TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mark Tomm 231073IACB

C++ varjatud efektid

Bakalaureusetöö

Juhendaja: Peeter Ellervee
Doktori kraad

Tallinn 2023

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Mark Tomm

10.05.2023

Annotatsioon

Käesoleva bakalaureusetöö eesmärk on määrata C++ keele varjatud efektidega omadusi, nendest kombineeritud tarkvara disaini mustreid ning lahendusi mis lasevad kasutada liides andmetüüpe polümorfses kontekstis. Antud töö raames käsitletakse C++ 17 standardit ja selle GNU kompilaatori implementatsiooni versiooni 12.1.0 x86-64 platvormil. Antud töö väljund on uuritud C++ omaduste käitusaja aja, käitusaja mälu ruumi ja mälu kujutise pindala kulud ning disaini mustrite ja mitmekujulisuse lahenduste ajalised võrdlusnäitajad.

Esimene samm on uurida C++ keele eritunnuste varjatud mõjusid, määrates kindlaks C++ keele GCC ehk GNU kompilaatori võimalikud käitusaja ja mälukujutise pindala kulud. Aluseks on võetud C keel, kui tööriist milles ei esine varjatud efektidega omadusi.

Teine etapp on võrrelda käitusaja kiirust eelnimetatud C++ eritunnuste erinevatel kombinatsioonidel, mille tulemuseks on teadaolevad tarkvara kujundus mustrid, ja selgitada, kuidas need aitavad programmikoodi korraldada tulevaste muudatuste ja laiendatavuse jaoks. Vaadeldud tarkvara kujundus mustrid on vajalikud, kuna neid kombineeritakse antud töö kolmandas osas et saavutada mitekujulisust.

Kolmas osa on võrrelda ajaliselt lahendusi mille abil on võimalik kasutada liides andmetüüpe polümorfses kontekstis ning määrata millisel moel iga lahendus piirab ja parendab programmikoodi laiendatavust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 62 leheküljel, 6 peatükki, 62 joonist, 1 tabelit.

Abstract

Hidden Effects in C++

The objective of this bachelor's thesis is to examine the hidden effects of C++ language features, combined software design patterns arising from these, and solutions that allow the use of specific types in a polymorphic context. This study focuses on the C++ 17 standard and its GNU compiler implementation version 12.1.0 on the x86-64 platform. The output of this work is the runtime time, runtime memory space, and memory image area costs of the investigated C++ features, as well as the temporal comparison metrics of design patterns and polymorphism solutions.

The first step is to investigate the hidden impacts of unique C++ language features by identifying the potential runtime and memory image area costs with the GCC, or GNU compiler of the C++ language. The base language for this comparison is C, which does not feature properties with hidden effects.

The second stage is to compare runtime speed on various combinations of the aforementioned C++ special features, resulting in known software design patterns. The aim is to elucidate how these patterns assist in organizing the codebase for future changes and extensibility. The software design patterns under consideration are essential as they are combined in the third part of this work to achieve polymorphism.

The third part involves benchmarking solutions that enable the use of specific types in a polymorphic context, and determining how each solution restricts and enhances the extensibility of the codebase.

The thesis is in Estonian and contains 62 pages of text, 6 chapters, 62 figures, 1 tables.

Lühendite ja mõistete sõnastik

OOP	Objekt orienteeritud programmeerimine
RAII	*Resource Acquisition is Initialisation*, lähenemine, kus ressurss on objekti elueaga seotud
RTTI	*Run-time type information*, käitusaja andmetüüpide informatioon
STL	*Standard Template Library*, Standardiga määratud C++ teek
Sub-klass	Alamklass, mis pärib kõik baasklassi omadused ja toimingud
Super-klass	Baasklass
OCP	*Open Closed Principle, SOLID* printsiipide teine punkt
OS	*Operation System*, operatsiooni süsteem
CPU	*Central Processing Unit*, Protsessor
RAM	*Random Access Memory*, Ajutine mälu
ABI	*Application Binary Interface*, kokkulepped kuidas teatud toimingud transleeritakse kõrgetasemelisest keelest Assemblerisse.
RVO	*Return Value Optimisation*, tuntud kompilaatori optimeerimise võte kus objekti enne tagastamist luuakse otse mälu regiooni, mis on eraldatud tagastamis väärtuse jaoks. On standardiga määratud alates C++17.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Kõrgetasemeliste programmeerimiskeelte tulekuga on C-l kahtlemata kõrgeim positsioon manus- ja madala latentsusega süsteemide tarkvara arendamisel [1]. See on tingitud C-keele disaini olemusest, mis liigitab selle kõrgetasemeliseks keeleks ja pakub samal ajal madalal tasemel juurdepääsu allolevale riistvarale. Selline kompromisside kogum on andnud tarkvaraarendus valdkonnale vastuvõetava suhte programmeerija töötundide tõhususe ja lõpp rakenduse käitusaja kiiruse vahel.

Tarkvara funktsionaalsusnõuded ja seega ka koodi keerukus suurenevad [2] ja see tähendab vajadust täiustatud tööriistade järele. Kuigi C-kompilaatorid pakuvad käitusaja ja mälu kulude osas enneolematut jõudlust, muutub tarkvarakoodi haldamine nõrgalt kontrollitud andmetüübi süsteemi, kaudsete teisenduste eri andmetüüpide vahel [3], osutite ja makrode abil üha raskemaks.

C++ on peaaegu tagasiühilduv C keelega [4] niivõrd, et seda võib vaadelda kui C koos lisafunktsioonidega [5]. Alates selle standardiseerimise algusaegadest on C++'i reklaamitud kui keel millel on otsene vastendamine riistvarale (sama mis C), null üldkulu abstraktsiooni mehhanismid (mis tähendab et kasutaja maksab vaid selle eest, mida ta kasutab), andmetüüpide kontrollimine, täiustatud vigade käsitlemine, kompileerimise aja arvutused, st mallid ja *constexpr* (alates C++11) [6].

C++ reklaamitakse aktiivselt kui C evolutsioonilist järeltulijat [5]. C++ on standardiseeritud juba üle 25 aasta ning seda on lihtne C keelest üle kanda, kuid C++ on endiselt C keelest üldise populaarsuse poolest maha jäänud [7]. Tekib mulje, et loomulik üleminek C++ peale pole kõigi nende aastate jooksul olnud nii lineaarne, kui seda keele kõiki eelnimetatud plusse vaadates esialgu järeldada võiks.

C on selles mõttes lihtne, et sellel ei ole palju eristatavaid tunnuseid ja abstraktsiooni mehhanisme. Hästi kogenud C-arendaja, kellel on põhiteadmised Assembleri keelest ja käesolevast arvuti arhitektuurist, suudab C-lähtekoodi uurides välja selgitada Assembleri loendi. Peamised kaudse suunamise mehhanismid on funktsioonid ja osutid (pointer).

Identifikaator (nimi) C keeles võib olla seotud objektiga [8], funktsiooniga, struktuuriga, struktuuri liikmega, *union'*iga, *union'*i liikmega, *enum'*iga, *typedef'*iga, *label'*iga, makroga, makro parameetriga [9]. Struktuur (*struct*) on oma kõige keerulisemal kujul erinevate eelmääratletud andmetüüpide kogum koos täiendava täidisega joondamise eesmärgil. Struktuuri liikme nihke saab arvutada käsitsi, uurides struktuuri sisu lähtekoodist. C ei varja arendaja eest ühtegi efekti [10].

C++ puhul ei pruugi andmetüübid olla nii triviaalsed kui struktuurid C keeles. Klassides võivad olla liikme funktsioonid, virtuaalsed ja puhtad virtuaalsed liikme funktsioonid, malli funktsioonid, sõbrad, *enum'*id, *enum* klassid, ülekoormatud operaatorid, üks kuni mitu konstruktorit, destruktor, kaudselt määratletud eri liikme funktsioonid jne. Klassid võivad olla tuletatud mitmest muust andmetüübist ja esineda mallidena. Niipea kui klassis on määratud liikme muutujad erinevate juurdepääsudega, ei ole standardiga enam garanteeritud nende füüsiline paigutus lähtekoodis määratud järjekorras. Samuti kui klassis on kasutaja poolt määratud mõni eriliikme funktsioon siis teatud juhtudel genereerib kompilaator taustal lisa koodi objektide haldamiseks. See loetelu ei ole ammendav. Ei ole harvad juhud, kui C keelt valdav arendaja eeldab, et lisakulud tekivad lihtsalt klassi kasutamisega andmetüübi määratlemiseks [10]. C++ keel on väga funktsionaalsuse rikas ja võib varjata toimingute mõju arendaja eest.

Antud töö eesmärk on uurida C++ neid omadusi, millel on varjatud mõjud ehk abstraktsiooni mehhanisme, mille kasutamisel ei ole lähtekoodist otseselt järeldatav lõpptulemuse keerukus.

Antud töö ülesanne on määrata C++ eri omaduste varjatud efektide käitusaja võimalikud kulud nii palju, et oleks võimalik põhjendada erinevate kujundus mustrite ja käitusaja mitmekujulisuse võtete toimingute aja kulu võrdlustestis Joonis 61 katse arvutiga Tabel 1.

Esimeses osas uuritakse erinevaid C++'i omadusi ükshaaval. Klassid, Pärimine, Polümorfism, käitusaja andmetüüpide info (RTTI), Erandjuhud, mallid ja standard teek (STL) sellises ulatuses, et saada andmeid käitusaja aja, käitusaja mälu ruumi ja mälu kujutise pindala kulude kohta.

Teises osas uuritakse levinud võtteid esimeses osas kirjeldatud C++'i omaduste kombineerimisel. Nende võtete ehk tarkvara kujundus mustrite eesmärk on teha

lähtekood paremini laiendatavaks ja lihtsustada selle haldamine. Koodi lõppkasutaja jaoks on nende mustrite eesmärk teha arusaadavamaks koodi eesmärk ilma, et peaks sekkuma implementatsiooni detailidesse ja vältida koodi kasutamist valel moel. Vaadeldud tarkvara kujundus mustrid on vajalikud antud töös, kuna neid kombineeritakse antud töö kolmandas osas et saavutada mitekujulisust või on nende abil realiseeritud mõni standard teegi osa.

Kolmandas osas uuritakse erinevaid lähenemisi mitmekujulisuse saavutamiseks. Tulemuseks on nende ajaline võrdlus ja näitajate analüüs.

Kui ei ole määratud teisiti, siis C tähendab ISO C11 ja C++ all on mõistetud ISO C++17. Vajadusel viitab antud töö C++17 mustandile N4713 ja C puhul mustandile N1570. GCC ehk GNU kompilaatori versioon on 12.1.0, platvorm on x86-64. Töös kasutatud arvuti parameetrid on Tabel 1.

OS	Manjaro Linux 22.0.3 Linux 5.15.93-1-MANJARO
CPU	Intel(R) Core(TM) i7-4790 CPU @ 4000.000MHz
RAM	4 x 8GB DDR3 1333MHz 1600MT/s
Kompilaator	GNU GCC v 12.1.0 -std=c++17

Tabel 1. Kaitse süsteemi parameetrid

Inglise keelsed terminid, sealhulgas ka C või C++ võtmesõnad, näiteks *for* või *new* on esitatud *kaldkirjas*. Identifikaatorid lähtekoodis, näiteks **CC** Joonis 1 või **std::unique_ptr**, muutujate nimed**,** kataloogi/faili nimed, kompilaatori argumendid ja samuti ka **this** võtmesõna klassi liikmefunktsiooni sees. **this** on ainuke erand kui C++ keele võtmesõna kirjutatakse **rasvaselt**.

```
// näidiskood
class CC {};
```

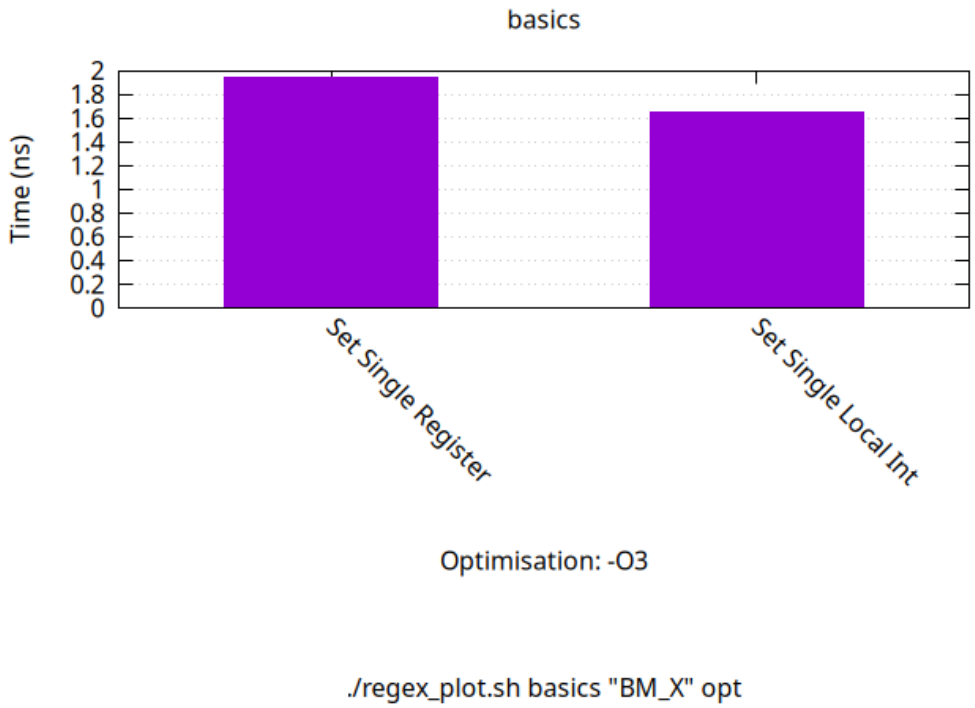
Joonis 1. C++. Klassi identifikaator ja deklaratsioon.

2 Võrdlustestid

Ajaliste võrdlustestide lähtekood täismahus *git* hoidlas järgneval lingil: https://github.com/marktomm/features-cpp

Võrdlustestide aluseks on antud töö lähtekoodis kasutatud *google-benchmark* C++ teeki. *google* on tõhus teek, mis on mõeldud C++ koodi jõudlusmõõtmiseks, keskendudes mikro võrdlustestidele. Muuhulgas pakub võimalust arvutada statistilisi mõõdikuid, nagu keskmine, mediaan ja standardhälve, pakkudes terviklikku ülevaadet koodi jõudlusest. Antud töös kasutatakse seda vaid keskmise arvutamiseks. Iteratsioonide arv on varieeruv kuni 10 astmes 9 ning seda otsustab põhi teek. Iteratsioonide arvu ei esitata antud töö joonistel.

Joonis 2 on toodud ühe ainsa toimingu: *movl* käsuga liigutada arvväärtus 1 **eax** registrisse *Set Single Register*. *Set Single Local Int* on ühele ainsale *int* tüüpi objektile pinus konstandi väärtustamine.



