



ÚJ Nemzeti Kiválóság Program
2018/19



EMBERI ERŐFORRÁSOK
MINISZTERIUMA

A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései

Név: Szalay Richárd

Kar: ELTE IK

Budapest, 2019. június 30.

Név: Szalay Richárd

ÚNKP 2018/19

Kar: ELTE Informatikai Kar

Képzési szint: mesterképzés (MSc)

„A” keret

Kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései

Azonosító adatok:

Név: Szalay Richárd

Lakcím: 1214 Budapest, Bánya utca 13. Fsz. / 2.

Adóazonosító jel: 8471253089

Szerződés száma: ÚNKP-18-2-I-ELTE-156

A szerződés tárgya/a kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései



A pályázó záró beszámolóját elfogadom.

Dátum: Budapest, 2019. május 24.

.....
Pál Zoltán
témavezető

Tartalom

Bevezetés.....	5
Motiváció.....	5
Kapcsolódó munkák.....	6
Kutatómunka összefoglalása.....	6
Eredmények áttekintése.....	8
Téma bemutatása.....	10
A „klasszikus” C++ fordítási modell.....	10
Problémák a klasszikus modellben.....	11
Modules.....	12
Automatikus modularizáció módszere.....	13
Eredmények.....	14
Publikálási terv.....	15
Irodalomjegyzék.....	17
Mellékletek.....	20
1. Összegzés.....	20
2. Az ÚNKP kapcsán konferenciaszereplések és publikációk megjelenésének felsorolása (MTMT-ből kinyert lista, ha van).....	22
3. Konferencia részvétel, kutatóút eredményeinek bemutatása.....	22
4. Csatolmányok listája.....	22

Bevezetés

Motiváció

A programozási nyelvek hosszú ideje támogatják a programegységek logikai hierarchiába szervezését, melyek mentén számos fordítóprogram optimalizáció végezhető el, amely mentén a kódot szervezik az adott nyelven fejlesztők. A programok logikai szervezettsége egy kívánt tulajdonság, mivel a jó szervezettség elősegíti a fejlesztők energiabefektetésének optimalizálását. A nyelvi szabályok, fordítóprogram implementációk és a programozói közösségek „jótanácsai” között számos érvrendszer terjedt el, melyekben nyelvről nyelvre kialakították a szokásaikat a fejlesztők. A mindennapi életben elterjedt programozási nyelvek közül sajátos a C++ két okból: az egyik, hogy a szoftverprojektek elemeinek logikai szervezésére nagyon kevés jó módszert biztosított (a szöveges beillesztésen alapuló fejlécállományoktól eltekintve), valamint a fejlesztés során a fordítással töltött időben is kiemelkedően rosszul teljesített. A C++ programozási nyelv ezen tulajdonságai meglepőek az elterjedtségéhez képest, a világ összes C++ programkódja a több tízmilliárd sor (lines-of-code, LoC) nagyságrendben mérhető – csak a Google saját bevallása szerint 600-800 millió LoC C++ kódot fejlesztett házon belül. A más mindennapi programozási nyelvekhez képest nehézkes szervezés és lassú inkrementális fejlesztés programozók tízezreinek életét keseríti meg – ebből fakadóan a kutatás eredményei, és bármely e témában végzett kutatás drasztikus minőségbeli javulást és költség- valamint erőforrásmegtakarítást okoz.

A C++ programozási nyelv legújabb kiadása (C++20) tartalmazza az új nyelvi elemek között a *Modules*-t (Smith, 2018), amely a programegységek logikai szervezését teszi lehetővé a korábbi fejlécállományos (header) rendszerhez képest. A szabványkiegészítés 2004 óta volt terítéken és a C++ közösség előtt, mely során számos ellenjavaslatot és diszkussziót élt meg. A diszkusszió során a C++ közösség változatos módon érvelt, ki-ki a saját álláspontja mellett, mutatva, hogy egy mindenki által várt, de nem ugyanolyan módon elképzelt nyelvi elemről van szó. A *Modules* célja Gabriel Dos Reis 2004-es első javaslata (Dos Reis, Hall, & Nishanov, 2016) óta a preprocesszoron való függés minimalizálása, és egy logikai szervezettség lehetővé tétele, amely során a programozónak már nem kell az egyes fájlok fizikai kialakításával foglalkoznia. Számos egyéb rosszul használható konstrukció is ellehetetlenül a modulrendszerre való áttéréssel, pl. a fejlécállományokból való makrók hatókörének kezelhetetlen kiszökése a főprogramba.

Azonban a nyelvi szabvány („jogszabály”) támogatása egy ilyen funkcióhoz nem elégséges a tökéletes kivitelezés érdekében, mivel azt a fordítóprogramoknak is támogatnia kell, még hozzá olyan minőségben, hogy a korábban említett fordítási idő legalább ne romoljon el a „hagyományos” eljáráshoz képest. A *Modules* céljai között sosem volt a fordítási idő enyhítése az eredeti javaslatot benyújtók szerint, de a kezdeti implementációk tapasztalatai azt mutatják, hogy akár drasztikus romlást is okozhat. A szabványjavaslat szövege a kutatás ideje alatt még végleges kidolgozás és elfogadás alatt állt, így a kutatás során az eddigi prototípus implementációkra és a közösség által kidolgozott lehetőségekkel dolgoztunk.

A motiváció fordítási időről szóló szignifikáns része személyes motivációból fakad: korábbi kutatás-fejlesztési projektek során aktívan fejlesztettem C++ nyelven programokat, és legtöbbször a legprogramozott algoritmusok lépéseinek szerkesztésekor újra kellett fordítani a programot, hogy a tesztet elvégezhessek – ez az újra-fordítás akár 6 percig is tartott még akár olyan apróságok esetében is, mint az iteráció határainak állítása.

Kapcsolódó munkák

A C++ programozási nyelv fordítási modelljének és a szoftverprojektek szervezéseinek problémáit számos korábbi kutatás elemezte. A nyelv újabb és újabb szabványainak megjelenésével a komplexitás egyre csak nő, amely számos lehetőséget ad a programozóknak, hogy rosszul használják a nyelvi szabályok adta lehetőségeket. Így a kutatói munka egy részét irodalomelemzés és nyílt forráskódú projektekben és azok történetében rossz szoftverfejlesztői minták keresése. Az irodalomelemzés során fényt derítettünk több korábbi erőfeszítésre, amely C és C++ szoftverprojektek architektúrális átalakítását célozták, kis és közepes méretű (10-1000 ezer LoC) projekteket használva a módszerek bemutatására. (Dayani-Fard, Yu, Mylopoulos, & Andritsos, 2005) (Yu, Dayani-Fard, & Mylopoulos, 2003) Több szabványmódosítási javaslat is volt a *Module*stől eltekintve, melyek közül végül egyik sem került a nyelv hivatalosan elfogadott szabványába, de fontos iránymutatást adtak a problémák körüljárásához.

