Tallinna Ülikool
Digitehnoloogiate Instituut

# C++ spetsifikatsiooni uuendusega kaasnevate muudatuste kasulikkuse analüüs.

Bakalaureusetöö

Autor: Roman Gorislavski
Juhendaja: Inga Petuhhov

Autor:……..…….. ” …… ” 2017

Juhendaja:……..……..…… ” …… ” 2017

Instituudi direktor:……..……. ” …… ” 2017

Tallinn 2017

# Autorideklaratsioon

Deklareerin, et käesolev bakalaureusetöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

| | |
|---|---|
| ..... | ..... |
| (kuupäev) | (autor) |

# Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina Roman Gorislavski (sünnikuupäev: 26.05.1989)

1. annan Tallinna Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „C++ spetsifikatsiooni uuendusega kaasnevate muudatuste kasulikkuse analüüs.“ mille juhendaja on Inga Petuhhov, säilitamiseks ja üldsusele kättesaadavaks tegemiseks Tallinna Ülikooli Akadeemilise Raamatukogu repositooriumis.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tallinnas, _____

(digitaalne) allkiri ja kuupäev

# Sisukord

# Sissejuhatus

C++ keele arengut uuriv teema arenes välja autori huvist C++ keele vastu. Teema on ka aktuaalne kuna C++ on viimase 6 aasta jooksul uuenenud märgataval määral ja C++11 loeti teiseks arendajasõbralikumaiks keeleks 2015 aastal (Stack Overflow, 2015). C++ on üldjuhul eelistatuim keel programmeerimisvõistlustel osalemiseks nagu Google Code Jam, Facebook Hacker Cup ja CodinGame võistlustel, millest viimane on ka automatiseeritud tööportaal arendajatele. C++ keel on tuntud kui ekspert sõbralik keel, mille õppimiseks kulub aastaid võhikutele. Selle õppimine pole raske, aga kuna ta lubab teha asju, mida keeled nagu Java ja C# piiravad, siis võhikute probleem seisneb eelkõige selles nad teeksid asju õigesti. Viimased 6 aastat on komitee püüdnud keelt ühtlustada ja muuta veelgi ohutumaks võhikutele, et nad ei tulistaks õppimise faasis koheselt oma jala otsast. Stoustrup ütles kunagi: "C++ on elav keel" (Stroustrup, 2007). Ajajooksul tingimused muutuvad ja need, kes kohanevad uute tingimustega jäävad ellu, see puudutab ka programmeerimise keeli. Lisaks sellele Standard C++ Foundation lasi välja ligi 2 aastat tagasi eksperimentaalse C++ Core Guidelines, mille abil saaks uued tulijada saaksid hoiduda keele vigadest. C++ Core Guidelines samuti hakkab ajaga kaasas käima, olles samas mõjutatud C++ kogukonna aktiivsemate liikmete otsustest.

Antud töö eesmärgiks on tuvastada C++ keele arenguga tekkinud potentsiaalsemad uued võimalused, mis ajajooksul kiiresti kasutust leiavad. Seetõttu antud tööd võivad pidada kasulikuks arendajad, kes tahavad saada ülevaadet uutest võimalustest ja samuti soovivad teada, millised võimalused populaarsemaks osutuvad.

Töö käigus otsitakse vastust järgmistele küsimustele:

- Millist koodi eelistaksid arendajad kirjutada kui neile antaks valida paari koodi stiili vahel?
- Kui suur oleks jõudluse võit või kadu uuema stiili puhul võrreldes vanema stiiliga?

Autor püstitab hüpoteesi, et 2011-2017 vahemikus toimunud C++ keele areng hakkab mõjutama seda kuidas C++ arendajad kirjutavad koodi läbi nende eelistuste. Kuna antud töö ikkagi on pilootprojekt, siis arvestatakse siin eelkõige aktiivsemate C++ kogukonna liikmete eelistustega.

# 1 Metoodika

Teine peatükk uurib uute keele võimaluste tagamaid kus iga alapeatükk jaguneb kaheks lõiguks samas järjekorras:

- Esimeses lõigus toimub alapeatükis vähemtuntud mõistete lahti seletamine, et mõista, paremini selle sisu. Samuti nimetab alapeatükkiga seotud uued keele võimalused. Samuti mainib keeleomaduste tuge kolmes populaarsemas kompilaatori süsteemis, õigemini selle toe puudumine.
- Järgnevates lõikudes uuritakse iga uue keele võimaluse põhiprobleemi ja selle võimaluse kasutust. Kuigi tihti on probleeme rohkem kui üks ja enamik neist on mainitud juba C++ standardile uute võimaluste ettepanekutes. Siis siin keskendutakse enamasti ühe keele võimaluse põhiprobleemi lahendamisele Küll aga autoripoolne valitud probleem ei pruugi väljendada standardikomitee põhiprobleemi.

Kolmas peatükk analüüsib uute keele võimalusi, kus iga alapeatükk, väljaarvatud viimane, jaguneb neljaks lõiguks samas järjekorras:

- Esimeses lõigus toimub alapeatükis käitusaegse programmi kiiruse tulemuste kirjeldamine. Kiirused mõõdetakse instruktsioonide hulga erinevuste abil. Antud tööraames eeldatakse, et enamus erinevaid instruktsioone on võrdväärse keerukusega. Instruktsioone eelistati ajamõõtmisele, kuna ajamõõtmisega ollakse rohkem seotud masinaga ja selle peal samal ajal jooksvate programmidega. Instruktsioonid on hea viis protsessori töö mõõtmiseks. Kuid ei mõõda üldse protsessori ja mäluvahelise liikumise ajakulu ega ka kompileerimise aega. Instruktsioonide mõõtmiseks kasutati GCC 8.0.0, kuna GCC oli ainuke kompilaator, mis juba mõnda aega toetas peaaegu kõiki keele omadusi. Selle konsooli sisendi parameetriteks olid: -std=C++1z -O0 -pthread. Kui täpset väärtust polnud võimalik anda instruktsioonides, siis kohati määrati suhteline hinnang kiiruse muutustele.
- Teises lõigus analüüsitakse alapeatükis vabas vormis keele omaduste potentsiaalseid ohtusid lahendamisest, mida autor antud töö käigus avastas. Antud töö tegevuse alla kuuluvad antud juhul küsitluse valmistamine, selle tulemused, koodinäidete

valmistamine, keele omaduste rakendamise uurimine ja loogilised järeldused vaatlusest.

- Kolmandas lõigus uuritakse iga uue keele võimaluse eelistuste põhjal. Antud eelistusi saadakse kvantitatiivsest uuringust, mille raames viidi läbi küsitluse abil Lisa 1. Selle sihtgruppiks on eelkõige aktiivsed C++ kogukonna liikmed. Kuigi tegu on kvantitatiivse uuringuga valim saab olema väike, kuna küsimused ei ole kerged ja neid on palju.

# 2 C++ keele võimaluste teooria

## 2.1 C++ ja sellega seonduvad mõisted

C++ on üldotstarbeline programmeerimise keel, mis on suunatud eelkõige süsteemide arenduseks, mis on (Stroustrup, 2007):

- parem kui C;
- toetab andmete abstraheerimist;
- toetab objektorienteeritud programmeerimist;
- toetab üldistavat programmeerimist.

ISO/IEC JTC1/SC22/WG21 ehk WG21 on rahvusvaheline standardiseerimisega tegelev töörühm, mis keskendub programmeerimiskeelele C++ (ISO/IEC JTC1/SC22/WG21, 2017).

ISO/IEC 14882:1998 ehk C++98 on C++ standarti esimene väljaanne, mis on aktsepteeritud ISO poolt (ISO/IEC, 1998). Kehtivuse vahemik 1998-2003.

ISO/IEC 14882:2003 ehk C++03 on C++ standarti teine väljaanne (ISO/IEC, 2003). Kehtivuse vahemik 2003-2011. Väljaande eesmärk on tuua sisse ainult vajalike veaparandusi, et arendajad harjuksid ISO standardiga ja kompilaatorite loojad jõuaksid järele antud standartiga.

ISO/IEC 14882:2011 ehk C++11 on C++ standarti kolmas väljaanne (ISO/IEC, 2011). Kehtivuse vahemik 2011-2014. Väljaande eesmärk on kaasajastada C++, et oleks kõlblik kasutamiseks tänapäeva tingimustes.

ISO/IEC 14882:2014 ehk C++14 on C++ standarti neljas väljaanne (ISO/IEC, 2014). Kehtivuse vahemik 2014-2017. Väljaande eesmärk on parandada, laiendada ja täiustada C++11 uusi võimalusi.

C++17 on tegemisel olev viies väljaanne, mille väljalaske aasta on 2017.
Väljaande eesmärk on paremini toetada suurte tarkvaralahenduste loomist, toetada kõrgetasemelisi parallelismi mudeleid ja lihtsustada keele põhielementide kasutamist.

Clang on populaarne kompilaatorite süsteem, mille on väljalasknud Google, mis sisaldab kaasaaegset C++ kompilaatorit. Kasutame viiendat versiooni toe testimisel.

GCC on teine populaarne kompilaatorite süsteem, mille on väljalasknud GNU Project, mis sisaldab kaasaaegset C++ kompilaatorit. Kasutame kaheksandat versiooni toe testimisel.

MSVC on kolmas populaarne kompilaatorite süsteem, mille on väljalasknud Microsoft, mis sisaldab samuti kaasaaegset C++ kompilaatorit. Kasutame 19.1-st versiooni toe testimisel.

## 2.2 Literaalid

Literaal on kindlas vormis koodi kirjutatud kirjamärkide jada, mis kujutab endas ajutist väärtust ja võib sisaldada prefiksi ja sufiksi. Literaale on erinevaid tüüpe: täisarvliteraal, märkliteraal, ujukomaarvliteraal, sõneliteraal, Boole´i literaal, viitliteraal ja kasutaja defineeritud literaal (ISO/IEC, 2016). C++14-s võeti kasutusele binaarliteraal ja järguühiku eraldaja (digit seperator). C++17-s võeti kasutusele u8 märkliteraal ja kuusteistkümmendsüsteemis ujukomaarvliteraal. MSVC hetkel ei toeta veel 16-nd süsteemis ujukomaarvliteraale.

Need uuenduste kasutuselevõtu põhjus ja nende kasutus:

Kahendkoodi sisestamise võimaluse puudumisel tuli arendajatel tihti manuaalselt teisendada arv teise süsteemi. Selle lahendamiseks on võetud kasutusele binaarliteraal (vt koodinäide 1), mille prefiks on 0B või 0b ja lubatud märkidest literaalis on ainult 0 või 1.

```
int main() {
     constexpr auto number{ 0b1010'0011'0001 };
}
```

**Koodinäide 1. Binaarliteraal kasutuses**

Pikka arvu jadaga numbrilisi literaale oli kohati raske lugeda (Crowl, 2013). Tihti ajati arvhaaval hiirega järge arvude kontrollimisel. Arvu loetavuse parandamiseks on nüüd võimalik sisestada numbrite vahele järguühiku eraldaja. Vastav tegevus ei mõjuta arvu väärtust. Selle kasutamiseks lisa ülakoma soovitud arvude vahele (vt koodinäide2).

