Tallinna Ülikool
Informaatika Instituut

C++11 ja C++03 spetsifikatsioonide võrdlus reaalajas töötava rakenduse loomise abil

Seminaritöö

Autor: Roman Gorislavski
Juhendaja: Inga Petuhhov

Tallinn 2012

Sisukord

Sissejuhatus

C++ spetsifikatsiooni võrdlev teema arenes välja autori huvist C++ keele vastu. Samuti on näha, et Eestis puudub antud teema kohta materjal, kuna teema on uus.

Inimesed elavad kiiresti muutuvas maailmas. See ei mõjuta ainult inimesi, vaid mõjutab ka programmeerimise keeli. See keel, mis ei kohane muudatustega, takerdub ja sureb välja kasutatavuse vaatepunktist. Enim kasutatavatest keeltest on C++ üks vanemaid. Põhjus, miks ta pole koos teiste vanemate keeltega kasutusest kadunud, on tõenäoliselt see, et ta on püsinud ajakohasena. See keel arendati 1979. aastal Stroustrupi poolt (AT&T B. L. Stroustrup, 1991) ja peaaegu 20 aastat hiljem see standardiseeriti (ISO/IEC, 1998). Viis aastat hiljem uuendati standard enamasti veaparanduste näol (ISO/IEC, 2003). 2011. aasta septembris uuendati jälle spetsifikatsioon (ISO/IEC, 2011), kuid seekord sisaldas see ka drastilisi muudatusi nagu lambda funktsioonid, `auto` lisamine ja palju muud. Stroustrup ütles:

"See tundub nagu uus keel" (B. Stroustrup, 2012). Järgmist spetsifikatsiooni autor ei ootaks enne 2016. aastat, aga ISO Komitee on juba jätkanud oma tööd C++ edasiarendamises ja praeguse versioonis olevate vigade parandamist.

Autorit huvitab eelkõige küsimus, kui palju keegi saab C++11 spetsifikatsiooniga tulnud laiendusi tegelikult kasutada. Kõige paremini saab sellele vastuse läbi praktilise kogemuse, luues rakenduse, mis võiks vastata kiiruse aspektist kaasaaegsetele nõudmistele ehk oleks edasiarendatav. C++ kasutatakse eelkõige rakendustel, kus rakenduse kiirus on äärmiselt tähtis või kus on limiteeritud ressurss. Seetõttu oleks sobilik luua reaalajas töötav rakendus. Teine aspekt oleks vananenud API kasutusele võtmine, et tuua nähtavale, kui hästi saab siduda kaasaaegset koodi vanaga, kuna uusi teeke C++ keelele igapäevaselt ei kerki ja tihti on arendaja sunnitud leppima C-stiili teekidega. Küsimus ei ole samas niivõrd rakenduse koheses loomises C++11 laiendustega, vaid C++11 laienduste rakendamises olemasoleva koodi uuendamisel. Samuti on küsimus selles, kuidas see muutus mõjutab koodi loetavust kiirust ja mälu.

Järgnevalt räägib autor sellest, kuidas struktuur välja hakkab nägema. Tuleb arvestada, et C++11 laienduste või uuendamise all mõeldakse võrdlust C++03 ja C++11 vahel, kus C++03

standardis see osa puudub või omab poolvalmis kuju. Sarnaselt hakatakse ka edaspidi nimetama standardeid lühendite järgi: C++03 (ISO/IEC, 2003) ja C++11 (ISO/IEC, 2011).

1 Kriteeriumite määramine

Selle seminaritöö kriteeriumid määratakse autori poolt siin peatükis, kuid kasutusse jõuavad nad järgnevas kolmes peatükis. Antud kriteeriumid määravad, kuidas saavad rakendatud järgnevate peatükkide struktuurne ja sisuline pool. Samuti määratakse, kuidas kasutatakse selles töös tabeleid, jooniseid ja koodinäited. Lisaks sellele määratakse ka rakenduse klassid.

1.1 C++11 laiendused

Kolm peatükki on omavahel eraldatud. Esimene käsitleb keeleomadusi ja teine teeke. Lisaks sellele käsitleb teine peatükk eelnevalt mainitud uute keeleomadust rakendamist rakenduses ning kolmas peatükk annab üldpildi muutustest rakenduses. Esimesed kaks peatükki jagunevad alapeatükkideks. Alapeatükid tegelevad ühe kindla C++ keele laiendusega.

Iga alapeatükk jaguneb viieks lõiguks samas järjekorras:

- Esimeses lõigus toimub alapeatüki või mõiste lahtiseletamine, et mõista, millest hakatakse selles alapeatükis rääkima.
- Teine lõik näitab, mis probleeme antud lahendus lahendab. Tihti on probleeme ka rohkem kui üks. Enamik neist olid mainitud juba C++ standardile uute lahenduste ettepanekutes ja need on ka vastavalt viidatud. Viite puudumisel on tegemist autori poolt C++03 ja C++11 võrdlusest järeldatud probleemiga. Autoripoolne väljatoodud probleem ei pruugi väljendada standardikomitee põhiprobleemi, miks antud laiendus keelele kasutusele võeti.
- Kolmas lõik räägib uuenduse rakendamisest ja läheneb uuele laiendusele praktilisemast poolest võrreldes eelmiste lõikudega. Sarnaselt sellele või isegi rohkemal määral lähenevad järgnevad lõigud uuele laiendusele praktilisemalt. Antud lõik seletab, kuidas on võimalik uusi laiendusi rakendada. Siinkohal ei pruugi olla välja kirjutatud kõik rakendamise viisid, vaid ainult need, mida autor pidas vajalikuks mainida. Enamasti olid need viisid kas otseselt seotud programmis kasutamisega või olid tihedalt seotud laiendusega, mida autor oma programmis kasutusele võttis.
- Neljandas lõigus räägib autor, kuidas ta rakendas eelnevalt mainitud laiendusi oma programmi koodis. Samuti toob ta välja sellega seonduva ajakulu. Ajakulu määramine toimus umbkaudselt ja sellega tahtis autor näidata, kui raske võib olla neid laiendusi

rakendada. Antud tulemusi korrates kellegi teise poolt tulevad ka teised tulemused. Ajakulu sõltub programmeerija taustast, tema kogemusest C++ kasutamisel, koodikirjutamise kiirusest ja paljust muust sarnasest. Samuti kasutas autor mõõtmiseks tavalist kella, aga ei teinud seda minutilise täpsusega 10 minutist pikemate mõõtmiste korral, kuna need sisaldasid kohvipausi ja muid sarnaseid tegevusi.

- Viies lõik tegeleb kiiruse ja mälu mõõtmisega. Kasutatava mälu mõõtmiseks kasutati `sizeof` funktsiooni. See määras ära objektide mahtu. Kiiruse mõõtmiseks kasutati `MyCounter` klassi, mis toetub `QueryPerformanceCounter` funktsioonile. Kiirust mõõdeti selle klassi `StartCounter` meetodi kutsumisest kuni `GetCounter` meetodi kutsumiseni, mille vahele jäi mõõdetav osa nagu näiteks funktsioonid, lihtne objekti kopeerimine või midagi muud sarnast. Täpseimaks ühikuks, mida `MyCounter` klass suutis mõõta, oli umbes 0,41 mikrosekundit, kuigi tuleb arvestada, et juba mitte millegi mõõtmine võttis keskmiselt aega 1,21 mikrosekundit. Võimalusel mõõdeti aegu rakendusesiseselt. Leidus ka laiendusi, mis ei olnud kasutuses antud rakenduses või siis olid rakenduses kasutusel, kuid seda liiga kõikuvate väärtustega, sest neist liikus läbi liialt varieeruv info. Sellisel juhul mõõdeti neid laiendusi spetsiaalses testalas, mis oli otseselt eraldatud ülejäänud programmist.

Testimise seadistused:

- Mälu mõõtmisel piisas ühest katsest. Kiiruse testimisel tehti enamasti 10 katset. Kui autori arvates polnud tulemus veel piisavalt selge, tehti veel täiendavad 30 katset.
- Testmasinaks oli raal Windows 7 64-bitise operatsioonisüsteemiga, mille mäluks oli 4 GB ja protsessoriks oli kahetuumaline Intel T9400, mille kumbagi protsessorituuma taktsageduseks oli 2.53GHz.
- Integreeritud arenduskeskkonnaks oli Visual Studio 2012 RC. Kompilaatoriks oli VC11. Kompilaatori seadistuseks kiiruse mõõtmise ajal olid: */Yu"stdafx.h" /FR"x64\Debug\" /GS /GL /W0 /Zc:wchar_t /Zi /Gm /Ox /sdl- /Fd"x64\Debug\vc110.pdb" /fp:precise /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D "_UNICODE" /D "UNICODE" /D "_ATL_STATIC_REGISTRY" /fp:except /errorReport:queue /WX- /Zc:forScope /Gd /MDd /Fa"x64\Debug\" /EHsc /nologo /Fo"x64\Debug\" /Fp"x64\Debug\SeminaritooVer1.pch"*.

Tabelite, jooniste ja koodinäidete kasutamine:

- Kõik koodinäited asuvad teksti sees, välja arvatud rakenduse kogukood, mis asub Lisa 2 all. Koodinäited on võimalusel võetud rakendusest, kuid sobiva näite puudumisel võis autor võtta koodi ka mujalt. Koodinäited kasutavad teist fonti ja kasutavad kergemaks lugemiseks erinevaid helendeid tekstivärvis.
- Kõik tabelid asuvad Lisa 1 all. Need sisaldavad tulemusi, mis on võrreldavate objektide vahel võrdlemisi muutumatud. Samuti sisalduvad siin ka objektide mahu mõõtmise tulemused.
- Kõik joonised asuvad teksti sees. Jooniste olemasolu korral puudub vastav tabel, kuna tabel oleks dubleerinud joonist. Joonis võeti kasutusele juhul, kui kahe või enama objekti võrdlemisel oli tulemus erinev.