Kutatómunka összefoglalása

Első lépésként a hosszú fordítási időkből adódó mély személyes motivációm kapcsán a kutatásom során megvizsgáltam, milyen eszközök állnak rendelkezésre a C++ projektek

fordítási idejének leküzdésére (ccache, distcc, BazelBuild, zapcc (Ltd., 2018)) és miként lehet a modularizációt elősegíteni meglévő projekteken.

A fordítást gyorsítani hivatott külső – a fordítási folyamatot körül ölelő – eszközökön méréseket végeztünk, azonban hosszú távon az ilyen megoldások nem segítenek érdemben leküzdeni a fordítóprogramok komplexitását. Ez különösképpen igaz a mindennapos inkrementális fejlesztés során – a tesztelésre/kiadásra szánt változatok elkészítésekor elhasznált energia és idő sem triviális környezetvédelmi (green computing) szempontból, azonban a kutatás a fejlesztők napi szintű munkáját hátráltató tulajdonságokra terjedt ki.

- A *ccache* esetében az elmentett korábbi fordítások csak tökéletes egyezés esetén hívhatók elő – egy karakter módosítása a forrásszövegben azonnal ellehetetleníti a találatot, és a fordítóprogramot a fordítás nyers elvégzésére (mintha nem is lenne gyorsítótár) kötelezi.
- A fordítási folyamat hálózaton szétterítését lehetővé tevő *distcc* környezetben a párhuzamos fordítások felső korlátja a hálózat átviteli képessége, valamint az összeszerkesztési fázis (*linker*) továbbra is a fejlesztői gépen zajlik. Inkrementális fejlesztés során, amikor csak néhány fordítási egységet kell újraépíteni, a párhuzamosítás nem segít.
- Az előfordított fejlécállományok (*precompiled headers*, (Litman, 1993), (Krishnaswamy, 2000)) technikája a fordítási rendszerben hozzáadott karbantartási lépéseket igényel, és nem segít a generikusok példányosításából adódó sokszor feleslegesen elvégzett munka – ld. a *Problémák a klasszikus modellben* alfejezetben – könnyítésére. Egyéb eszközöket, pl. *template repozitóriumok*, vagy memóriában gyorsítótárazó fordítószerverek is megvizsgáltunk.

Ezt követően terelődött a figyelmünk a modulrendszerre, annak elveire, implementációjára és viszonyaira. A C++ programozói közösség szignifikáns része hitt abban, hogy a modulrendszer többek között a fordítási idő kérdésére is megoldást ad majd. Kérdéses azonban, hogy mennyire lehetséges és kifizetődő a modulrendszerre átállni a fejlesztői közösségnek. Érdemes megemlíteni a visszafelé kompatibilitást számos területen eltörő Python 3 verziót, melynek a kiadásából a Python közösség azóta se állt helyre és frissítette a kódbázist, a Python 2-es és 3-as verzió mind a mai napig párhuzamosan fejlődik. A C++

kódbázis mérete hatalmas, többek között olyan szoftverprojektek milliói létezhetnek, amelyeket akár lehetetlen tovább fejleszteni/frissíteni, többnyire jogi okokból.

Készítettünk tehát egy prototípus implementációt az egyik legelterjedtebb build-rendszerben (*CMake* (Whisperity, 2018)), amellyel a szoftverprojektek fordítási lépései között kifejezhető a modulépítés, valamint a kutatás során témavezetőmmel kialakítottunk egy módszert, amely folyamalgoritmusok felhasználásával modulokba szervezi a projektet. A modularizáció során ún. „unity build”-et készítünk a projekt egyes részeiből, amely során meg kell küzdenünk az esetleges ütközések feloldásával. Ezeket a lépéseket a módszerhez elkészített prototípus implementáció felismeri és átírja az ütköző neveket, elősegítve ezzel, hogy a keletkező rendszer a fordítás során ekvivalens viselkedést generáljon. Az implementáció Python nyelven íródott, és számos ismert gráfalgoritmus- és vizualizációs alkalmazás-könyvtárat használ fel, így könnyen kiegészíthető, valamint a belső reprezentáció könnyen vizualizálható.

Eredmények áttekintése

A kutatói munka során számos korábbi javaslatot tekintettünk át, beleértve a fordítóprogram-fejlesztéseket is. Az elkészített modellt szintetikus példákon felül elterjedt, nyílt forráskódú és különböző méretű szoftverprojekteken ellenőriztük. A „kézzelfogható eredményeket” a modularizációt végrehajtó algoritmus, annak prototípus implementációja, és a mögöttes, a C++ programozási nyelv szabályaira alapuló matematikai modell testesíti meg. A vizualizációs lehetőségeknek köszönhetően a bemenetként használt projekt fejlesztői iránymutatást kaphatnak, hogy a projektjük mely részén találhatóak olyan belső függőségek, melyek tapasztalataink szerint azt mutatták, hogy pusztán a fordítási egységek (forrásfájlok) szintjén történő architektúrális átszervezést végrehajtó klaszterizáció nem elég egy – valamilyen metrika szerint – hasznos modularizáció eléréséhez, azaz szükségszerűen a szoftverprojekt belsejét ismerő fejlesztők időbefektetése és munkája, valamint a belső és külső interfész megtörése szükséges. Ennek az állításnak formális bizonyításvázlatot is készítettünk, nem csak tapasztalati eredményt közlünk.