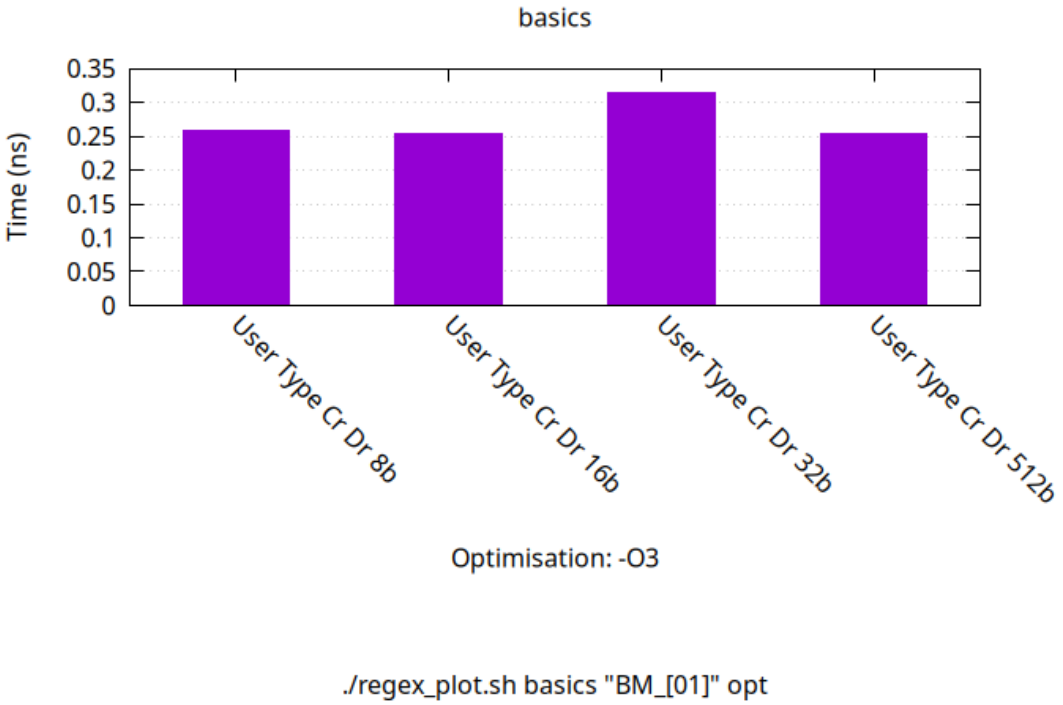
Joonis 2. Võrdlustest. Primitiivsed toimingud.

Testide jaoks kasutatud arvuti taktsagedus on 4000Mhz ning levinud on arvamus et lihtsale konstandi *movl* toimingule registrisse kulub ligikaudu 3 tsüklit. Kui üks tsükkel Tabel 1 arvuti peal on 0.25ns, siis kolm on kokkuvõttes 0.75ns. *google* teek näitab ligikaudu 1.8ns. Järeldus on et *google* teek võib anda kuni 1ns lisa kulu. *google* teek lingib vaikimisi *pthread* teegiga. Seetõttu võib arvata et mõni proloogi Assembleri käsk jääb mõõtmise aja vahemikusse. *Set Single Local Int* puhul on võimalik, et kompilaator kasutab kõige kiireima viisi programmi jooksutamiseks lõpuni, näiteks *xor* **eax** registrit ise endaga, et tagastada *google* teegi sisemise **main** funktsioonist nulli. Seega on see ka mõni tsükkel kiirem.

Igal võrdlustesti joonisel genereeritakse automaatselt:

1. Pealkiri päises
2. Ajavahemik (y–telg) nullist kuni optimaalne arvväärtus
3. Tulemused (tulemuste arv on varieeruv). Sildi tekst tuleb testi nimest lähtekoodis
4. Optimeerimise seis. Sees on **-O3,** mis on kõrge optimeerimise lipp GNU GCC'l. **-O0** tähendab väljas
5. Skript ja argumendid võrdlustesti käivitamiseks peatüki alguses antud *git* hoidlas

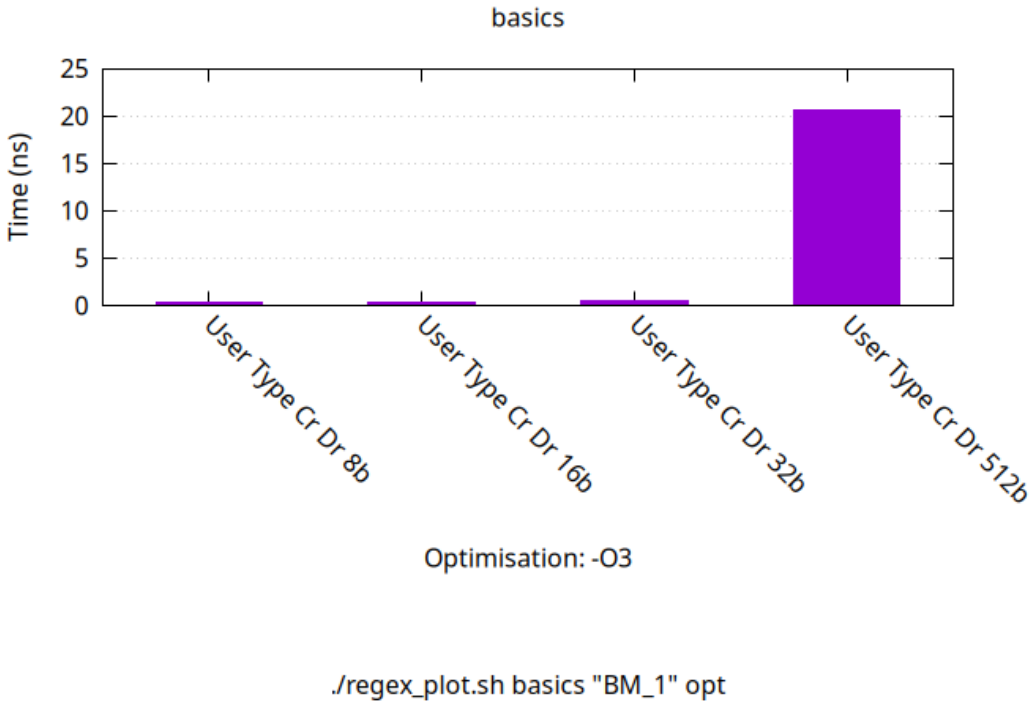
Cr Dr tulemuse sildi tekstis viitab sellele, et mõõte aja vahemikus on tehtud konstruktori ja destruktori rakendamine sildi nimes esimesena antud andmetüübi objektile. Kui ei ole kirjas teisiti (*Heap*, *Up*, *Ptr*) siis toimus lokaalse muutuja tekitamine pinus. Näitab, et objekti suurusest 8 ja 512 baidi vahemikus ei ole mõju konstruktori ja destruktori aja kulule juhul kui ei ole tarvis väärtustada kogu objekti mälu regioon.



Joonis 3. Objekti initsialiseerimine pinus juhul kui väärtustatakse ainult esimesed 8 baiti.

Antud võrdlustestis kui on vaja tekitada andmetüübi eksemplar pinus ning väärtusata ainult esimesed 8 baiti, siis on testi tulemus 1 tsükli ehk 0.25ns piires konstantselt. Mälu eraldamised üle üldiselt antud töös töötud võrdlustestides ei ületa 16 baiti.

Joonis 4 näitab ajalisi tulemusi objekti initsialiseerimisel pinus juhul kui peab algväärtustama kogu objektile kuuluva regiooni.



Joonis 4. Objekti initsialiseerimine pinus juhul kui väärtustatakse kogu objekt.

Joonis 37 näitab tulemusi mälu eraldamisel kuhjal.

3 C++ omadused

Ajaliste võrdlustestide lähtekood täismahus *git* hoidlas järgneval lingil: https://github.com/marktomm/features-cpp

Kataloogi tee failini esimesel real koodi joonistes tähendab, et joonise kood asub eelnevalt toodud *git* hoidlas. Andmetüüpide suhteid illustreerivad diagrammid genereeritud *PlantUML* abil. Lähtekood samuti eelneval lingil.

PlantUML legend:

- A täht sinine taust – abstraktne klass
- C täht roheline taust – terviklik klass
- Punase raamiga ruut – privaatse juurdepääsu täpsustajaga liikmemuutuja
- Rohelise taustaga ring ja ümarsulud lõpus – avaliku juurdepääsu täpsustajaga liikme funktsioon

Muul tekst koodi joonise esimesel real on viide koodi allikale, välja arvatud juhul kui kommentaari tekst on **näidiskood**.

C++ pärib suuremas osas kõike, mida oskab pakkuda C. Üks väljapaistev erinevus on et C++'s on rangemad andmetüüpide kontrollid [4]. C keeles on lubatud ümber määrata (*cast*) osuteid erinevatele andmetüüpidele otse.

C++ keeles annab Joonis 5 kood kompileerimise ajal veateate. C keeles on see vastuvõetav.

```
// näidiskood
// g++ prog.cc -Wall -Wextra -O2  -std=gnu++17
typedef struct S{}S;
typedef struct Z{}Z;

int main () {
    S s;
    Z z;
    S* sp = &s;
    Z* zp = sp; // error: cannot convert 'S*' to 'Z*' in initialization
    return 0;
}
```

Joonis 5. C++. Andmetüüpide ümber määramine.

C keelt võib sisuliselt mõista kui portatiivset masinkoodi generaatorit [11], mille esialgses kujutuses kõrgetasemelise koodi kujul puudub peidetud mõju lõpptulemusele. Antud peatükis uuritakse neid C++ omadusi, mille kasutamise tulemusel ei ole otsekoheselt (vaadates lähtekoodi transleerimise üksust) võimalik määrata käitusaja lisakulu, lõpp rakenduse mahtu (*image footprint*) lisakulu ja kujutise mälu jaotust *(image memory layout*).

3.1 Klassid

C++'s määratakse klassi ja *struct'*i defineerimisega eristatav andmetüüp. Andmetüüp on kompileerimise ajal teatud hulk objekti omadusi [1]. Nimelt:

6. Suurus ja jaotus mälus
7. Hulk lubatud väärtuseid (olekuid)
8. Hulk lubatud toiminguid

C++ klassid on sarnased C keele *struct* üksusega selle poolest, et nende eeldeklaratsioon (*forward declaration*) määrab kompileerimise ajal eraldiseisva andmetüübi olemasolu. Nende definitsiooniga määratakse kindel arv muid andmetüüpe, mis selle klassi või *struct'*i andmetüübi eksemplar hakkab endaga kaasas kandma.

3.1.1 Standardne paigutus

C++ keeles võib andmetüübil olla teatud juhtudel standardne paigutus [12]. See omadus garanteerib tüübi andmete paigutust mälus lähtekoodis deklareeritud järjekorras. Järgnev nimekiri määrab ära, millisel juhul andmetüübil puudub standardne paigutus:

1. Erinevad juurdepääsu täpsustajad mitte staatilistele andmetele

C++'s on klassidel (ja *struct*'idel) olemas juurdepääsu täpsustajad (*access specifiers*) *public*, *protected* ja *private*.

```
// näidiskood
#include <iostream>
#include <cstdint>

class A {
public:
  uint32_t a{5};
  uint32_t q{65535};
};

int main() {
  A a;
  uint8_t *a8p = (uint8_t *)&a;
  a8p += 4;
  uint32_t a32 = *(uint32_t *)a8p;
  cout << a32 << '\n'; // väljund: 65535, kõik on OK
}
```

Joonis 6. C++. Standardne paigutus, liikmemuutuja positsiooni määramine nihke abil.

Standardse paigutusega objekti eksemplaridel on C++ standardi poolt tagatud eelnevalt määratud füüsilise paigutuse järjekord.

Juhul kui klass **A** Joonis 6 oleks defineeritud nagu toodud Joonis 7, siis **main** funktsioon Joonis 6 oleks tähendanud määratlemata käitumist (*undefined behavior*) [13].

```
// näidiskood
// klass A versioon 2
class A {
public:
  uint32_t a{5};
protected:
  uint32_t q{65535};
};
```

Joonis 7. C++. Erinevad juurdepääsu täpsustajad.

Klassi **A** Joonis 6 ja klassi **A** Joonis 7 objektid võivad kasutada erinevat hulka mälu käitusaja jooksul. See omadus ei mõjuta käitusaega ega lõpp rakenduse mahtu (*image footprint*) muul viisil.

Joonis 8 toodud C++ koodi optimeerimata masinakood Joonis 9 kinnitab, et klasside **A** konkreetsete liikmete ligipääsemisel ja objektide kopeerimisel tehakse sama palju tööd.

```
// access_specifier/include/lib.h
class A {
public:
    uint32_t a{5};
    uint32_t q{65535};
};

class A2 {
public:
    uint32_t a{5};

protected:
    uint32_t q{65535};
};

// access_specifier/src/main.cpp
int main() {
    A a;
    cout << a.a;
    A2 a2;
    cout << a2.a;

    [[maybe_unused]] A aa = a;
    [[maybe_unused]] A2 aa2 = a2;
}
```

Joonis 8. C++. Erinevate juurdepääsu täpsustajatega klassid.

```
# optimisatsioonid off GCC -O0
./asm_prettify.sh access_specifier | grep '[0-9]* main' -A25
8 main:
; proloog eemaldatud
; a konstruktor
20      mov     DWORD PTR -24[rbp], 5
21      mov     DWORD PTR -20[rbp], 65535
; edasta a2.a to cout
22      mov     eax, DWORD PTR -24[rbp]
23      mov     esi, eax
24      lea     rax, _ZSt4cout[rip]
25      mov     rdi, rax
26      call    _ZNSolsEj@PLT
; a2 konstruktor
27      mov     DWORD PTR -16[rbp], 5
28      mov     DWORD PTR -12[rbp], 65535
; edasta a2.a to cout
29      mov     eax, DWORD PTR -16[rbp]
30      mov     esi, eax
31      lea     rax, _ZSt4cout[rip]
32      mov     rdi, rax
33      call    _ZNSolsEj@PLT
; aa = a
34      mov     rax, QWORD PTR -40[rbp]
35      mov     QWORD PTR -24[rbp], rax
; aa2 = a
36      mov     rax, QWORD PTR -32[rbp]
37      mov     QWORD PTR -16[rbp], rax
```

```
38      mov     eax, 0
```

Joonis 9. Asm. Joonise 4 Assembleri tulemus.

2. Virtuaalsed liikme funktsioonid

Virtuaal funktsioonide kasutamine Joonis 10 toob kaasa teatud käitusaja ajakulud.

```
// näidiskood
class NonStandard {
public:
  int a;
  virtual void doSomething() {}
};
```

Joonis 10. C++. Klass virtuaalse liikme funktsiooniga.

Selle asemel, et teha tavaline funktsiooni väljakutse, peab sisenema virtuaal funktsioonide tabeli kaudu, et määrata kindlaks, mis funktsioon peab tulemusena olema rakendatud käitusajal.

Käitusaja mälu kulud suurenevad ühe osuti suuruse võrra iga eksemplari kohta, kuna on vaja viidata õigele virtuaal tabelile (*vpointer overhead*). Samuti kulud lisa mälu virtuaal tabeli sisu jaoks, kuna virtuaal tabel hoiab endast osuteid õigetele funktsioonidele. Üks virtuaal tabel on jagatud ühe klassi kõikide eksemplaride vahel *(vtable overhead*) [14].

Mälu kujutise pindala suurust mõjutab antud juhul virtuaalne tabel ja lisa keerulisem kood, et teha topelt väljakutseid.

Rohkem detaile ja ajalised võrdlustestid on peatükis 3.1.5.

3. Virtuaalne baas klass

Virtuaalset pärimist määratakse enne päritud klassi ligipääsu täpsustajat võtmesõnaga *vurtual* Joonis 11.

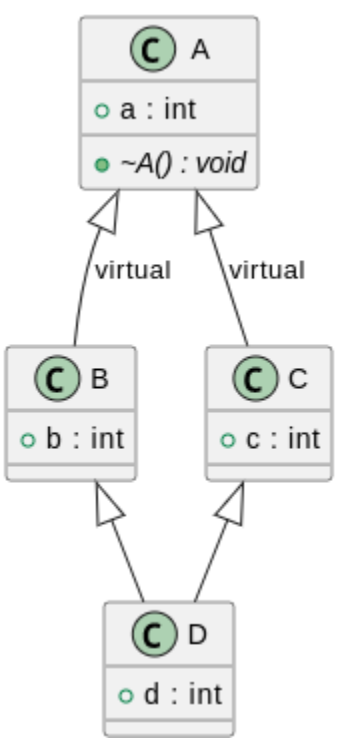
```
// näidiskood
class Base {};

class NonStandard : virtual public Base {
public:
  int a;
};
```

Joonis 11. C++. Virtuaalne baas klass.