Algväärtustamise loendi võimalused laiendati std::initializer_list abil ka kasutaja defineeritud konteineritele ning seda on võimalik kasutada nii konstruktoris kui ka funktsioonis.

```
int main() {
     constexpr auto number{ 1'000'000'000 };
     constexpr auto phone{ +372'800'7226 };
}
```

**Koodinäide 2. Kohajärgu eraldaja kasutuses**

C++-s oli 5 varianti sõneliteraale kuid ainult 4 varianti märkliteraale - puudus u8 (Smith, 2014a). Lisaks puudus viis garanteerida ASCII kasutamist tähemärgi formaadina, juhul kui mõni teine formaat oli sätitud seadmetesse vaikimisi (Smith, 2014a). Selle kasutamiseks tuleb nüüd kirjutada, prefiks u8 ja järgnevalt ülakomade vahele tähemärk (vt koodinäide 3).

```
#include <string>
using namespace std::string_literals;
int main(){
     constexpr auto narrow{ '1' };
     constexpr auto utf8{ u8'2' }; //C++17
     constexpr auto utf16{ u'3' };
     constexpr auto utf32{ U'4' };
     constexpr auto wide{ L'5' };

     auto s_narrow{ "11"s };
     auto s_utf8{ u8"22"s };
     auto s_utf16{ u"33"s };
     auto s_utf32{ U"44"s };
     auto s_wide{ L"55"s };
}
```

**Koodinäide 3. Märk- ja tekstliteraalid kuvatuna**

Puudus viis ujukomaarvu kuvamiseks literaalina 16-nd süsteemis koos eksponendina, kuigi kohati oli see literaal aktsepteeritud väljundina ja sisendina. Selle kasutamiseks tuleb lisada 0x prefiks, peale seda lisada kuueteistkümnendsüsteemis ujukomaarv, kohustuslikku eksponendi prefiks p ja eksponent ise kümnendsüsteemis (vt koodinäide 4).

```
int main() {
     constexpr auto number{ 0x1AFF.C7p15 };
}
```

**Koodinäide 4. Mittekorrektne delegeerimine**

## 2.3 Teisendamised ja avaldised

Teisendamine on ühe muutuja väärtuse edastamine teise teist tüüpi muutujasse. Väärtus kategooriad (value categories) määravad avaldise õiguse viidata muutuja nimega objektile ja õiguse vaikimisi liigutada. C++14-s tehti muudatusi konteksti põhistes teisendustes. C++17-s tehti muudatusi teisendatud konstantsetes avaldistes, lihtsustati väärtus kategooriaid, parandati tehete järjekorda avaldistes ja lubati loendtüübi muutuja otsest loend algväärtustamist. MSVC hetkel ei toeta muudatusi teisendatud konstantsetes avaldistes, lihtsustatud väärtus kategooriaid, parandatud tehete järjekorda avaldistes ja loendtüübi muutuja otsest loend algväärtustamist.

C++ ei lubanud klassil käituda erinevalt tema tüüpi konstantsete ja mitte konstantsete objektidega ja ei tunnistanud teisendamise funktsioone ülemklassist (Brown, 2011). Selle lahendamiseks lubati klassidel vastavalt kontekstile vaikimisi teisendada end täisarv- või loendtüübiks. Selle kasutusega enam ei pidanud kasutama 0+i või +i vormi (vt koodinäide 5), mis sundis klassi ennast vaikimisi teisendama täisarvtüübiks.

```
template< class T = int >
//requires  std::is_integral<T>::value
class Wrapper {
public:
     operator T& () { return member; }
```

```
     constexpr operator T () const { return member; }
private:
     T member{};
};
int main() {
     Wrapper<> i{};
     switch (+i) {}
}
```

**Koodinäide 5. Teisendatakse väärtus + abil**

Ajaloolistel põhjustel oli pandud piirang, mis tüüpe võis kasutada parameetrina teisendatud konstantses avaldistes, sest eelnvates C++ spetsifikatsioonides viit, viide ja viit klassi liikme konstantse avaldise tüüp (vt koodinäide 6) ei olnud piisavalt tugev (Smith, 2014b). Nüüd, millal see ei ole enam probleem, on süsteem ühtlustatud, seetõttu kus enne sai kasutada ainult otse viita (pointer), siis nüüd seda saab kasutada parameetrina ka funktsioone, mis tagastavad viita.

```
template<int* pointer>
struct X {};

struct S { static int member; } instance;
int S::member;
int main() {
     X<&instance.member> variable;
}
```

**Koodinäide 6. Instantsist staatilise liikmekutsumine**

Koopiast hoidumine (copy elision) toimub kui ajutine objekt on kasutatud teise objekti algväärtustamiseks (Smith, 2015). Mis tõttu optimeerimine ei olnud garanteeritud alati toimima kindlatel juhtudel. Selle lahendamiseks lihtsustati väärtus kategoorias geneerilise vasakpoolse väärtuse (glvalue) ja puhast parempoolse väärtuse (prvalue) mõisteid, millega esimene annab nüüd asukoha ja teine algväärtustab. See muudatus lisaks lubab luua objekte,

millel on kopeerimise konstruktor keelatud, aga lubab omastada puhas parempoolse väärtusega avaldisi teiselpool võrdusmärki (vt koodinäide 7). Kuna need ei ole enam seotud mingi asukohaga, siis need ei loeta enam kopeerimiseks.

```
struct Class {
     constexpr Class(int) {};
     constexpr Class(Class&) = delete;
     constexpr Class& operator=(const Class&) = delete;
};
int main() {
     constexpr Class x = 5;
};
```

**Koodinäide 7. Koopilast hoidumine**

Loendtüüpi (enum) muutuja väärtuse algväärtustamiseks tuli reeglina seni teha läbi tahtlikku teisendamise (Reis, 2015). See eristas seda algväärtustamist tavalisest klassis ja sisseehitatud tüüpide algväärtustamisest. Nüüd teisendus toimub vaikimisi (vt koodinäide 8) kui algväärtustatakse loendtüüpi muutuja tema alustüübiga lubades sellega arendajatel kirjutada ühtses stiilis.

```
enum Number : unsigned int{};

int main() {
     Number instance{ 27 };
}
```

**Koodinäide 8. Loendtüübi algväärtustamine**

## 2.4 Tüübituvastamine ja struktuursidemed

Struktuurside (structured binding) on viis lahti pakkida avaldist tema alamosadeks, eraldi muutujateks, eeldusel, et too sisaldab mingis vormis alamosi. C++14-s võeti kasutusele

tagastatava tüübi tuvastamine funktsioonis. C++17-s võeti kasutusele uued reeglid muutuja tüübi tuvastamisel ja struktuurside. MSVC hetkel ei toeta struktuursidemeid.

Iga C++ kasutaja kellele tutvustati C++11-s auto, lambda avaldisi ja tagastatava tüübi tuvastamist koos sabaga (vt koodinäide 9) koheselt mõtiskleb, miks nad ei saa lihtsalt kirjutada auto tüübi asemel ja automaatselt tuvastada sellega tagastatav tüüp (Merrill, 2012). See oli selge, et tüübi tuvastus ilma lõputa oli plaanis, aga ajapuuduse tõttu jäi tollal tegemata. Nüüd saab hoiduda pikkade tagastavate tüüpide korduvast kirjutamist funktsiooni deklareerimisel (vt koodinäide 10).

```
constexpr auto function(int a, float b)
     -> decltype(a + b) {
     return a + b;
}
int main() {
     function(5, 4.5f);
};
```

**Koodinäide 9. Sabaga tagastatava tüüp**

```
constexpr auto function(int a, float b) {
     return a + b;
}
int main() {
     function(5, 4.5f);
};
```

**Koodinäide 10. Tagastatava tüübiga funktsioon.**

Loend algväärtustamine koos auto-ga tekitavad probleeme õpetamisel (Voutilainen, 2013a). Kuna auto ja loend algväärtustamine enamus ajast kattus ja seetõttu loodi tahtmata ühe muutuja asemel loendi ühe elemendiga. Selle lahendamiseks on võimalik määrata otsesel algväärtustamisel muutuja tüübi ühe väärtusega muutujaks ja koopia algväärtustamisel määrata muutuja loendiks (vt koodinäide 11). Tulemuseks on natuke erinev süntaks, mis

eristab massiivi sisendit üksikelemendist aga samas propageerib ühtset algväärtustamist. funktsioonis.

```
#include <initializer_list>
int main() {
     auto x{ 5 };
     //auto y{ 1,5 };
     auto z = { 5 };
     auto w = { 1,5 };
};
```

**Koodinäide 11. Loend algväärtustamine ja auto**

Meie juba saame tagastada mitu väärtust, kuid puudub sellele süntaks, mis oleks nii elegantne kui ka kergesti kasutatav (Sutter, Bjarne, & Reis, 2015). Mitme väärtuse tagastamine on olnud võimalik juba C++ algusaegadest kasutades struct-klassi või kasutades tuple funktsionaalsust (vt koodinäide 12). Kuid mõlemal olid omad probleemid. Esimene oli kasulik ainult korduva tagastusmustri puhul ja teise puhul muutujad olid tihti algväärtustamata jäetud, mis üldjuhul loetakse tänapäeval halvaks stiiliks. Uus pakutud struktuursideme (vt koodinäide 13) süntaks parandab need probleemid visuaalse ja funktsionaalse lihtsuse abil samas suutes koos töötada eelnevate viisidega.

```
#include <tuple>
std::tuple<int, float, double>
function(int a, float b, double c){
     return { a, b, c };
}
int main() {
     int a{}; float b{}; double c{};
     std::tie(a, b, c) = function(5, 4.5f, 3.5);
};
```

**Koodinäide 12. Tagastamine tie-funktsiooniga**

```
#include <tuple>
std::tuple<int, float, double>
function(int a, float b, double c){
     return { a, b, c };
}
int main() {
     int a{}; float b{}; double c{};
     std::tie(a, b, c) = function(5, 4.5f, 3.5);
};
```

**Koodinäide 13. Tagastamine struktuursidemetega**

## 2.5 Tingimuslaused ja silmused

Vahemiksilmus (range loop) on silmus, mis loetleb (iterate) läbi konteineri algusest lõpuni. C++17-s võeti kasutusele päisekontroll, kompileerimisaegne tingimuslause, vahemiksilmuse üldistamine ja tingimuslause päises algväärtustamise lubamine. MSVC hetkel ei toeta päisekontrolli, kompileerimisaegset tingimuslauset ja tingimuslause päises algväärtustamise lubamist.