Érdemes kiemelni az eredmény fontosságát szempontjából, hogy az *LLVM Compiler Infrastructure* nyílt forráskódú fordítóprogram szoftverprojekt hibakereső keretrendszerét (*LLVM Debugger*, *LLDB*) fejlesztő csapat a kutatással párhuzamosan – bár formális modellek és bizonyítékok nélkül, kizárólag a programozói megérzésre és a fordítóprogram által adott hibaüzenetekre hagyatkozva – elkezdte a *Modules* támogatási lehetőségeit a saját projektjükbe

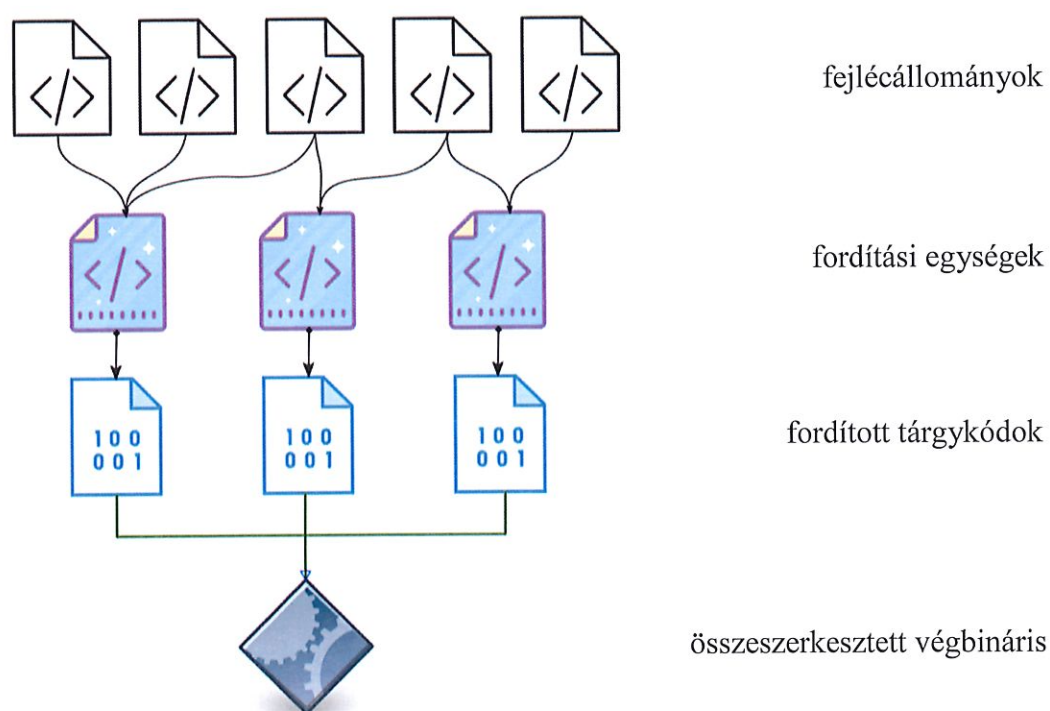
belevinni, és szembesültek azokkal a belső függőségi hibákkal, amik alapján a mi eszközünk is kijelzi, hogy a modularizációhoz a kód interfészének megtörése szükséges (Turner, 2019).

Az Új Nemzeti Kiválóság Program során végzett kutatómunkánk a „vadon fellelt” problémák körüljárása kapcsán az alapjait fektette le egy hosszabb távú kutatás-fejlesztési feladatnak, amely elsődleges célja a fordítási idő és számítási igény csökkentése. Ezt az erőfeszítést a nyelvi szabályokkal és lehetőségekkel – ideértve természetesen a *Modules*-t is – való kompatibilitás megtartása mellett, viszont a célszoftvert fejlesztő programozókra minimális többletmunkát előíró fejlesztésként kell elkészíteni.

Téma bemutatása

A „klasszikus” C++ fordítási modell

A C++ programozási nyelven írt projekteket külön-külön fordítási egységekben készítik el a szoftverfejlesztők. Az egyes fájlok fordítását egymástól függetlenül futtatható fordítóprogram-processzusok végzik, melyek semmilyen információval nem rendelkeznek a többi fordítási egységről, kizárólag azt látják, amelyet a processzus indításakor a parancssori argumentumokban megkapnak (Phases of translation). A projekt teljes vagy inkrementális fordítási folyamatát egy fordító keretrendszer (*build tooling system*) vezényeli, melynek feladata a függőségek számontartása, a fordító argumentumainak összeállítása és a programok indítása, és a futtatható állományok megfelelő mappahierarchiába rendezése. Klasszikus ilyen eszközök a *make* és *ninja*. Ezeket a rendszereket leíró fájlokat egy magasabb absztrakciós szinten konfigurálják (pl. *autoconf* vagy *CMake*).



A fordítási folyamat alapja a **fordítási egység** (*translation unit*), amely általában egy konkrét forrásfájl, annak beágyazott fejlécei, és a konkrét fordítási paraméterek által beállított értékek közössége ad ki. Kezdetben nem csak C++ kód, hanem a preprocesszornak utasításokat adó ún. „direktívák” is megtalálhatóak az állományokban (Ernst, Badros, & Notkin, 2002). A `#include` direktívában megnevezett fejlécállományokat a *preprocesszor* a fordítás

klasszikusan „lexikális elemzésnek” nevezett fázisában szövegszerűen „bemásolja” a bemenetbe, amíg talál ilyen direktívát. Más preprocesszor direktívák (`#define`, `#ifdef`, ...) segítségével a programszöveg egyes részeit eltüntethetjük, vagy felhasználói/környezeti változó értékére cserélhetjük. Az így összeállt bemenet már csak C++ nyelven írt kódot tartalmaz, amelyen a fordítóprogram lefuttatja a fordítóprogramok tudományterületéből ismert módokon a szintaktikus és szemantikus elemzést, és tárgykódot generál. A különböző fordítási egységekből előálló tárgykódot a bináriszerkesztő (*linker*) építi végfelhasználó által is futtatható állománnyá.

Problémák a klasszikus modellben

Nincs megkötés arra nézve, hogy milyen kódszöveg szerepelhet egy fordítási egységben, amíg az szemantikusan helyes C++ programot eredményez – egy sor alkotóelemei pl. több különböző fájlból érkező makrókon keresztüli behelyettesítéssel is összeállhatnak. Más programozási nyelvekkel – pl. Java esetében nagyon szigorú szabályokat ír elő a fordító rendszer – ellentétben nincs megkötés arra, hogy miként szervezzék a programegységeket. A *fejlécállományok* szükségessége a nyelv elődjeit képező nyelvek időszakából hagyaték: a különálló fordítás miatt a programegységek közötti interfészek megteremtésére a fejlécállományok szolgáltak, például egy felhasználói típust használó összes fordítási egységben ismerni kell a típusnak a definícióját, így egy közös fejlécállományt mindegyikben beillesztve ezt a fordító elvégzi automatikusan, míg a programozók által kezelt kódszövegben a definíciót csak egy helyen kell leírni.