Virtuaalse baasklassi omaduse ülesanne on lahendada “teemant probleem” (*diamond problem*), mis võib tekkida mitmekordse pärimise korral Joonis 12 ja Joonis 13.

```
// näidiskood
class A {
  public: int a;
};

class B: virtual public A {
  public: int b;
};

class C: virtual public A {
  public: int c;
};

class D: public B, public C {
  public: int d;
};
```

Joonis 12. C++. Teemant klass



Joonis 13. UML. Teemant klass

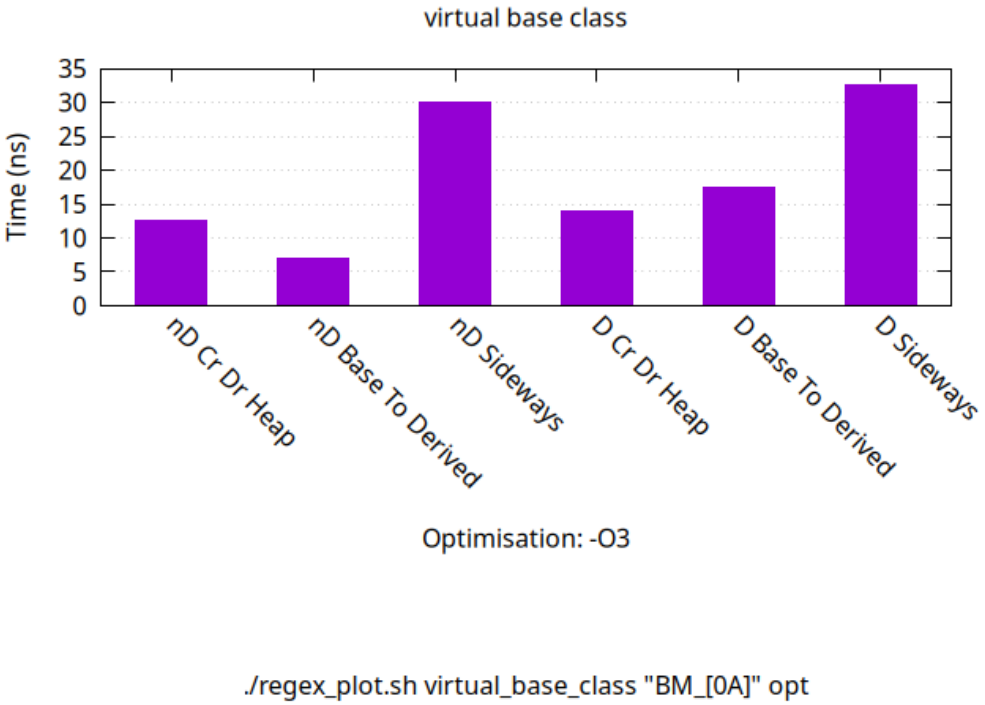
Virtuaalse pärimise korral tagatakse klasside hierarhias, et esineb ainult üks eksemplar baas klassist ja ei teki duplikaat andmeid.

Käitusaja aja lisa kulud tekivad virtuaal baas klassi andmete juurde pääsemisel, kui on vaja teha keerulisemat lisa loogikat, et arvutada baas klassi andmete paigutus [14].

Käitusaja mälu lisakulud tekivad kuna iga klass mis pärib virtuaalselt (Joonis 12 klassid **B** ja **C**), peab omama osuti virtuaal baas klassile.

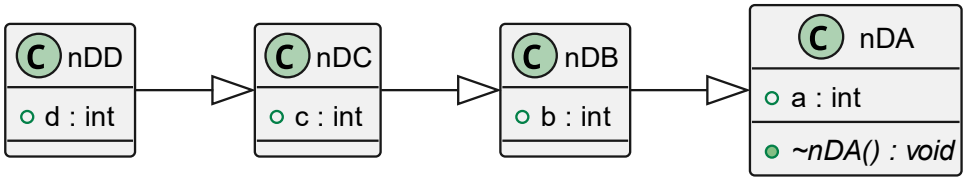
Mälu kujutise pindala suurenemine on tingitud lisa loogikast ehk koodist, mida peab kompilaator genereerima, et tagada õige nihe ümber määramise ajal ning ligipääs virtuaal tabelile programmi käimise ajal. Samuti tekib lisa initsialiseerimise kood klass eksemplari jaoks ehk kompilaator peab genereerima konstruktorisse lisa loogikat, et taga virtuaalse baas klassi eksemplari loomist ainult üks kord [15].

Joonis 14 *nD* on mitte teemant omadusega virtuaalse destruktoriga baas klassiga andmetüüp. *D* on Teemant omadusega virtuaalse baas klassiga andmetüüp. *Cr Dr Heap* on konstruktor ja destruktor objekti koostamisel kuhjal. *Base To Derived* on **dynamic_cast** baas andmetüübist tuletatud andmetüüpi. *Sideways* on horisontaalne **dynamic_cast**.



Joonis 14.Võrdlustestid. dynamic_cast toimingute võrdlus: virtuaalse baas klassiga objekti ja tavalise baas klassi objekti vahel, optimeeritud.

Horisontaalne **dynamic_cast**, näiteks klassist **B** klassi **C** Joonis 13, on kõige kulukam kuna lisanduvad toimingud ümber määramise valideerimiseks [14]. Tavalise (mitte teemant) klassi pärimise UML diagramm on toodud Joonis 15.



Joonis 15. UML. Lihtne pärimine.

4. Mitte staatilised andmed nii baas klassis kui ka pärijal Joonis 16.

```
 // non_standard_member_vars/include/lib.h
class Base {
public:
    int a{65535};
};

class NonStandard: public Base {
public:
    int b{65525};
};

class StandardClass {
public:
    int a{65515};
    int b{65505};
};
```

Joonis 16. C++. Mitte staatilised andmed nii baas klassis kui ka pärijal.

Joonis 16 toodud andmete jaotus klasside vahel ei lisa käitusaja ega mälukujutise pindala lisa kulusid. Seda näitab Joonis 17 toodud C++ keele koodi Assembleri listing Joonis 18.

```
// non_standard_member_vars/src/main.cpp
NonStandard ns;
fn(ns);
StandardClass sc;
fn(sc);

[[maybe_unused]] int nsa = ns.a;
[[maybe_unused]] int sca = sc.a;
```

Joonis 17. C++. Toimingud standard ja mitte standard paigutusega andmetüübi eksemplariga.

```
./asm_prettify.sh non_standard_member_vars | grep '^[0-9]* main' -A 35
42 main:
; proloog eemaldatud
; tekitame NonStandard
54      mov     DWORD PTR -24[rbp], 65535
55      mov     DWORD PTR -20[rbp], 65525
; edastame NonStandard ns to fn
56      mov     rax, QWORD PTR -24[rbp]
57      mov     rdi, rax
58      call    _Z2fnN24non_standard_member_vars11NonStandardE
; tekitame StandardClass
59      mov     DWORD PTR -16[rbp], 65515
60      mov     DWORD PTR -12[rbp], 65505
; edastame StandardClass sc to fn
61      mov     rax, QWORD PTR -16[rbp]
62      mov     rdi, rax
63      call    _Z2fnN24non_standard_member_vars13StandardClassE
; int nsa = ns.a
64      mov     eax, DWORD PTR -24[rbp]
65      mov     DWORD PTR -32[rbp], eax
; int sca = sc.a
66      mov     eax, DWORD PTR -16[rbp]
67      mov     DWORD PTR -28[rbp], eax
68      mov     eax, 0
69      mov     rdx, QWORD PTR -8[rbp]
```

Joonis 18. Asm. Toimingud standard ja mitte standard paigutusega andmetüübi eksemplariga.

5. Pärimine mitte standardse paigutusega klassist Joonis 19

```
// näidiskood
class NonStandardBase {
private:
  int a;

public:
  int b;
};

class NonStandard : public NonStandardBase {
};
```

Joonis 19. C++. Pärimine mitte standardse paigutusega klassist.

Sellisel koodil ei ole käitusaja ega mälukujutise pindala lisa kulusid.

Standardne paigutus annab teatud garantiisid klassi eksemplari mälu jaotuse kohta:

1. Etteaimatav objekti andmete täpne paigutus mälus.
2. Pärimise puhul on teada, et baas klass on kõige alguses mälus.

Sellistel omadusel on hulk kasulikke faktoreid:

3. Objekti võib kindlalt liidestada C teekidega ja teiste keeltega.
4. Otsene objekti kaardistamine riistvara registritesse.
5. Lihtsam andmete serialiseerimine.

C++ standard teegis on olemas mall *std::is_standard_layout* Joonis 20, mis aitab määrata kompileerimise ajal kas andmetüüp on standardse paigutusega:

```
// näidiskood
#include <type_traits>

class Standard {
  public: int a;
  float b;
};

class NonStandard {
  public: int a;
  private: float b;
};

int main() {
  static_assert(std::is_standard_layout < Standard > ::value);
  static_assert(false == std::is_standard_layout<NonStandard>::value);

  return 0;
}
```

Joonis 20. C++. std::is_standard_layout.

3.1.2 Lihtne andmetüüp

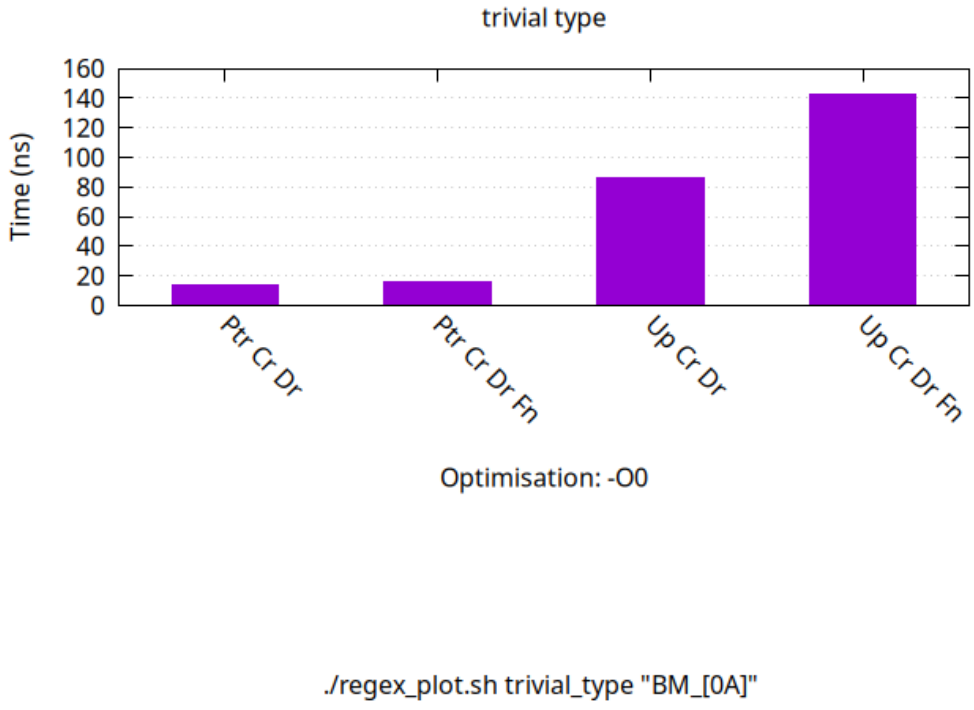
Lihtne ehk triviaalne andmetüüp C++ keeles on klass või *struct*, mille kohta kehtib järgmine [cppref]:

1. Ei oma virtuaalseid funktsioone.
2. Ei oma kasutaja poolt määratud konstruktorit.
3. Ei oma kasutaja poolt määratud destruktorit.
4. Ei oma kasutaja poolt määratud kopeerimise konstruktorit.
5. Ei oma kasutaja poolt määratud liigutamise konstruktorit.
6. Ei oma kasutaja poolt määratud kopeerimise operaatorit.
7. Ei oma kasutaja poolt määratud liigutamise operaatorit.

Kõikide mitte staatiliste andme liikmete suhtes on eelnevad 7 punkti tõesed.

Kui klass on tuletatud, siis kõik 7 punkti peavad olema tõesed ja baas klassi suhtes.

Mitte lihtsate andmetüüpide eksemplaride jaoks peab kompilaator genereerima lisa koodi sõltuvalt sellest, mis eri liikme funktsioonid on määratud ning mitte *default*. Juhul kui andmetüübile on kasutaja poolt määratud destruktor, siis peab kompilaator genereerima selle andmetüübi eksemplari jaoks destruktori väljakutse iga kord kui objektil toimub ulatusest väljumine. Seetõttu peab näiteks **std::unique_ptr** olema mõne võrra kulukam kui tavaline osuti Optimeerimata koodi võrdlustest näitab selgelt seda vahet Joonis 21. *Ptr Cr Dr* on tavalise osuti konstruktori ja destruktori väljakutsumine, teine tulemus *Ptr Cr Dr Fn* on alguses sama mis eelmine ning lisaks veel edastatakse osuti funktsiooni ning funktsiooni sees toimud osuti dereferentseerimine. *Up Cr Dr* ja *Up Cr Dr Fn* on samad toimingud kuid tavalise osuti asemel kasutatakse **std::unique_ptr**. Igale võrdlustestile lisandub 4 baiti mälu eraldamist *new* operaatori kaudu, mille kulu on antud võrdlustestis 10ns piires Joonis 38. Oluline on toimingute ajakulude erinevused primitiivse ja mitte lihtsa andmetüübi vahel.



Joonis 21. Võrdlustest: Toimingud. Tavaline osuti ja std::unique_ptr, optimeerimata.

Optimeeritud kujul on **std::unique_ptr** samaväärne primitiivse osutiga Joonis 36.

3.1.3 Liikme funktsioonid

Peale andmetüüpide, mis lisavad mälu koormust igale tekitatud objektile, võivad C++ klassid omada oma deklaratsioonis liikmetena ka funktsioone. C++ klassi eksemplarile lisavad mälu mahtu vaid andmetüüpidest liikmed (*data members*) Joonis 22 ja virtuaalsete liikme funktsioonidega lisandub üks osuti virtuaalsele tabelile.

```
// näidiskood
// g++ prog.cc -Wall -Wextra -O2 -std=gnu++17
#include <cstdint>
#include <iostream>
using namespace std;

class S {
  uint32_t a;
  uint32_t b;
  uint32_t c;
};

class Z {
  uint32_t a;
  uint32_t b;
  uint32_t c;
  uint32_t fn1() { return a; }
};

int main() {
  cout << sizeof(S) << endl; // väljund: 12
  cout << sizeof(Z) << endl; // väljund: 12
}
```

Joonis 22. C++. Liikme funktsioon

C++ klassi liikme funktsioonil on olemas teistsugune varjatud efekt. Liikme funktsiooni argumentide arvule lisandub alati veel üks osuti antud objektile. Selle argumendi muutuja nimetus on liikme funktsiooni sees **this**. Klassi objekti liikme funktsiooni välja kutsumine on esisatud Joonis 23.

```
// näidiskood
// g++ prog.cc -Wall -Wextra -O2 -std=gnu++17
#include <cstdint>
#include <iostream>
using namespace std;

class Z {
public:
  Z *fn3(uint32_t x) { return this; }
};

uint32_t fn4(uint32_t x) { return x; }

int main() {
  Z z;
  z.fn3(65535);
  fn4(65535);
  return 0;
}
```

Joonis 23. C++. Liikme funktsiooni rakendamine.

Joonis 23 C++ koodi Assembleri väljund *godbolt.org web* kasutaja liideses on Joonis 24.

```
; godbolt.org, x86-64, C++, GCC 12.1, -O0
16      main:
17      push    rbp
18      mov     rbp, rsp
19      sub     rsp, 16
; main fni stack frame’i algusest võetakse 1 bait
; tulemus on et rax register on nüüd osuti Z z eksemplarile
20      lea     rax, [rbp-1]
; System V ABI järgi läheb funktsiooni teine argument rsi registrisse
21      mov     esi, 65535
; System V ABI järgi läheb funktsiooni esimene argument rdi registrisse
22      mov     rdi, rax
23      call    Z::fn3(unsigned int)
; fn4 puhul rdi’sse läheb üks ainus argument.
24      mov     edi, 65535
25      call    fn4(unsigned int)
26      mov     eax, 0
27      leave
28      ret
```

Joonis 24. Asm. Liikme funktsiooni rakendamine.