Seni oli väga raske C++ programmil otseselt ja kindlalt määrata kas antud teegi päis on kättesaadav kasutamiseks või mitte (Nelson & Smith, 2015). Lahendus lisati makrofunktsiooni (vt koodinäide 14) kujul, kuna päiste lisamine toimub eeltöötluse (preprotsessor) ajal, siis on mõttekas seda ka tollel hetkel kontrollida.

```
#if __has_include(<optional>)
#    include <optional>
#    define optional 1
#elif __has_include(<experimental/optional>)
#    include <experimental/optional>
#    define optional 1
#    define experimental 1
#else
```

```
#    define optional 0
#endif
```

**Koodinäide 14. Päise kontroll**

Puudus kompileerimisaegne tingimuslause, mis lubaks teha kompileerimis aegseid otsuseid ilma, et peaks kasutama mitmeid üle-laadimisi (Voutilainen, 2015). Kompileerimisaegne tingimuslause võeti kasutusele loomuliku lisana makro ja käitusaegsele tingimuslausele. Kuigi kohati too kattub üle tulevase kontseptide (consepts) ülesannetega, on ta mõeldud kasutamiseks eelkõige lokaalsel tasemel samal ajal kui kontsepte kasutataks globaalsel tasemel.

```
#include <type_traits>
template<class T,
    typename std::enable_if<std::is_integral<T>::value>::type >
     constexpr auto function(T input) -> T {
     return 20;
}
template<class T,
     typename
std::enable_if<std::is_floating_point<T>::value>::type >
     constexpr auto function(T input) -> T {
     return 50.f;
}
template<class T>
constexpr auto function(T input) -> T {
     return input;
}
int main() {
     constexpr auto result{ function(3) };
}
```

**Koodinäide 15. Funktsioonide üle-laadimine**

```
#include <type_traits>
template<class T>
constexpr auto function(T input) {
     if constexpr (std::is_integral_v<T>) {
          return 20;
     } else if constexpr (std::is_floating_point_v<T>) {
          return 50.f;
     } else {
          return input;
     }
}
int main() {
     constexpr auto result{ function(3) };
}
```

**Koodinäide 16. Kompileerimisaegne if**

Kuna lubati vahemiku lõputüübil erineda vahemiku algus tüübist, tekitati kasutajale probleeme kui too püüdis kasutada neid sisseehitatud vahemiksilmusega (Niebler, 2016). Vahemiksilmus (vt koodinäide 18) üldistati kuna vastasel juhul erineva algus- ja lõputüüpi kasutajad peavad kasutama klassikalist for-silmust (vt koodinäide 17).

```
#include <string>
struct FirstWord {
     std::string member;
     struct EndOfString {
          bool operator()(std::string::iterator it) {
               return (*it) != '\0' && (*it) != ' ';
          }
     };
     std::string::iterator begin() { return member.begin(); }
     EndOfString end() { return EndOfString(); }
};
```

```
bool            operator!=(std::string::iterator            it,
FirstWord::EndOfString p) {
     return p(it);
}
int main() {
     auto text{ FirstWord{ "Hello World!" } };
    for (auto iterator{ text.begin() }; iterator != text.end();
++iterator)
          auto charr = *iterator;
}
```

**Koodinäide 17. Tavaline silmus ja erinev lõputüüp**

```
#include <string>
struct FirstWord {
     std::string member;
     struct EndOfString {
          bool operator()(std::string::iterator it) {
               return (*it) != '\0' && (*it) != ' ';
          }
     };
     std::string::iterator begin() { return member.begin(); }
     EndOfString end() { return EndOfString(); }
};
bool            operator!=(std::string::iterator            it,
FirstWord::EndOfString p) {
     return p(it);
}
int main() {
     auto text{ FirstWord{ "Hello World!" } };
     for (auto character : text)
          character;
}
```

**Koodinäide 18. Vahemiksilmus ja erinev lõputüüp**

Seni toimus algväärtustamine (vt koodinäide 19) enne lauset (statement) ja lekkis vanema skoopi või kasutas täiendavat skoopi (Köppe, 2016). Kuna silmuse päises algväärtustamine oli lubatud juba mõnda aega, siis selle lubamine tingimuslause päises on keelearenduses loomulik osa (vt koodinäide 20).

```
#include <array>
int main() {
     constexpr std::array<int, 6>
          container{ 4, 5, 6, 6, 7, 8 };
     auto index{ 4 };
     {
          constexpr auto half{
               container.max_size() / 2
          };
          if (index < half) ++index;
          else if (index > half) --index;
          else index;
     }
}
```

**Koodinäide 19. Tavaline tingimus lause**

```
#include <string>
struct FirstWord {
     std::string member;
     struct EndOfString {
          bool operator()(std::string::iterator it) {
               return (*it) != '\0' && (*it) != ' ';
          }
     };
     std::string::iterator begin() { return member.begin(); }
     EndOfString end() { return EndOfString(); }
```

```
};
bool            operator!=(std::string::iterator            it,
FirstWord::EndOfString p) {
     return p(it);
}
int main() {
     auto text{ FirstWord{ "Hello World!" } };
     for (auto character : text)
          character;
}
```

**Koodinäide 20. Algväärtustamisega tingimuslause**

## 2.6 Agregaadid ja konstantavaldised

Konstantavaldis (constant expression) on avaldis mida arvutatakse välja üldjuhul kompileerimise ajal. Agregaat (aggregate) on struktureeritud komponendikogum, mille komponentidel võib olla ühesugune või erinev andmestruktuur, kogumi enda andmestruktuur võib aga olla vastava liittüübi koostisosa (Eesti Standardikeskus, 2001). C++14-s lõdvestati konstantavaldise funktsioonide tingimusi (nt. lubati mitme realised konstantavaldised) ja parandati ühilduvust liikme ja agregaat algväärtustamise vahel. C++17-s võeti kasutusele inline-muutujad ja lubati agregaat algväärtustamist koos ülemklassidega klassidel. MSVC hetkel ei toeta veel inline-muutujaid ja ei luba agregaat algväärtustamist koos ülemklassidega klassidel.

C++11-s constexpr-i süntaksi piirangud, mis siiani väga piiravad, ja nende puudulik väljendusrikus on põhimured, mis on suunatud constexpr võimalusele (Smith, 2012). Selle probleemideks olid keelatud mitme-realised funktsioonid (vt koodinäide 21), keelatud mitme return süntaksi kasutamine, keelatud enamus tingimuslauseid, silmuseid, constexpr funktsioone ja vaikimisi const asus constexpr funktsioonis. See tekitas probleeme literaal klassi tüüpidele, mis soovivad olla kasutatud nii konstantses avaldistes kui ka väljaspool (Smith, 2013). Neist tähtsaim uuendus on mitme realised funktsioonid (vt koodinäide 22), kuna nüüd on võimalik kasutada konstantavaldise meetodis tavalisi tingimuslauseid, silmuseid ja lokaalseid muutujaid.

```
enum class Messages {
     Hi, Quest, Success, Fail
};
constexpr Messages getMessage(int input)
{
     return (input==0) ? Messages::Quest :
            ((input==1) ? Messages::Success :
            ((input==2) ? Messages::Fail :
                          Messages::Hi));
}
```

**Koodinäide 21. Üherealine constexpr funktsioon**

```
#include <array>
int main() {
     constexpr std::array<int, 6>
          container{ 4, 5, 6, 6, 7, 8 };
     auto index{ 4 };
     if (constexpr auto half{
          container.max_size() / 2
          }; index < half) ++index;
     else if (index > half) --index;
     else index;
}
```

**Koodinäide 22. Mitmerealine constexpr funktsioon**

Puudus viis kasutada agregaat algväärtustamist kui klass omas liikme algväärtustamist (Voutilainen, 2013b)(vt koodinäide 23). Agregaat algväärtustamine on väga tundlik muutustele keeles, kuna peab järgima rangeid tingimusi. Tänu agregaat algväärtustamise reeglite kohandamisele on võimalik nüüd kasutada agregaate sujuvalt koos liikme algväärtustamisega (vt koodinäide 24).

```
struct Aggregate {
     int member1;
     char member2;
};
int main() {
     Aggregate instance1{ 5, 'b' };
     Aggregate instance2{ 7, 'b' };
}
```

**Koodinäide 23. Tavaline agregaatklass**

```
struct Aggregate {
     int member1;
     char member2{ 'b' };
};
int main() {
     Aggregate instance1{ 5 };
     Aggregate instance2{ 7 };
}
```

**Koodinäide 24. Agregaatklass koos liikmealgväärtustamisega**

Puudub võimalus defineerida inline-muutujaid (vt koodinäide 26), mis paneb tõsised piirangud teegi disainile (Finkel & Smith, 2015). Selle lubamisel on nüüd võimalik algväärtustada staatilist muutujat ilma, et kohati peaks seda topelt deklareerima (vt koodinäide 25).

```
#include <vector>
struct Class {
     static constexpr auto member{ 42 };
};
constexpr int Class::member;
int main() {
     std::vector<int> container{};
```

```
     container.push_back(Class::member);
}
```

**Koodinäide 25. Topelt deklareeritud staatiline liige.**

```
#include <vector>
struct Class {
     static constexpr auto member{ 42 };
};
int main() {
     std::vector<int> container{};
     container.push_back(Class::member);
}
```

**Koodinäide 26. Inline-muutuja kasutuses**

Agregaat algväärtustamine ei toimi kui ülemklass eksisteerib, mis ei ole kasulik (Smolsky, 2015). Piiratud ülemklassi (vt koodinäide 28) lubamine lubab agregaatklassidel sarnaselt tavalistele klassidele omada ülemklassi ilma, et seda peaks looma agregaatklassis muutujana (vt koodinäide 27).

```
struct Derived {
     struct Base {
          double member1;
     } base;
     int member2;
};
int main() {
     Derived instance{ { 1.0 }, 0 };
}
```

**Koodinäide 27. Agregaatklass muutuja klassiga**

```
struct Base {
     double member1;
```

```
};
struct Derived : Base {
     int member2;
};
int main() {
     Derived instance{ { 1.0 }, 0 };
}
```

**Koodinäide 28. Agregaatklass ülemklassiga**

## 2.7 Lambda avaldised

Lambda avaldis on puhas parem poolne avaldis, mis tagastab sulund tüüpi ehk anonüümse funktsiooni tüüpi objekti (ISO/IEC, 2016). C++14-s võeti kasutusele lambda avaldise hõivamine liikumisega ja geneerilised lambdad. C++17-s võeti kasutusele lambda avaldise *this hõivamine väärtusena ja lambda avaldised konstantavaldises. MSVC hetkel ei toeta veel lambda *this hõivamist väärtusena ja lambda avaldisi konstantavaldises.

C++11 lambda avaldised ei toeta hõivamist liikumisega (capture-by-move) (Voutilainen, 2013). Hõivamine liikumisega ei aitaks ainult tüüpe mida on kulukas kopeerida, vaid aitaks ka tüüpe mida ei saagi kopeerida, seetõttu võetigi see kasutusele (vt koodinäide 29).

```
#include <memory>
struct Class{
     constexpr auto method()->void {};
};
int main() {
     auto pointer{ std::make_unique<Class>() };
     [l_pointer{ move(pointer) }]() { l_pointer->method(); };
}
```

**Koodinäide 29. Liikumise hõivamine otse**

Lambda avaldiste kasutamine võib olla tülikas, kuna arendaja peab tihti nimetama välja tüüpe mida oleks võimalik kompilaatoril ise automaatselt tuvastada (F. Vali, Sutter, & Abrahams,

2012). Geneeriliste lambda (vt koodinäide 31) avaldiste puhul tuleb arvestada et nad käituvad sarnaselt geneeriliste funktoritega (functor) (vt koodinäide 30), kuid kasutavad kaasaaegsemat süntaksi.

```
struct {
     template<typename T, typename U>
     auto operator()(T x, U y) const
          -> decltype(x + y) {
          return x + y;
     }
} closure;
int main() {
     closure(0.5, 1.5);
     closure(1, 2);
}
```

**Koodinäide 30. Geneeriline funktor**

```
int main() {
     auto lambda{ [](auto x, auto y)
          {return x + y; }
     };
     lambda(0.5, 1.5);
     lambda(1, 2);
}
```

**Koodinäide 31. Geneeriline lambda**

Probleem seisnes, et sa ei saanud püüda (capture) *this hõivata väärtusena (Edwards et al., 2015). Väärtusena hõivamine on vajalik kui eeldata, et objekti võidakse kutsuda peale lambda avaldiste hävitamist.