A szövegszerű beillesztéssel és a különálló fordítással azonban számos probléma akad: az egyik legkritikusabb a névütközések vagy felüldefiniálások esete, mely során preprocesszor makrók – vagy egyes szimbólumnevek – jelentése a fordítási egységek között különböző lehet, ezzel nehezévé téve a projekt fejlesztését és megértését. Bizonyos esetekben előfordulhat, hogy az eddig jól működő program egy módosítást követően akár rejtett módon elromlik, akár csak a fejlécállományok sorrendjének megváltoztatásától. Bár ezek a hibák kiküszöbölhetőek statikus analízis eszközökkel vagy teszteléssel, súlyosabb probléma, hogy bármilyen szimbólum ami a fordítás során deklarálásra kerül a fordítási egységben, azt többet onnan eltávolítani nem lehet. Így azon integrált fejlesztői környezetekben dolgozó szoftverfejlesztők, akiknél az *IDE* a kódszöveget aktívan értelmezi, felajánlasként megkaphatnak olyan belsőséges, klienskódban használatra ellenjavalt – ún. implementációs

részlet – szimbólumokat, amelyeket minden jelzés vagy rossz viselkedés nélkül felhasználhatnak, ezzel újfent kiteve a projektet egy későbbi megváltozás esetén „mágikus” hibák előtörésének.

A Bevezetőben említésre került a fordítóprogram teljesítmény- és időigénye. Ennek egy szignifikáns része a klasszikus fordítási modellből fakad: a fejlécállományokat minden fordítóprogram processzusnak külön-külön fel kell olvasnia a háttértárról, értelmezni a tartalmukat, a fordítási egységből kódot generálni. Ennek a kódgenerálásnak legnagyobb részét a C++ generikusok által lehetővé tett fordítási idejű programozás (*template metaprogramming*) teszi ki. Amennyiben egy generikust több fordítási egységben is példányosítunk, az összeszerkesztett végbinárisban csak egy példány marad meg, így látszólag az esetek nagy többségében „felesleges” munkát végez a fordító (Mihalicza).

Modules

A klasszikus modell számos problémájára megoldást kínál az először 2004-ben felvetett, és a nyelvi lehetőségek közé a C++ 2020-as kiadású változatába elfogadott *Modules* (Smith, 2018) rendszer. A fő cél jobban elszeparálható és kezelhető, logikailag szervezett programkönyvtárak készítését lehetővé tegye. A programkönyvtárat felhasználó kliens kód fejlesztőjének nem szükséges azzal foglalkoznia, hogy milyen fejlécállományokra van szüksége. A felhasználást egy új utasítás, az ***import ModulNeve;*** vezeti be, a pontos összekapcsolás megvalósítása pedig a fordítóprogram és a fordítást vezérlő keretrendszer feladata.

A modulok expliciten meghatározzák a saját interfészüket amit a kliensek láthatnak – olyan szimbólumok (típusok, függvények, ...) amelyek nem részei ennek az exportált interfésznek úgy viselkednek a kliensben, mintha sosem lettek volna definiálva. A preprocesszor direktívák hatókörét a modulból az új nyelvi szabályok értelmében nem engedi ki a fordító, így elkerülhetők azok a problémák, amelyek a fejlécállományokból „kiszökő” definíciók – akár makró, akár szimbólum – lehetősége okoz.

Az *import* utasítás konkrét viselkedésének kialakítása a szabványszöveg értelmében a fordítóprogram és fordítási rendszer fejlesztőire van bízva. Számos optimalizációs lehetőséget meg kell majd a közösségnek vizsgálnia ezen a ponton, ugyanis a modulrendszer csak a bemutatott problémák egy részét oldja meg. A számomra legfontosabb nyitott kérdés a

fordításhoz szükséges idő, melyre a modulrendszer úgy tűnik, inkább káros hatással van. Ezt a megállapítást nagy C++ kódbázissal rendelkező cégek – pl. Google – már a C++20 funkcióinak első megbeszéléseikor felvetették (Klimek, 2016).

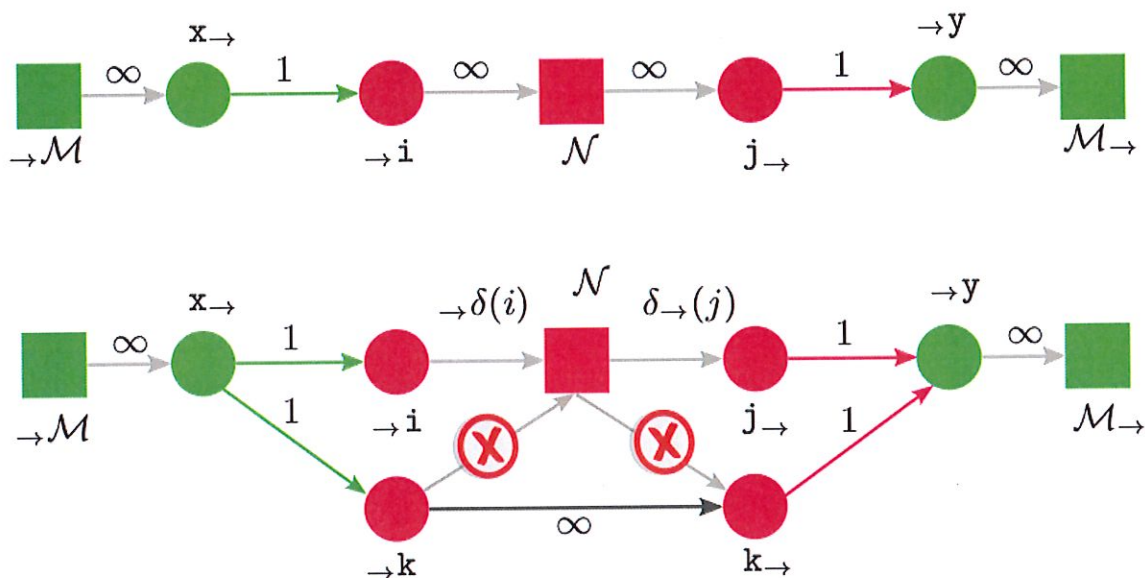
Ezekről a kérdésekről eltekintve azonban fontos kiemelni, hogy több száz milliárd sornyi C++ kód létezik a világon különböző rendszerekben, amelyeket évek-évtizedek folyamán a „klasszikus” – korábban bemutatott – fordítási modell alapján készítettek el. A modulrendszerre áttérés nem triviális lépések megtételét igényli, melyet a kutatásunk során elkészített módszerrel próbálunk meg könnyen lehetővé tenni.