Eri liikme funktsioonid (konstruktorid, destruktor, ülekoormatud operaatorid) käituvad **this** suhtes samamoodi nagu tavalised liikme funktsioonid. Kuid nende kasutamisel kompilaatori poolt genereeritud eri liikme funktsioonide asemel toovad endaga kaasa teistsuguseid varjatud efekte, mida täpsemalt uuritakse teemas 3.1.2.

3.1.4 Pärimine

Pärimine on tuntud mehhanism C++ valdkonnas, mille abil on võimalik kaasata loogika ja andmed ühest andmetetüübist teise.

Pärimine ilma virtuaalsete funktsioonideta C++ keeles ei põhjusta käitusaja aja, mälukasutust ega lõpp rakenduse mahu lisakulu. Kõik ümbermääramised (*cast*) on staatilised ja ära määramist tehakse kompileerimise ajal.

C++'s on ühel klassil võimalik pärida mitmest teisest klassist korraga. Mitme pärimise korral, kus mõlemas päritud klassis on andmed tekib varjatud efekt.

Joonis 25 klassi **CC** eksemplari andmete jaotust mälus ei ole standardiga defineeritud [12].

```
// näidiskood
// g++ prog.cc -Wall -Wextra -O2 -std=gnu++17
class A {
  /* omab andmeid */
};

class B {
  /* omab andmeid */
};

class CC : public A, public B {
  /* ... */
};

int main() { /* ... */ }
```

Joonis 25. C++. Minimaalne mitmene pärimine.

Antud klassi eksemplari C keele stiilis ümber määramine (*cast*) Joonis 26 on antud juhul implementatsiooniga määratud käitumine (*implementation defined behavior*) ehk GCC kompilaator määrab mis juhtub.

```
// multi_inherit/include/lib.h
class A {
public:
  uint32_t a{5};
};

class B {
public:
  uint32_t b{255};
};

class CC: public A, public B {
  /* ... */
};

// multi_inherit/src/main.cpp
int main() {
  CC c;
  A* a = (A*)&c;
  B* b = (B*)a;
  B* b2 = (B*)&c;
  cout << b->b << endl;  // väljund: 5. NB! B::b on 255
  cout << b2->b << endl; // väljund: 255
}
```

Joonis 26. C++. Mitte standard paigutsega andmetüübi ümbermääramine. Andmetüübis esineb mitmene pärimine.

Joonis 26 toodud C++ keele koodi Assembleri tulemus optimeerimata kujul koos selgitustega on Joonis 27.

```
// ./asm_prettify.sh multi_inherit/
8 main:
9 .LFB1761:
; proloog eemaldatud
; CC objekti loomine id’ga c
20  mov   DWORD PTR -16[rbp], 5
21  mov   DWORD PTR -12[rbp], 255
; laetakse c aadress rax registrisse
22  lea   rax, -16[rbp]
; c aadress laetakse a muutjasse
23  mov   QWORD PTR -40[rbp], rax
; laetakse aadress millele viitab a rax registrisse
24  mov   rax, QWORD PTR -40[rbp]
; a nimega osuti laetakse b muutujasse
25  mov   QWORD PTR -32[rbp], rax
; b2 part. load c object address to rax
26  lea   rax, -16[rbp]
; tehakse vajalik nihe et saada B::b liikme muutuja
27  add   rax, 4
28  mov   QWORD PTR -24[rbp], rax
; valmistatakse b->b cout fni jaoks, aga tegelt on viidatud A::a’le
29  mov   rax, QWORD PTR -32[rbp]
30  mov   eax, DWORD PTR [rax]
31  mov   esi, eax
32  lea   rax, _ZSt4cout[rip]
; valmistatakse b2->b cout fni jaoks
39  mov   rax, QWORD PTR -24[rbp]
40  mov   eax, DWORD PTR [rax]
41  mov   esi, eax
42  lea   rax, _ZSt4cout[rip]
```

Joonis 27. Asm. Mitte standard paigutsega andmetüübi ümbermääramine. Andmetüübis esineb mitmene pärimine.

3.1.5 Mitmekujulisus

C++’s on mitmekujulisus mehhanism, mille abil on võimalik kasutada teatud koodi osas erinevaid andmetüüpide ilma, et oleks vaja neid eristada. Süsteemi kujundamisel (*system design*) tasub läheneda, et ei ole olemas polümorfseid andmetüüpe, on olemas vaid koodi regioonid, kus andmetüüpe kasutatakse polümorfselt [16].

Klassikaline otsekohene tarkvaraarendus viis C++ keeles dünaamilise mitmekujulisuse puhul on tekitada abstraktne baasklass (mõiste *interface* on levinud teistes OOP keeltes) ning pärineda sellest. Sellisel juhul on pärinemine ära määratud konkreetse lõppklassi (leaf) deklaratsioonis, mis tähendab, et mitmekujulisus kusagil mujal koodis vajab sekkumist andmetüübi implementatsiooni detailidesse Joonis 28.

```
// näidiskood
class Base { virtual ~Base() {} };

class Derived : public Base {};
```

Joonis 28. C++. Baas klass virtuaalse destruktoriga.

Suurtes süsteemides, kus on palju alamsüsteeme ning nende vahel suhtlevaid komponente, pärinevad sellisel juhul osad klassid suurest hulgast teistest klassidest Joonis 29.

```
// https://github.com/OpenXRay/xray-16.git
// commit: bccfae0f3049b0ed67fcde8cd27576885785b968
// src\xrGame\Car.h
class CCar : public CEntity,
             public CScriptEntity,
             public CPHUpdateObject,
             public CHolderCustom,
             public CPHSkeleton,
             public CDamagableItem,
             public CPHDestroyable,
             public CPHCollisionDamageReceiver,
             public CHitImmunity,
             public CExplosive,
             public CDelayedActionFuse
```

Joonis 29. C++. Suur hulk *super*–klasse.

Antud lähenemisviis tähendab, et primitiivsete sisseehitatud andmetüüpide (nt. *char*) kasutamiseks polümorfses koodi regioonis on vajalik tekitada nende ümber mähis klass ehk andmetüüp, mis pärineb vajalikust baas klassist. Sama nõue tekib ka kõikide muude osapoolte andmetüüpide jaoks.

Sellise klassi eksemplar peab alati kaasas kandma baasiga seotud lisa koormust ka väljaspool polümorfset koodi regiooni. Iga baas klass lisab vähemalt ühe osuti suuruse, kuna GCC C++ implementatsioon tekitab virtuaal liikme funktsioonide kasutamisega osuteid virtuaal tabelile. Lisaks sellele käivad alati kaasas baas klasside andmed, juhul kui nad on olemas. Virtuaal funktsiooni väljakutse on kõige halvemal juhul kaks taset (*two levels of indirection*) kulukamad.

Ainuke pluss on, et andmetüübi deklaratsioonis on kirjeldatud kavatsus seda kasutada teatud mitmekujulises kontekstis.

Joonis 30 tuleneb veel üks miinus, mis kaasneb antud lähenemisel dünaamilisele mitmekujulisusele. Vajalik on kuhja kasutus ja opereerimine osutitega (*reference*

semantics), mis ei ole esmane ettenähtud lihtne viis teha toiminguid andmetüüpide eksemplaridega, nagu seda on opereerimine väärtustega (*value semantics*).

```
// näidiskood
class Base {
public:
  virtual ~Base() {}
  virtual void fn() = 0;
};

class Derived : public Base {
public:
  virtual void fn() {}
};

int main() {
  Base *b = new Derived{};
  b->fn();
  delete b;
  return 0;
}
```

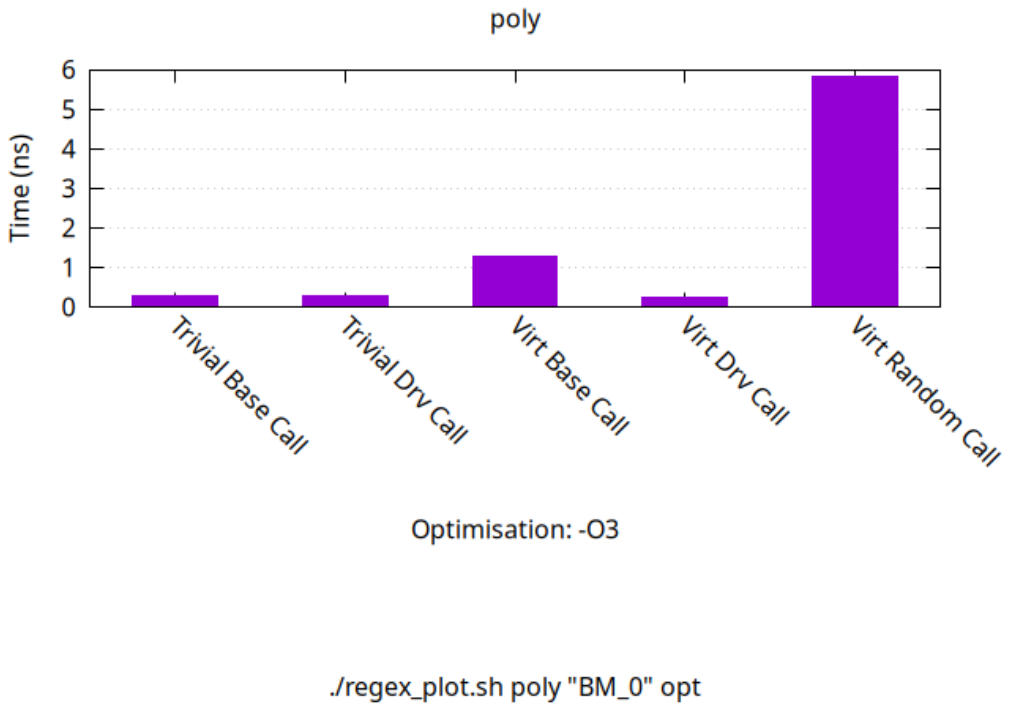
Joonis 30. C++. Pärimine ning virtuaalsed liikmefunktsioonid nõuavad toiminguid osutitegaJoonis 30Joonis 30 programmi Assembleri väljund godbolt.org *web* kasutaja liideses on Joonis 31:

```
; godbolt.org, x86-64, C++, GCC 12.1, -O0
1       main:
; proloog eemaldatud
; operaatorile new antakse ette 8 baiti, sest ainuke asi mis võtab antud
näites andmemahtu on osuti virtuaal tabelile
6       mov     edi, 8
7       call    operator new(unsigned long)
8       mov     rbx, rax
9       mov     QWORD PTR [rbx], 0
10      mov     rdi, rbx
11      call    Derived::Derived() [complete object constructor]
12      mov     QWORD PTR [rbp-24], rbx
13      mov     rax, QWORD PTR [rbp-24]
; osutamine virtuaal tabelisse
14      mov     rax, QWORD PTR [rax]
; osutamine esimesele elemendile virtuaal tabelist
15      mov     rdx, QWORD PTR [rax]
16      mov     rax, QWORD PTR [rbp-24]
17      mov     rdi, rax
18      call    rdx
19      mov     rax, QWORD PTR [rbp-24]
20      test    rax, rax
21      je      .L4
22      mov     esi, 8
23      mov     rdi, rax
24      call    operator delete(void*, unsigned long)
```

Joonis 31. Asm. Pärimine ning virtuaalsed liikmefunktsioonid nõuavad toiminguid osutitega.

See on võimalik varjatud efekt virtuaal funktsioonide kasutamisel ja antud lihtsa näitega on seda Assmebleri listingust näha vaid **-O0** GCC kompilaatori lipuga. **-O1,2,3,s,g** lippudega eemaldatakse sarnase lihtsa koodi tühja virtuaal funktsiooni väljakutse ehk toimub devirtualiseermine.

All on toodud virtuaalsete funktsioonide väljakutsetega seotud ajalised võrdlustestid (*benchmarks*) optimeeritud mälukujutise jooksutamisel. Andmetüübid on määratud feature-cpp *git* hoidlas failis **poly/include/lib.h** ja ajalise võrdlus testi kood on failis **poly/bench.cpp**. Kõik tulemused 0.25ns ringis on võrdelised pinus olevate lokaal objektide liikme funktsioonide rakendamisega. Nende puhul lõplik andmetüüp on kompileerimise ajal teada. *Virt Base Call* määrab ümber baas klassi enne liikme funktsiooni rakendamist. *Virt Random Call* on virtuaalsete liikme funktsioonide väljakutse polümorfses kontekstis, kus on teada vaid baas klass ning tuletatud andmetüüp varieerub korrapäratult iga testi käivitus. Viimase puhul peab tegema rohkem otsustamist käitusajal ning ajaline lisa kulu on ligi 20 korda kulukam võrreldes lihtsa (*Trivial*) kasutaja poolt määratud andmetüübi liikme funktsiooni väljakutsega.



Joonis 32. Võrdlustest. Erineva struktuuriga andmetüüpide liikme funktsioonide rakendamine, optimeeritud.

3.1.6 Käitusaja andmetüüpide informatsioon (RTTI)

Käitusaja andmetüüpide informatsiooni abil on võimalik käitusajal määrata kas väljend (*expression*) kuulub etteantud polümorfsete andmetüüpide hulka kasutades *dynamic_cast* operaatorit. *dynamic_cast* operaatorid tohib kasutada ka mitte polümorfsete andmetüüpidega juhul kui operaatori tulemust on võimalik tuletada kompilatsiooni ajal [17].

Teine RTTI operaator on *typeid*. *typeid* operaatoriga saadakse käitusajal andmetüübi infot ja võib ette anda ükskõik mis andmetüübi eksemplari. Staatiliste andmetüüpide puhul tehakse *typeid* töö kompileerimise ajal, mitmekujulise andmetüübi (juhul kui andmetüübi hierarhias on kasvõi üks virtuaal funktsioon) puhul käitusaja jooksul [18]. Mitmekujulise andmetüübi eksemplari puhul tagastatakse infot kõige tuletatud (*most derived*) andmetüübi kohta [19].

RTTI funktsionaalsus on mitte deterministlik [20], selle olemasolu lisab mitmekujuliste andmetüüpide eksemplaride virtuaaltabelistesse osuti andmetüüpi informatsioonile, mis

loomulikult suurendab lõpp rakenduse mahtu. Piiratud aja ja mäluga keskkonnas jooksva programmi koodi kompileerimisel on võimalus see funktsionaalsus välja lülitada. GNU GCC kompilaatoril on olemas selleks argument **-fno-rtti**.

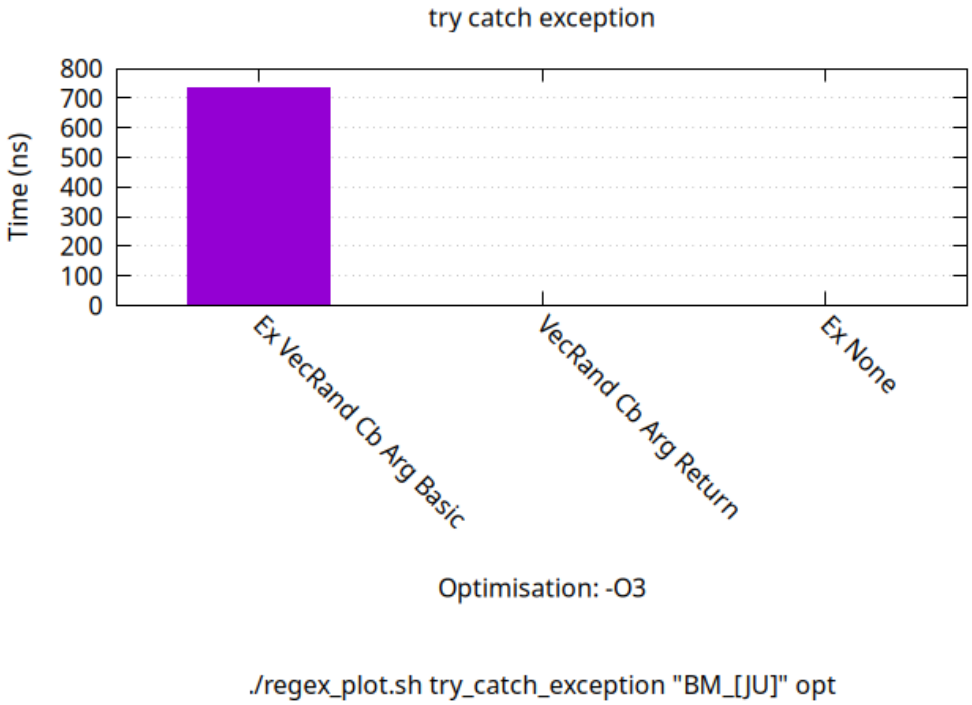
3.1.7 Erandid

Erandid on vea haldamis mehhanism, mille abil on võimalik eraldada vea tuvastamise koht vea haldamis asukohast. See on eriti vajalik olukordades kus vea toimumis kohas ei ole alati võimalik tegeleda veatöötlusega, n.t. klassi konstruktor. Konstruktorist ei saa tagastada veakoodi tavalise funktsiooni tagastus meetodil.