Varasemalt lambda avaldised olid keelatud kasutada konstantsetes avaldistes (F. S. Vali, Voutilainen, & Reis, 2015). Kuigi lambda avaldised sobiksid süntaksi poolest kenasti konstantavaldistega kokku ja tunduksid selle loomuliku osana. Selle kasutuselevõtuga on

võimalik nüüd kirjutada kergemini loetavamat koodi võrreldes funktoritega konstantsetes avaldites.

## 2.8 Erindid ja tõendajad

Erindi (exception) spetsifikatsioon koosneb määratud tüüpidest, mis näitavad et funktsioon võib väljuda nendest ühe määratud tüüpidest erindi kaudu (ISO/IEC, 2016). C++17-s võeti kasutusele funktsioon püüdmata erindide loetlemiseks, automatiseeritud sõnumiga staatiline tõendaja ja erindi spetsifikatsiooni integreerimist tüübi süsteemi. MSVC hetkel ei toeta veel erindi spetsifikatsiooni integreerimist tüübi süsteemi.

Kood mis on sihiliselt kutsutud destruktorist, mis võib iseennast kutsuda kuhja (stack) lahti pakkimisel, ei ole suuteline tuvastama kas ta on ise ka lahti pakkimise osa (Sutter, 2014). Seetõttu on kasutusele võetud sisseehitatud funktsioon mis loetleb püüdmata erindeid.

Arendajad kes on kasutanud BOOST_STATIC_ASSERTi, on harjunud sellega, et neil pole käsitsi sõnumi lisamise võimalus (Brown, 2014). Automatiseerud sõnumiga tõendaja lisab võimaluse, kus arendaja ei pea olema veateate sõnastamise ekspert.

Erindi spetsifikatsioon ei olnud kunagi sellega seotud tüübi osa, mis viis nii mõnegi kummalise probleemini C++-s. (Maurer, 2014). Erindi spetsifikatsiooni integreerimine tüübi süsteemi parandas tüübi teisendamisega seotud probleemide leidmist.

## 2.9 Mälu hõivamine ja vabastamine

Objekti tüüpidel on joondus nõuded, mis panevad piirangud, milliseid aadresse seda tüüpi objekt võib hõivata (ISO/IEC, 2016). C++14-s võeti kasutusele globaalne operaator delete koos parameetriga suurus ja mälu hõivamise ümbersõnastamine. C++17-s võeti kasutusele mälu hõivamise viis üle-joondatud andmete korral. MSVC hetkel ei toeta veel mälu hõivamise viise üle-joondatud andmete korral ja mälu hõivamise ümbersõnastamist. GCC ei toeta veel mälu hõivamise ümbersõnastamist.

Globaalne operaator delete koos parameetriga suurus ei ole saadaval seni olnud, mille tulemusel jõudlus kannatas (Crowl, 2012). Selle lisamine samuti ühtlustas delete funktsionaalsust ka üleüldiselt.

Mälu hõivamine ja vabastamine on tänapäeval muutunud tõsiseks kuluks kaasaaegsetes süsteemides (Crowl & Carruth, 2012). Seetõttu see sõnastati paremini ümber, aga erinevalt enamikust muutustest on selle täitmine kompilaatori loojatele vabatahtlik osa. Mälu hõivamise ümbersõnastamisel kohati tugeva optimeerimise all võib instruktsioonide arv väheneda, kuna lubatakse kohati välja optimeerida new ja delete, mida enne kompilaator eriti puutuda ei tohtinud.

Kahjuks C++11-s pole määratud ühegi mehhanismiga kuidas üle-joondatud (over-aligned) andmed korrektselt hõivaksid dünaamilist mälu (Nelson, 2012). Õnneks selle kasutusala piirdub vahemälu täitmise optimeerimisega ja seadmete vaheliste mõõtmete ühtlustamisega ühilduvuse põhimõttel.

## 2.10 Atribuudid ja nimeruumid

Atribuudid (attributes) määravad täiendavat informatsiooni erinevatele konstruktsioonidele nagu tüüpidele, muutujatele, nimetustele, blokkidele või tõlkeüksustele (ISO/IEC, 2016). C++14-s võeti kasutusele kasutamisest hoidumise (deprecated) atribuut. C++17-s võeti kasutusele muutuja mitte kasutamise atribuut, funktsiooni tagatise kasutamise atribuut, mitme tingimuse läbimise atribuut, tuvastamata atribuutide ignoreerimise lubamine, nimeruumi (namespace) kontrolli atribuut, atribuutide lubamine nimeruumides ja loendites ja astmeliste nimeruumide ühendamine. MSVC hetkel ei toeta veel muutuja mitte kasutamise atribuut, funktsiooni tagatise kasutamise atribuut, tuvastamata atribuutide ignoreerimist ja nimeruumi kontrolli atribuuti.

Mitme praegused kompilaatorid on pakkunud viise kuidas märgistada objekte, mille kasutamine on vananenud (Barbati, 2012). Põhiprobleem seisnes kompilaatori loojate oma süntaksi kasutamises ehk puudus ühtne süntaks. Seetõttu võeti kasutusele ühtne hoidumise atribuut.

Hoiatuste eesmärk on püüda vigu, kuid tihti on olukordi kus kompilaator ei saa kindlusega tuvastada kas antud koodi jupp on tahtlik või mitte (Tomazos, 2015). Seetõttu sai lisatud kolm uut atribuuti: muutuja mitte kasutamise atribuut, funktsiooni tagatise kasutamise atribuut ja tingimuslause mitme tingimuse läbimise atribuut.

Meil pole määratud nõudmistes, et kompilaatorid ignoreeriks atribuutide nimeruume, mida nad ei suuda tuvastada ja seetõttu selle kasutajad peidavad atribuute makrodesse (Garcia, 2016). Lubati ignoreerida tuvastamata atribuute, mis võimaldas kompilaatori loojatel lisada oma atribuute vähendamaks sellega kolmandate osa poolte makrosid.

Juhtudel kui on vaja kasutada mitu atribuuti ühes annotatsioonis, siis tulemuseks on sõnaohtrus, mis paneb enamik atribuudi nimeruumide kasutajaid määrama neile lühikesi nimetusi (Garcia, Sanchez, Torquati, Danelutto, & Sommerlad, 2015). Seetõttu lisati veel üks atribuut, mis kontrollib atribuutide nimeruume ja lubab vähendada nimeruumide üleliigseid deklareerimisi.

Atribuudid ei olnud lubatud kasutada ei loenditel ega ka nimeruumides (Smith, 2014d). Vananenud atribuuti lisamisel avastati, et loendid ja nimeruumid polnud toetatud ja seetõttu on see nüüd lubatud.

Pole haruldane leida mitme sügavustasemega nimeruume suurtes projektides (Kawulak, 2014). Seetõttu loodi süntaks ühendamaks erineva sügavustasemega nimeruume omavahel, vähendades sellega oma korda süntaksi pikkust.

## 2.11 Mallid ja volditud avaldised

Malle võib nüüd jaotada üldiselt kolmeks tüübiks: geneerilised klassid, geneerilised funktsioonid ja geneerilised muutujad. Volditud avaldised (folding expressions) jagunevad kaheks: binaarvoltimine ja monaadvoltimine (ISO/IEC, 2016). C++14-s võeti kasutusele geneeriline muutujad (variable templates). C++17-s võeti kasutusele type_name malli geneerilise parameetrina, geneeriliste argumentide tuvastamine geneerilises klassis, väärtus parameetrite deklareerimist auto-ga, voltivad avaldised ja tühjade parameeter pakkide lubamist monaad voltides. MSVC hetkel ei toeta veel geneeriliste argumentide tuvastamist geneerilises klassis, väärtus parameetrite deklareerimist auto-ga, voltivad avaldisi ega ka tühjade parameeter pakkide lubamist monaadvoltides.

C++ ei oma süntaksit geneerilise muutujate kirjutamiseks nagu ta omab seda funktsioonide ja klasside puhul (Reis, 2013). Geneerilise muutuja kasutusele võtuga lihtsustati geneerilise konstandi loomist.

C++ omab geneerilist tüüpi, mis ei ole geneeriline klass, aga malli geneerilised parameetrid ikkagi vajavad class võtmesõna (Smith, 2014c). Võtmesõna typename lubamine malli geneerilise parameetrina lubab paremini arendajal kuvada oma kavatsusi läbi parema sõnastuse.

Loomisfunktsioonide kasutamine nagu make_tuple on segadusse ajav, kunstlik, lisab liigset koodi ja erineb mitte geneeriliste klasside loomisest (Spertus & Vandevoorde, 2013). Selle lahendamiseks lubati võimalusel tuvastada geneerilisi argumente geneerilises klassis. Lisaks võeti kasutusele tuvastamisejuhend (deduction guide), mis võimaldas tuvastada tüüpi argumentidest, millest kompilaator tüüpi tuvastada iseseisvalt ei suuda.

Seni pidi spetsiifiliselt eraldi välja tooma väärtus parameetrite tüüpe, mis tõi kaasa endaga paljusõnalisust ja vähendas paindlikkust malli loomisel, mis aktsepteeris argumentideks igat tüüpi (Touton & Spertus, 2015). Nüüd tüüp tuvastatakse väärtus parameetrist automaatselt, hoides sellega tüübi nimetuse mainimist.

Seni tuli meil tulemuse saamiseks välja arvutada volte üle parameeter pakki, selleks meie pidime kasutama hulgi variaarsete mallide (variadic template) üle-laadimisi (Sutton & Smith, 2014). Volditud avaldiste kasutusele võtuga lihtsustati arvutamist parameeter pakkidega märgatavalt.

```
constexpr auto sum() -> int {
     return 0;
}
template<typename T>
constexpr auto sum(T t) -> T {
     return t;
}
template<typename T, typename... Ts>
constexpr auto sum(T t, Ts... ts)
-> decltype(t + sum(ts...)){
     return t + sum(ts...);
}
int main() {
```

```
     constexpr auto result{
          sum(21,sum(22,1.2))
     };
}
```

**Koodinäide 1. Mittekorrektne delegeerimine**

```
template<typename... Ts>
constexpr auto sum(Ts... ts){
     return (ts + ... + 0);
}
int main() {
     constexpr auto result{
          sum(21,22,1.2)
     };
}
```

**Koodinäide 2. Mittekorrektne delegeerimine**

Tühi volt, mis kasutab liitmist, alati tagastab 0 kui argumendid puuduvad (Le Jehan, 2015). Nüüd saame, aga eristada tühja volti, 0 väärtus voldist. Tänu millele võime kohati hoiduda tõsistest loogika vigadest.

# 3 C++ keele võimaluste analüüs

## 3.1 Literaalid

Käitusaegse programmi kiirus ei muutu, sest literaalid teisendatakse väärtusteks kompileerimise ajal. Literaalide puhul võib oletada, et isegi kompileerimiseaegne kiiruse muutus on märkamatu.