1.2 Rakendus

Töö eesmärgi saavutamiseks autor loob rakendust, arvestades sissejuhatuses määratud nõudeid. Selle rakenduse põhiülesandeks on kuvada ekraanile reaalajas Kinect-seadmest tulenevad pilte eraldi operatsioonisüsteemi aknas.

Sellest tulenevalt peaks sisaldama vähemalt järgnevaid klasse:

- akna loomise ja haldamise eest vastutav klass;
- klass, mis haldab Direct2D kasutamist;
- klass, mis haldab Kinect-seadme kasutamist;
- klass, mis haldab Kinect-seadmest tulenevaid pildiandmeid;
- klass, mis haldab nuppude loomist;
- klass, mis haldab nuppude väljanägemist;
- klass, mis haldab ajamõõtmist;
- klass, mis loeb faile;
- klass, mis aitab faili andmete töötlemisel;
- klass, mis hoiab töödeldud andmeid.

2 Süntaks

2.1 Tühi viit: `nullptr`

Tühi viit (null pointer) on literaal, mida kasutatakse näitamaks, et viit on tühi ja algväärtustatud. Tühi viit on `nullptr_t` tüüpi puhas parempoolne väärtus (*prvalue*) (ISO/IEC, 2011).

See uuendus kaasnes järgnevate probleemide lahendamiseks:

- Puudus korrektne viis tühi viida eristamiseks nullist. Antud probleem raskendas funktsioonide ülelaadimist, mis sisaldasid viita ja täisarvu (Shutter & Stroustrup, 2007).
- Tühi viidal puudus oma literaal. Programmeerija silmis `NULL` makro tegi oma töö ära. Kuid kompilaatori silmis `NULL` makro oli tavaline täisarv väärusega 0 (Shutter & Stroustrup, 2007).
- Täiuslik edastamine (ingl. k. *perfect forwarding*) sisaldas viga viida edastamisel (Lavavej, 2009). Probleem asus selles, et täisarv 0 loeti 2003 aasta standardis erandiks, mis lubas tal vaikimisi olla teisendatud viida tüübiks. Täiuslikul edastamisel tüüp aga määrati enne sisendi järgi ja alles siis edastati see väärtus koos tüübiga. 0 on olemuselt täisarv ja seetõttu täisarvu tüüp ka edastati. Täisarvu tüüpi ei ole võimalik aga vaikimisi teisendada viida tüübiks, mis andis ka vastava veateate.

Uuenduse rakendamise võimalused:

- Võimalik on kasutaja defineeritud `NULL` makro ümber defineerimine 0-st `nullptr`'iks, kui see oli kasutaja defineeritud. Ajaliselt võtaks see sekundeid, kui just ei teki probleeme koha leidmisega, kus defineeriti `NULL` makro. Samas tuleb arvestada, et kui kood peale seda ei kompileeri, on võimalik, et `NULL` makro leidis antud koodis väärkasutust (vt koodinäide 1).

```
int number = NULL;
char letter = NULL;
```

Koodinäide 1. `NULL` makro väärkasutamine

- Võimalik on asendada kõik `NULL` makrod `nullptr`'i vastu, seda kas käsitsi või mõne tekstitöötlus programmi abil. Tuleb olla tähelepanelik sarnaselt eelmisele punktile `NULL` makro väärkasutamise suhtes.
- Juhul, kui viida algväärtustamisel vanas koodis oli kasutatud 0-i, siis on võimalik ettevaatlikult asendada viitade ees asuvad 0-id `nullptr`'iga. See on ajaliselt kõige kulukam, kuna tuleb jälgida, mis on muutuja tüüp, enne kui teha koodis muudatusi.

Valitud sai teine variant, kuna see tundus antud töö autorile on kõige korrektsemana. Samas võib käsitsi ümber kirjutamine olla ajaliselt kulukas. Visual Studio–l on õnneks olemas lihtne tekstiredaktor, mis lubas hõlpsasti vahetada `NULL` makro `nullptr`'iks. Koodi uuendamine rakenduses lubas autoril näha näpukat, kus ta kirjutas `NULL` makro juhuslikult täisarvutüübi väärtuseks. Autori arvates on tühja viida korrektse kasutamise stiil kokkuvõttes muutunud (vt koodinäide 2).

```
static MyWindow* pThis = NULL; //Korrektne C++03 järgi
static MyWindow* pThis = nullptr; //Korrektne C++11 järgi
```

Koodinäide 2. Tühi viit enne ja praegu

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- Käitusaegne (ingl. k. *runtime*) kiirus `nullptr`'i kasutusele võtmisega ei muutu või pole märgatav (vt Lisa 1, tabel 1)
- Mälu kasutatavus võib muutuda, kuna `nullptr_t` suurus peab olema võrdväärne `void*` (ISO/IEC, 2011), mis ei pruugi olla, aga võrdväärne mõne täisarvu tüübiga (vaata Lisa 1, tabel 4).

2.2 Tüübi määraja: `auto`

Tüübi-määrajana `auto` näitab, et deklareeritava muutuja tüüp tuvastatakse selle algväärtuse kaudu või selle deklareeritav funktsioon sisaldab lõplikku-tagastatavat tüüpi (ISO/IEC, 2011).

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- puudus võimalus geneerilist funktsiooni tagastava muutuja tüübi määramiseks, kuna antud tagastatavat tüüpi oli võimalik tuvastada alles geneerilise funktsiooni kompileerimise ajal;
- pikad või raskesti loetavad tüüpide nimetused tuli välja kirjutada (Järvi & Stroustrup, 2004). Tüüpiline näide on konteinerite iteraator (vt koodinäide 3 esimene rida), milles võisid kergesti tekkida näpuvead.

```
std::map <WS, T>::const_iterator iValue = enumMap.find(value);
auto iValue = enumMap.find(value);
```

Koodinäide 3. Enne ja pärast `auto` kasutusele võtmist

- Tüüpide pedantsel väljakirjutamisel suurenes veatekke tõenäosus. Vaadates koodinäidet 3, on näha, et autor kirjutas uuesti välja `std::map`'i ja `WS`'i, mis tähendab, et juhul, kui autor muudaks muutuja tüübid vastavalt `std::unordered_map`'iks või `std::string`'iks, siis ta peaks ka meeles pidama nende muutmist igas iteraatoris.

Uuenduse rakendamise võimalused:

- võimalik on asendada algväärtustamisel muutuja tüüp `auto`'ga, mis lubab tuvastada muutuja tüübi algväärtuse tüübist;
- eelnevat punkti on võimalik kasutada kas igal pool või siis ainult pikemate ja keerukamate tüübinimetuste puhul. See jääb stiiliküsimuseks.
- Lisaks eelnevatele on võimalik kasutada `auto`t geneerilise funktsiooni tagastatava tüübi määramiseks `decltype` abiga. `decltype` määrab tüübi ja `auto` püüab antud tüübi kinni (vt koodinäide 4).

```
template <typename T> //Kui oletame, et T on double
auto func(T x, int y) -> decltype(x + y) { return x + y; }
```

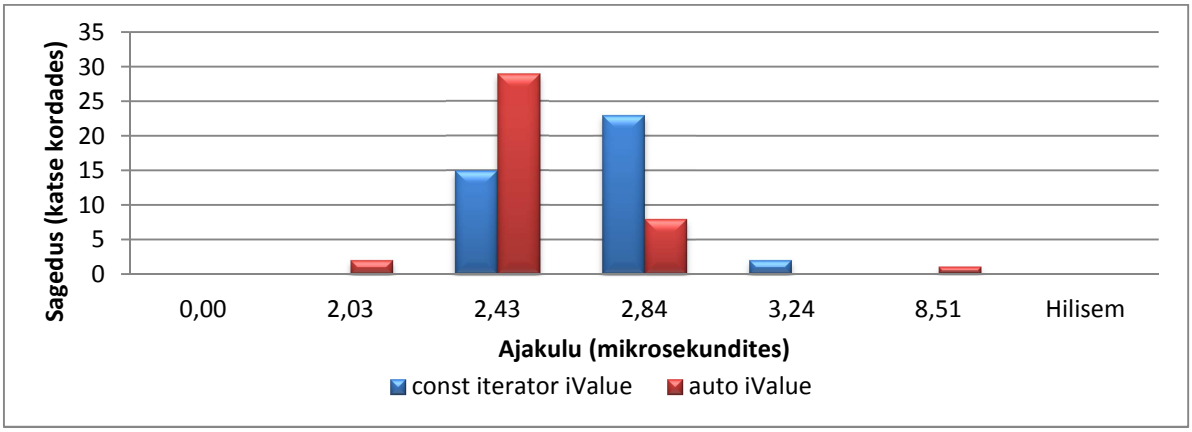
Koodinäide 4. Geneerilise funktsiooni tagastatava tüübi määramine

Rakendamisel kasutas autor eelviimast viisi, kus jättis vahetamata `HRESULT`, `BOOL` tüüpi ja baastüüpi muutujaid (`int`, `bool` jne.). `HRESULT` ja `BOOL` tüüpi ei vahetanud autor välja, kuna need on tegelikult defineeritud `long` ja `int`, mis ei ole väljendamise kontekstis nii kohane kui

seda on HRESULT tüüp vigadest teatamiseks. Autori arvates on tüübi määraja korrektse koodi kirjutamise stiili kokkuvõttes muutnud (vt koodinäide 4), asendades eelkõige veaohtlikud tüübid auto'ga. Ajakulu sellele oli suur, kuna tuli välja otsida koodi, mida taheti muuta. Antud programmi koodis oli otsimise ajakulu 30 minutit.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- käitusaegne kiirus auto kasutusele võtmisega ei muutu või siis muutub kiiremaks vaadates joonist 1, kus näeme koodinäite nelja kiiruse testi tulemusi. Selliseid tulemusi autor ei oodanud, kuna eeldas kiiruse samaks jäämist, kuid tulemustes oli näha selge väike nihe (u. 0,4 mikrosekundit) uue viisi kasutusele võtmise kasuks. Nihke põhjuseid antud töö ei kirjelda.