Erandite kasutavas koodibaasis peab tagama andmetüüpide eksemplaride ressursside vabastamist destruktoris (RAII) kuna erandi tekkimine tähendab alati pinu lahti kerimist (*stack unwinding*).

C programmeerimiskeele ajast olemasolevad alternatiivid on kas vea tagastus funktsioonist tagastuskoodina (või osuti argumendina) või globaalse lipu muutuja kasutamine (*errno*).

Joonis 33 tulemused viitavad, et erandi töötlemise toimingud tekitavad umbes 700ns lisakulu. Esimene mõõtetulemus *Ex VecRand Cb Arg Basic* määras minimaalse erandi sündmust. Erandi objektiks oli primitiivne *int* tüüp. Teine *VecRand Cb Arg Return* mõõtetulemus määras kõige muu loogika ajalist koormust, mida kasutati esimeses peale erandi tekitamist. Viimane tulemus *Ex None* määras ajakulu juhul kui lähtekoodis on määratud *try catch* blokk, aga erandi sündmust ei teki.



Joonis 33. Võrdlustest. Erandi lisakulu 700ns, optimeeritud.

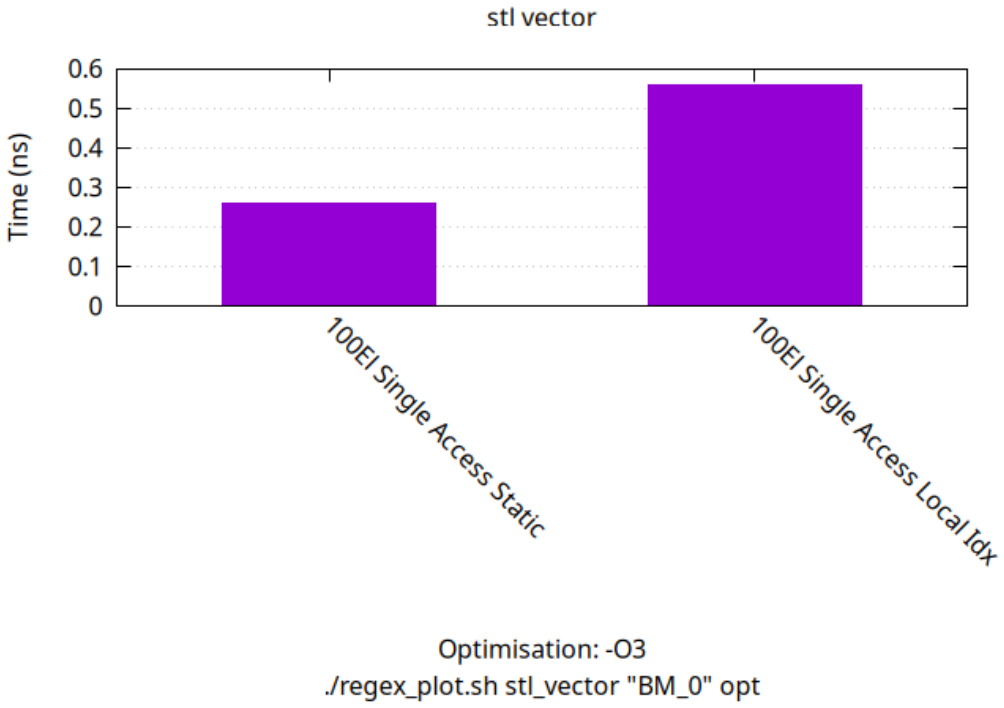
Antud töös kasutatud kood ei oma loogikat kus on olemas toimingud mille taga järel tekkivad erandjuhtumi sündmused. Välja arvatud antud peatükkis uuritud **./try_catch_exception** kataloogi sisu ning C++ standard teegi konteiner **std::vector** mis võib mälu eraldamisel tekitada **std::bad_alloc**. Autor eeldab antud töös, et **std::bad_alloc** erandi sündmuse tekkimise tõenäosus on praktiliselt null, mis tähendab, et erandid ei ole antud töös mõjuv omadus. Erandid ei tekita antud töö raames käitusaja aja lisakulusid.

3.2 Standard teek (STL)

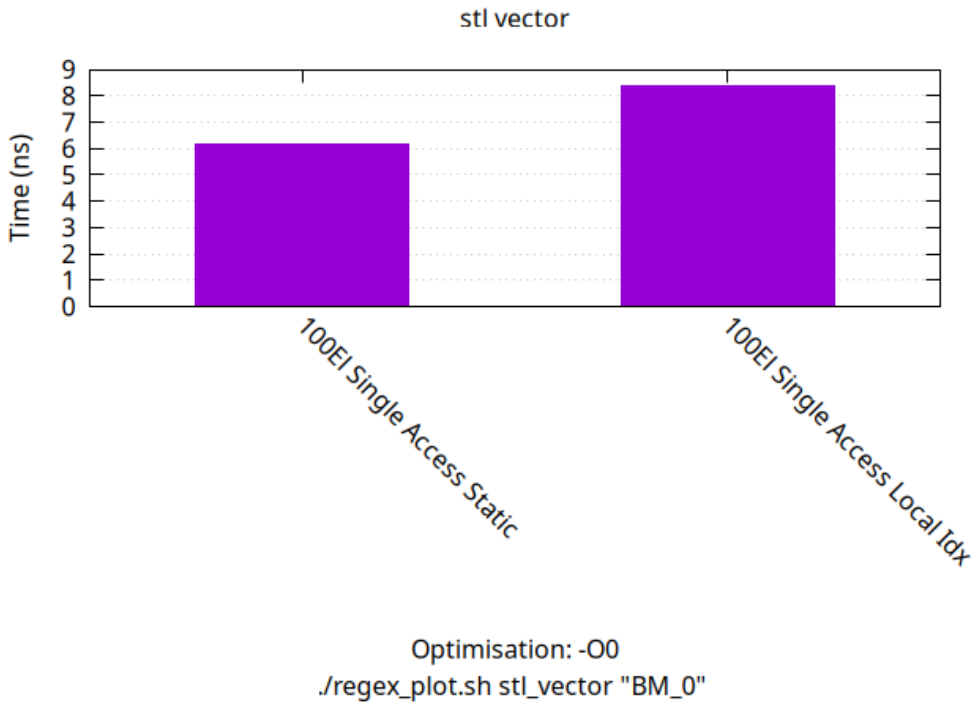
3.2.1 std::vector

std::vector on ainuke konteiner, mida kasutatakse antud töös aja mõõtmise koodis ja näidetes. Mitmekujulisuse ajaliste kulude mikro mõõtmisteks (*micro benchmark*s) on vaja teha baas klassi osutiga virtuaalsete funktsioonide väljakutseid ning põhiline on, et reaalne tuletatud andmetüüp oleks iga iteratsiooniga korrapäratu, et *branch predictor* ei saaks teha optimeerimist. Kirjeldatud loogika teostamiseks tehakse mõõtmiste ajal lisa toiminguid, milleks on indeksi järjestikune suurendamine ja rotatsioon. Joonis 34 näitab

kõigepealt ajakulu staatiliseks ligipääsuks koguaeg samas indeksiga, seejärel järjestikust ligipääsu 100 elemendi suuruse konteineri piires.



Joonis 34. Võrdlustest. std::vector järjestikune ligipääs ajalised kulud, optimeeritud.

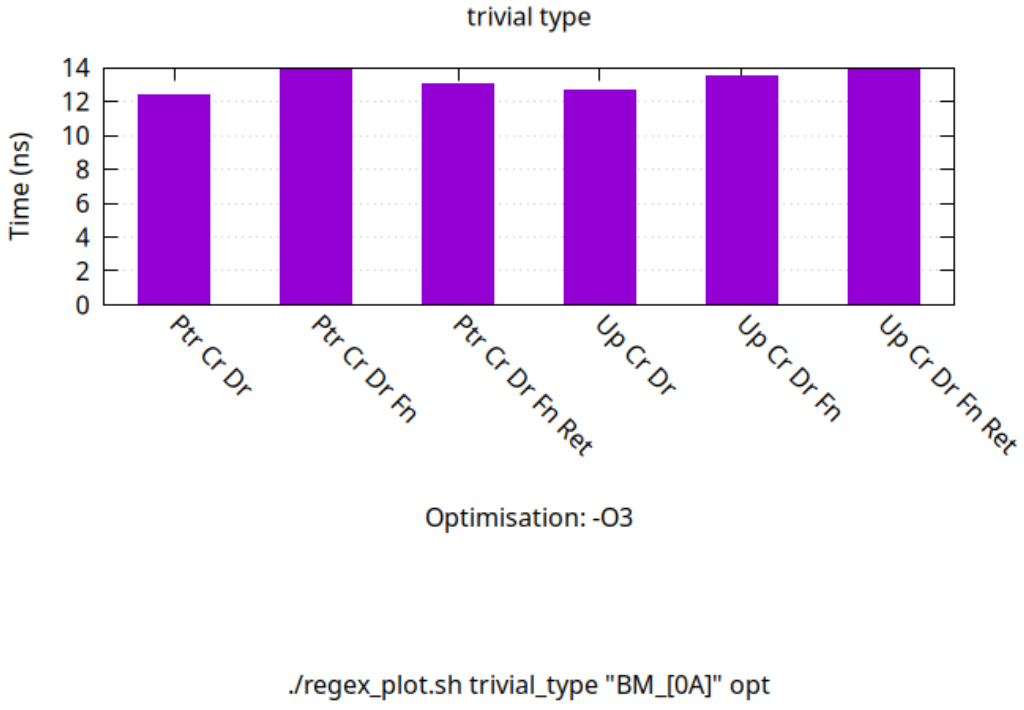


Joonis 35. Võrdlustest. std::vector järjestikune ligipääs ajalised kulud, optimeerimata.

3.2.2 std::unique_ptr

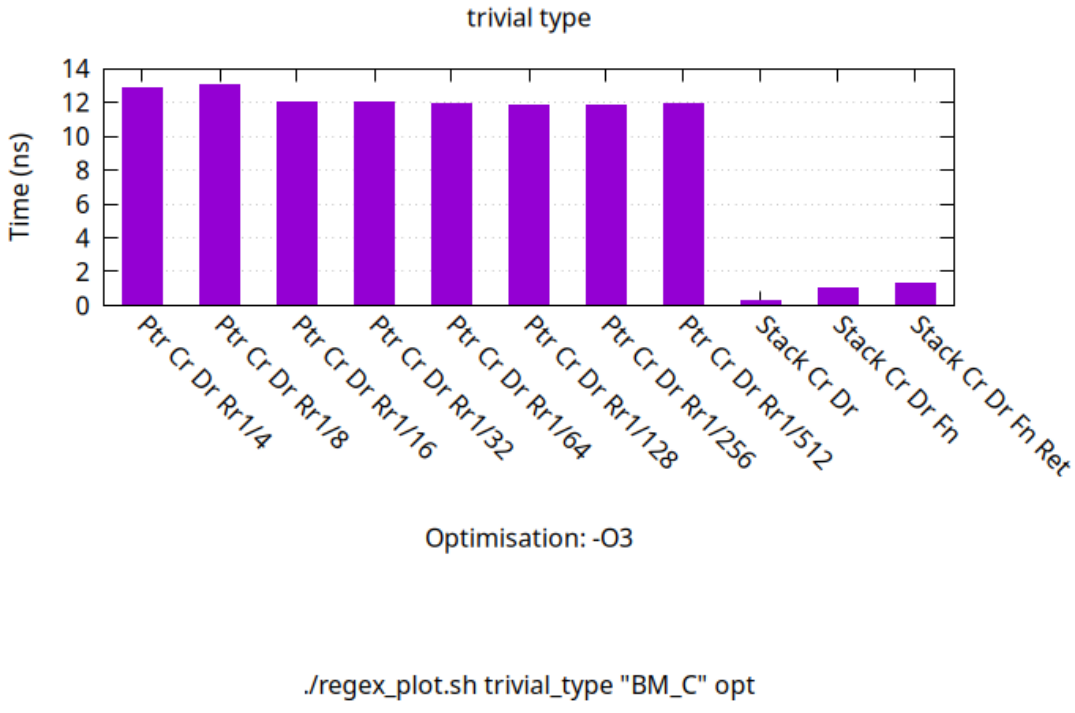
std::unique_ptr tagab automaatse mälu vabastamise, kui kontrollitav objekt väljub elutsüklist, ning tagab ka unikaalse omandiõiguse, mis tähendab, et korraga saab olla ainult üks *unique_ptr*, mis osutab antud objektile.

Ajalised võrdlustestid on toodud Joonis 36. *Ptr Cr Dr* on tavalise osuti konstruktori ja destruktori väljakutsumine, teine tulemus *Ptr Cr Dr Fn* on alguses sama mis eelmine ning lisaks veel edastatakse osuti funktsiooni. *Ptr Cr Dr Fn Ret* teeb peale funktsiooni ka funktsioonist tagastust. *Up Cr Dr*, *Up Cr Dr Fn* ja *Up Cr Dr Fn Ret* on samad toimingud kuid tavalise osuti asemel kasutatakse **std::unique_ptr**. Igale võrdlustestile lisandub 4 baiti mälu eraldamist *new* operaatori kaudu. See mälu eraldamine võtab optimeeritud koodi puhul suurema osa ajast (vt. Joonis 37) ning selle võrdlustesti tulemuste põhjal on võimalik järeldada, et optimeeritud kujul on **std::unique_ptr** praktiliselt identne tavalise osutiga (*raw pointer*). Praktikas kulub tavaliselt palju rohkem aega mälu haldamisele.



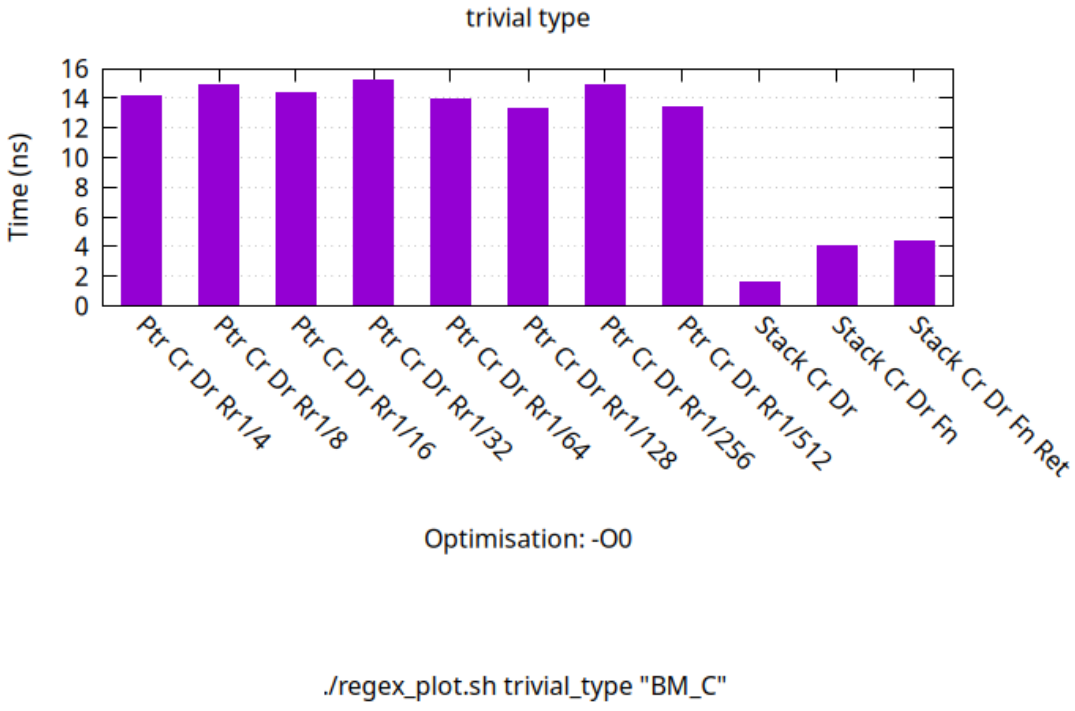
Joonis 36. Võrdlustest: Tavaline osuti ja std::unique_ptr toimingud, optimeeritud

Joonis 37 on toodud samad toimingud pinus 4 baidi kohta ning kuhjal on võrdluseks toodud, lisaks konstruktoritele, mälu eraldamine vahemikus 4 kuni 512, ning vahemiku suurendamine toimub kordaja 2 abil.