Automatikus modularizáció módszere

Az automatikus modularizáció egy klaszterizáció segítségével történik: a projekt forrásfájljait modulegységekhez rendeli, majd az így előálló egységeket modulként fordítjuk, és a klienskódban felhasználjuk. Ehhez a módszerünk több lépést végez el:

1. Kezdeti bementként szükséges egy „durva klaszterizáció”, amely kezdetben a fordítási egységeket egy „modulhoz” rendeli. Ez nem szükséges, hogy egy lefordítható modul-hozzárendelés legyen.
2. A tényleges projekt mély analízise és tényállítások kiemelése – A projekt fordítási információ alapján számos belső kapcsolatot kigyűjtünk, melyek szükségesek a modularizáció szempontjából fontos függőségi gráfok előállításához. A kutatás során megalkottunk egy modellt, amely a nyelvi szabályok alapján leírja azokat a központi programtulajdonságokat, amelyek mentén a modulok közötti átszervezést végre kell hajtani.
3. A függőségi gráf felépítése után abban egyszerű köröket keresünk a kezdeti modulok között. Mivel a modulok egymástól való függése a fájlrendszer szintjén megjelenik, így nem lehetséges a C/C++ világból a fejlécállományok körkörös függését feloldó *header guardok* használata. Az egymástól körkörösén függő modulok köreit feldaraboljuk, megvizsgálva, hogy pontosan mely fejlécállományok okozzák a körkörös függést. Ezt folyamalgoritmusok segítségével végezzük (Cormen, Leiserson, Rivest, & Stein, 2009), a függőségi gráf előállításának szabályát az alábbi ábra szemlélteti, ahol az M, N, ... négyzetek modulokat jelölnek, x, y, i, j, k, ... körök pedig a fájlokat. A nyilak arra utalnak, hogy a függőségi körben – melyet determinisztikusan elvágunk egy adott modulnál – a fájl és a modul a függés okozója,

vagy „elszenvedője”. A készített gráfban elindítunk egy folyamatot, és a minimális vágás mentén illesztett fájlokat külön modulba szervezzük.



4. Az így kapott modulok az interfészük szintjén már nem függnek körkörös egymástól, azonban ellenben a klasszikus modellel, ahol például a típusok definíciói a fejlécben, az implementáció pedig egy fordítási egységbe került, a modulok esetében a kettőnek ha nem is egy fájlban, de mindenképpen egy modulban kell szerepelnie. Így egy lépés az implementációs egységek átmozgatása az interfész egység moduljába. Ez viszont újabb körkörös függéseket eredményezhet a modulok között.
5. Az így keletkező körkörös függéseket átszervezésekkel és modulok összevonásával oldjuk fel. Az összevonás során egy új modul keletkezik, amely magában foglalja a két „szülőmodulnak” az összes hozzá rendelt állományát.

Az algoritmus eredménye az elkészített modul-hozzárendelés. A projekt fájljait az elkészített prototípus implementációnk átírja úgy, hogy a kiszámított modulokat használják a nyers `#include` direktívák helyett. Az algoritmus futásideje a gyakorlatban elenyésző, a legnagyobb részét a kód analízise teszi ki, amelynek körülbelül fordítási idővel megegyező időigénye van.

Eredmények

A módszert és annak implementációját több nyílt forráskódú projekten is teszteltük. A legfontosabbak ezek közül a *Vim* (Community, dátum nélk.), az *Apache Xerces* (Foundation,

1999), és az *LLVM/Clang* (Project, 2003) C++ fordító keretrendszer voltak. Azt tapasztaltuk, hogy a bemutatott modularizáció – amelyet a projektben található fájlok szintjén, így a projekt belső logikai, szimbólumok szintjén definiált szervezésére hatás nélkül végeztünk – nem volt képes a projektet érdemben felbontani. A fájlok összesen egy, néha két nagy modulba estek össze, a szoftverprojekt készítői által kialakított határmezsgyék a végbinárisok – futtatható állományok és alkalmazáskönyvtárak – között pedig legtöbbször egy szigorú határt jelentettek a modularizációban is, vagy egyes modulok több binárishoz is elkezdtek illeszkedni.

Ennek megfelelően a végső következtetés, hogy a modularizáció automatikus, a szoftverprojekt belső szervezettségét a nyelvi szabályokon túlmenően nem ismerő és azt nem megváltoztató módszere nem ad jól használható eredményt. Szükséges tehát a projekten dolgozó szoftvertervezők és –fejlesztők energiabefektetése abba, hogy a modularizációt egy meglévő projektre kivitelezzék. Ez azonban szükségszerűen maga után fogja vonni a projekt külső és belső interfészének átalakítását, mely okán a terméket használó klienskódokban is változásokat kell bevezetni. Ezen a ponton félok, hogy a modularizáció jelenleg még csak egy nyelvi újítás és álom marad, és még sok évig párhuzamos világban fognak élni a C++ fejlesztők, mivel a használt alkalmazáskönyvtáraik egy – tetemes méretű – része a régi, klasszikus modellt fogja továbbra is használni, és csak a most újonnan elkészített projektek készülnek *Modules*-elven írva, ha egyáltalán.

Publikálási terv

A kutatás eredményeit prezentáltam az *STCS (Conference on Software Technology and Cyber Security) 2019* konferencián Szombathelyen 2019. februárjában. A konferenciakiadványban egy publikáció fog ebből megjelenni. A kutatási munka ipari jelentőségét és időszerűségét alátámasztandó, hogy a *C++Now* konferencián teljes hosszúságú (90 perc) előadást tartottam Aspen, Colorado állam városában. Ez a legnagyobb C++ konferencia, amelyen részt vesz a C++ világ krémje, a nyelvi szabályokat tartalmazó szabványért felelős bizottság tagjai, és így azok is, akik ilyen folyóiratcikket bírálnak.

