció változások jelzésére úgy, hogy a programkönyvtár két verziójának binárisaiból visszafejti az implementációkat assembly kódba és ezeket (lexikálisan) összehasonlítja.

Lehetőségünk van csak a fejállományok, illetve csak a bináris kód alapján való ellenőrzésre is, viszont ekkor csak kevesebb hibát képes detektálni. Bizonyos szempontból érdekes az eszköz megvalósítása, így tekintsük át, hogy hogyan működik (a dolgozat írásakor a legfrissebb verzió a 1.99.8.5 volt, így az itt leírtak arra vonatkoznak):

Az abi-compliance-checker egy Perl nyelven megírt szkript, amely az aktuális platformtól függően különböző eszközöket használ az elemzés elvégzésére. A platformok támogatottsága igen széleskörű, támogatja az összes elterjedt platformot¹. A binárisokból az információkat a platformnak megfelelő eszközökkel gyűjti ki, de a fejállományok feldolgozásához mindig a GCC²-t használja. Mint azt a 5.2.1 részben írtam én is vizsgáltam a GCC felhasználását az elemzésre, de végül elvettem mert nem tartottam elég stabilnak a generált fájlformátumot. A közelmúltban abi-compliance-checker fejlesztői is tapasztalhatták ezt

¹A következő platformokat támogatja: GNU/Linux, FreeBSD, Mac OS X, MS Windows, Haiku/BeOS

²GNU Compiler Collection

a problémát: az elemzéshez a `—fdump—translation—unit` kapcsolót használják (amely egy nem dokumentált, belső használatra szánt kapcsoló), ami nem úgy működik a GCC 4.8.0 verzióban mint az GCC előző verziójában³. A döntésüket az indokolhatja, hogy az abi-compliance-checker első verziója 2009-ben jelent meg, amikor a clang még nem volt olyan elterjedt mint napjainkban.

Bár az abi-compliance-checker messze a legkomplettebb megoldás amit találtam, a kivételek kérdésével nem foglalkozik eltekintve a binárisokban található implementációk lexikális összehasonlításától.

A nyílt forráskódú világban viszonylag elterjedtnek mondható. Többek között a glibc-t is ennek segítségével ellenőrzik, valamint a <http://upstream-tracker.org> weboldal is ezt használja a programkönyvtárak ellenőrzésére. A jelentéseket XML valamint HTML formátumban is képes előállítani, így azok gépi valamint emberi feldolgozásra is alkalmasak.

6.2. shlib-compat

Ez az eszköz az adott dinamikus programkönyvtár DWARF szimbólumait használja a bináris kompatibilitás ellenőrzéséhez. Ezt az eszközt főként C programozási nyelven írt programokhoz fejlesztik de bizonyos korlátok között C++ nyelvű programkönyvtárakhoz is használható. Érdeemes megemlíteni, hogy támogatja a szimbólumverziózást. Mivel Unix specifikus eszközöket használ, ezért nem támogat olyan sok platformot mint az abi-compliance-checker.

³Részletek az alábbi hibaleírásban: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=57850

7. fejezet

Tanácsok, technikák

Ebben a fejezetben ismertetek néhány olyan technikát amellyel bizonyos feltételekkel részben megőrizhető a bináris kompatibilitás, valamint ismertetem az adott technika előnyeit és hátrányait.

7.1. Verziózás

Ebben a szakaszban bemutatok pár módszert a szimbólumok verziózására. A verziózó technikák lényege, hogy úgy vezetünk be módosításokat, hogy megtartsuk a korábbi implementáció másolatát is.

7.1.1. Szimbólum verziózás bináris szinten

A bináris kompatibilitás elvesztésének egyik oka az lehet, ha egy már létező szimbólumot törölnek vagy a jelentését megváltoztatják. Erre a problémára vezették be a Linux rendszereken a glibc 2.1-es verziójától kezdve a szimbólum verziózást [2] amely a Sun egy korábbi megoldásán alapul. A megoldás lényege, hogy egy dinamikus programkönyvtárban egy szimbólumnak több verziója is jelen lehet. A kliens alkalmazás fordításkor megadhatja, hogy mely verziót szeretné használni, ekkor tulajdonképpen nem csak egyszerűen a szimbólumnévre fog hivatkozni a lefordított kódban, hanem a verziózott szimbólumnévre.