Joonis 37. Võrdlustest. Toimingud pinus ja kuhjal, optimeeritud.

Joonis 38 on võrdlused samad toimingud, aga optimeerimata mälukujutisega. Järeldus on et *google–benchmark* lisab muude toimingute hulka ligikaudu 10ns kui tehakse ühekordne mälu eraldus kuhjal.



Joonis 38. Võrdlustest. Toimingud pinus ja kuhjal, optimeerimata.

Lisaks tuleb pöörata tähelepanu et antud töös kasutusel oleva süsteemi operaator *new* kutsumisel kasutatakse **libstdc++** teegi deklaratsiooni mis kutsub **malloc**. *google-benchmark* teek ei täpsusta, mis on tulemus juhul kui testi käivitus võtab iga 1024 iteratsioon kordades rohkem aega kui eelmised.

4 C++ omaduste kombinatsioonid

Ajaliste võrdlustestide lähtekood täismahus järgneval lingil: https://github.com/marktomm/design-cpp

Kataloogi tee failini esimesel real koodi joonistes tähendab, et joonise kood asub eelnevalt toodud *git* hoidlas. Andmetüüpide suhteid illustreerivad diagrammid genereeritud *PlantUML* abil. Lähtekood samuti eelneval lingil.

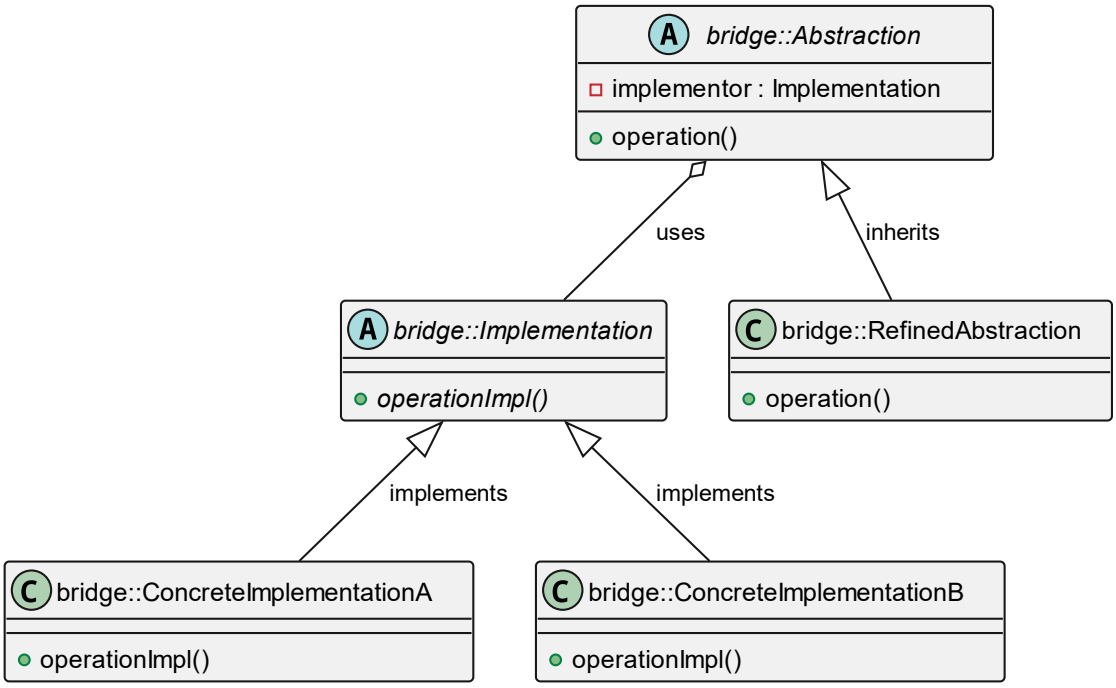
PlantUML legend:

- A täht sinine taust – abstraktne klass
- C täht roheline taust – terviklik klass
- Punase raamiga ruut – privaatse juurdepääsu täpsustajaga liikmemuutuja
- Rohelise taustaga ring ja ümarsulud lõpus – avaliku juurdepääsu täpsustajaga liikme funktsioon

4.1 Sild (*Bridge*)

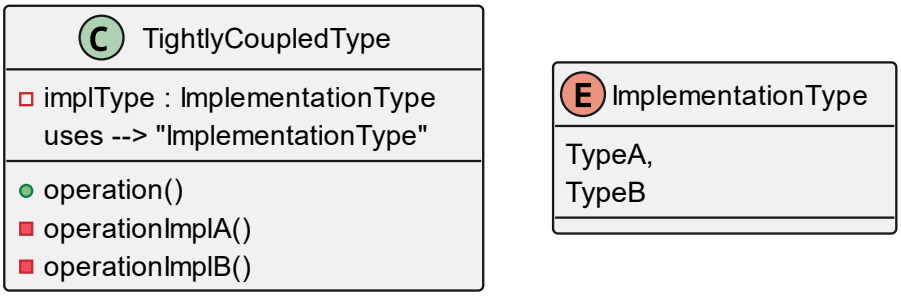
Kaks andmetüüpi. Ühte nimetatakse Abstraktsiooniks, teist Implementatsiooniks. Implementatsioon peab omama ühist *interface*'i, ehk omama abstraktset baas klassi. Viit Implementatsiooni *interface*'le on Abstraktsioonis liikme muutujana. Abstraktsioon võib samuti omada abstraktset baasi ning varieeruda. Ehk siis kaks eraldi andmetüüpide hierarhiat mille implementatsioon võib muutuda kuid peab olema kindel ühesuunaline seos kus Abstraktsioon on teadlik Implementatsiooni *interface*'st ning toetub sisemiselt sellele [21].

Joonis 39 on toodud UML diagramm mis illustreerib andmetüüpide suhteid *Bridge* kujundusmustri puhul.



Joonis 39. UML. Sild kujundusmuster.

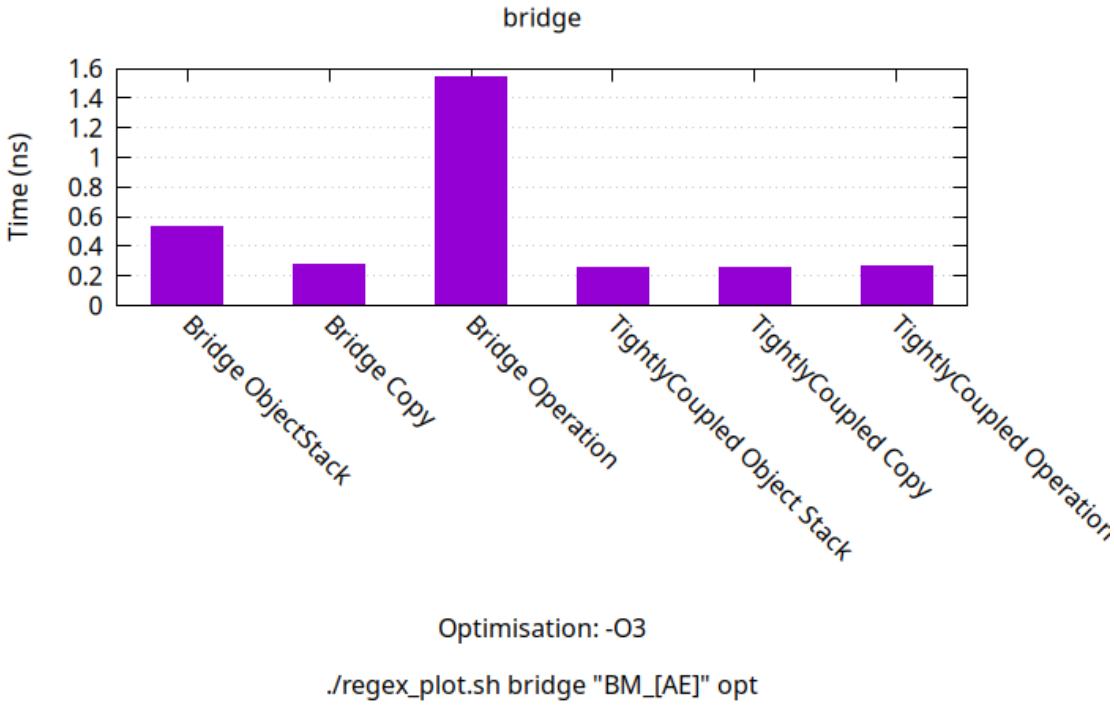
Joonis 39 ühe toimingu kohta tehakse kaks virtuaal liikme funktsiooni väljakutset. Tulemuseks kahes eraldi suunas arenev hierarhia. Kiirem kui paindlikum andmetüüp võib olla realiseeritud *enum*'i abil eristamisega.



Joonis 40. UML. Tihedalt seotud andmetüüp.

Tihedalt seotud andmetüübi puhul on lähtekoodis lisa loogika liikme funktsiooni **operation** sees mille põhjal tehakse kindel kumb **operationImpl** kasutada.

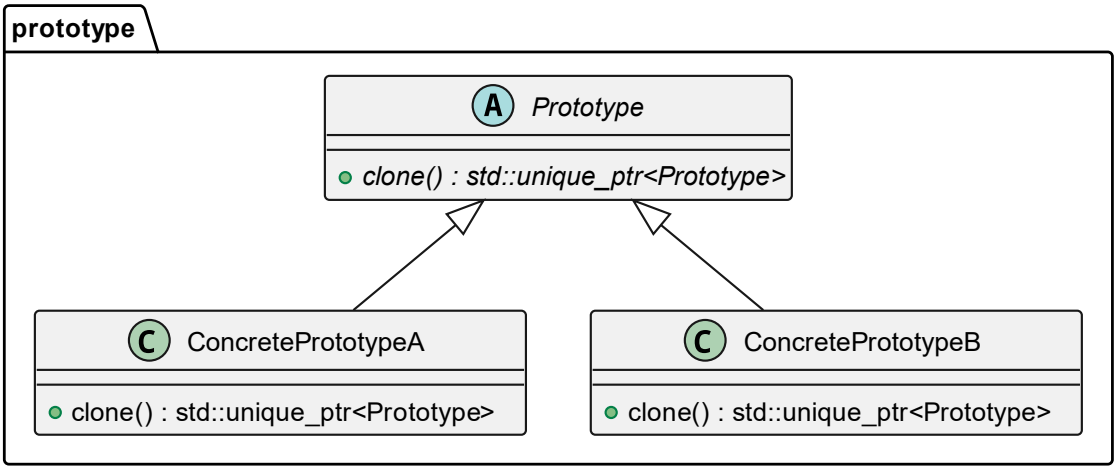
Ajalised võrdlustestid Joonis 41 näitavad, et vaatamata lisa keerukusele **TightlyCoupled** liikme funktsiooni **operation** sees on *Bridge* kujundusmustrit kasutav kood viis korda kulukam ajaliselt.



Joonis 41. Võrdlustest. Sild kujundus mustri objekti käitusaja aja kulu, optimeeritud.

4.2 Prototüüp (*Prototype*)

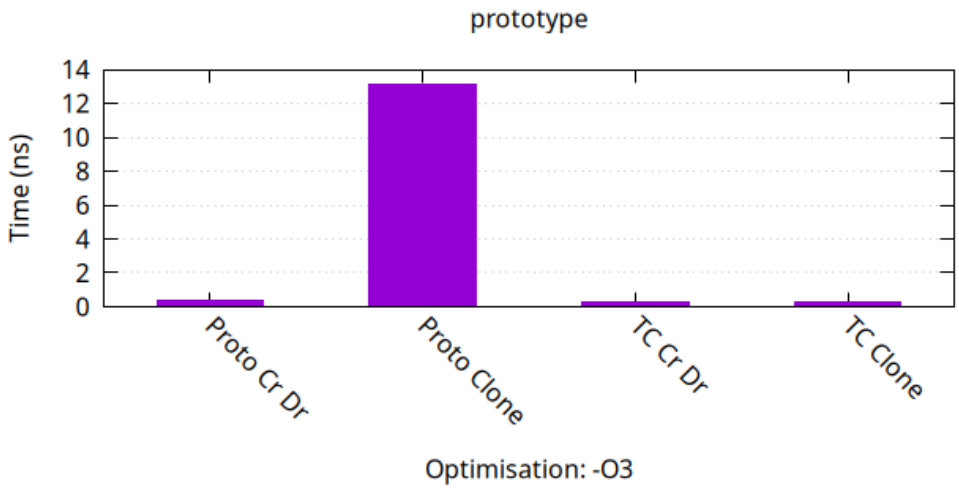
Baas klass omab virtuaalset liikme funktsiooni **clone**, mille ülesanne on tagastada endast koopiaid. *Sub*-klass vajadusel laiendab seda funktsiooni Joonis 42.



Joonis 42. UML. Prototüüp kujundusmuster.

Omab kasu kui on tegemist andmetüübiga mille struktuuris esineb pärimine ja koopiat on vaja teha kontekstis kus on olemas osuti või viit baasklassile (*sub*-klass on teadmata) [21].

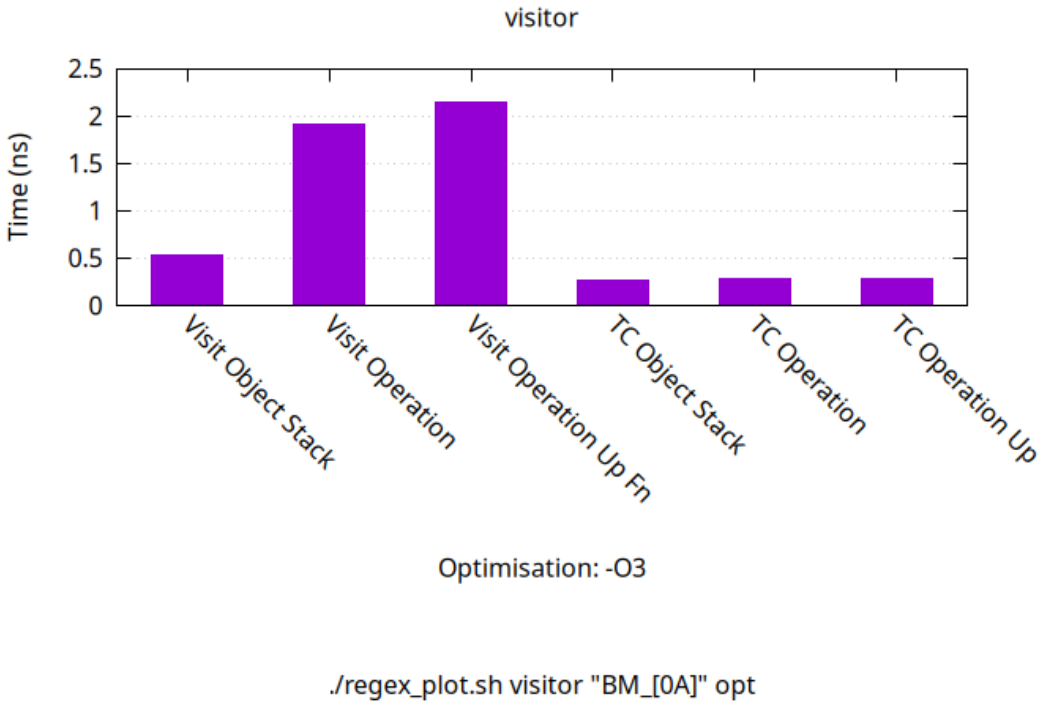
Kõik toimub *stack*'il. *Proto Clone* tagastab **std::unique_ptr**. Optimeeritud koodis ei tekitanud see kõige lihtsamate toimingutega lisa kulusid . Lisa kulud on kuna kloonimine toimub alati kuhjal.