A kutatást összefoglaló művet az *Informatikai Kar Tudományos Diákköri Konferenciáján* előadtam. A matematikai modell, a problémák leírása és a megoldási módszerről folyóiratcikk készült, melyet a *Software – Practice & Experience* szakfolyóiratba küldtünk. Jelen összefoglaló dolgozat készítésének pillanatában a mű bírálata folyamatban van.

Irodalomjegyzék

- Community, T. V. (dátum nélk.). *Vim: the ubiquitous text editor*. Forrás: <http://vim.org>
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press. Forrás: mitpress.mit.edu/books/introduction-algorithms
- Dayani-Fard, H., Yu, Y., Mylopoulos, J., & Andritsos, P. (2005. 4 04). Improving the Build Architecture of Legacy C/C++ Software Systems. *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, (old.: 96-110). doi:10.1007/978-3-540-31984-98
- Dos Reis, G., Hall, M., & Nishanov, G. (2016. 2 15). *N4720 -- A Module System for C++*. Tech. rep., ISO/IEC JTC1/SC22/WG21 - Open Papers. Forrás: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0142r0.pdf>
- Ernst, M. D., Badros, G. J., & Notkin, D. (2002. 12). An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.*, 28, 1146-1170. doi:10.1109/TSE.2002.1158288
- Foundation, T. A. (1999). *Xerces: A processor for parsing, validating, serializing and manipulating XML*. Forrás: <http://xerces.apache.org/>
- ISO/IEC JTC 1/SC, 2. (2017. 12). *ISO/IEC 14882:2017 Information technology --- Programming languages --- C++, version 17 (C++17)*. Geneva, Switzerland: International Organization for Standardization. Forrás: iso.org/standard/68564.html
- Johnson, D. B. (1975. 3). Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4, 77-84.
- Keith, M. J., & Martin, M. C. (1994. 4 07). Genetic programming in C++: Implementation issues. *Advances in genetic programming*, 1, 285-310.
- Klimek, M. (2016. 9 18). Deploying C++ Modules to 100s of Millions of Lines of Code. Bellevue, Washington, United States of America. Forrás: <https://www.youtube.com/watch?v=dHFNpBfemDI>

- Krishnaswamy, T. (2000. 1). Automatic Precompiled Headers: Speeding up C++ Application Build Times. *Proceedings of the First Workshop on Industrial Experiences with Systems Software, WIESS 2000, October 22, 2000, San Diego, CA, USA*, (old.: 57-66).
Forrás: <http://www.usenix.org/events/wiess2000/krishnaswamy.html>
- Litman, A. (1993. 3). An implementation of precompiled headers. *Software: Practice and Experience*, 23, 341-350. doi:10.1002/spe.4380230308
- Ltd., C. S. (2018. 6 14). zapCC -- A (Much) Faster C++ Compiler. Forrás: <http://zapcc.com>
- Malloy, B. A., & Power, J. F. (2017. 11). Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, (old.: 314-323). doi:10.1109/ESEM.2017.45
- Martin, K., & Hoffman, B. (2015. 1 16). *Mastering CMake: a cross-platform build system* (3.1. kiad.). Kitware Inc.
- Mihalicza, J. (dátum nélk.). How #s Affect Build Time in Large Systems. *Proceedings of the 8th International Conference on Applied Informatics (ICAI)*, (old.: 343-350). Forrás: icai.ektf.hu/pdf/ICAI2010-vol2-pp343-350.pdf
- One-definition rule. (dátum nélk.).
- Oracle. (dátum nélk.). The Template Repository.
- Phases of translation. (dátum nélk.).
- Project, T. L. (2003). *Clang: C Language Family Frontend for LLVM*. Forrás: <http://clang.llvm.org>
- Smith, R. (2018. 11 26). *P1103r2-- Merging Modules*. Tech. rep., JTC1/SC22/WG21 - Papers Mailing List. Forrás: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1103r2.pdf>
- Stroustrup, B. (2004. 4 12). *#scope: A simple scoping mechanism for the C/C++ preprocessor*. Tech. rep., ISO/IEC JTC1/SC22/WG21 - Open Papers. Forrás: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1614.pdf>

Turner, Z. (2019. 3 8). Host is now dependency free. Forrás:
<http://lists.llvm.org/pipermail/lldb-dev/2019-March/014846.html>

Whisperity. (2018. 10 8). Add experimental support for C++ Modules TS if the compiler supports it. Forrás: https://gitlab.kitware.com/cmake/cmake/merge_requests/2482

Yu, Y., Dayani-Fard, H., & Mylopoulos, J. (2003. 10 06). Removing false code dependencies to speedup software build processes. *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research, October 6-9, 2003, Toronto, Ontario, Canada*, (old.: 343-352). Forrás: <https://dl.acm.org/citation.cfm?id=961375>

Mellékletek

1. Összegzés

Kar: ELTE Informatikai Képzési szint: mesterképzés ÚNKP 2017/18 „A” keret
Kar (MSc)
Név: Szalay Richárd Kutatás címe: A C++ programozási nyelv moduláris
fordíthatóságának kutatási kérdései

A programozási nyelvek hosszú ideje támogatják a programegységek logikai hierarchiába szervezését, melyek mentén számos fordítóprogram optimalizáció végezhető el, amely mentén a kódot szervezik az adott nyelven fejlesztők. A mindennapi életben elterjedt programozási nyelvek közül sajátos a C++ két okból: az egyik, hogy a szoftverprojektek elemeinek logikai szervezésére nagyon kevés jó módszert biztosított (a szöveges beillesztésen alapuló fejlécállományoktól eltekintve), valamint a fejlesztés során a fordítással töltött időben is kiemelkedően rosszul teljesített. A C++ programozási nyelv legújabb kiadása (C++20) tartalmazza az új nyelvi elemek között a *Modules*-t, amely a programegységek logikai szervezését teszi lehetővé a korábbi fejlécállományos (header) rendszerhez képest. A szabványkiegészítés 2004 óta volt terítéken és a C++ közösség előtt, mely során számos ellenjavaslatot és diszkussziót élt meg.

Azonban a nyelvi szabvány („jogszabály”) támogatása egy ilyen funkcióhoz nem elégséges a tökéletes kivitelezés érdekében, mivel azt a fordítóprogramoknak is támogatnia kell, méghozzá olyan minőségben, hogy a korábban említett fordítási idő legalább ne romoljon el a „hagyományos” eljáráshoz képest – tekintve, hogy több tíz, vagy akár néhány százmilliárd sornyi (LoC) programkódbázis épült fel az évtizedek alatt C++ nyelven írva.