A dinamikus programkönyvtár írói amikor egy szimbólum jelentését megváltoztatják, akkor egy új verziót adnak a könyvtárhoz az adott szimbólumból, de a régebbit is meghagyják, így a korábban fordított alkalmazások továbbra is működni fognak, mivel azok a régebbi verzióra hivatkoznak, valamint garantáltan az általuk elvárt (korábbi) működést fogják tapasztalni. Bár ezt a fajta verziózást alapvetően a C programozási nyelven írt programokhoz találtál ki, de van példa C++ nyelvű programkönyvtáraknál való alkalmazásra is (erre egy példa a GNU C++ standard könyvtár implementációja [6]).

7.1.1.1. Esettanulmány

Példaként tekintsük az alábbi C programozási nyelven megírt vektor implementációt¹:

¹A példa a FreeBSD projektből egyik példája alapján készült: http://people.freebsd.org/~deischen/symver/freebsd_versioning.txt (2013 November)

```

1 struct __vector;
2 typedef struct __vector* vector_t;
3
4 ...
5 vector_t v_create(int initial, int max);
6 ...

```

A fentebbi `v_create` függvényt egy későbbi verzióban ki szeretnénk egészíteni egy új paraméterrel az alábbi módon:

```

1 struct __vector;
2 typedef struct __vector* vector_t;
3
4 ...
5 vector_t v_create(int initial, int extent, int max);
6 ...

```

Tisztán látszik, hogy az interfész inkompatibilis módon megváltozott, de a C szimbólum maradt `v_create`, ezért a korábban lefordított alkalmazások rosszul fogják használni az új függvényt. A megoldás ilyenkor, hogy létrehozunk két külön implementációt a két verzióhoz például `__v_create_old` és `__v_create_new` néven, majd az egy speciális szintaxissal megjelöljük, hogy melyik függvény felel meg korábbi és melyik az új verziónak:

```

1 .symver __v_create_old, v_create@VER_1.0
2 .symver __v_create_new, v_create@@VER_1.2

```

A `.symver` assembler utasítás után a valódi implementáció szimbólumnevet kell írni, majd vesszővel elválasztva a verziózott szimbólumnevet szimbólum@verzió formában. A két `@` jellel az alapértelmezett implementációt lehet kijelölni. Ebben az esetben a dinamikus programkönyvtár betöltő már tudni fogja, hogy melyik implementációt kell rendelnie az adott alkalmazás `v_create` hívásához.

7.1.1.2. Limitációk

A szimbólumverziózás egyik előfeltétele, hogy a fordító, a linker és a betöltő is támogassa a szimbólumverziózást. Másfelől az esettanulmányból látható, hogy ez egy erősen platformfüggő megoldás, így nem használható általánosan. A szimbólum verziózás igen nehezzé válik C++ programozási nyelvre alkalmazva, főleg a virtuális táblákat érintő változások verziózásakor, ezért nem igazán ajánlják a használatát ([6] „Outstanding Issues” rész).

Továbbá látható, hogy a programozó feladata, hogy kézzel megjelölje a szimbólumok különböző verzióit, ami megterhelő feladat, valamint könnyen hibákhoz vezethet².

7.1.1.3. Konklúzió

Ez a megoldás elsősorban a C programozási nyelvű programkönyvtárakhoz lett kifejlesztve így nem veszi figyelembe a C++ nyelv működését, ezért csak korlá-

²Például véletlenül valaki rossz verzióval jelöl meg egy szimbólumot: például a `libstdc++` könyvtár ABI verziója a GCC 4.2.1 és 4.2.2 verzióiban is 6.0.9 volt, de történ inkompatibilis módosítás (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=33678).

tozottan alkalmazható C++ kódokra. Az is látható, hogy ez a megoldás korántsem platformfüggetlen, valamint sok feladatot hárít a programozókra. A programkönyvtár mérete az idő haladtával igencsak megnőhet a különböző verziók tartalmazása miatt, ami a gondot okozhat a programkönyvtár felhasználóinak.

7.1.2. Verziózás névterekkel

Névterekkel lehetséges a 7.1.1 szakaszban bemutatott technikához hasonló verziózást megvalósítani tisztán C++ nyelven, azaz platformfüggetlen módon. A technika lényege, hogy minden könyvtárverzióhoz tartozik egy saját, verziózott névtér, de a régebbi verziókat is megtartják.