Joonis 43. Võrdlustest. Prototüüp omadusega minimaalse objekti kopeerimise aja kulu, optimeeritud.

4.3 Külaline (*Visitor*)

Kasulik kui andmetüübi struktuur on konstantne ning vajalik on varieeruv hulk algoritme [21]. See tekitab lisa käitusaja aja koormust nendele toimingutele mille arvuga tahetakse varieeruda. Joonis 44 toodud ajaline lisa koormus *Visit Operation* tekib topelt virtuaal funktsioonide väljakutsete tõttu kui kõik eksemplarid asuvad stackil (võrreldes tavalise väljakutsega *TC Operation*). *Visit Operation Up Fn* on toiming **std::unique_ptr**'ga.



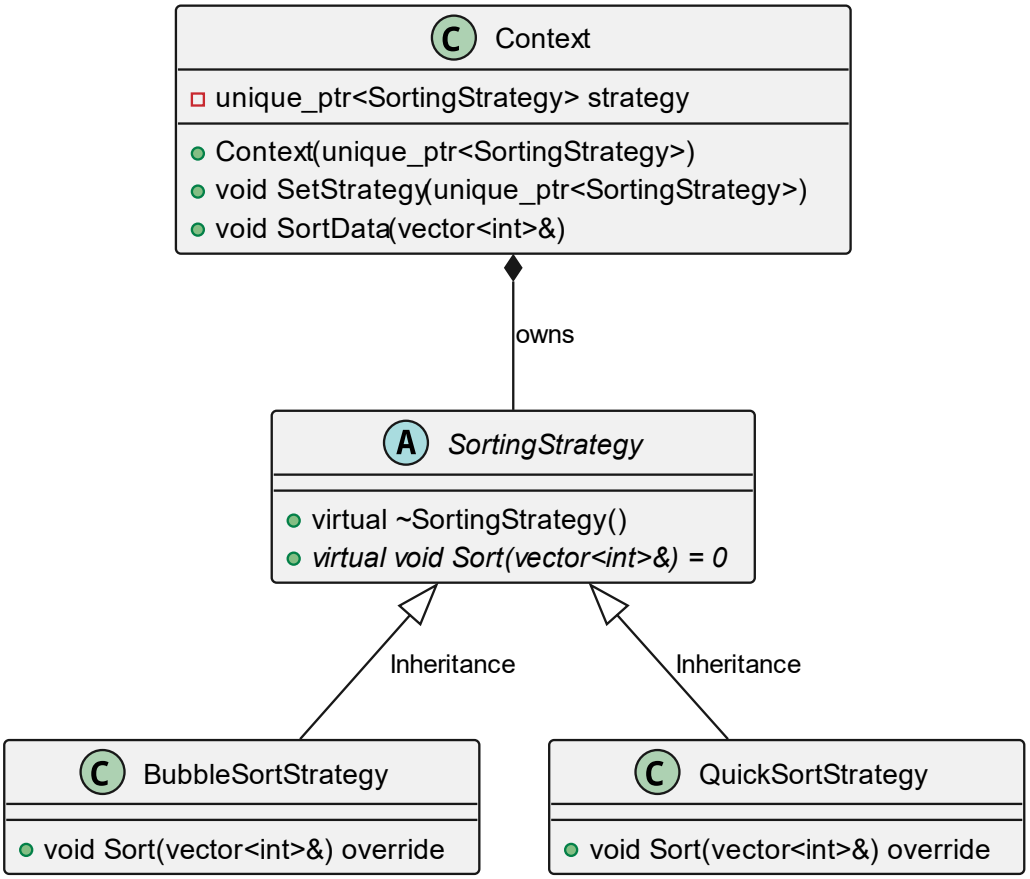
Joonis 44. Võrdlustest. Külaline kujundusmustri toimingud, optimeeritud.

Mälu lisa koormus käitusajal kuna tekitatakse eraldi virtuaalse baas klassiga struktuuri omav andmetüüp (*Visitor*) ning tema iga tema eksemplar on üks lisa osuti suurune objekt.

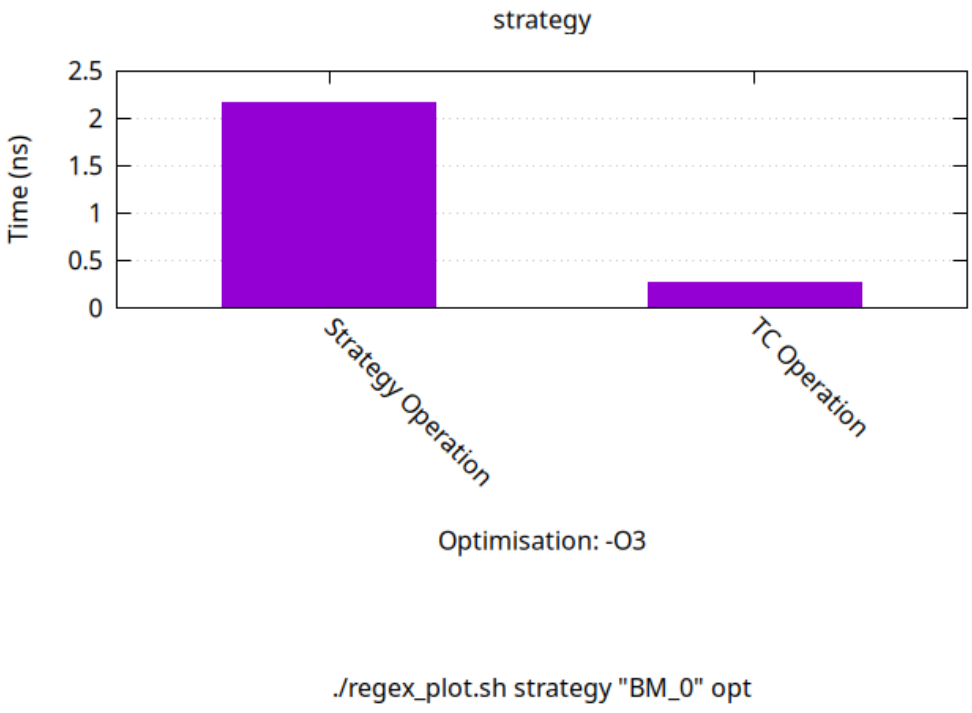
4.4 Strateegia (*Strategy*)

Ühe andmetüübi (*Strategy*) struktuur sisaldab puhtalt virtuaalset baasi ja puhtalt virtuaalset meetodit, mis aktsepteerib 0 kuni n argumenti. Teine andmetüüp (Context) omab liikmemuutujat, mis hoiab viidet *Strategy* andmetüüpidele, ning tavaliselt aktsepteerib strateegiaid esialgu konstruktori kaudu.

Põhitoiming mis rakendab käitusaja jooksul määratud strateegia on Joonis 45 **Context::SortData**. Joonis 46 on selle toimingu silt *Strategy Operation*. Lisa topelt väljakutse virtuaal liikme funktsiooni on kaks taset kulukam. **Context** klassi objekt oli pagutatud pinus, **SortingStrategy** objekt oli tekitatud kuhjal.



Joonis 45 UML. Strateegia kujundusmuster.



Joonis 46. Võrdlustest. Strateegia kujundusmustri toimingud, optimeeritud.

5 Käitusaja mitmekujulisus

Ajaliste võrdlustestide lähtekood täismahus järgneval lingil: https://github.com/marktomm/poly-cpp

Kataloogi tee failini esimesel real koodi joonistes tähendab, et joonise kood asub eelnevalt toodud *git* hoidlas. Andmetüüpide suhteid illustreerivad diagrammid genereeritud *PlantUML* abil. Lähtekood samuti eelneval lingil.

PlantUML legend:

- A täht sinine taust – abstraktne klass
- C täht roheline taust – terviklik klass
- Punase raamiga ruut – privaatse juurdepääsu täpsustajaga liikmemuutuja
- Rohelise taustaga ring ja ümarsulud lõpus – avaliku juurdepääsu täpsustajaga liikme funktsioon

Mitmekujulisus võimaldab erinevaid objekte käsitleda ühise alamtüübi objektidena, võimaldades ühel liidesel esindada erinevaid andmetüüpe.

Käitusaja mitmekujulisus ehk dünaamiline polümorfism võimaldab konkreetsete andmetüüpide määramist edasi lükata käitusajani.

Iga lähenemine omab ühe või mitu omadust:

1. Struktuuri laiendatavus
2. Funktsionaalsuse laiendatavus
3. Väärtuste semantika (*value semantics*)
4. Vaba tuletatud andmetüüp. S.t. tuletatud andmetüüp ei pärine otseselt ühisest alamtüübist kui vaadata lähtekoodi

Lisaks sellele varieerub iga lähenemine käitusaja kiiruse poolest andmetüübi eksemplari:

1. loomisel (konstruktor)
2. kopeerimisel
3. liigutamisel
4. vabastamisel (destruktor)

5. algoritmide väljakutsumisel

5.1 Enum

1. Mitte abstraktne baas klass mis omab andmeid päritud andmetüübist Joonis 47
2. Pärija määrab baas klassile konstruktoris oma andmetüübi Joonis 48
3. Algoritmid kutsutakse välja eraldiseisvates funktsioonides mille loogikaks on tuletatud andmetüüpide määramine Joonis 49

```
// enum_type/include/lib.h
enum PortType { tcp, serial, end_ = 0xFFFFFFFF };
class Port {
public:
    explicit Port(PortType pt) noexcept: type_{pt} {}
    PortType GetType() const noexcept { return type_; }
private:
    PortType type_;
};
```

Joonis 47. C++. Enum baas klass

```
// enum_type/include/lib.h
class TcpPort final: public Port {
public:
    explicit TcpPort(std::string ip, uint16_t pn) noexcept;
    ~TcpPort() = default;

    void Read(BufferData&) const noexcept;
/* ... */
};

class SerialPort final: public Port {
/* ... */
};
```

Joonis 48. C++. Enum alam klassid

```
// enum_type/include/lib.cpp
void read(TcpPort const& port, BufferData& data) noexcept { port.Read(data);
}

void readPort(unique_ptr<Port> const& port, BufferData& output) noexcept {
    switch (port->GetType()) {
    case tcp:
        read(*static_cast<TcpPort const*>(port.get()), output);
        break;
    case serial:
        read(*static_cast<SerialPort const*>(port.get()), output);
        break;
    default:
        break;
    }
}
```

Joonis 49.C++. Enum eraldiseisvad funktsioonid mis tegelevad alamtüübi määramisega.

Kasutajasõbralikud omadused on funktsionaalsuse laiendatavus.

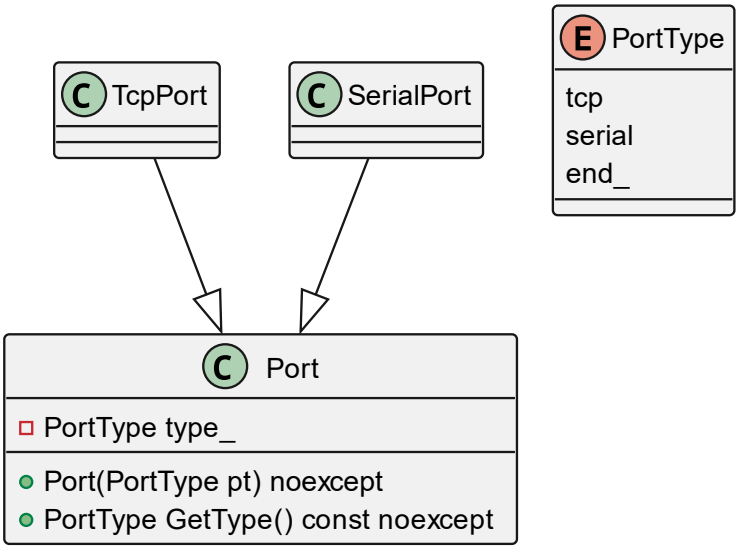
Kuna algoritmid ei ole registreeritud virtuaaltabelis siis on antud võttega võimalik teha alati juurde uut funktsionaalsust ilma, et tekiks ABI kokkusobimatus.

Sama asi peab paika ka uute andmetüüpide jaoks. Kuid andmetüüpidega tuleb arvestada kahe asjaga:

1. Peab uuendama kõik eraldiseisvad funktsioonid Joonis 49 mis tegelevad algoritmide valimisega, s.t. olemasoleva koodi modifitseerimine (OCP struktuur). Kompilaator antud juhul hoiatada ei saa.
2. Kui ABI tagasiühilduvus on nõutud, siis peab ette arvestama enumeratsiooni suurusega. Antud näites on lõppu lisatud liige end_, mis teeb enumeratsiooni vähemalt 4 baidi suuruseks.

Antud lähenemine on kõige otsekohesem võte mitmekujulisusele ning kõige lähedasem oma kiiruse poolest tavalise funktsiooni väljakutsele Joonis 50.

Kõige kiirem käitusaja mitmekujulisuse mehhanism selle arvelt, et C++ keele mehhanismid, s.t. pärimine, tegelevad ainult struktuuriga. Algoritmide valikuga tegeletakse käsitsi.



Joonis 50. UML. Enum ei oma abstraktset baas klassi.

5.2 Klassikaline OOP

Enim tuntud mitmekujulisuse võte. Mitmekujulisus paneb paika algoritmid ehk abstraktne baas klass, milles tuleb kirjeldada funktsionaalsuse liides puhtade virtuaal (*pure virtual*) funktsioonide näol Joonis 51.

```
// oop/include/lib.h
class Port {
public:
    Port() = default;
    virtual ~Port() = default;
    virtual void Write(BufferData const&) noexcept = 0;
}
```

Joonis 51. C++. Klassikaline OOP virtuaalse destruktoriga baas klass.

Kasutajasõbralikud omadused on struktuuri laiendatavus.

Kuna algoritmid määratakse virtuaalsete liikme funktsioonidega, siis ei ole võimalik hiljem lisada hierarhiasse uusi algoritme, selleks et kasutada neid mitmekujulisust nõudvas kontekstis ilma et ei peaks olemasolevat koodi täiendama (OCP funktsionaalsus) ja lõhkuma ABI. Üks positiivne osa on see, et uute algoritmide lisamisega tagab kompilaator, et kõik andmetüübid hierarhias neid implementeerivad.

Uute andmetüüpide lisamiseks hierarhiasse on see lähenemine sobiv.

5.3 Külastaja muster

Topelt virtuaal lähetamisega mitmekujulisusega muster, s.t. võrdlemisi aeglane käitumise ajal. Uute andmetüüpide lisamine nõuab ABI piiride ületamist (OCP struktuur) ehk muster pakub kinnist andmetüüpide hulka.

Võimalus eraldada algoritmid implementatsioonist ja lisada vabalt uusi.

Nõuab teatud järjekorda koodis:

1. külastaja eeldeklaratsioon Joonis 52
2. lõppklasside deklaratsioonid Joonis 53
3. külastajate vastuvõtjad (acceptors) on defineeritud Joonis 54
4. algoritmid on defineeritud Joonis 55

```
// visitor/include/lib.h

class Visitor;
```

Joonis 52. C++. Külastaja eeldeklaratsioon.

```
// visitor/include/lib.h
class TcpPort final: public Port {
public:
```

Joonis 53. C++. Alamtüüpi deklaratsioonid

```
// visitor/include/lib.h
    void accept(Visitor const&) const noexcept override;
    void Read(BufferData&) const noexcept;
};
```

Joonis 54. C++. Külastajate vastuvõtjad

```
// visitor/include/lib.h
class Read final: public Port::Visitor {
public:
    void visit(TcpPort const&) const noexcept override;
};
```

Joonis 55. C++. Algoritmide deklaratsioonid

Vastasel juhul on olukord, kus kõik kood sõltub ja teab teistest.