Ennek megfelelően – és a hosszú fordítási időkből adódó mély személyes motivációm kapcsán – a kutatásom során megvizsgáltam, milyen eszközök állnak rendelkezésre a C++ projektek fordítási idejének leküzdésére (ccache, distcc, BazelBuild, zapcc, ...) és miként lehet a modularizációt elősegíteni meglévő projekteken.

A kutatás során témavezetőmmel kialakítottunk egy módszert, amely folyamalgoritmusokkal modulokba szervezi a projektet. A kimenet megfelelő fordítói környezet támogatás esetén integrálható a projektbe. A modularizáció során ún. „unity build”-et készítünk a projekt egyes részeiből, amely során meg kell küzdenünk az esetleges ütközések feloldásával.

Név: Szalay Richárd

ÚNKP 2018/19

Kar: ELTE Informatikai Kar

Képzési szint: mesterképzés (MSc)

„A” keret

Kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései

Tapasztalataink azt mutatták, hogy pusztán a fordítási egységek (forrásfájlok) szintjén történő klaszterizáció nem elég egy – valamilyen metrika szerint – hasznos modularizáció eléréséhez, azaz szükségszerűen a szoftverprojekt belsejét ismerő fejlesztők időbefektetése és munkája, valamint a belső és külső interfész megtörése szükséges.



2. Az ÚNKP kapcsán konferenciaszereplések és publikációk megjelenésének felsorolása (MTMT-ből kinyert lista, ha van)

- Konferenciaszereplések:
 - o STCS-2019: (Conference on Software Technology and Cyber Security), 2019. február 21-23., Szombathely, Magyarország
 - o C++Now: 2019. május 5-10, Aspen, Colorado, Amerikai Egyesült Államok
- Publikációs lehetőségek, melyek **jelenleg még elbírálás alatt** állnak.
 - o Az STCS konferenciáról készülő CEUR-WS konferenciakiadványban egy publikáció.
 - o Folyóiratcikk a Wiley & Sons: Software Practice & Experience folyóiratban.
- Megjelent – idézhető – publikáció a dolgozat benyújtásakor még nem áll rendelkezésre.

3. Konferencia részvétel, kutatóút eredményeinek bemutatása

- STCS-2019 konferencia: részben dologi keretből – a utazáshoz a **vonatjegy dologi keretből**, a szállásfoglalás központilag, **nem** dologi keretből. A konferenciának egyéb költsége, pl. regisztrációs díja nem volt.
- C++Now 2019 konferencia.

Név: Szalay Richárd ÚNKP 2018/19
 Kar: ELTE Informatikai Kar Képzési szint: mesterképzés (MSc) „A” keret
 Kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései

Az ÚNKP 2018/2019. dologi keretéből megvalósult utazás beszámolója

Név:	Szalay Richárd
az utazás jellege, <u>kérjük aláhúzni</u>	<i>tanulmányút / kutatóút / konferencia részvétel</i>
úti cél(ok)	Szombathely, Magyarország
pontos dátum (indulás és érkezés napja)	2019. február 21 – 23.
az ÚNKP dologi keretének terhére <u>elkölthött összeg</u> / Önköltséges	12 920 HUF
a kutatás, konferencia helyszíne (i)	ELTE Savaria Egyetemi Központ, Szombathely, Magyarország
szakmai kapcsolatfelvétel	Simon J. Thompson, University of Kent, United Kingdom
Az Informatikai Kar a 2018/19-es tanév hallgatói kutatásainak összefoglalása és publikálása céljából konferenciát szervezett az ELTE nemrég hozzáolvasztott szombathelyi kampuszán. A rendezvényen nagy többségben ELTE-s hallgatók és doktoranduszok adták elő jelenlegi kutatásaiknak eredményét, ezen felül külföldi cserediákok, BME, PPKE és OE hallgatók is jelen voltak. A konferencián az ÚNKP kutatásom eredményeinek akkori állapotát mutattam be, az előadásban kihangsúlyozva a probléma fontosságát és a megoldási módszer vázlatát. Nagyon sok érdekes kérdést kaptam főleg a fent említett professzor úrtól, aki hasonló problémákat kerülgetett már néhány éve Erlang környezetben. Több más ELTE-s hallgató saját témájának előadása is rendkívül érdekes volt, és segített, hogy még a doktori képzés megkezdése előtt lehetőségem legyen megismerni leendő tanszéke, az ELTE IK Programozási Nyelvek és Fordítóprogramok tanszék által folytatott munkát, amelynek eddig csak a C++ specifikus részét láttam témavezetőmön, Dr. Porkoláb Zoltánon keresztül. Thompson professzor úrral azóta is tartjuk e-mailben a kapcsolatot, számos érdekes publikációt küldtünk már egymásnak korábbi konferenciákról.	