Ohutusest rääkides järguühiku eraldaja eelkõige vähendab vigade arvude kirjutamisel, kuna nüüd saab pikki arve grupeerida väiksemateks loetavateks osadeks ja vähendab vajadust käsitsi teisendada arve kahendsüsteemist. Samuti garanteeritud ASCII formaadi võimalus lubab ohutumalt erinevate seadmete vahel kasutada ühtset koodi. Samas tuleb mainida, et 16-nd süsteemi ujukomaarvu eksponent on 10-nd süsteemis, mis võib kaasa tuua valeväärtuse kirjutamise.

Kasutus eelistustes, enamus arendajaid eelistas kopeerida ette antud väärtusi/konstante, vastavalt 94,4% binaarset notatsiooni ja 83,3% teaduslikku notatsiooni 16-nd süsteemis, otse koodi teisendamise asemel. Samas koodiloetavuse nimel 72.2% oli valmis suurte väärtuste korral lisama järguühiku eraldajaid. Literaalidega seoses suurimaks vaidlus punktiks osutus u8 literaali lisamine tähemärkidele, kus arendajatest 55,6% leidsid seda kasulikuna. Samas tervelt 22,2% arendajatest valis muu, mis võib olla määratud puudulikkust valikust küsitluses või vastajatel puudus lihtsalt eelistus antud punkti suhtes. Literaalide valdkonnas uudsusvalmidus on keskmiselt 76,4%. (vt Lisa 1)

## 3.2 Teisendamised ja avaldised

Käitusaegse programmi kiirus ei muutu täiendavate konteksti põhiste teisenduste tõttu, kuna teisendused toimuvad ikkagi nüüd ainult edaspidi vaikimisi. Muudatused teisendatud konstantsetes avaldistes ei mõjuta ka otseselt käitusaegset kiirust, kuna konstantsed avaldised lahendatakse kompileerimise ajal. Lihtsustatud väärtus kategooriad ei muuda enamikel kompilaatoril käitusaegset kiirust, kuna lihtsustamine muutis osa koopiast hoidumise viise soovituslikust kohustuslikuks. Küll aga parandab kiirust neil kellel puudub või on keelatud koopiast hoidumine, kuna sõnastab üle nii, et kohati ei olegi tegu kopeerimisega. GCC puhul tühjast kopeerimise konstruktorist hoidumisega näiteks vähendaks koodi 12 instruktsiooni

võrra. Muutused tehete järjekorras otseselt muutusi käitusaegses kiiruses sisse ei too, kuid võivad tulevikus pakkuda kompilaatori loojatele uusi viise optimeerimiseks koodi. Loendtüübi loendi otsese algväärtustamisega käitusaegne kiirus ei muutu, kuna eelnevalt nõutud teisendamine loend tüübiks toimus kompileerimise ajal.

Ohutus paraneb selle võrra, et nüüd osa ohutuid teisendusi toimuvad vaikimisi ja see vähendab koodi ridade kirjutamist ja sellega väheneb tõenäosus veatekke tekkimiseks. Tehete järjekorra muutused parandavad marginaalselt ohutust, kuna varem isegi tihti kasutatud koodid kohati ei töötanud korrektselt määramata tehete järjekorra tõttu. Teised muudatused minimaalselt kui üldse mõjutavad ohutust.

Näite põhjal 77.8% arendajatest eelistas kasu saada konteksti põhistest teisendustest. Samal ajal teisendatud konstantses avaldise vastu oli huvi madal, ainult 16.7%, eeldusel, et teised teisendatud konstantses avaldise kasutusviisid on sama potentsiaalse kasutusega nagu näide oli. Lisaks 83.3% arendajatest leidis, et standard peaks nõudma optimeerimist kompilaatori loojatelt kui see on võimalik. 77,8% arendajatest leidis, et otsene kui ka koopiaga algväärtustamine peaks olema võimalik enamus tingimustes. Tehete järjekorra muutuste koha pealt tekkis vaidlus koht, kus küll C++17 rakendatav tulemus, 38,9%, oli küll eelistatud valik kuid tulemus oli kaugel üksmeelsusest. Tehete järjekorras võib oletada puudus veel üks valik kuna muu valisid 38,9% arendajatest. Lisaks tasukaalukas vaidlus punkt oli loendtüübi muutuja algväärtustamine, kus väikese edumaaga, 55,6%, eelistati antud muutujat algväärtustada läbi vaikiva teisendamise. Teisendamise ja avaldiste valdkonnas uudsusvalmiduse on keskmiselt 58,4%. (vt Lisa 1)

## 3.3 Tüübituvastamine ja struktuursidemed

Tagastatava tüübi tuvastamine funktsioonis käitusaegse programmi kiirust otseselt ei muuda, kuid kaudselt võib hoida ära mõnda peidetud teisendust ja sellega võibolla parandada käitusaegset kiirust. Uued reeglid seoses tüübituvastusega ei muuda käitusaegset kiirust, kuna tüübituvastus lahendatakse kompileerimise ajal. Struktuur side aga parandab märgatavalt käitusaegset kiirust võrreldes tie funktsiooniga. Näiteks näites langeb instruktsioonide arv 649-lt 404-le.

Tagastatava tüübi tuvastamine parandab ohutust, kuna nüüd saab hoiduda tüüpide korduvast deklareerimisest. Tüübituvastuse uued reeglid samas võivad kogenud arendajatele tuua segadust nende harjumuse tõttu. Samas seda on kergem ja ohutum õpetada uutele õpilastele, sest nüüd on üksikelement eristatav massiivi algväärtustamisest. Struktuurside on kasulik mitme tüübi tagastamisel ohutuse vaatepunktist, kuna enam ei pea tagastatud muutujaid deklareerima ette ära.

72,2% arendajatest eelistasid kirjutada funktsioone tuletatud tagastustüübiga, samal ajal traditsiooniliseks eelistas jääda ainult 16,7%. Kui vaadata tuvastatud tüübiga algväärtustamist, siis 38,9% arendajatest eelistas tuvastada massiivtüüpi ainult juhul kui oli tegu rohkem kui elemendiga. Samas C++17 rakendatav algväärtustamine, mis tuvastab massiivi võrdusmärgi abil, oli eelistatud teisel kohal, 33,3%. Arendajatest 94,4% eelistas tagastada mitme muutujaga tüübi funktsioonis peaaegu üksmeelselt struktuursidemega. Tüübituvastamise ja struktuursideme valdkonna uudsusvalmidus oli keskmiselt 66,6%. (vt Lisa 1)

## 3.4 Tingimuslaused ja silmused

Päisekontroll käitusaegset kiirust ei muuda, kuna kontroll toimub juba eeltöötluse ajal. Kompileerimisaegne tingimuslause samuti ei mõjuta käitusaegset kiirust. Vahemiksilmuse üldistamise kasutamisel kui võrrelda näiteid, siis uus üldistatud silmus on 5x-4 instruktsiooni, kus x on tsüklite arv silmuses, võrra kiirem, kuna see toob täiendava ühe funktsiooni silmuse sisse. Viimane sõltub rohkelt arendaja oskustest. Tingimuslauses algväärtustamine kiirust otseselt ei muuda, kuna muudab ainult muutuja skoopi.

Ohutusest rääkides päisekontroll tõstab langetab keerukust, kuna lubab kontrollida päiste olemasolu samas programmis, samas kui teised teevad seda väliste rakenduste kaudu. Oht, midagi valesti teha välises rakenduses on võrdlemisi suur, kuid tuleb arvestada kuna tegu on makroga, siis pakutud alternatiiv ei ole ise ka ohutu, kuna viga võib märkamata jääda samuti. Kompileerimisaegne tingimuslause on märgatavalt ohutum kui selle üle laetud funktsioonide alternatiiv, kuna viimasele on võimalik teha kergesti loogika viga sisse seoses geneeriliste funktsioonide spetsialiseerumisega. Lubades vahemiku lõputüübil erineda tõuseb ka oht, et seda võidakse kasutada valesti mitte teadlikult. Samas teadlikult kasutades võib see natuke hoopis paraneda võrreldes iteraatoriga tingimuslause väljakirjutamisega, kus on kergesti

võimalik teha trükkiviga. Algväärtustamine tingimuslauses muudab koodi ohutumaks, kuna vähendab ohtu, kus muutuja lekkib vanema skoopi ja sellega väldib sellega seotud ohtusid.

72,2% arendajatest eelistasid kasutada programmi sisest päisekontrolli. Samas kompileerimisaegse tingimuslausest oli huvitatud umbes 1/3 arendajatest, täpsemalt 38,9%, võisteldes tugevasti funktsioonide üle-laadimisega ja kontseptidega (concepts). Vahemiksilmuse üldistamise kasulikkus aktsepteeriti üksmeelselt, 100%-iliselt. Tingimuslause päises algväärtustamist eelistaksid kasutada 66,7% ja ülejäänud eelistaksid kasutada selle alternatiive. Tingimuslausete ja silmuste valdkonna uudsusvalmidus oli keskmiselt 69,5%. (vt Lisa 1)

## 3.5 Agregaadid ja konstantavaldised

Käitusaegse programmi kiirus üldiselt ei muutu kuna enamus konstantavaldise funktsioone lahendatakse kompileerimise ajal. Kuid see võib mõjutada kaudselt konstantavaldiste funktsioone, mida ei ole kompileerimise ajal lahenda, vastavalt arendaja oskustele, kuna neil on nüüd suurem vabadus muuta funktsionaalsust. Parandatud ühilduvus liikme ja agregaadi algväärtustamine käitusaegset programmi kiirust ei mõjuta, kuna mõlemad on algväärtustamised lahendatakse ühtemoodi programmi jooksmisel. Käitusaegne programmi kiirus ei muutu inline-muutujatega kui nad kasutuses ainult ühes transleerimisüksuses, kuid võib muutuda muutuja kasutamisel mitmes transleerimisüksuses. Agregaat algväärtustamisega koos ülemklassidega käitusaegne programmi kiirus ei muutu, kuna oleks võrdväärne sellega kui ülemklass oleks hoopis muutuja alamklassis kompileeritud koodi suhtes.

Ohutusest rääkides konstant avaldise piirangute vähendamine toob endaga kaasa loetavama ja ohutuma koodi, kuna enam ei pea kogu koodi ühele reale ära mahutama. Ohutum on see kuna viga on leida kergem loetavast lühikesest koodireast kui krüptilisest pikast koodireast. Agregaat algväärtustamine koos liikme algväärtustamisega on ohutum lubatuna, kuna see on lubatud ka tavalistel klassidel ja tooks segadust kui ühes neist kasutada ei saaks. Inline-muutujad parandavad ohutust sellega et neid ei pea topelt deklareerima erinevalt teistest staatilistest muutujatest, mis tekitab segadust tihti kuna topelt deklareerimine on haruldane. Agregaatide lubamine ülemklassidega parandab ohutust, kuna kohati tekitas segadust see, et tavalistel klassidel oli lubatud omada ülemklasse aga agregaatidel mitte.