Kasutajasõbralikud omadused on funktsionaalsuse laiendatavus.

5.4 std::variant

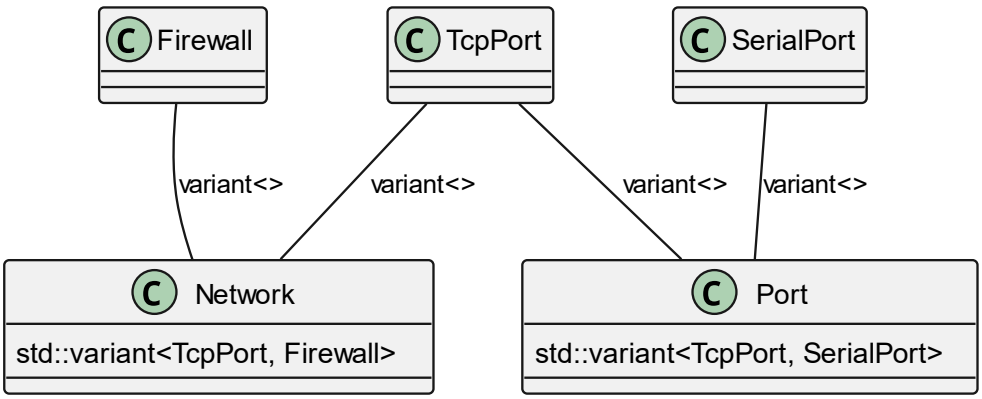
Toetab lõplikku andmetüüpide hulka ja avatud algoritmide arvu. Standardi poolt määratud külastaja mustri implementatsioon GNU GCC **libstdc++** poolt.

Kasutajasõbralikud omadused:

1. Toetab väärtuste semantikat
2. Vaba tuletatud andmetüüp
3. Üks andmetüüp võib omada mitut ühist alamtüüpi Joonis 56. Sisuliselt nagu mitmene pärinemine Joonis 57
4. Funktsionaalsuse laiendatavus

```
// variant_t/include/lib.h
// 3. vt TcpPort
class TcpPort {
};

class SerialPort {
};
using Port = std::variant<TcpPort, SerialPort>;

class Firewall {
};
using Network = std::variant<TcpPort, Firewall>;
```

Joonis 56. C++. std::variant mitu alamtüüpi



Joonis 57. UML. std::variant mitu alamtüüpi.

5.5 Strateegia muster

Algoritmide eraldamine struktuurist kasutades sõltuvuste süstimist konstruktoritesse osutite abil Joonis 58.

Kasutajasõbralikud omadused on funktsionaalsuse laiendatavus.

Lihtsustab algoritme muutmist andmetüüpi eksemplaris käitusajal.

```
// strategy/src/main.cpp
SerialPort p1("/dev/ttyUSB1", SyslogReadStrategy{}, SyncWriteStrategy{});
p1.SetWriteStrat(UnbufferedWriteStrategy{});
```

Joonis 58. C++. Strateegia orienteerub võimalusele muuta algoritme käitusajal.

5.6 Type Erasure

Nagu **std::variant** kuid lisaks veel toetab piiramatu arvu alamtüüpe.

1. Toetab väärtuste semantikat
2. Vaba tuletatud andmetüüp
3. Üks andmetüüp võib omada mitut ühist alamtüüpi
4. Funktsionaalsuse laiendatavus
5. Struktuuri laiendatavus

Realiseerimiseks peab kasutama kolm kujundus mustrit: *Bridge* + *Prototype* + *External Polymorphism* Joonis 59.

```
// type_erasure_up/include/lib.h
struct concept_t {
    virtual ~concept_t() = default;
    virtual void _read(BufferData&) const noexcept = 0;
};
template<typename T> struct model final: concept_t {
    model(T x) : _data(std::move(x)) {}
    void _read(BufferData& output) const noexcept override {
        read(_data, output);
    }
    T _data;
};
// auto x = model<int>();
// x->_read(BufferDataVar); // vabalt olev funkstioon read on vajalik
```

Joonis 59. C++. Väline mitmekujulisus

Väline mitmekujulisus toetab vabu tuletatusd andmetüüpe. Aga väärtuste semantika puudub.

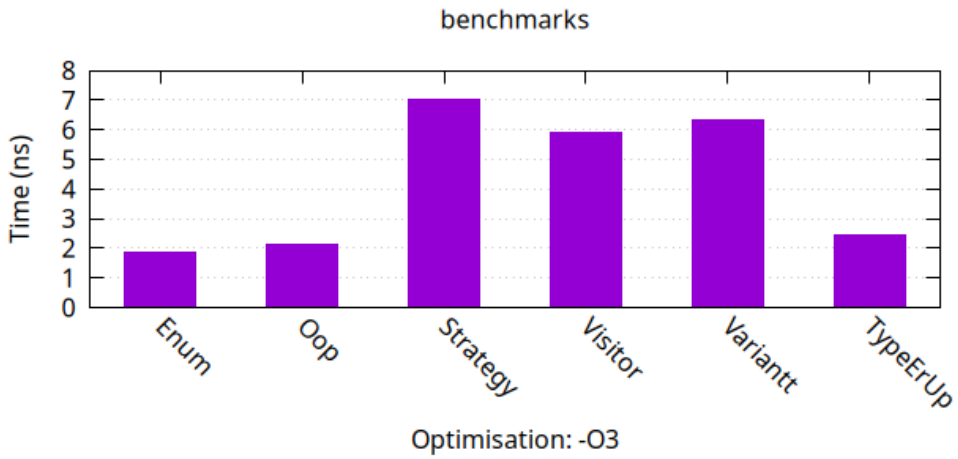
Kui lisada *Brigde* kujundus mustrit liikmemuutujana andmetüübiga **std::unique_ptr** baasklassile, millel on alamtüüp mall konstruktoriga, siis on võimalik lisada tugi ka väärtuste semantikale.

```
class Readable {
public:
    template<typename T>
    Readable(T x) noexcept: _self(std::make_unique<model<T> >(std::move(x)))
{}

// eri liikme funktisiooni kasutavad Prototype kujundusmustrit. Eemaldatud
// näidiskoodist.

// sõber funktsioonid eemaldatud näidiskoodist.

private:
    struct concept_t {
    };
    template<typename T>
    struct model final: concept_t {
    };

    // GOF: Bridge kujundusmuster
    std::unique_ptr<concept_t> _self;
};
```

Joonis 60. C++. Type Erasure

5.7 Käitusaja mitmekujulisuse võtete võrdlustestid

Joonis 61 on toodud keskmine aeg, et teha ühe liikme funktsiooni rakendamise toiming polümorfses kontekstis iga antud peatükkis toodud võtte kohta. Lisa koormuseks on 0.5ns **std::vector** järjestikune lugemine lokaalse indeksiga Joonis 34. *Enum* on kõige kiirem, seda võib võrrelda C keelele omase käsitsi tehtud lahendusega. *Oop* toimib väga lähedaselt *Enum*'iga ning palju kiirem võrreldes muude lahendustega, kus on andmetüüpides lähtekoodis otse kasutatud baas klass virtuaalse desktruktoriga. *Variantt* on tulemus, kus esines **std::variant**. C++ standardi poolt määratud andmetüüp, mis teostab Külastaja võtte. Esialgelt oli eeldus, et GNU GCC kompilaator peaks saama lihtsamad toimingud optimeerida oma standard teegi suhtes sama hästi kui oli toimunud devritualiseerimine *Oop* puhul. *TypeErUp* on *Type Erasure* võtte aja näitaja, ning on paari tsükli võrra kulukam kui *Oop* arvatavasti vaba funktsiooni lisa väljakutse tõttu.



./regex_plot.sh benchmarks "BM_Y" opt

Joonis 61. Võrdlustest. Liikme funktsiooni rakendamise toiming mitmekujulisuse võtetel.

6 Kokkuvõte

Ühelt poolt C++ annab hulga uusi omadusi, et oleks võimalik standardiga ette nähtud omaduste abil rakendada mitemekujulisus. Kõige otsekohesemalt ainult pärimisega mis lõppkokkuvõttes viib objektide jaoks mälu eraldamiseni kuhjal. See tähendab, et puudub väärtuste semantika, millele toetub näiteks standard malli teegi (STL) **std::vector** .

```
// näidiskood
vector<uint32_t*> x;
x.push_back(new uint32_t{1});
x.push_back(new uint32_t{1});
x.push_back(new uint32_t{1});
for (auto i : x) {
    cout << i << " "; // väljund: 0x0ffd9990 0x0ffd9986 0x0ffd9980
}
```

Joonis 62. C++. Osutite semantika ja std::vector.

Väljundiks ei ole „1 1 1“, vaid on kolm mälu aadressi. See tähendab, et kasutades C+ keele mitmekujulisuse saavutamiseks *out of the box* omadusi nagu virtuaalsed liikme meetodid baas klassis, kaob nendel andmetüüpidel võimalus kasutada teisi standardi poolt pakutud valmislahendusi kõige otsekohesemal moel.

Selleks, et C++ keeles oleks võimalik andmetüüpi kasutada mitmekujulisust vajavas kontekstis (s.t. kus on objektide kogumi jaoks teada vaid baas klass) sellisel moel, et ei peaks alam–andmetüüpi definitsiooni koormama teadmisega mis polümorfses keskkonnas on sead tarvis kliendi koodil kasutada, peab rakendama mitu kujundus mustrit (*Bridge* ja *Prototype*) ning lisaks veel põhi andmetüüpi siseselt ka OOP võtet nimetusega *External Polymorphism*. Võib vaielda, et kuigi C++ keel pakub hulga abstraktsiooni mehhanisme erinevate võtete saavutamiseks, siis osad kõige rohkem kasulike funktsioone omavad asjad nagu tüüpide kustutamine (*type erasure*), on vaja kolm *high–level* keerukust omavahel kokku siduda. **std::variant** pakub head alternatiivi.

Ajaliste kulude uurimise käigus selgus, et optimeeritud kujul on kõige kulukamate mitmekujulisus võtete toimingute lisa kulu ligikaudu 10 kordne võrreldes minimalistliku C keele taolise alternatiiviga. See tähendab, et iga pöördumine polümorfses kontekstis

objekti poole virtuaalse liikme funktsiooni rakendamisega võib olla 5ns ulatuses kulukam antud katse arvuti peal Tabel 1. See tulemus viitab, et optimeeritud mälu kujutise puhul on mitmekujulisus realiseeritud x86-64 arhitektuuri peal loetud arvuga masinakäske ning mõned mitte standard paigutusega ja mitte lihtsad standard teegi mallid, nagu **std::unique_ptr**, omavad sarnast lisa kulu primitiividega.

Käesoleva töö tulemusena jõudsime järeldusele, et ükskõik mis dünaamilise mitmekujulisuse võtte rakendamisel on puhtalt polümorfsete mehhanismide lisa kulu vahe kordades suurem kui C keelele omane lihtne lahendus, kuid samas on tegemist vaid mõningate nanosekunditega. Tulemuseks peab praktikas optimeerimisi läbi viima antud järjekorras: andmestruktuurid, algoritmid, mäluhaldust ja alles seejärel pöörata tähelepanu, et teatud andmetüübid võtavad lisa baidi jagu mälu virtuaal tabeli osuti jaoks ning tekib topelt kulu iga liikmefunktsiooni väljakutsega.

Muud teemad, mis on antud töö raames asjakohased, kuid ei olnud uuritud vajalikus ulatuses:

- Staatiline mitmekujulisus
- Staatiline *vs* dünaamiline linkimine
- Muudatuste/laiendatavuste mõju mälukujutise ABI'le
- Väärtuste kategooriad (*value categories*)
- RVO ehk *Return Value Optimisation*
- Mälu haldamine ja *new* operaatori ümber määramine
- Andmetüübi mälu paigutus strateegiad
- Mälukujutise pindala ja mälukujutis kui andmetüüpide kogum
- Võrdlustestid muudel kompilaatoritel ja arhitektuuridel
- Erandjuhud pinus

Kasutatud kirjandus

[1] D. Saks, "Meeting Cpp. Writing better embedded Software," Youtube, 2019. [Online]. Available: https://youtu.be/3VtGCPIoBfs?t=1801. [Accessed 17 01 2023].

[2] R. Lindberg, "Alan Kay notes," 25 09 2019. [Online]. Available: http://rickardlindberg.me/writing/alan-kay-notes/. [Accessed 11 01 2023].

[3] B. Stroustrup, "1.4.2 The Early Years," in *The C++ Programming Language (4th Edition)*, Addison-Wesley, 2013, p. 23.

[4] B. Stroustrup, "Chapter 44 Compatibility," in *The C++ Programming Language (4th Edition)*, Addison-Wesley, 2013, pp. 1268-1279.

[5] B. Stroustrup, "www.stroustrup.com; From C to C++," 2011. [Online]. Available: https://www.stroustrup.com/From-C-to-Cpp-bs.pdf. [Accessed 11 1 2023].

[6] B. Stroustrup, "www.stroustrup.com; Foundations of C++," 2011. [Online]. Available: https://www.stroustrup.com/ETAPS-corrected-draft.pdf. [Accessed 11 1 2023].

[7] TIOBE Software BV, "TIOBE Index," TIOBE Software BV, 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed 11 1 2023].

[8] "ISO/IEC 9899:2011; 3.15," ISO-9899, 2011. [Online]. Available: https://iso-9899.info/n1570.html#3.15. [Accessed 16 2 2023].

[9] "ISO/IEC 9899:2011; 6.2.1," ISO-9899, 2011. [Online]. Available: https://iso-9899.info/n1570.html#6.2.1. [Accessed 16 2 2023].

[10] "CppCon. extern c: Talking to C Programmers about C++," YouTube, 2016. [Online]. Available: https://www.youtube.com/watch?v=D7Sd8A6_fYU. [Accessed 12 1 2023].

[11] B. W. Kernighan and D. M. Ritchie, "Preface," in *The C Programming Language*, Prentice-Hall Inc., 1978, p. 9.

[12] cppreference.com, "Classes - cppreference.com - Standard-layout class," 2023. [Online]. Available: https://en.cppreference.com/w/cpp/language/classes#Standard-layout_class. [Accessed 13 3 2023].

[13] ISO/IEC, "12.2 Class members; paragraph 25," in *Working Draft, Standard for Programming Language C++ N4713*, International Organization for Standardization, 2017, p. 218.

[14] *dynamic_cast From Scratch.* [Film]. CppCon, 2017.

[15] B. Stroustrup, "21.3.5 Virtual Base Classes," in *The C++ Programming Language (4th Edition)*, Addison-Wesley, 2013, pp. 632-636.

[16] "GoingNative. Inheritance Is The Base Class of Evil," YouTube, 2013. [Online]. Available: https://www.youtube.com/watch?v=bIhUE5uUFOA. [Accessed 2 2 2023].

[17] ISO/IEC, "8.5.1.7 Dynamic cast; paragraphs 5, 6," in *Working Draft, Standard for Programming Language C++ N4713*, 2017, p. 103.

[18] cppreference.com, "typeid - cppreference.com," 2023. [Online]. Available: https://en.cppreference.com/w/cpp/language/typeid. [Accessed 11 3 2023].

[19] ISO/IEC, "8.5.1.8 Type identification; paragraph 2," in *Working Draft, Standard for Programming Language C++ N4713*, 2017.

[20] ISO/IEC, "TR 18015:2006(E) Technical Report on C++ Performance," 15 2 2006. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf. [Accessed 4 4 2023].

[21] R. H. R. J. J. V. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Mark Tomm

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose C++ varjatud efektid, mille juhendaja on Peeter Ellervee
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

15.05.2023

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.