Joonis 1. Auto kasutamine iValue määramisel (vt koodinäide 4).

- Mälu kasutatavus sellega ei muutu, kuna võtab tüübiks sama tüübi, mida oleks suure tõenäosusega ka programmeerija ise valinud (va Lisa 1, tabel 4).
- Samas tuleb pidada meeles, et auto süntaks kopeerib väärtust vaikimisi ja juhul, kui on soov objektile viidata, tuleb kirjutada auto& või auto&& (Järvi, Stroustrup, & Reis, 2004).

2.3 Kompileerimisaegne veatuvastus: static_assert

static_assert on deklaratsioon, kus konstantset avaldist (ingl. k. *constant expression*) on konteksti raames võimalik muuta bool'iks. Tõese väärtuse korral ei muuda see midagi, aga

avaldise väära väärtuse korral loetakse programm halvasti koostatuks (ingl. k. *ill-formed*) ja kuvatakse diagnostiline teade, mis sisaldab kasutaja poolt märgitud sõnumit. (ISO/IEC, 2011)

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- puudus veatuvastusviis, mis suutnuks tuvastada vigu, mida ei olnud võimeline tuvastama `#error` direktiiv. Erinevalt `assert` makrost hoidus `static_assert` süntaks täiendavast käitusaegsest ressurssikulust (Klarer, Maddock, Dawes, & Hinnant, 2004).
- Puudus kompileerimise ajal töötav veatuvastaja, mis oleks saanud täiendada oma C++ veatuvastusvõimalustega eelmainitud veatuvastusviise. Antud veatuvastusviis sobib suurepäraselt kasutamiseks koos geneeriliste funktsioonidega, olgu see veatuvastamiseks või geneeriliste funktsioonide spetsialiseerumiseks.

Uuenduse rakendamise võimalused:

- seda on võimalik kasutada veatuvastamiseks funktsioonides, kui antud viga muutub tuvastatavaks juba kompileerimise ajal. See hoiab ära vea esinemise sügavamal koodis, kus ta ilmub välja tihtipeale ainult sümptomi kujul. Kui enne seda kasutati funktsioonis veatuvastamiseks kompileerimisaegset veatuvastajat emuleerivat makrot, siis viimase võib asendada antud veatuvastajaga ehk `static_assert` võtmesõnaga.
- Uuendust on võimalik kasutada geneerilise funktsiooni spetsialiseerumiseks. Seda illustreerib koodinäide 5, kus kompileerimisaegne tuvastaja kontrollib, kas antud tüüp on loendtüüp või mitte.

```
E MyXmlReader::readValueEnum()
{
      static_assert(std::is_enum<E>::value,
      "readValueEnum specified type must be some enum type");
      …
}
```

Koodinäide 5 `static_assert` kasutuses

Kuna autori loodud programmis ei kasutatud sarnaseid makrosid, siis selles programmis asusid enne kompileerimiseaegset veatuvastamist võimalikud veatekkekohad. Antud veatuvastajaid kasutas autor ainult geneerilistes funktsioonides, kus autor pigem kontrollis, kas kõik järgib geneerilise funktsiooni spetsialiseerumist. Erandiks on `ParseSomeEnum` funktsioon, kus autor kontrollib, et tüübid oleksid ühesugused. Sellega tegelemise ajakulu oli 15 minutit, kuna autor oli teadlik, kus koodis olid potentsiaalsed veakohad, võrdlemisi palju võttis aega veakohtade asukohaga seotud teadmiste sõnastamine.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- käitusaegne kiirus `static_assert`i kasutusele võtmisega ei muutu või siis pole muutused märgatavad (vt Lisa 1, tabel 1).
- Mälu kasutatavus sellega ei muutu, kuna tegemist on kompileerimisaegse funktsiooniga.

2.4 Range tüüpsusega loendtüüp

Range tüüpsusega loendtüüpi nimetatakse ka skoobitud loendtüübiks. Antud loendtüüpi deklaleeritakse `enum class` või `enum struct` võtmesõna abil, mõlemad võtmesõnad on semantiliselt võrdväärsed (ISO/IEC, 2011).

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- vaikteisendamine loendtüübist täisarvuks on lubatud, kuigi ei tohiks olla. (Shutter, Miller, & Stroustrup, 2007)
- Puudub võimalus määrata baastüüp, seetõttu pole võimalik ennustada ega ka määrata suurust. Samuti puudub võimalus ennustada või määrata loendtüüpi (nimelt märgisust) (Shutter et al., 2007).
- Skoobi puudumine, mille tõttu skoobita loendtüüp lubab kirjutada veaohtliku koodi tänu kahele või enamale erinimelisele loendtüübile, kus asub samanimeline andmeobjekt (Shutter et al., 2007).
- Mitteühilduvad laiendused, mis adresseerivad neid vigu (Shutter et al., 2007).

Uuenduse rakendamise võimalused:

- On võimalik lisada olemasolevatele loendtüüpidele enne loendtüübi nimetust võtmesõna `class` ja pärast koolonit täisarvutüüp (vt koodinäide 6). Lisaks eelnevale tuleks lisada ka skoop loendtüübi kasutamisel (vt koodinäide 7).

```
enum class DataInXml : int {...}; //C++11
```

Koodinäide 6. Loendtüübi põhi

```
enumMap[L"NAME"] = SYSTEM_BUTTON_NAME; //C++03
//C++11 lühendamata
enumMap[L"NAME"] = SystemButtonInXml::SYSTEM_BUTTON_NAME;
enumMap[L"NAME"] = SystemButtonInXml::NAME; //C++11
```

Koodinäide 7. Loendtüübi kasutus enne ja pärast

- Võimalik on ka jätta määramata täisarvu tüüp (vt koodinäide 6) ja lisada ainult `class`, loendtüübi kasutamisel ka skoop. Määramata jätmisel loetakse see vaikimisi `int`'iks.
- Kui oled järginud kumbagi kahest eelnevast punktist, siis võimaluse korral on soovitatav lühendada loendtüübi elemendi nimetus, kuna puudub vajadus unikaalsete nimede järele (vt koodinäide 7 viimast kaks koodirida).

Rakendamisel asendas autor kõik skoobita loendtüübid range tüüpsusega loendtüüpideks, kuna pidas seda kaasaaegsemaks ja ohutumaks kui skoobita loendtüüpi. Ajakulu selle rakendamiseks sõltus sellest, kui paljudele skoobita loendtüübi objektidele oli vaja lisada skoopi, aga umbkaudseks väärtuseks sai 10 minutit antud programmi koodis.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- käitusaegne kiirus range tüüpsusega loendtüüpide kasutusele võtmisega ei muutu või siis pole muutus märgatav (vt Lisa 1, tabel 1).
- Mälu kasutatavus selle uuendusega võib muutuda, kuna on nüüd võrdväärne määratud täisarvu tüübiga (ISO/IEC, 2011) (vt Lisa 1, tabel 4). Enne oli loendtüübi suuruse ja selle baastüübi määramine fikseeritud kompilaatoriga (ISO/IEC, 2003).

2.5 Klassi meetodite ülekate: `override` ja `final`

Antud võtmesõnad on mõeldud ülekattega kasutamiseks, et kindlustada soovitud tulemuste saamist.

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- alamklassi meetod katab juhuslikult üle ülemklassi meetodi, kuigi seda ei kavatsetud teha (Voutilainen, Meredith, Maurer, & Uzdavinis, 2009);
- alamklassi meetod ei kata ülemklassi meetodit üle tehtud trükivea tõttu (Voutilainen et al., 2009);
- alamklassi meetod ei kata ülemklassi meetodit üle, kuna eksiti argumendi tüübiga (Voutilainen et al., 2009);
- alamklassi meetod kaetakse ülemklassi meetodiga programmeerija poolt üle teadlikult, kuid see meetod ei olnud mõeldud ülekatmiseks;
- püütakse pärida klassilt, mis ei ole mõeldud pärimiseks.

```
class MySysBtn final : public MySysBtnBase
{
      virtual HRESULT Draw() override;
```

Koodinäide 8. `override` ja `final` kasutuses

Uuenduse rakendamise võimalused:

- tekkis võimalus lisada `override` igale poole, kus on soov teha ülekatet meetodile (vt koodinäide 8 viimane koodirida);
- võimalus kirjutada `final` klassile, mis ei ole mõeldud pärimiseks (vt koodinäide 8 esimene koodirida);
- tekkis ka võimalus ülekatet lubavale meetodile kirjutada `final`, kuid autor ei näe ühtki selget põhjust, miks peaks keegi seda tegema.

Rakendamisel püüdis autor hoiduda koodi `final` kasutamisest, eriti funktsiooni tasemel. Samas `override`'i kasutas ta igal võimaliku hetkel, sest `override` tegi kindlaks, et autor kirjutas funktsiooni üle. `final` on pigem ennetav omadus ja mõeldud rohkem teistele koodi

kasutajatele. Ajakulu oli oodatust suurem, kuna `final`'i lisamine on klassi disainimise ajal tekkiv küsimus, millega muutmisega ei tahaks iga kord tegeleda nii autor kui autori arvamuse kohaselt ka teised programmeerijad. Umbkaudselt kulus selle rakendamisele 40 minutit.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- käitusaegne kiirus `override`'i ja `final`'i kasutusele võtmisega ei muutu või siis pole muutus märgatav (vt Lisa 1 all tabel 2).
- Mälu kasutatavus sellega ei muutu, kuna tegemist on kompileerimisaegse kontrolliga. Testides koodi sarnase koodinäite 8 näitel saime tulemuseks tabeli, mida näeme Lisa 1 all tabel 4. See, mis kirjas tabelis, näitab tõesti, et antud võtmesõnade lisamine klassi mälu suurust ei muuda.