Név: Szalay Richárd

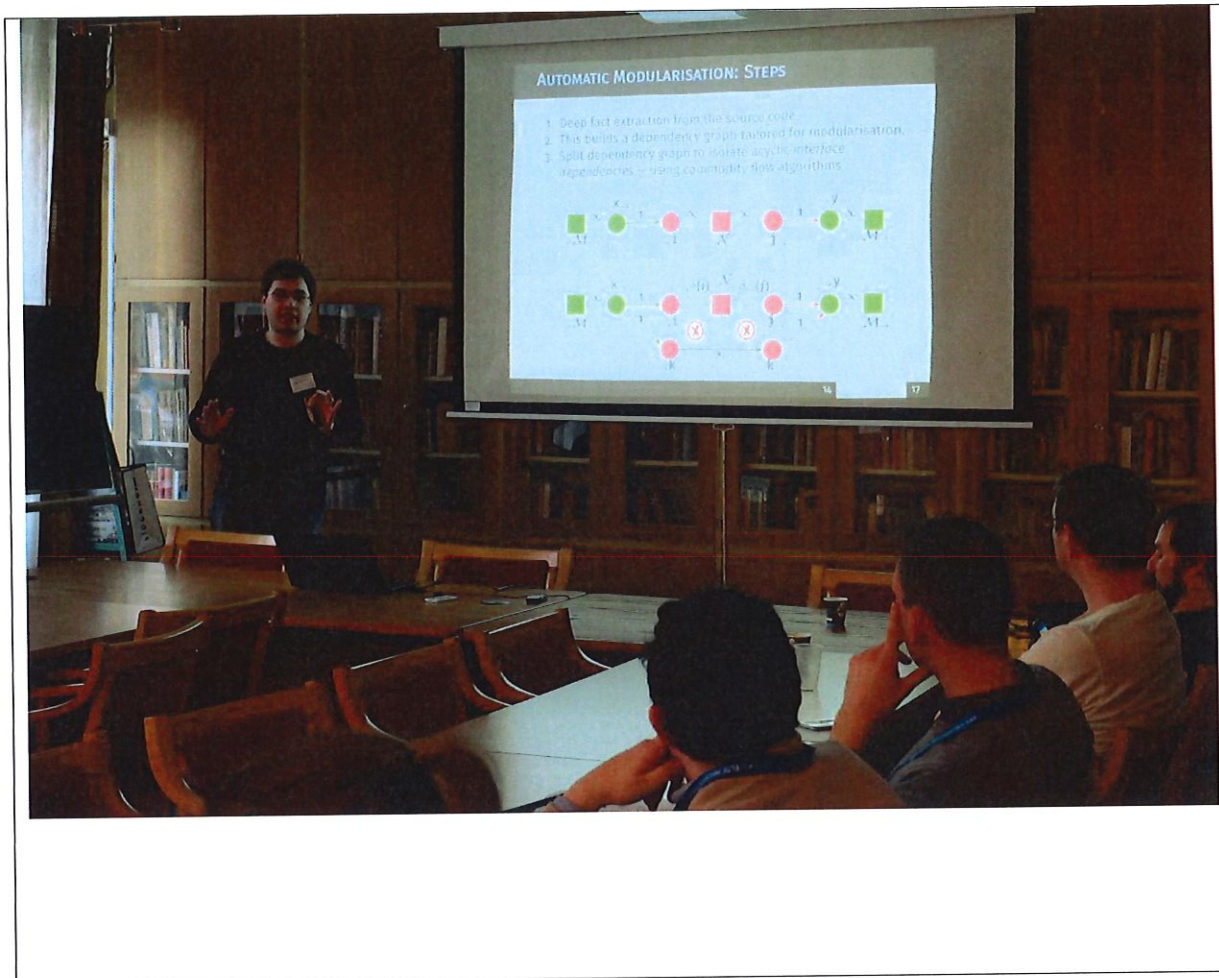
Kar: ELTE Informatikai Kar

Képzési szint: mesterképzés (MSc)

ÚNKP 2018/19

„A” keret

Kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései



Szalay Z

Pecsi Zoltan

Név: Szalay Richárd
Kar: ELTE Informatikai Kar Képzési szint: mesterképzés (MSc) „A” keret
Kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései

ÚNKP 2018/19

Az ÚNKP 2018/2019. dologi keretéből megvalósult utazás beszámolója

Név:	Szalay Richárd
az utazás jellege, <u>kérjük aláhúzni</u>	<i>tanulmányút / kutatóút / konferencia részvétel</i>
úti cél(ok)	Aspen, Colorado, Amerikai Egyesült Államok
pontos dátum (indulás és érkezés napja)	2019. május 3 – 13.
az ÚNKP dologi keretének terhére elköltött összeg / Önköltséges	Önköltséges
a kutatás, konferencia helyszíne (i)	Aspen Center for Physics, Aspen, CO USA
szakmai kapcsolatfelvétel	C++ szabványbizottsági tagok: Marshall Clow, Bryce Lebach Számos C++-ban érdekelt cég (Bloomberg, nVidia, Microsoft, stb.) tagjai, pl. John Lakos, Doug Buttler.
A konferenciát évente rendezik meg a C++-t (és nem C++-szal) fejlesztők – úgy mint szabványbizottsági tagok, a legnagyobb alkalmazáskönyvtárak (STL, Boost) fejlesztői – egy kis (legfeljebb 140 fős) közösségének. A konferencia fő témája nem a nemsokára kiadásra kerülő C++20 szabvány, hanem az a következő (C++2y, várhatóan C++23) változatba bekerülni szándékozott elemek – természetesen azonban sokan előadják az elfogadásra került funkciók vagy segédkönyvtárak történetét is. A kis létszám miatt egy nagyon közvetlen hangulatú konferencia alakul ki, ahol mindenki ismer mindenkit – na persze én újoncként nem ismertem senkit. Ez természetesen hamar megváltozott, a 1,5 órás előadás szekciók közötti szünetekben folyamatosan az elhangzott előadásokról és új ötletekről beszélgettünk. A teljesség igénye nélkül az általam meghallgatott más előadások között szerepelt: reflection, fordítás idejű reguláris kifejezések (véges determinisztikus automata kiértékelés), hangkártya-vezérlés az STL-ben, modern C++-ban írt rossz kódok kihasználása, párhuzamos végrehajtási vezérlők. Részt vettem továbbá két előadáson, mely a modern C++ egyetemi környezetben való tanításának tapasztalatait, nehézségeit mutatja be az előadó oktató szemszögéből. Sajnos a saját előadásom nem a legszerencsésebb időpontra került, így a konferencia résztvevők egy része már hazament, de a saját blokkomban szereplő három előadás közül	

még ígyis körülbelül 30-40 fő jött el. Számos hozzászólás érkezett a témához, bár a kutatás maga alapvetően „negatív eredményt” mutatott be, többen örültek annak, hogy nem féltünk a negatív eredmény bemutatásától.

Meghívást kaptam a szabványbizottság inkubátor osztályának „Tooling and Developer Infrastructure” tanulmányi csoportjába, hogy résztvegyek a videókonferenciákon.

Összességében úgy érzem, hogy a konferencián való részvétel egy nagyon fontos sarokköve volt annak, hogy saját magam és az ELTE IK PNYF tanszék élvonalas tagja lehessen a C++ nyelv fejlődését szervező közösségnek. Remélem, hogy többször részt vehetek még ezen a konferencián.



Szalay Z

PhdLab 2018

Név: Szalay Richárd

ÚNKP 2018/19

Kar: ELTE Informatikai Kar

Képzési szint: mesterképzés (MSc)

„A” keret

Kutatás címe: A C++ programozási nyelv moduláris fordíthatóságának kutatási kérdései

4. Csatolmányok listája

- Egyéb csatolmányok nincsenek.