Üksmeelselt, 100%, eelistasid arendajad kasutada mitme reaga konstantavaldise funktsioone. Samuti olid kõrged eelistused kasutada liikme algväärtustamist koos agregaat algväärtustamisega, milleks oli 77,8%, ja hoiduda staatilise liikme topelt deklareerimisest, milleks oli 77,8%. 77,8% arendajatest eelistas lisada agregaat klassile ülemklassi sarnaselt tavalistele klassidele. Agregaatide ja konstantavaldiste valdkonna uudsusvalmidus oli keskmiselt 83,4%. (vt Lisa 1)

## 3.6 Lambda avaldised

Käitusaegse programmi kiirus muutub märgatavalt kui lubada hõivamist liikumisega, sest see aitaks hoiduda mähise loomisest. Kui võrrelda näiteid siis väheneks instruktsioonide arv 50-ne võrra. Enamus lambda avaldisi kutsuvad täiendava ühe instruktsiooni võrra aeglasemad võrreldes nende alternatiiviga, funktoritega. Sellest on mõjutatud nii geneerilised lambda avaldised kui ka käitusaegsed konstant lambda avaldised. Kompileerimiseaegsed konstant lambda avaldised ei mõjuta otseselt käitusaegset programmi kiirust. Samuti *this hõivamine väärtusena ei mõjuta käitusaegset programmi kiirust võrreldes selle mähitud vormiga, sest mähitud osa optimeeritakse välja.

Ohutusest rääkides hõivamine liikumisega hoiab arendajaid mähise loomisest, mida nad tihti on jõudluse nimel on valmis tegema. Mähisel loomisel on kergesti võimalik vigu teha. Geneerilised ja konstantsed lambda avaldised muudavad ohutumaks kirjutamist üleüldiselt kuna võimaldavad hoiduda nende kergelt krüptilistest alternatiividest, funktoritest, milles võidakse kergemini vigu sisse kirjutada. Samas geneerilised lambdad on ohtlikumad kui teised mallid, kuna puudub template võtmesõna, millega geneerilisust tugevalt seostatakse, ja sellega seoses võivad tekkida ebameeldivad üllatused. Kõigele lisaks *this väärtuse hõivamise lubamine, vähendab ka natuke ohtu, mis on seotud alternatiivide otsimisega ja nendes vigade tegemisega.

100% arendajatest eelistas kasutada hõivamist liikumisega lambda avaldises otse. Lisaks toetas arendajatest 88,9% geneeriliste lambda avaldiste kasutamist üle teiste alternatiivide. 77,8% arendajatest leidis kasulikuna *this hõivamise väärtusena võrreldes pakutud alternatiividega. 94,6% arendajatest eelistas kasutada konstantavaldises samuti lambda avaldisi. Lambda avaldiste valdkonna uudsusvalmidus oli keskmiselt 90,3%. (vt Lisa 1)

## 3.7 Erindid ja tõendajad

Käitusaegse programmi kiirus võib muutuda natuke aeglasemaks uncaught_exception asendamine uncaught_exceptions-iga kuid minimaalselt, kuna tihti on vaja luua, 1-s instruktsioon, lokaalset muutujat, mis hoiab püüdmata erindite arvu ja antud funktsioonide omavahelised erinevused instruktsioonide arvus. Staatiline tõendaja ei mõjuta käitusaegse programmi kiirust, kuna tõendamine toimub kompileerimise ajal. Erindi spetsifikatsiooni integreerimine tüübi süsteemi kiirust otseselt ei muuda, sest ka see osa lahendatakse kompileerimise ajal.

Funktsioon püüdmata erindite loetlemiseks on ohutum kui selle eelkäia, mis lihtsalt tuvastas kas püüdmata erindid eksisteerivad, kuna nüüd on ta kohati võimeline reageerima kui erindite hulk muutub nendega tegelemise jooksul. Automatiseeritud sõnumiga tõendaja ohutust parandab selle võrra, et hoiab koodi juppe kopeerimaks sõnumi tekstiks, sest koodi jupp võib muutuda ja teksti muutmist võidakse ära unustada. Erindi spetsifikatsiooni integreerimine tüübi süsteemi muutis tüübid ohutumaks kohati kohtades, kus erindid pidid töötama koos otsendusoperaatoriga (dereference) või koos teisendusoperaatoritega.

38,9% arendajatest eelistasid kasutada uuemat funktsiooni püüdmata erindite loetlemiseks kui samas selle traditsioonilist variatsiooni eelistas tervelt 44,4%. 88,9% eelistas kasutada automatiseeritud sõnumiga staatilist tõendit eeldusel, et sõnum ei vajanud korrigeerimist arendaja poolt. 72,2% arendajatest arvas, et erindi spetsifikatsioon peaks olema tüübi süsteemi osa. Erindite ja tõendajate valdkonna uudsusvalmidus oli keskmiselt 66,7%. (vt Lisa 1)

## 3.8 Mälu hõivamine ja vabastamine

Käitusaegse programmi kiirus on mõjutatud seoses globaalse operaator delete koos suurus parameetriga, kuna lisandub täiendav instruktsioon seoses täiendava argumendi lisamisega. See võib veel täiendavalt olla mõjutatud delete operaatorite instruktsioonide arvu erinevustest. Samas võib toimuda hoopis kiiruse paranemine kui see koodiosa sattuks vahemällu, sest vahemälus olles ta peaks küsima ka objekti suurust, mis võtab aega kui objekt ise ei asu vahemälus ja suurus pole määratud parameetrina. Üleüldiselt mälu hõivamine üle-joondatud andmetega on aeglasem kui tavaline mälu hõivamine või C-stiili mälu hõivamine. Kuid võib osutada kiiremaks vale jagamise (false sharing) situatsiooni kohtades kui üle-joondamise

suurus on võrdväärne vahemälu liinisuurusega (line size). Vale jagamine langetab kiirust sellega, et küsib andmeid RAMist ja mitte vahemälust otse.

Ohutusest rääkides delete koos parameetriga suurus on ohtlik kui eksitakse korrektse suuruse sisestamisega arendaja poolt. Mälu hõivamise ja vabastamise ümbersõnastamine on ka kohati ohtlik, kuna optimeerimisel võib tekkida vigu mida pole varem nähtud. Antud vead on kompilaatori loojate ülesanne parandada ega üldjuhul ei mõjuta otseselt teisi arendajaid. Mälu hõivamine koos üle-joondamisega on kohati ohtlik, kuna selle kasutamises on kergesti võimalik eksida. Samas seda on ohutum kasutada üle-joondatud andmetega võrreldes selle alternatiividega.

44,4% arendajatest eelistas kasutada suurus parameetriga operaator delete-i kui operaator new oli määratud samuti suurus parameetriga. Ülejäänud, 55,6%, eelistas kasutada klassikalist operaator delete-i. 94,4% arendajatest oleks nõus lubama kompilaatoritel optimeerida välja viita. Vale jagamise korral 38,9% arendajatest oli valmis kasutama üle-joondatud viitasid ja 22,2% oli valmis kasutama C-stiili üle-joondatud mäluhaldust. Mälu haldamise valdkonna uudsusvalmidus oli keskmiselt 59,2%. (vt Lisa 1)

## 3.9 Atribuudid ja nimeruumid

Käitusaegse programmi kiirus ei muutu seose atribuutidega ja nimeruumidega, kuna tegu on kompileerimisaegsete muudatustega.

Atribuudid keskenduvadki ohutuse tagamisele. Võime hoiatada, et antud funktsionaalsus on vananenud on äärmiselt tähtis ohutusele. Muidugi on tähtis ka teatada kompilaatorit, et mõni muutuja ei pruugi olla kasutuses, mõni funktsioon peab kindlasti tagastama mingi väärtuse ja kohati tingimuslauses on lubatud mitme tingimuse läbimine. Lisaks on tähtis, et kompilaatorite loojad saaksid lisada oma atribuute samas stiilis. Sellega hoidudes makrode kasutamisest ja oma variatsioonidest. Atribuutide lubamine nimeruumidele ja loend tüüpidele on ka tähtis, kuna hoiab ära segadust, miks atribuudid mujal toimivad, aga mitte nendel objektidel. Atribuutide nimeruumide kasutamine vähendab ohutuse mõttes ainult trükivea esinemise tõenäosust. Astmeliste nimeruumide ühendamine omakorda vähendab tekstijoondamisega seotuid probleeme.

94,4% arendajatest eelistas kasutada hoidumise atribuuti üle alternatiivide. 83,3% arendajatest eelistas kinnitada ebatraditsioonilist koodikirjutamist atribuutide abil. 72,2% arendajatest arvas, et kompilaatori loojatel peaks olema võimalus lisada oma atribuute standard atribuutide stiilis. 50% arendajatest eelistas kasutada nimeruumi kontrolli atribuuti. Teine osapool eelistas nimetada atribuute välja koos nende nimeruumidega. 83,3% arendajatest eelistas astmeliste nimeruumide ühendamist. Atribuutide ja nimeruumide valdkonna uudsusvalmidus oli keskmiselt 76,6%. (vt Lisa 1)

## 3.10 Mallid ja volditud avaldised

Mallid ja volditud avaldised üldiselt käitusaegse programmi kiirust ei mõjuta, kuna lahendatakse nad suuresti kompileerimise ajal. Kuid geneeriliste argumentide dedutseerimine võib natuke aeglustada käitusaegset programmi kohati, kui kasutatakse mallita dedutseerimisjuhendit (deduction guide), umbes 2 instruktsiooni võrra.

Ohutusest rääkides geneerilised muutujad on ohutumad kui selle alternatiivid juba lihtsalt potentsiaalsete trükivigade pärast. Geneeriliste argumentide tuvastamine omakorda aitab hoiduda valede tüüpide sisestamisest ja ühtlustab algväärtustamist tavaliste klassidega, mis vähendab vigade arvu seoses loomisfunktsioonide loomisega. Väärtus parameetrite deklareerimine autoga parandab ohutust ainult selle võrra, et aitab hoiduda juhuslikust vale tüübi määramisest. Ohutuse vaatepunktist voltimine lihtsustab arvutamist parameeter pakkidega marginaalselt võrreldes alternatiividega ja seetõttu aitab hoiduda paljudest potentsiaalsetest loogika ja trükkivigadest. Tühja voldi eristamine 0 väärtusega voldist on kohati tähtis, kuna võib viia vigadeni, mis toovad esile näiteks massiivi loomist ühe 0 väärtusega elemendiga kuigi oli vaja saada tühi massiiv. Antud loogika vead on ohtlikud kuna neid on raske tuvastada, näiliselt korrektse koodi alt.

83,3% arendajatest eelistas kasutada geneerilisi muutujaid. 22,2% arendajatest eelistas kasutada vastavalt kontekstile mallides typename või class võtmesõna parameetri loomisel, aga enamus ehk 61,1% eelistas kasutada hoopis ainult typename võtmesõna. 77,8% arendajatest eelistas lasta kutsutud geneeriliste konstruktorite ja funktsioonide argumente tuvastada kompilaatori poolt. 94,4% arendajatest eelistas kasutada voltimise avaldisi parameeter pakkide arvutamiseks. Mallide ja volditud avaldiste valdkonna uudsusvalmidus oli keskmiselt 84,7%. (vt Lisa 1)

## 3.11 Järeldus

Uued C++ keele võimalused toovad palju huvitava ja mõned võimalused on võetud vastu kõigi poolt lausa üksmeelselt:

- mitmerealised konstantavaldised;
- lambda avaldises hõivamine liikumisega;
- vahemiksilmuse üldistamine;
- binaarnotatsioon;
- struktuursidemed;
- konstantavaldistes lambda avaldised;
- viita välja optimeerimine;
- volditud avaldised;
- hoidumise atribuut.