2.6 Parempoolse väärtuse viide: &&

&& abil deklareeritud viidet (*reference*) nimetatakse parempoolse väärtuse viiteks (C++11). Parempoolne viide käitub sarnaselt vasakpoolsele viitele, kuid lubab ennast siduda ka parempoolse väärtusega.

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- liigutamise semantika puudumine. Eelnevalt oli võimalik ainult kopeerida, üksikute eranditega (Hinnant, Abrahams, & Dimov, 2004).
- Täiusliku edastamise viisi puudumine. Selle saavutamine eeldas eelnevalt tohutul hulgal funktsioonide ülelaadimist (Hinnant et al., 2004).
- Mittekonstantne viide ei saanud ennast eelnevalt siduda parempoolsete väärtustega (Hinnant et al., 2004).

Uuenduse rakendamise võimalused:

- on võimalik lisada && funktsiooni parameetriks peale tüüpi kui soovid objekti liigutada funktsiooni argumendist funktsiooni sisse (vt koodinäide 9). Tulemuseks oleks see, et funktsiooni argumendina asuvast muutujast jääks ainult tühi konteiner kuna selle andmed liigutati ümber funktsiooni sisesele muutujale.

```
MyXmlReader(std::wstring&& fileName);
```

Koodinäide 9. Liigutamiseks mõeldud parameeter.

- on võimalik lisada && geneerilise funktsiooni parameetriks peale määramata tüüpi, kui soovid muutujat edastada, kasuta funktsioonis sellisel juhul võimalusel ka `std::forward` täiuslikuks edastamiseks (vt koodinäide 10).

```
template<typename WS>
T ParseSomeEnum(WS&& value) {
      ...
      auto iValue = enumMap.find(std::forward<WS>(value));
```

Koodinäide 10. Standardne täiuslikku edastamise viis.

- Erandlikel juhtudel on võimalik kasutada && ka koos võtmesõnaga `const` ühel ja samal tüübil.
- Samuti on võimalik kasutada kombinatsiooni `auto&&`, kuid tuleb arvestada, et see ei pruugi olla sama tüüp kui määratud tüüp, mille ees asub &&. See sarnaneb käitumiselt teisele punktile.

Rakendamisel vajas see osa kõige rohkem tähelepanu, kuna on võrreldes eelnevatega üks raskemini arusaadavaid uuendusi. See mõjutab suuresti ka koodidisaini, kuna nüüd on võimalik lisaks kopeerimisele muuta klassi liikuvateks objektideks. Sellega tekib disainiküsimus – tuleb otsustada, millal kasutada võimalust liigutada objekte ja millal mitte. Seetõttu on kõnealuse võimaluse rakendamisel suur ka ajakulu, autoril kulus umbes 10 tundi selleks, et otsustada, kuhu seda on vaja lisada ning kuhu mitte. Antud rakendamise ajakulu on tingitud ka sellest, et taoline omadus ei ole veel täielikult rakendatud VC11-s. Autor sai seda rakendada liikumise konstruktoris, liikumise operaatoris ning ka mõnes funktsioonis. Järgneva peatüki I alapeatüki funktsioonide rakendamise ajakulu autori loodud programmis sai ka siia alla arvestatud.

Selle uuendusega kaasnevat muutused kiiruses ja mälus:

- käitusaegne kiirus muutub vastavalt sellele, kuidas && on rakendatud. Vaadates Lisa 1 all tabel nr 5 on näha, et see on tõesti erinev. Kui objekte liigutatakse või edastatakse, siis kasutusel on funktsioonid, mille parameetriks on parempoolne viide. Erinevus

sõltub sellest, kas koodi muudetakse, kas kood on määratud, kas tüüpi proovitakse liigutada või mitte. Üldjuhul on selle kasutamine kiirendanud funktsioone võrreldes kopeerimisega märgatavalt.

- Mälu kasutatavus sellega otseselt ei muutu.

3 Teegid

3.1 Utiliidi teek: Utiliidi komponendid

Siin teegi osas asuvad funktsioonid, mida kasutatakse läbi kogu STL teegi (ISO/IEC, 2011). Teisisõnu, siin asuvad koos võrdlemise operaatoritega järgnevad funktsioonid: `std::swap`, `std::move`, `std::forward` ja `std::declval`.

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- funktsioon `std::swap` ei olnud piisavalt geneeriline C++03 ajal ja ei olnud suuteline ka omavahel vahetama C-stiili massiive (Krügler, 2009).
- Puudus lihtne ja kontekstiliselt arusaadav funktsioon vasakpoolse väärtus parempoolseks väärtuseks muutmiseks (Krügler, 2009).
- Puudus lihtne ja kontekstiliselt arusaadav abifunktsioon, mis edastaks objekti täielikult.
- Puudus funktsioon, mille ülesanne on saada tuntud tüüpi objekt mitte kasutamise kontekstis (Krügler, 2009).

Uuenduse rakendamise võimalused:

- tekkis võimalus kasutada `std::move` või `std::swap` väärtuste liigutamiseks muutujate vahel liikumise konstruktoris (vt koodinäidet 11).

```
XmlGeometry(XmlGeometry&& th):
      thickness(std::move(th.thickness)),
      xmlFill(std::move(th.xmlFill)),
      xmlConnection(std::move(th.xmlConnection)),
      xmlShape(std::move(th.xmlShape)){}
```

Koodinäide 11. Liigutamise konstruktor

- Tekkis võimalus liigutada mittekonstantseid objekte parempoolsest tehteosast vasakpoolsele tehteosale `std::move` abiga (vt koodinäide 12).

```
m_xmlData = std::move(xmlReader.m_xmlData);
```

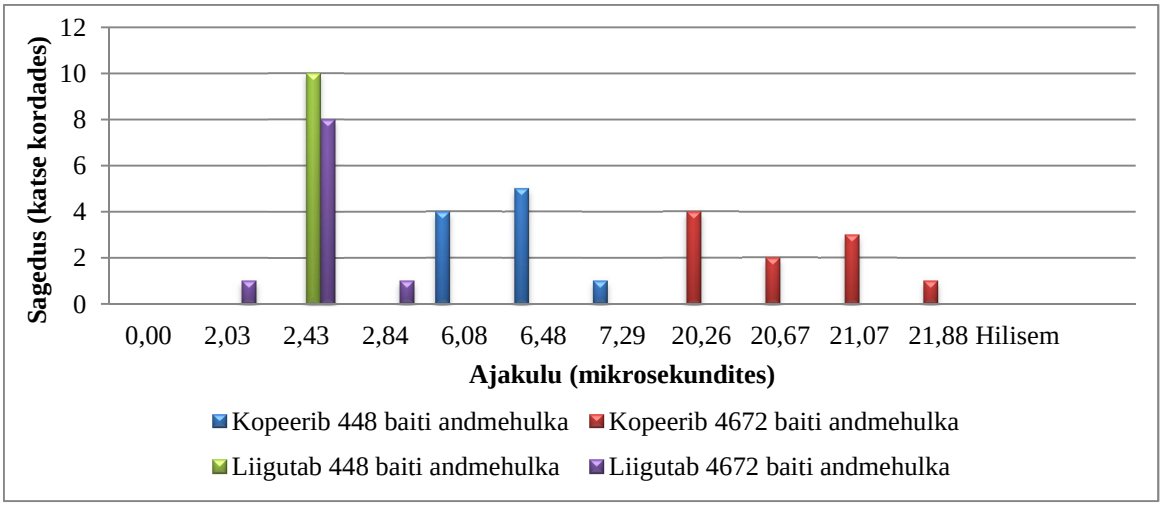
Koodinäide 12 Objekti liigutamine

- Tekkis võimalus edastada objekt eeldusel, et objekti ei muudeta, objekt algab välisfunktsiooni parameetrist määramata parempoolse viite tüübina ja lõpeb `std::forward` funktsiooniga, mis asub sisefunktsiooni argumendi või tagastatava väärtuse kohal (vt koodinäide 10).
- Võimalus on ka kasutada `std::move` funktsiooni argumendina, aga soovitatavalt ainult juhul, kui parameeter eeldab määratud parempoolse viite tüüpi. Lisa 1 all tabelis nr 5 vaatamisel on märgata selle funktsiooni rakendamisel mõju rakenduse kiirusele erinevates situatsioonides.

Antud uuenduse rakendamine toimus paralleelselt parempoolsete viidete rakendamisega, kuna uuendus ja parempoolsed viited on üksteisega sügavalt seotud. Samas kasutas autor rakendamisel antud teegi osa isegi rohkem kui parempoolseid viiteid, kuna andmete liigutamiseks lihtsat `std::move` funktsiooni oli kasutatud erinevalt teistest ka muudes kohtades sarnaselt koodinäitele 12. Rakendatud sai ka edastamise laiendus ja autor otsustas seda eelistada mõnes kohas, kus oleks eeldatud liigutamise laienduse kasutusele võtmist. Ajakulu, milleks oli 10 tundi, sai juba lisatud parempoolsete viidete teema alla.