7.1.2.1. Esettanulmány C++03 szabványú kóddal

Annak érdekében, hogy a kliens kódban ne kelljen feltüntetni a verziókat, létrehozhatunk egy fő névteret amelybe a **using namespace** kifejezéssel beemeljük a legutolsó verziót. A verziózott névterek természetesen lehetnek a főnévtérbe beágyazott névterek is. A korábbi vektoros példa (lásd 7.1.1.1 szakaszban a(z) 31. oldalon) alapján nézzük az alábbi példát:

```
namespace lib {
namespace v1 {
    struct Vector;
    Vector* create(int initial, int max);
}
namespace v2 {
    struct Vector;
    Vector* create(int initial, int extent, int max);
}
using namespace v2;
}
```

Ez a technika egész jól működik amennyiben nem találhatóak sablonok a kódban. Abban az esetben viszont amikor sablonosztályokat és/vagy sablonfüggvényeket is definiál a programkönyvtár, beleütközhetünk egy limitációba: bár a deklarációk a kliens nézőpontjából látszólag a főnévtérben vannak (esetünkben a lib névtér), de ennek ellenére a sablon specializációkat csak a verziózott névterekben tudja deklarálni mivel a C++ szabvány [7] szerint a sablonokat csak abban a névtérben lehet deklarálni amiben az eredeti sablon is deklarálva volt.

7.1.2.2. Esettanulmány C++11 szabványú kóddal

A C++11 szabványban megjelent az **inline namespace** fogalma. Ha egy beágyazott névteret **inline**-ként definiálunk, akkor azt mondjuk, hogy a benne található deklarációknak pontosan ugyanúgy kell működniük mintha a külső névtérben lennének. Módosítsuk ez előző példát ennek megfelelően:

```
namespace lib {
namespace v1 {
    struct Vector;
    Vector* create(int initial, int max);
}
```

```

}
inline namespace v2 {
    struct Vector;
    Vector* create(int initial, int extent, int max);
}
}

```

Látható, hogy így már nem szükséges a `using namespace` használata valamint mivel a legfrissebb verzió `inline`, ezért a sablonok is specializálhatóak a külső - esetünkben lib - névtérben.

7.1.2.3. Konklúzió

A két esettanulmány megmutatta, hogy ezzel a technikával fenntartható a bináris kompatibilitás, mivel egy újabb verzió tartalmazza a régebbi verzió összes kódját³. Az is látható, hogy szemben a 7.1.1.1 szakaszban megismert módszerrel ez a módszer tisztán a C++ programozási nyelvet használja a verziózás megvalósítására, így nem platformfüggő.

Sajnos azonban ez a módszer sem praktikus megoldás nagy méretű programkönyvtárak esetén, mivel a kódmásolatok miatt a programkönyvtár mérete verzióról-verzióra egyre csak növekedik, így ez idővel gondot okozhat mind karbantartási, mind terjesztési szempontból.

7.1.3. Az új viselkedés elérhetőségének szabályozása

A 4.1 részben láthattuk, hogy az implementáció változások miatti viselkedés módosulás milyen hatással van a kliensekre. Amennyiben az új funkciókat úgy implementálják, hogy az külön bekapcsolható a kliens által, de alapértelmezés szerint nincs bekapcsolva, akkor elkerülhetők az új implementáció miatti inkompatibilitások. Erre sok módszer létezik: használhatunk funkcionként vagy programkönyvtár verzióként egy globális változót vagy akár egy inicializáló függvényt, amelynek megadjuk, hogy mely könyvtárverzió szerinti működést szeretnénk használni.

Látható, hogy ez a módszer kellő odafigyeléssel és tervezéssel megoldhatja a 4.1 részben tárgyaltakat. Mindazonáltal a módszer hátrányaként róható fel, hogy itt is kód többszörözés történik, ami növeli a programkönyvtár méretét, továbbá a sablonokat nagyon nehéz úgy tervezni, hogy támogassák ezt a fajta verziózást.

7.2. Híd programtervezési minta használata

Egy jól ismert strukturális programtervezési minta a híd (angolul Bridge pattern), amelyet a C++ világban szokás `pimpl`-nek is nevezni. A módszer lényege, hogy az implementációs részleteket teljesen elrejtjük egy átlátszatlan mutató (angolul opaque pointer) mögé. A felhasználó interfészként csak egyszerű függvényeket illetve metódusokat kap, az adattagok az átlátszatlan mutatóba vannak

³Érdemesnek tartom megjegyezni, hogy a kódDuplikálás miatt ezzel a megoldással még akkor is képesek vagyunk a bináris kompatibilitás fenntartására amikor maga az új verzió már nem is forrás kompatibilis a régi verzióval szemben.

elrejtve, így mutató által hivatkozott memória szerkezete bármely későbbi verzióban binárisan kompatibilis módon megváltoztatható. Az alábbi kódrészlet mutatja a megoldás vázlatát:

```
// L.hpp
class X {
    struct XImpl;
    XImpl* pimpl;
    ...
public:
    ...
    int f();
};
// L.cpp
...
int X::f() {
    return pimpl->f();
}
```

Mivel az `XImpl` osztály szerkezete nem elérhető a kliens kódból, ezért a szerkezetén történő módosításoknak nincs a bináris kompatibilitásra gyakorolt hatásuk.