Eelnevates peatükkides sai näidatud, et C++14 ja C++17 keele võimalused enamus ajast käitusaegselt ei muutunud. Erinevalt C++11 uuendustest, mis nii mõnigi kord mõjutas käitusaega. C++14 ja C++17 uuendused olid eelkõige mõeldud C++11 täiendamisele. Tundub et igale lisatud käitusaegsetele omadustele C++11 lisati C++14 ja C++17 raames kompileerimisaegsed muudatused. Käitusaegsetes süsteemides suuri kiiruse paranemisi ei ootaks.

Kood on ohutuse mõttes ühtlustunud. Paljud keele uuendused, mis iseseisvalt on mahult väiksed on mõeldu nii öelda keele lappimiseks, mille tulemusena keele süntaks tundub palju sujuvama maastikuna. Suured muudatused on muutnud keele ühtlasemaks.

# Kokkuvõte

C++14 ja C++17 võimaluste uurimistega sai autor teada, mis on need 9 potentsiaalset uut keele võimalust, mis istuvad peaaegu igale C++ arendajale. Autor mõistis, et kui palju võimalusi tekkis kompileerimisaegsete lahenduste loomisega.

Edasiarenduseks on siin ruumi küllaga. Vaadates käesoleva bakalaureuse teksti, jääb viimane peatükk eelnevatega võrreldes üldiseks, sest küsimustiku vastanute valim oli väike sihtgruppi spetsiifika tõttu ja analüüs kiiruse mõõtmise analüüs oli piiratud ühe viisiga. Üldpilti on võimalik süveneda rohkem laiemale sihtrühmale suunates. Samuti on võimalik jõudlus analüüsi teha alternatiivsete viisidega. Muidugi on võimalik oodata C++20 versiooni tulekut.

# Kasutatud kirjandus


Barbati, A. G. (2012). [[deprecated]] attribute. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3394.html

Brown, W. E. (2011). A Proposal to Tweak Certain C++ Contextual Conversions. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3253.pdf

Brown, W. E. (2014). Extending static_assert. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3846.pdf

Crowl, L. (2012). C++ Sized Deallocation. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3432.html

Crowl, L. (2013). Digit Separators. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3661.html

Crowl, L., & Carruth, C. (2012). Clarifying Memory Allocation. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1634.htm

Edwards, H. C., Vandevoorde, D., Trott, C., Finkel, H., Reus, J., Maffeo, R., & Sander, B. (2015). Lambda Capture of *this by Value as [=,*this]. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0018r2.html

Eesti Standardikeskus. (2001). INFOTEHNOLOOGIA. SÕNASTIK Osa15: Programmikeeled, *2001*. Retrieved from https://www.evs.ee/eelvaade/evs-iso-iec-2382-15-2001-et.pdf

Finkel, H., & Smith, R. (2015). Inline Variables. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4424.pdf

Garcia, J. D. (2016). Standard and non-standard attributes. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0283r0.pdf

Garcia, J. D., Sanchez, L. M., Torquati, M., Danelutto, M., & Sommerlad, P. (2015). Using non-standard attributes. *WG21 Document*. Retrieved from http://www.open-

std.org/jtc1/sc22/wg21/docs/papers/2015/p0028r0.pdf

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).

ISO/IEC. (2014). Information Technology — Programming languages — C++, *2014*(Fourth edition).

ISO/IEC. (2016). Working Draft , Standard for Programming Language C ++. *WG21 Document*.

ISO/IEC JTC1/SC22/WG21. (2017). ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee - ISOCPP. Retrieved May 1, 2017, from http://www.open-std.org/jtc1/sc22/wg21/

Kawulak, R. (2014). Nested namespace definition. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4026.html

Köppe, T. (2016). If statement with initializer. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0305r0.html

Le Jehan, T. (2015). Unary Folds and Empty Parameter Packs. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4358.pdf

Maurer, J. (2014). N4320: Make exception-specifications be part of the type system. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4320.html

Merrill, J. (2012). Return type deduction for normal functions. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3386.html

Nelson, C. (2012). Dynamic memory allocation for over-aligned data. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3396.htm

Nelson, C., & Smith, R. (2015). Feature-testing preprocessor predicates for C++17. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0061r0.html

Niebler, E. (2016). Generalizing the Range-Based For Loop. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0184r0.html

Reis, G. Dos. (2013). Constexpr Variable Templates. *WG21 Document*. Retrieved from https://isocpp.org/files/papers/n3615.pdf

Reis, G. Dos. (2015). Construction Rules for enum class Values. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0138r0.pdf

Smith, R. (2012). Relaxing syntactic constraints on constexpr functions. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3444.html

Smith, R. (2013). constexpr member functions and implicit const. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3598.html

Smith, R. (2014a). Adding u8 character literals. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4197.html

Smith, R. (2014b). Allow constant evaluation for all non-type template arguments. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4198.html

Smith, R. (2014c). Allow typename in a template template parameter. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4051.html

Smith, R. (2014d). Attributes for namespaces and enumerators. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4196.html

Smith, R. (2015). Guaranteed copy elision through simplified value categories. *WG21 Document*. Retrieved from http://www.open-

std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html

Smolsky, O. (2015). Extension to aggregate initialization. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0017r0.html

Spertus, M., & Vandevoorde, D. (2013). Template parameter deduction for constructors. *WG21 Document*, (Revision 1). Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3602.html

Stack Overflow. (2015). Stack Overflow Developer Survey 2015. Retrieved May 1, 2017, from http://stackoverflow.com/research/developer-survey-2015#tech-lang

Stroustrup, B. (2003). Literals for user-defined types. *WG21 Document*.

Stroustrup, B. (2007). Evolving a language in and for the real world: C++ 1991-2006. *3rd ACM SIGPLAN Conference on History of Programming Languages*, 4:1-4:59. http://doi.org/10.1145/1238844.1238848

Sutter, H. (2014). uncaught _exceptions. *WG21 Document*, (Revision 1). Retrieved from https://isocpp.org/files/papers/N4152.pdf

Sutter, H., Bjarne, S., & Reis, G. Dos. (2015). Structured bindings. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r0.pdf

Sutton, A., & Smith, R. (2014). Fold expressions. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4191.html

Tomazos, A. (2015). Proposal of [[unused]], [[nodiscard]] and [[fallthrough]] attributes. *WG21 Document*. Retrieved from http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0068r0.pdf

Touton, J., & Spertus, M. (2015). Template Argument Type Deduction. *WG21 Document*, (Revision 1). Retrieved from http://open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4469.html

Vali, F. S., Voutilainen, V., & Reis, G. Dos. (2015). Constexpr Lambda. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4487.pdf

Vali, F., Sutter, H., & Abrahams, D. (2012). Proposal for Generic (Polymorphic) Lambda Expressions. *WG21 Document*, (Revision 2). Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3418.pdf

Voutilainen, V. (2013). Auto and braced-init-lists. *WG21 Document*. Retrieved from http://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3681.html

Voutilainen, V. (2013). Generic lambda-capture initializers, supporting capture-by-move. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3610.html

Voutilainen, V. (2013). Member initializers and aggregates. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3605.html

Voutilainen, V. (2015). Static if resurrected. *WG21 Document*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4461.html

# Summary


## *Analysis of Usefulness on Changes in C++ Specification*

*Bachelor´s Thesis*

This bachelor´s thesis focuses on changes in C++ based on its new language features. The aim of this paper is to introduce new potentially useful C++ language features, which will find its use over time. The reason why this topic was chosen was because of authors interest in current C++ language development.

Over the course of this thesis author writes about C++14 and C++17 language features, about issues they solve and solutions their offer. Also, author writes about their performance, safety and its value in developer eyes. By learning about these features author found out:

- what are 9 features, on which usage all developers agree on almost in unison;
- how much potential these offer to compile-time dependent solutions;
- how much more smoother and safer language has become.

There is still a lot of room for further developments. By looking at this bachelor´s thesis, it is possible to see that questionnaire was more focused on smaller target group, because its difficult questions and performance analysis was done with only one type of measurement. Therefore, one could expand target group though more simplified questionnaire or do performance analysis based on other type of measurements. C++20 language features are up next, so it is also possible to wait for those.

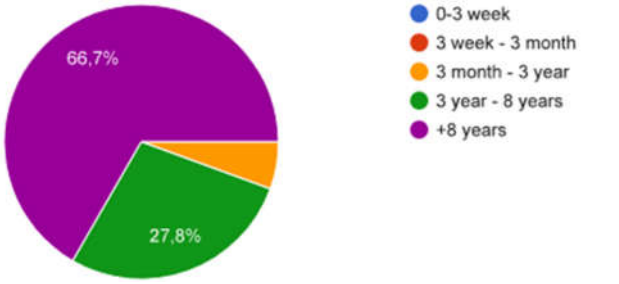
# Lisad

## Lisa 1 Küsitluse tulemused

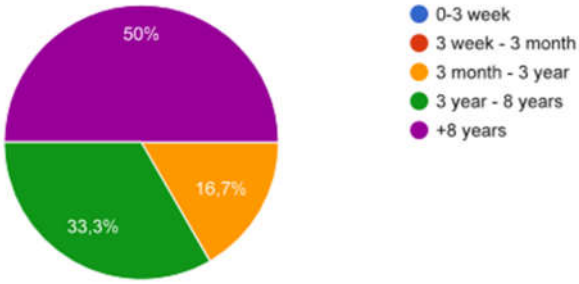
C++

18 vastust

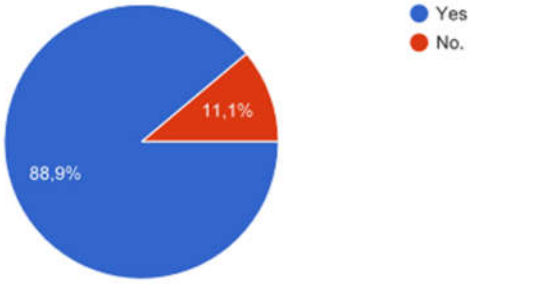
Experience in programming (studying included) (18 vastust)




Experience in C++ (studying included) (18 vastust)

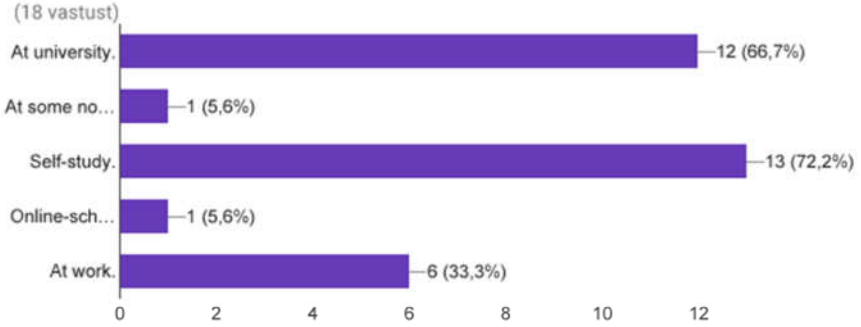



Have you worked with C++ last 3 months? (18 vastust)



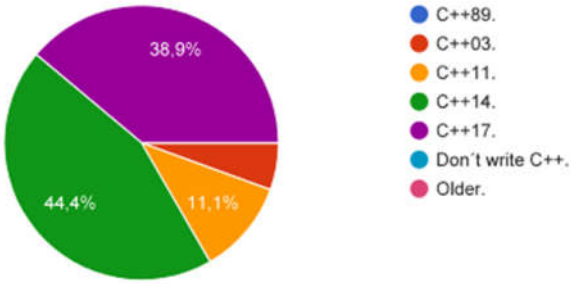
Did or are you studying computer science related fields somewhere?