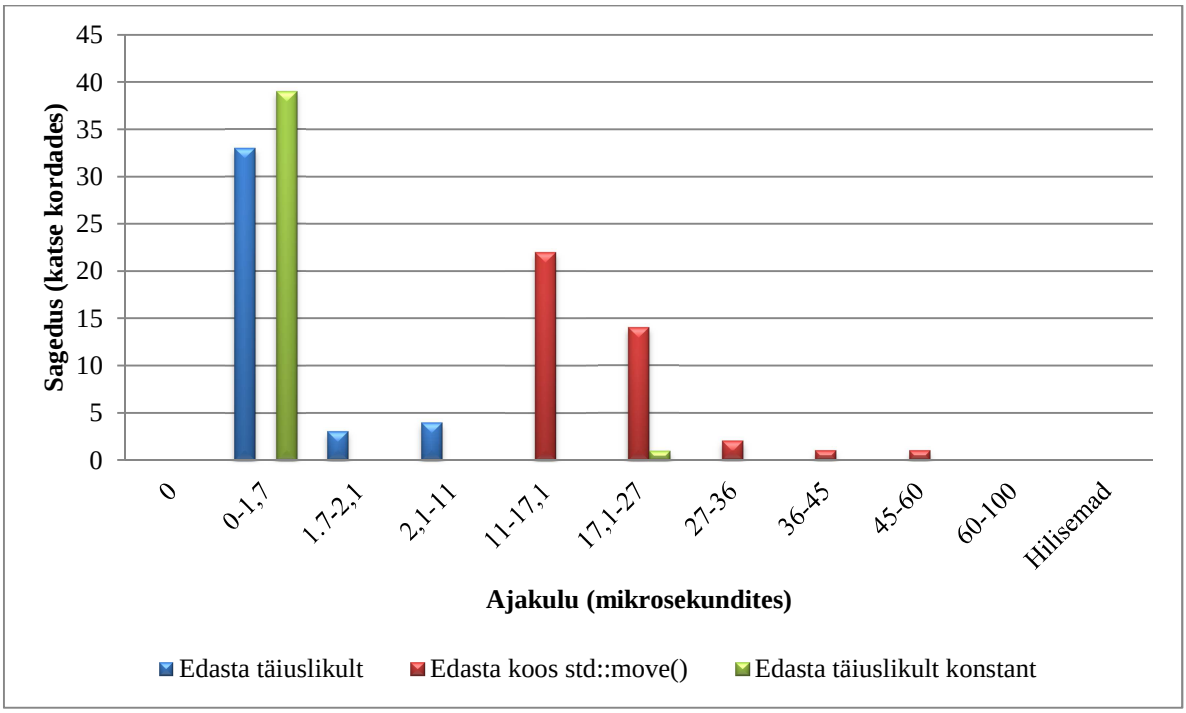
Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- `std::swap` muutub mõnel ajal kiiremaks, kuna on võimeline nüüd ka objekte liigutama.
- `std::move` muudab kiiremaks objekti liigutamise teatud juhtudel. Vaadates joonist nr 2 on näha, et selle rakendamine tõepoolest kiirendab andmete edastamist ja seda lausa mitmekordselt. Lisaks on näha, et andmehulga suurenemisel liikumise ajahulk ei muutu, kuid andmehulga suurenemisega suureneb kopeerimisele kuluv ajahulk.



Joonis 2. Andmete liigutamise ja kopeerimise võrdlus (vt koodinäide 12).

- `std::forward` muudab objekti edastamise kiiremaks, seda tingimusel, et objekti ei liigutata. Vaadates joonist nr 3 on näha, et antud võimaluse kasutamine vähendab märkimisväärselt ajakulu funktsiooni rakendamisel. Kui aga kasutada funktsiooni argumendiks `std::move`, siis kiirus langeb kümnekordselt, kuna too loob ajutise objekti. Täiuslikul edastamisel ei ole näha märgatavaid erinevusi kiiruses konstantsete ja mitte konstantsete objektide võrdlemisel.



Joonis 3. Täiuslik edastamine (vt koodi Lisa 2 all funktsioonis `TestMain()` `MoveorVector == 0`).

3.2 Utiliidi teek: Targad viidad

Targad viidad on objekt, mis omab ja haldab teist objekti läbi viida (ISO/IEC, 2011, üldistatud `std::unique_ptr` mõiste).

See uuendus kaasnes, lahendamaks järgnevaid probleeme:

- `std::auto_ptr` ei töötanud hästi koos STL konteineritega ja vajas seetõttu paremat asendust, mis oleks suuruselt võrdne C-stiili viidaga (Alistar Meredith, 2009) ning jätkaks omaduste edastamise semantika põhimõtet.

- Puudus viit, mis järgiks jagatud omaduste semantika põhimõtet, aga nõudlus sellise viida järele oli suur. (Dimov, Dawes, & Colvin, 2003).

- Puudus viit, mis järgiks ajutist omaduste semantika põhimõtet, see on mõeldud täiendamaks eelneva viida kasutamise lihtsustamiseks.

- Lisaks puudus palju abifunktsoone, mis lihtsustaksid tarkade viitade kasutamist.

Uuenduse rakendamise võimalused:

- nüüd on võimalik rakendada `std::auto_ptr` asemel `std::unique_ptr` ja sellega saavutada parem töökindlus ja kõrgemal tasemel ühilduvus konteineritega.

- On võimalik asendada C-stiili viitasid targa viidaga, et saavutada rakenduse parem töökindlus ja niiviisi hoiduda mälulekete võimalikest tekkekohtadest (vt koodinäide 13).

```
MyWindow* pMyWindow;
std::unique_ptr<MyWindow> pMyWindow;
```

Koodinäide 13. C-stiili ja tark viit

Tarkasid viitasid rakendas autor juba enne koodi uuendamist, aga need olid mittestandardsed `CComPtr`id. Mujal C-stiili viitade asemel sai rakendatud tarku viitasid. Tarkadest viitadest kasutas autor ainult `std::unique_ptr` viita, kuna tal polnud vajadust antud objekti jagada

viitade vahel. Aega kulus sellele umbes 10 minutit, kuna koodis ei olnud alles palju C-stiili viitasid. Samuti jättis autor puutumata staatilised viidad.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- kuigi on oletatav, et tarkade viitade kasutusele võtmine aeglustab koodi, ei muutu koodi kiirus või siis pole muutus nendes kontekstides märgatav (vt Lisa 1, tabel 3).

- Mälu võib suureneda C-stiili viitade kasutamisega võrreldes, sõltuvalt sellest, milline tark viit kasutusele võtta (vt Lisa 1, tabel 4).

3.3 Utiliidi teek: Tüübi eripärad

Antud teegi osa sisaldab komponente, mida kasutatakse C++ programmides põhiliselt mallides, et toetada võimalikult laia hulka tüübi eripärasid, optimeerida malli koodi kasutamist, tuvastada tüübiga seotud kasutaja vigu. Samuti kasutatakse antud teegi osa komponente tüübi teisendamiseks kompileerimise ajal (ISO/IEC, 2011).

See uuendus kaasnes, lahendamaks järgmisi probleeme:

- puudus võimalus toetada võimalikult laia hulka erinevaid tüüpe ühesuguste eripäradega, et lubada mallil spetsialiseeruda ülelaadimisel (Maddock, 2002)(vt koodinäide 14);
- puudus võimalus kontrollida, kuidas malli argumendid järgivad lepet ja ilmutada kontrollimise tulemusi veateatena sellele sobivas kohas (Maddock, 2002);
- puudus võimalus kirjutada sobivaimat algoritmi vastavalt igale eripärale (Maddock, 2002). Tihtipeale tuli kirjutada algoritm kas igale tüübile eraldi või kirjutada üks üldine algoritm, mis ei olnud enam nii optimeeritud.

Uuenduse rakendamise võimalused:

- võimalik lisada tüübi eripära koos `static_assert`’iga, et tuvastada juba kompileerimise ajal, kas vastav tüüp on kindla leppe alusel sobiv või mitte (vt koodinäide 5).

- Võimalik on lisada geneerilisele funktsioonile või klassile `std::enable_if` koos tüübi eripäraga. Sellisel juhul lisatakse antud `typename` kas malli parameetriks, funktsiooni parameetriks või funktsiooni tagastatavaks tüübiks (vt koodinäide 14).

```
template <typename T = typename std::enable_if
<std::is_enum<T>::value, T>::type, typename S = std::wstring>
class EnumParser {...}
```

Koodinäide 14. `std::enable_if` kasutuses

Selle rakendamine võttis rohkem aega kui `static_assert`'i rakendamine. Kuigi nad on üksteisega põimitud, ei ole sellest teegist arusaamine nii lihtne, kui arvata võiks. Ka eripärade tuvastamine ei käitu alati nii, nagu võiks oletada. `std::enable_if` oli autori arvates väga sarnane `static_assert`'ile, aga tüübi eripärasid, millega koos `std::enable_if` sai kasutada, ei olnud nii laias valikus kui oli eelneval. Selle võimaluse rakendamisele kulus kokkuvõttes umbes 50 minutit.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- käitusaegne kiirus tüübi eripärade kasutusele võtmisega otseselt ei muutu (vt Lisa 1, tabel 1), kuid võib muutuda kaudselt;

- mälu suureneb, kuid jääb minimaalseks, kuna enamiku tüübi eripärade funktsioonidest tagastatakse booli tüüpi väärtust. Mälu suurus võib ka mitte suureneda, juhul kui tüübi eripärade funktsioonid kasutatakse ära juba kompileerimisaegselt.

3.4 Sõne teek

Antud teek sisaldab komponente, mis hoiavad endas objekte, mis manipuleerivad ja hoiavad sõnesid ja tähemärke.

See uuendus kaasnes lahendamaks järgmisi probleeme:

- põhisõned ei sobinud kasutamiseks koos paljude funktsioonidega, kus samas sobis kasutamiseks null-lõpetatud sõnesid, kuna antud funktsioonid polnud vastavalt üle laetud (Becker, 2006);

- puudus standardne võimalus teisendada põhisõnesid numbrilisteks väärtusteks ja vastupidi;
- puudus standardne võimalus teisendada põhisõnesid paiskväärtuseks. Antud võimalus on tekkinud ka paljudes teistes teekides;
- puudus võimalus liigutada põhisõnesid, enne oli võimalik vaid kopeerida;
- puudus kiire võimalus vahetada antud konteiner teise tüübi konteineri vastu.