Látható, hogy bár az adatszerkezetek módosításával kapcsolatos problémák egy részét ezzel a módszerrel orvosolhatjuk, de ez csak egy kis szelete a bináris kompatibilitással kapcsolatos témának. Beágyazott rendszereken egyes esetekben túl költségesnek bizonyulhat a `pimpl` mutató használata, valamint mivel (majdnem) minden alkalommal amikor egy interfészhívás történik egy mutató dereferálás is történik, ezért teljesítménycsökkenést is tapasztalhatunk.

8. fejezet

Eredmények

Dolgozatomban céloom főként a bináris kompatibilitás, mint koncepció vizsgálata volt. Az elméleti kutatások mellett elkészítettem egy kísérleti szoftvert, amely a gyakorlatban is alkalmazható a bináris kompatibilitás ellenőrzésére. A szoftver már a jelenlegi állapotában is alkalmas számos (főként a függvényekre és metódusokra vonatkozó) binárisan inkompatibilis változtatást kijelezni. A bináris kompatibilitás teljes eldöntésére, az összes részlet felfedezésére még további fejlesztésekre van szükség.

Most összehasonlítom az én megoldásomat a 6.1 részben bemutatott abi-compliance-checkerrel¹. Mind két programmal a Xerces nyílt forráskódú XML feldolgozó programkönyvtár C++ programnyelvű változatának 3.1.0 és 3.1.1 verzióját hasonlítottam össze. Azért választottam a Xerces programkönyvtárat mert egy népszerű C++ nyelvű programkönyvtár, mindemellett a több mint négyszáz fejláblományában elég sok ellenőrizhető minta található.

A programkönyvtár mind két verzióját lefordítottam és külön-külön könyvtárakba feltelepítettem. Ezután elsőként az abi-compliance-checkert futtattam a két verzió összehasonlítása érdekében. Az eszköz tíz futtatást tekintve, átlagban negyvenöt másodperc alatt végezte el a teljes analízist. *Az ellenőrzés eredményeként* az eszköz egy HTML jelentést generált, amelyben *egy súlyos hibát nevezett meg*. A programkönyvtárban található egyik osztály (Schema-Info) egyik konstruktorának argumentumlistája megváltozott, ezért az eredeti változathoz tartozó szimbólum eltűnt az új verzió binárisából. Az eredményt ellenőriztem és azt tapasztaltam, hogy a jelzett hiba valóban fenn áll.

Az általam elkészített szoftver egy adott programkönyvtárhoz egy adatbázis alapú modellt épít, majd egy újabb futtatással két ilyen módon létrehozott adatbázis lehet összehasonlítani. Ez azt jelenti, hogy a két fázis, az adatgyűjtés és az összehasonlítás teljesen elkülönül. A teljes analízishez, tehát programkönyvtár verzióként egy modellépítésre és egy összehasonlításra volt szükség, ez összesen hét másodpercet vett igénybe, a modellek adatbázisai egyenként két megabyte-osak lettek. A modell építés közben három fejláblományra fordítási hibát jelzett a clang keretrendszer, de ezek a hibák mind inline függvények törzsében jelentkeztek, így a modell építését nem befolyásolták.

A saját megoldásom hat súlyos hibát talált. Egyrészt megtalálta az abi-compliance-checker által is megtalált hibát, valamint további öt függvényátne-

¹A teszthez az eszköz 1.97.7 verzióját használtam.

vezésből adódó hibát. A Xerces 3.1.0 verzióban több osztályban is megtalálható volt egy `insureCapacity` nevű metódus, amelyeket az új verzióban már `ensureCapacity` néven találunk meg. A változtatás oka nyilvánvalóan egy elírás orvoslása volt. A találatok különbsége abból adódhat, hogy az abi-compliance-checker nem ellenőrzi a privát metódusokat, ami jogos abban az esetben amikor a kliens nem érheti el azokat. Ebben az esetben viszont az `XMLBuffer` osztály esetén az `insureCapacity` metódust publikus, **inline** metódusokban is használták, így az elérhetővé vált a kliens kód számára is. A többi négy esetben valóban nem okozhat problémát a metódus átnevezés. Az eredmény tesztelésére létrehoztam egy egyszerű példaprogramot amely az `XMLBuffer` `append` metódusát hívja meg, amely egy olyan publikus **inline** metódus, amely használja az `insureCapacity` metódust. Mikor a dinamikus programkönyvtár új verziójával próbáltam meg elindítani a programot, a betöltő az elvárásoknak megfelelően ki jelezte, hogy hiányzik az `insureCapacity` metódus szimbóluma, így a programot nem lehet elindítani.