Uuenduste rakendamise võimalused:

- alati on võimalik jätkata vana koodi kirjutamist, aga see ei ole soovitatav;
- on võimalik lisada `std::to_string` sinna, kus on vaja teisendada numbreid sõnedeks;

```
dataShape.points.emplace_back(std::to_string(x),
      std::to_string(y));
```

Koodinäide 15. `std::to_string` kasutuses

- nüüd on võimalik lisada `std::stof` ja teisi sarnaseid teisendusfunktsioone, et saavutada vastupidine efekt eelmisele punktile (vt koodinäidet 16);

```
localColor.r = std::stof(std::wstring(pwszLocalValue));
```

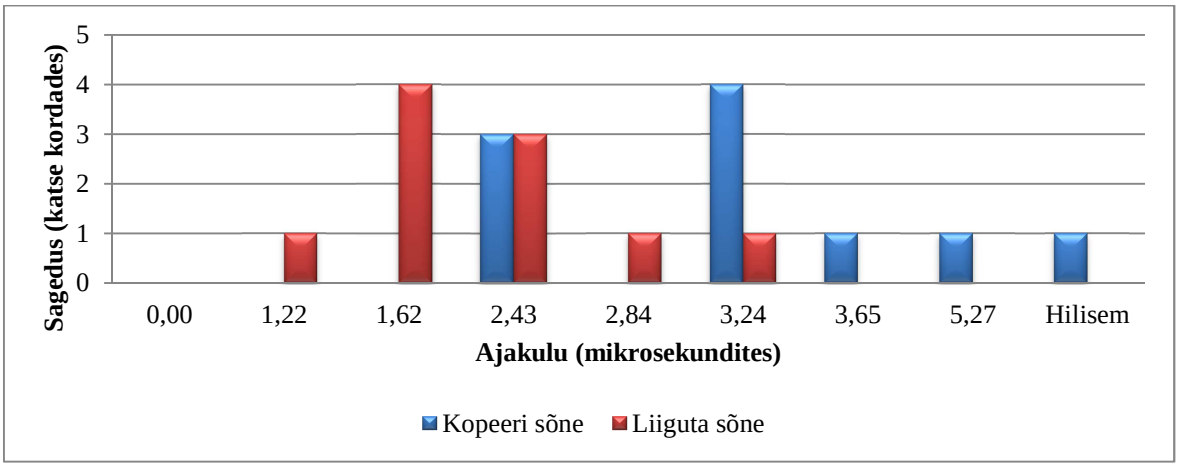
Koodinäide 16. `std::stof` kasutuses

- nüüd on võimalik kergemini vahetada välja põhisõne teist tüüpi konteineri vastu, kuna uuema standardis asuvad põhisõned sisaldavad rohkem sarnaseid funktsioone, mis vähendab ajakulu nende muutmisele;
- nüüd on võimalik kiiresti ja kergesti muuta sõnesid paiskväärtuseks.

Enamik C++ kasutajaid on kasutanud põhisõnesid, samuti sai selle programmi loomisel kasutatud sõnesid. Võimalused, mida sai kasutatud antud programmis, on vajalikud, kuna eemaldasid vajaduse mõõta teksti pikkust ja kasutada C-stiili sõnesid – selle eesmärk on luua võimalus muundada sõna arvuks. Ajakulu sellele oli lühike, rakendamiseks kulus umbes 10 minutit, aga ajavõit oli suur, kuna puudus vajadus lugeda igat tähemärki eraldi.

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- põhisõned muutuvad kiiremaks tänu parempoolse vääruste viidetele. Seetõttu on ka funktsioonide keerukus langenud lineaarselt konstantseks, eeldusel, et jaotaja (ingl. k. *allocator*) on ühesugune (ISO/IEC, 2011). Joonisel 4 on näha mõlemast viisist tekkivast ajakulu moodi võrdlusest, et põhisõne loomine läbi teise sõne liigutamise on peaaegu kaks korda kiirem kui seda on kopeerimine.



Joonis 4. Põhisõne kopeerimine ja liigutamine (vt koodi Lisa 2 all funktsioonis `TestMain()` `MoveorVector == 1`).

- Maht on sõltuv kompilaatorist, aga suure tõenäosusega suudeti mälu kasutatavust paremini vähendada VC11-s võrreldes selle eelkäijatega just tänu oskuslikumalt optimeeritud põhisõnedele.

3.5 Konteinerite teek

Antud teek sisaldab komponente, mida võidakse kasutada andmete organiseerimiseks (ISO/IEC, 2011).

See uuendus kaasnes, lahendamaks järgmisi probleeme:

- puudus massiiv, millel oleks fikseeritud arv elemente (Alisdair Meredith, 2003);
- puudusid alternatiivid kahendpuude põhimõttel töötavatele assotsiatiivsetele konteineritele;

- konteineritel puudus funktsioon, mis lubaks edastada elementi täielikult talle ette nähtud kohale konteineris ilma mingi liigutamise või kopeerimiseta;
- konteinerid olid optimeerimata, vastamaks kaasaegsetele tingimustele.

Uuenduste rakendamise võimalused:

- konteinerite valik on suurenenud ja tänu sellele on rohkem eri võimalusi konteinerite valikuks vastavalt olukorrale.

```
std::map<S, T> enumMap;
std::unordered_map <S, T> enumMap;
enumMap.find(std::forward<WS>(value));
```

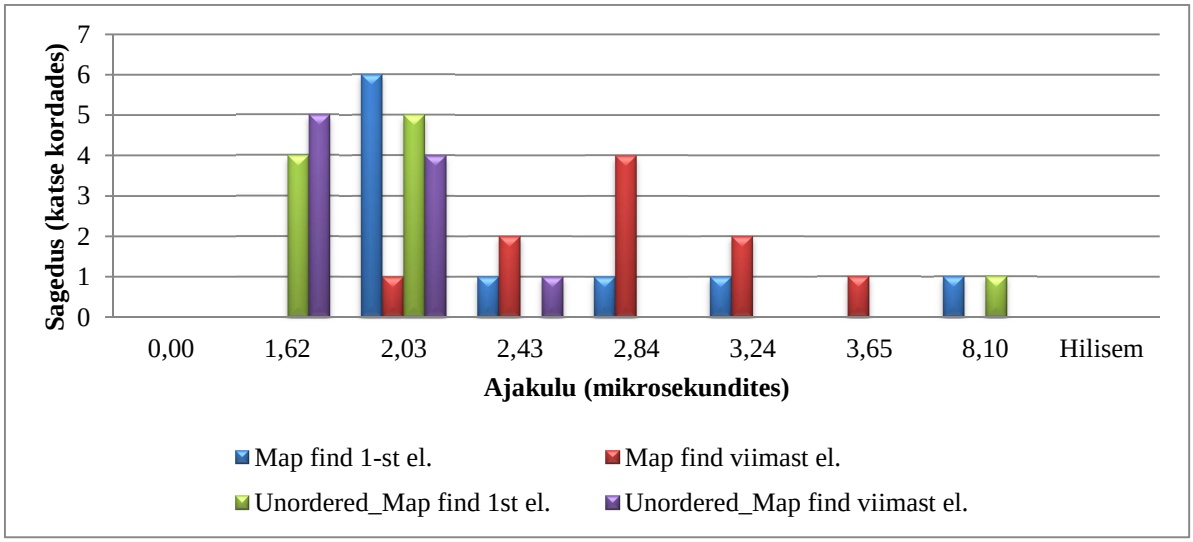
Koodinäide 17. Binaarne puu või paisktabel

- Võimalik on kasutada uut `emplace` funktsiooni, tänu millele muutus elemendi edastamine efektiivsemaks, kuna hoidub ajutiste objektide loomisest.

```
enumMap[L"MOUSE_STATES"]   = SystemButtonInXml::STATES;
enumMap.emplace(L"MOUSE_STATES", SystemButtonInXml::STATES);
```

Koodinäide 18. `emplace` ja `operator[]` kasutuses

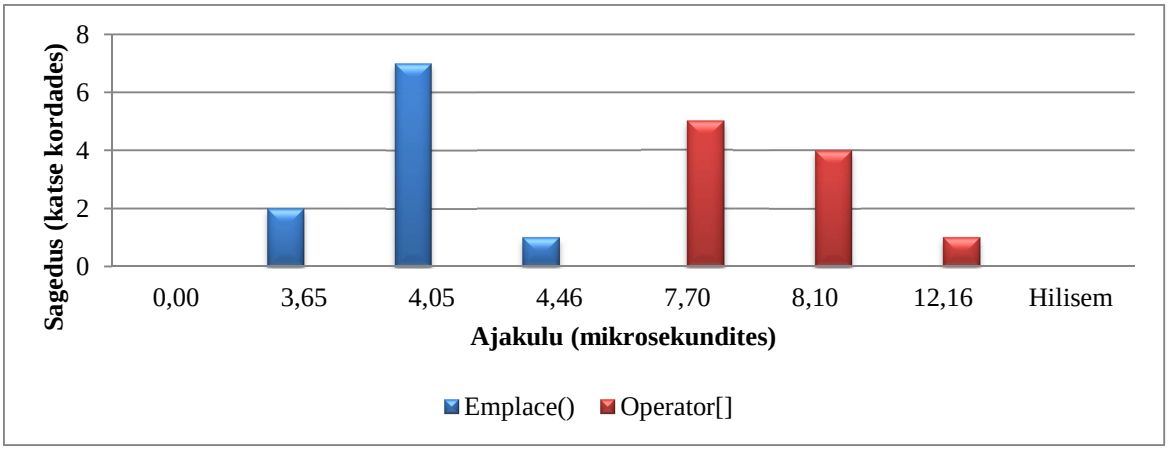
Nende uuenduste rakendamine toimus väga erinevatel aegadel. Kuigi konteinereid rakendas autor juba programmi loomisel, sai tänu uuele `data` funktsioonile asendatud C-stiili massiiv vektoriga, seejärel tagastas autor C-stiili viita. Samuti sai kasutusele võetud `emplace_back` funktsioon, asendamaks `push_back` funktsiooni. Alguses, kui autor hakkas uuendama `EnumParser`'it, oli too lihtsalt klass, kuid on nüüdseks arenenud kõige rohkem uuendusi kasutavaks klassiks antud programmis. Sellel klassis sai välja vahetatud põhiline konteiner, kuna numbrid näitasid, et uus kasutusele tulnud konteiner on kiirem ja efektiivsem kui see, mida autor kasutas enne (vt joonis 5). Samuti sai vahetatud välja viis, kuidas lisati eelnevale konteinerile elemente, et kulutada vähem aega funktsiooni täitmisele. Ajakulu oli 40 minutit, kuna tuli mõõta, kuhu mis täpselt sobis.