9. fejezet

Konklúzió

A bináris kompatibilitás feltételeinek vizsgálata során látható volt, hogy a kompatibilitást meghatározó tényezőket nagyban befolyásolja az éppen használt C++ fordítóprogram által használt ABI, valamint bizonyos esetekben az éppen használt C++ nyelvi szabvány is befolyásolhatja a generált kódot. Amennyiben a C++ nyelvű kódokra betartjuk a felsorolt megszorításokat, szabályokat, úgy nagy eséllyel binárisan kompatibilis kódot kapunk a jelenlegi szabvány és a jelenleg elterjedt fordítóprogramok vizsgálata alapján. Egy újabb C++ szabvány vagy egy merőben eltérő bináris interfészt használó fordítóprogram megjelenésekor egyes szabályok módosulhatnak, vagy újak is megjelenhetnek.

Leggyakrabban egy szimbólum megváltozása vezet bináris inkompatibilitáshoz. Láthattuk, hogy a diplomamunkában bemutatott módszerekkel az ilyen típusú változások jól ellenőrizhetők programmal. A meglévő szabályok nagy része a programkönyvtárak fejállományai alapján szintén ellenőrizhető szoftverrel, így elmondható, hogy bár nem minden esetben fedezhető fel az összes változás, de a változások túlnyomó többségét képesek jelezni az ellenőrző programok.

A bináris kompatibilitás automatikus ellenőrzése ha nem is teljes mértékben, de elég jól megoldott ahhoz, hogy egy szoftver tesztelési folyamatába beépíthető legyen. Mint azt láthattuk, egy egyszerű privát metódusnév elírásának javítása is okozhat bináris inkompatibilitást, az ilyen hibákat egy automatikus ellenőrzés kijelezhette volna.

Annak érdekében, hogy még pontosabban lehessen ellenőrizni a bináris kompatibilitást, további kutatás szükséges. Az implementáció megváltozásából adódó inkompatibilitások ellenőrzése még nem megoldott, csak durva módszerek állnak rendelkezésre. Ez alól kivételt képez a kivételkezelés ellenőrzése: a forráskód ismeretében megvalósítható egy elfogadható szintig [8] a kivételek ellenőrzése, ennek ellenére nem találtam olyan bináris kompatibilitást ellenőrző szoftvert amely ezt megtenné. A jelenlegi módszerek nem alkalmasak a sablonok belső szerkezetének vizsgálatára, csak azok példányait tudjuk ellenőrizni, de mint láthattuk a sablonok jelentős mértékben befolyásolhatják mind a kliens működését, mind a programkönyvtár interfészét, így a sablonok bináris kompatibilitására gyakorolt hatásainak további analízise is indokolt.

Irodalomjegyzék

- [1] Itanium C++ ABI, 2013. URL: <http://mentorembdedded.github.io/cxx-abi/abi.html>.
- [2] Ulrich Drepper. ELF Symbol Versioning, November 2013. URL: <http://www.akkadia.org/drepper/symbol-versioning>.
- [3] DWARF Debugging Information Format Committee. DWARF Debugging Information Format Version 4, November 2013. URL: <http://dwarfstd.org>.
- [4] Michael J. Eager and Eager Consulting. Introduction to the DWARF Debugging Format. 2012.
- [5] Agner Fog. Calling conventions for different C++ compilers and operating systems, November 2013. URL: http://www.agner.org/optimize/calling_conventions.pdf.
- [6] GNU. The GNU C++ Library Manual: ABI Policy and Guidelines, November 2013. URL: <http://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html>.
- [7] ISO. International Standard ISO/IEC 14882:2011(E) – Programming Language C++, 2011.
- [8] Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivancic, and Aarti Gupta. Interprocedural Exception Analysis for C++. In Mira Mezini, editor, *ECOOP*, volume 6813 of *Lecture Notes in Computer Science*, pages 583–608. Springer, 2011. URL: <http://dblp.uni-trier.de/db/conf/ecoop/ecoop2011.html#PrabhuMBIG11>.
- [9] Pavel Shved and Denis Silakov. Binary Compatibility of Shared Libraries Implemented in C++ on GNU/Linux Systems. 2009.
- [10] Bjarne Stroustrup. *The C++ Programming Language (4th Edition)*. Addison-Wesley Professional, 2013.
- [11] tiobe.com. TIOBE Programming Community Index for October 2013, Október 2013. URL: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.