Joonis 5. Binaarse puu (`map`) ja paisktabeli (`unordered_map`) võrdlus otsingu kasutamisel (vt. koodinäide 17).

Selle uuendusega kaasnevad muutused kiiruses ja mälus:

- kiirus suureneb mitmel viisil. Vaadates joonist nr 5, on näha, kuidas muutub ajakulu funktsiooni `find` kasutamisel kumbagi konteineri koodinäitest 17. Paisktabel on märgatavalt kiirem ja selle otsimise kiirus pidi olema konstantne võrreldes binaarse puu omaga, kus see pidi olema logaritmiline (ISO/IEC, 2011). Antud joonis toob seda ka hästi esile. Samas on paisktabel antud kontekstis kiirem isegi sellisel juhul, kui oleks mõlemate konteinerite otsimise algoritmi keerukus võrdväärne.
- Vaadates joonist nr 6 on näha, et eelnev punkt ei ole ainukene viis kiirendada programmi selle teegi abil. Lisaks on võimalik kasutada uusi `emplace` funktsioone. Antud juhul kasutame seda võrdluses paljudele tuttava `operator[]` funktsiooniga. Joonis 6 võrdlusest (koodinäide 18) on näha, et `operator[]` on aeglasem `emplace` ist ja selle põhjuseks on ajutise objekti loomisest hoidumine.



Joonis 6. `emplace()` ja `operator[]` võrdlus (vt. koodinäide 18).

- Maht on sõltuv kompilaatorist, aga suure tõenäosusega suudeti mälu kasutatavust vähendada VC11-s võrreldes selle eelkäijatega just tänu paremini optimeeritud konteineritele.

4 Üldpilt ja järeldused

Eelnevates peatükkides sai vaadeldud detailselt, kuidas C++11 on laiendanud C++ võimalusi pärast C++03. Vaadeldi, kuidas C++11 laienduste rakendamine muudab terve koodi loetavust, ajakulu ja kiirust. Mida ei vaadeldud, on koodi üldpilt. Kogu programmi kood asub Lisa 2 all ning on vajalik osa sellest peatükist arusaamiseks.

Üldpildis jäi ka osa C++11 laiendusi sellisel juhul rakenduses kasutusse viimata, seda kahel põhjusel:

- esiteks puudus vajadus kasutada teatud laiendusi loodud programmis. Näiteks lambda-funktsioonid, mis on toetatud küll kompilaatori poolt, kuid vastavat kasutusala ei leidnud;
- teiseks ei toetanud autori poolt valitud kompilaator kõiki C++11 laiendusi. Seetõttu sai laienduste valik piiratud.

4.1 Koodi loetavus

Kuigi eelnevates peatükkides mainitud muutuste tõttu võiks arvata, et koodi loetavus on ka üldpildi tasemel muutunud, on autori arvates kood muutunud üldpildis ainult osaliselt pisut loetavamaks.

Põhjuseid on selleks kolm:

- Laienduste tegemisel eeldati, et kood peaks jääma äratuntavaks. Vastasel juhul võib see kogenud C++ arendajaid, kuna nende arust on toimunud liiga suured muutused, mistõttu nad võivad eeldada, et sellega kaasneb tohutul hulgal ümberõppimist. Samuti häiriks see ka koodi loetavust, kuna süntaks ei oleks arendaja poolt kergesti seostatav vana ära harjutud süntaksiga.
- Eelnevates peatükkides mainitud laiendused ei tõsta eriti loetavust, võrreldes mõnede rakendamata jäänud C++11 laiendustega, näiteks kasutaja poolt defineeritud literaalid (B. Stroustrup, 2003). Eelnevalt mainitud laiendus võib muuta loetavuse paremaks või väärkasutusel hullemaks sellisel määral, et tekitab küsimuse, kas see on veel seesama keel. Erinevalt enamikest teistest laiendustest ongi loetavuse parandamine või hullemaks muutmine kasutaja poolt defineeritud literaalide ülesanne.

- Autor valis, nagu eelnevalt mainitud, C-stiili teeke ja APIsid, mis piirasid uute laienduste kasutusalasid märgataval tasemel.

Seega võib järeldada, et kuigi loetavus pole palju muutunud, on C++11 laiendustel seda siiski võimalik parandada ja aja möödudes viia parandusi sisse ka suuremal määral.

4.2 C++11 laiendused, vead programmis ja ajakulu

Eelmistes peatükkides sai mainitud ajakulu. Üldkokkuvõttes ei kesta C++11 laienduste rakendamine autori näidatud programmile sarnase suurusega programmi puhul kauem kui nädal. Samas võiks arvestada ka ajakulu nende selgeks õppimiseks, mis samuti ei kestaks tõenäoliselt rohkem kui nädal. See teeb ajakuluna kokku maksimaalselt kaks nädalat, arvestades, et seda rakendaval isikul on ka muud tegevused, millega ta antud nädalate sees tegeleb.

Teine ajakulu programmi loomisel ja uuendamisel on vigade parandamine, mis võtab enda alla kohe kindlasti suurema osa. Autoril oli koodis nii pisivigu, mis võtsid kuni viis minutit parandamiseks, kui ka suuremaid vigu, mille parandamiseks võis minna nädalaid, arvestades muidugi, et autor ka parandas ja uuendas koodi samal ajal. Autor pole ka käesoleva töö esitamise ajaks suutnud kõiki suuremaid vigu eemaldada.

Autor märkas, et C++11 laiendused aitavad vähendada vigade teket koodis. Ent enamik vigu, millest antud uuendused aitavad hoiduda, on reeglina väikse või keskmise suurusega. Näiteks `NULL` makro väärkasutamine ja `static_assert` aitab lahendada keskmise suurusega vigu. Autor märkas, et muutused ei toimunud suuremates vigades, nagu näiteks selles, et ei olnud võimalik kasutada süsteeminuppe või olukorras, kus terve aken vilkus.

Sellest võib järeldada, et uued laiendused aitavad parandada pisemaid vigu nendest hoidumist soodustades. Samas ei saa uued laiendused aidata programmeerijaid suuremate vigade parandamisel. Küll aga saab ajakulu, mis tuleb suuremate programmide loomisel olema tõenäoliselt väiksem kui vead, mida nad aitavad ära hoida.

4.3 Programmi kiirus

Eelnevates peatükkides sai näidatud, et C++11 laiendused enamuse ajast kas kiirendasid funktsioone või ei muutnud ajaliselt midagi. Samas on küsimus, kui palju üldpildis kiirus siis muutus.

Autor pidas seda huvitavaks, kuna kiirenes enamjaolt ainult akna avamine. Põhjus selleks on lihtne, C++11 laiendusi sai kasutada kõige rohkem avamise protsessis, kuna põhiprotsess oli tugevalt seotud C-stiili teekide ja APIdega. Kuid tuleb märkida, et akna avamine tõesti tundub kiiremana. Ent selle esialgne versioon omas märkimisväärselt teistsugust kuju disainiküsimuste vastuste muutumises, seetõttu seda ajaliselt mõõtes oleksid tulemused ebamäärased, seega arvestab autor ainult eelnevates peatükkides saadud tulemusi.

Autor ei maininud, et osa C++ laiendusi saavad mitu korda olema kasutatud ja sellega väheneb ajakulu märkimisväärselt. Näiteks vaadates joonist nr 6, on näha, et erinevus on umbes kahekordne, kuigi numbrid on väiksed. Samas, sarnast koodi kutsutakse akna avamisel 21 korda iga täiendava andmefaili puhul. Hetkel on andmefaile ainult kaks, ehkki suurematel rakendustel võib neid olla sadu ja need võivad reeglina sisaldada rohkem kui 21 erinevat tüüpi elemente.

Sellest võib järeldada, et programmi kasvamisega kasvab ka C++11 laienduste võit sellisel määral, et ühe funktsiooni vahetamisega võib suure programmi puhul võita terve sekundi ja kui kasutada mitu uuendust, siis on ajavõit vastavalt ka suurem.

Kokkuvõte

Programmi uuendamisel C++11 laiendustega sai autor teada, kui palju ta saab koodi kasutada, kui tal on antud määratud tingimused. Autori sai koodi kasutada paljudes kohtades, kuid see sõltus vägagi olukorrast. Mõnes kohas polnud võimalik kasutada laiendust ning selleks polnud ka vajadust. Teisest küljest aga oli võimalik täiendada antud koodi paljude uute laiendustega. Tihtipeale suurenes tänu uuendustele ajavõit, loetavus ning programmis vähenes vigade parandamise vajadus. Autor mõistis, et antud C++11 laiendused aitavad tõepoolest võita rohkem, kui neile kulub ressursse.

Edasiarenduseks on siin ruumi küllaga. Vaadates käesoleva seminaritöö teksti, jääb viimane peatükk jääb eelnevatega võrreldes üldiseks, sest eelnevais peatükkides keskenduti detailidele. Üldpilti on võimalik süveneda rohkem siis, kui kompilaatoritesse jõuavad ka teised C++11 laiendused. Samas on võimalik ka uurida mainimata jäänud C++11 laiendusi. Muidugi on võimalik jätkata tööd programmiarendusega ning uurida selle abil midagi uut.

Kasutatud kirjandus

Becker, P. (2006). Uniform Use of std::string Revision 1. *WG21 Document*, (Revision 1). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2006/n1981.html

Dimov, P., Dawes, B., & Colvin, G. (2003). A Proposal to Add General Purpose Smart Pointers to the Library Technical Report. *WG21 Document*. Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1431.htm

Hinnant, H. E., Abrahams, D., & Dimov, P. (2004). A Proposal to Add an Rvalue Reference to the C++ Language. *WG21 Document*. Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1690.html

ISO/IEC. (1998). Information Technology — Programming languages — C++, *1998*(First edition).

ISO/IEC. (2003). Information Technology — Programming languages — C++, *2003*(Second edition).

ISO/IEC. (2011). Information Technology — Programming languages — C++, *2011*(Third edition).

Järvi, J., & Stroustrup, B. (2004). Decltype and auto (revision 3). *WG21 Document*, (revision 3). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1607.pdf

Järvi, J., Stroustrup, B., & Reis, G. D. (2004). Decltype and auto (revision 4). *WG21 Document*, (revision 4). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1705.pdf

Klarer, R., Maddock, J., Dawes, B., & Hinnant, H. (2004). Proposal to Add Static Assertions to the Core Language (Revision 3). *WG21 Document*, (Revision 3). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1720.html

Krügler, D. (2009). Moving Swap Forward. *WG21 Document*. Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2958.html

Lavavej, S. T. (2009). *Everything you ever wanted to know about nullptr*. Channel 9. Vaadatud http://channel9.msdn.com/Shows/Going+Deep/Stephan-T-Lavavej-Everything-you-ever-wanted-to-know-about-nullptr

Maddock, J. (2002). A proposal to add type traits to the standard library. *WG21 Document.* Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2002/n1345.html

Meredith, Alisdair. (2003). A Proposal to Add a Fixed Size Array Wrapper to the Standard Library Technical Report. *WG21 Document*. Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1479.html

Meredith, Alistar. (2009). Constraining unique_ptr. *WG21 Document*, (revision 4). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2853.pdf

Shutter, H., Miller, D., & Stroustrup, B. (2007). Strongly Typed Enums (revision 2). *WG21 Document*, (revision 2). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2213.pdf

Shutter, H., & Stroustrup, B. (2007). A name for the null pointer: nullptr (revision 4). *WG21 Document*, (revision 4). Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2431.pdf

Stroustrup, B. (2003). Literals for user-defined types. *WG21 Document*. Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1511.pdf

Stroustrup, B. (2012). C++11 - the new ISO C++ standard. Loetud November 1, 2012, http://www.stroustrup.com/C++11FAQ.html#think

Stroustrup, B. (AT&T B. L. (1991). *A History of C ++ : 1979 − 1991*. New Jersey. Loetud http://www.stroustrup.com/hopl2.pdf

Voutilainen, V., Meredith, A., Maurer, J., & Uzdavinis, C. (2009). Explicit Virtual Overrides. *WG21 Document*. Loetud http://open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2928.htm

Lisad

Lisa 1. Tabelid

Tabel 1. Objektid ja funktsioonid, mille ajakulu on märkamatu

(Ülevalt alla näitab katsete hulka ja lahtris näidatakse aega (mikrosekundites))

Tühiaeg (aeg, mis kulub mitte millegi mõõtmisele)	`CComPtr<INuiSensor> pNuiSensor =`		`static_assert (std::is_same <std::decay <WS>::type, S>::value, "...");`	`SystemButtonInXml systemButtonInXml = SystemButtonInXml::STATES;`	
	`NULL;`	`nullptr;`		`enum SystemButtonInXml : int {};`	`enum class SystemButtonInXml : int {};`
1,215652	1,215652	1,215652	1,215652	1,215652	1,215652
1,215652	1,215652	1,215652	1,215652	1,215652	1,215652
1,215652	1,215652	1,215652	1,215652	1,215652	1,215652
1,215652	1,215652	1,215652	1,215652	1,215652	1,215652
1,215652	1,215652	1,215652	0,810435	0,810435	1,215652
1,215652	1,215652	1,215652	1,620869	1,215652	1,215652
1,215652	1,215652	1,215652	1,215652	1,215652	1,215652
0,810435	1,215652	1,215652	1,215652	1,215652	0,810435
1,215652	1,215652	1,215652	1,215652	0,810435	1,215652
1,215652	0,810435	1,215652	1,215652	0,810435	1,215652

Tabel 2. `override` ja `final` mõju klassis ja funktsioonis kasutamisel

(Ülevalt alla näitab katsete hulka ja lahtris näidatakse aega (mikrosekundites))

`MyClass myClass;`	`MyClass (final) myClass;`	`MyFunc()`	`MyFunc() override`	`MyFunc()final`
2,431304	2,026086	2,026086	1,620869	1,215652
1,620869	1,620869	1,620869	1,215652	1,620869
1,215652	1,215652	1,620869	1,215652	1,215652
2,026086	1,620869	1,620869	1,215652	1,215652
1,215652	1,620869	0,810435	1,215652	1,215652
8,104345	1,620869	1,620869	1,620869	1,215652
8,104345	1,215652	1,620869	1,215652	1,215652
1,620869	1,215652	1,215652	1,215652	1,215652
2,026086	0,810435	1,620869	1,620869	1,620869

1,215652	1,215652	1,215652	1,620869	1,215652
1,215652	1,215652	1,620869	1,215652	1,215652
1,620869	1,215652	1,620869	1,620869	1,215652
1,215652	1,620869	1,620869	1,215652	1,620869
1,620869	1,215652	1,215652	1,620869	1,215652
1,215652	1,215652	1,215652	2,026086	1,215652
1,620869	1,620869	1,215652	1,215652	1,215652
1,215652	1,215652	1,215652	1,215652	1,215652
1,215652	1,215652	1,215652	1,215652	1,215652
1,620869	1,215652	1,215652	1,215652	1,215652
1,620869	0,810435	1,620869	1,215652	1,215652

Tabel 3. Tarkade viidate kasutusega tulenev ajakulu või pigem selle puudumine (Ülevalt alla näitab katsete hulka ja lahtris näidatakse aega (mikrosekundites))

`DataShape* pMyCount`		`std::unique_ptr <DataShape> pMyCount`		`std::shared_ptr <DataShape> pMyCount`	
`new DataShape ()`	`pMyCount -> pot = 1;`	`new DataShape ()`	`pMyCount -> pot = 1;`	`new DataShape ()`	`pMyCount -> pot = 1;`
9,725214	1,215652	7,699128	1,215652	13,77739	1,620869
10,940866	1,215652	15,398256	0,810435	8,509562	8,104345
7,293911	1,620869	8,104345	1,215652	10,53565	1,215652
14,182604	1,215652	10,940866	1,215652	14,1826	1,620869
12,156518	2,026086	10,940866	2,026086	14,58782	1,620869
4,862607	1,215652	18,234776	1,215652	7,699128	1,215652
19,855645	1,620869	12,561735	2,026086	8,91478	1,620869
10,535649	1,215652	19,045211	1,215652	12,96695	1,215652
11,7513	1,215652	11,346083	1,620869	8,91478	1,215652
10,940866	1,620869	9,319997	1,620869	10,13043	1,620869

Tabel 4. Testitud mälu kasutusega seotud muutused			
sizeof()	**Suurus (baitides)**	**sizeof()**	**Suurus (baitides)**
`NULL`	4	`std::map<S,T>::const_iterator iValue;`	24
`Nullptr`	8	`auto iValue`	24
`pNuiSensor = NULL;`	8	`enum SystemButtonInXml : int {};`	4
`pNuiSensor = nullptr;`	8	`enum class SystemButtonInXml : int {};`	4
`MySysBtn` (koos `override`)	536	`MyWindow* pMyWindow;`	8
`MySysBtn` (koos `class final`)	536	`std::unique_ptr<MyWindow> pMyWindow;`	8
`MySysBtn` (koos `final`)	536	`std::shared_ptr<MyWindow> pMyWindow;`	16
`MySysBtn` ()	536		

Tabel 5. Optimeeritud ja standardse funktsiooni ülelaadimise lahenduste võrdlus (vt koodi Lisa 2 all funktsioonis `TestMain()` `MoveorVector == 0`).						
	Optimeeri-tud funktsioon, mis muudab ja edastab objekti.	**Optimeeri-tud funktsioon, mis muudab ja liigutab objekti.**	**Optimeeri-tud funktsioon, mis muudab ja kopeerib objekti.**	**Standardne funktsioon, mis muudab ja kopeerib mitte konstantset objekti.**	**Standardne funktsioon, mis muudab ja liigutab objekti.**	**Standardne funktsioon, mis muudab ja kopeerib objekti.**
Mediaan	1,216	17,019	22,490	30,391	24,993	22,692
Vahemike mood	0,8-10,3	10,3-19,8	19,8-29,2	19,8-29,2	10,3-19,8	19,8-29,2
Mood esines	39	24	27	14	24	22
Kõrvalekalle	1	16	13	26	16	18
Üldarv (*Count*)	40	40	40	40	40	40

Lisa 2. Programmi kood

Kogu programmi koodi asub digitaalsel kujul kaasaantud CD-plaadil.