(18 vastust)




What C++ specification version you currently follow (work with) mostly, when programming?

(18 vastust)



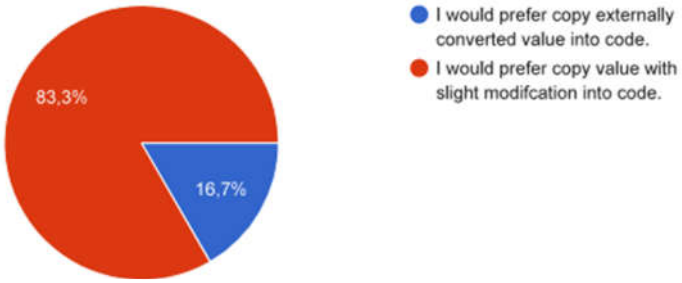

## Literals

You were asked to add following binary constant 1010 0011 0001 (binary notation) to your code:

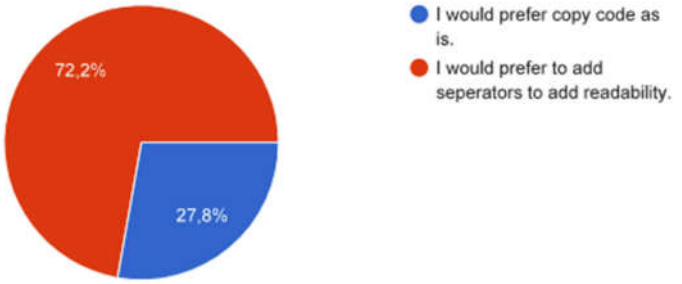
(18 vastust)

You were asked to add following constant 1AFF.C7p15 (hexadecimal exponential notation) to your code:

(18 vastust)

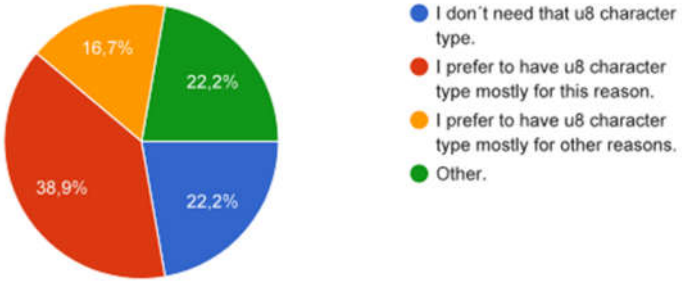


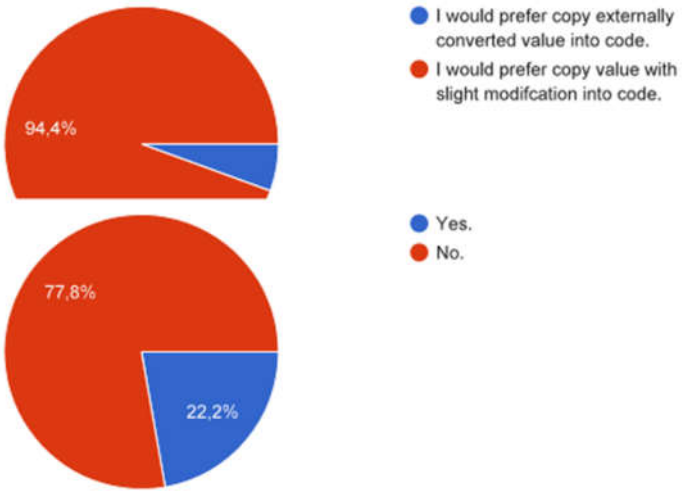

You were asked to add few long constants to your code: (18 vastust)




When working with characters would you prefer have less character types or you prefer have each string type to have corresponding character type?

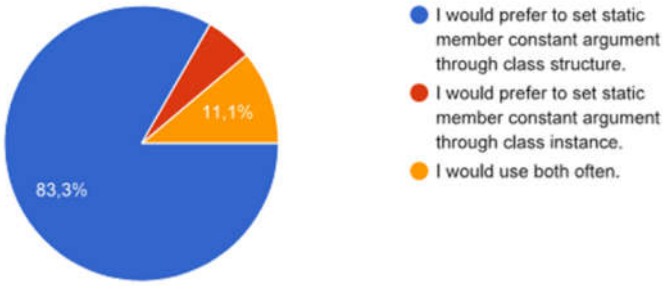
(18 vastust)





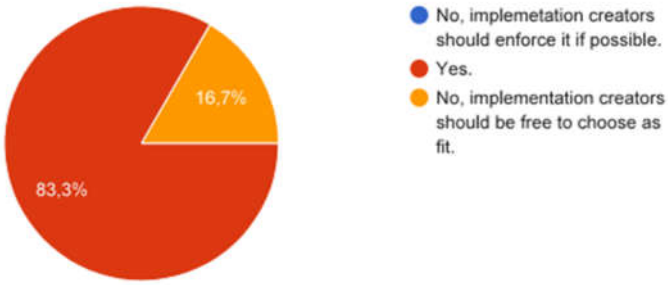

How would you prefer to set static member as constant argument?

(18 vastust)

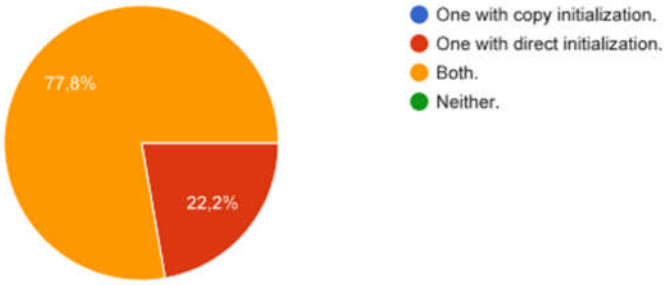



Should be optimizations like copy elision enforced by standard?

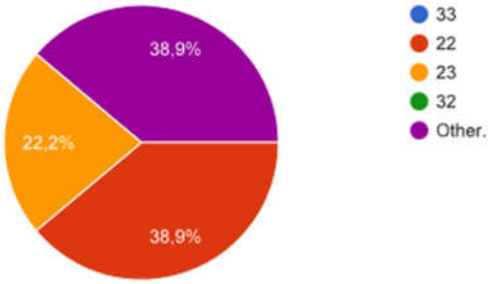
(18 vastust)



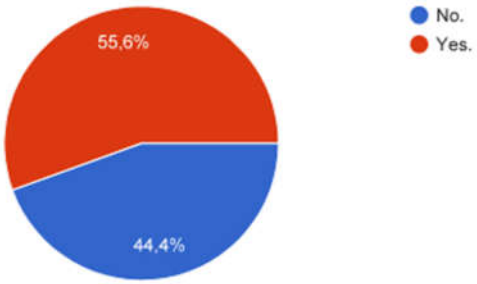
Which one should be valid code if not both in your opinion?
(18 vastust)




What program should print in your opinion? (18 vastust)



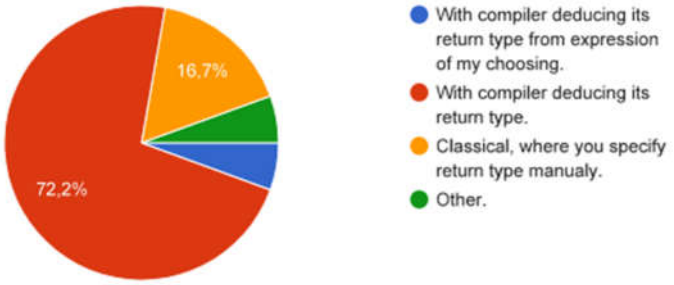

Should enum class type be allowed to be cast implicitly from its underling type during initialization?
(18 vastust)




Type deduction and structural bindings

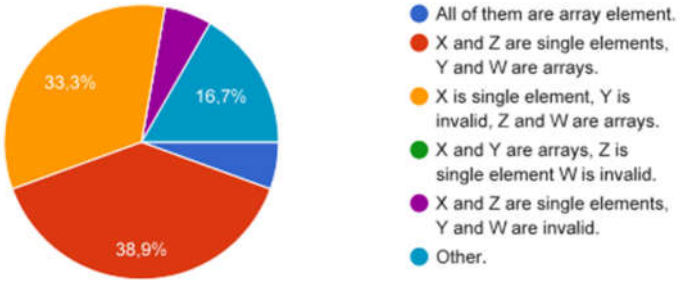
Which way would you prefer to write functions most of time:

(18 vastust)



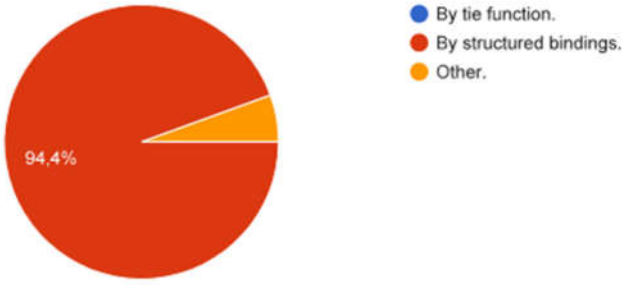

Which of following types should in your opinion single element and which ones should contain arrays?

(18 vastust)




Which way would you prefer to have multiple values returned most of time?

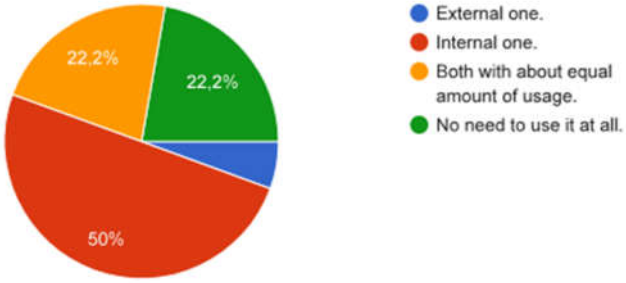
(18 vastust)




Conditions and loops

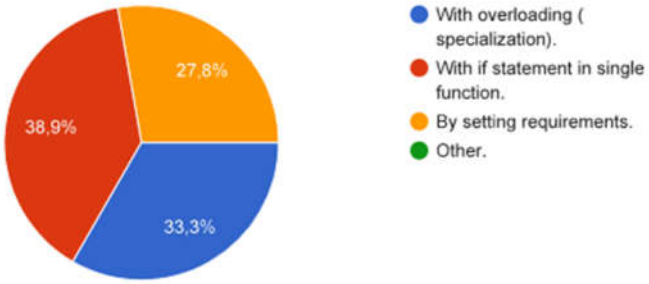
Would you prefer to use internal or external way to detect headers?

(18 vastust)



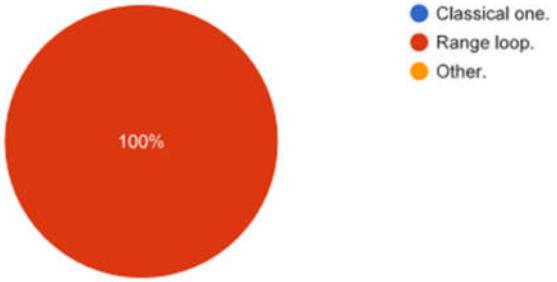

How would you prefer to change behavior of a function depending of type during compile-time most of time?

(18 vastust)

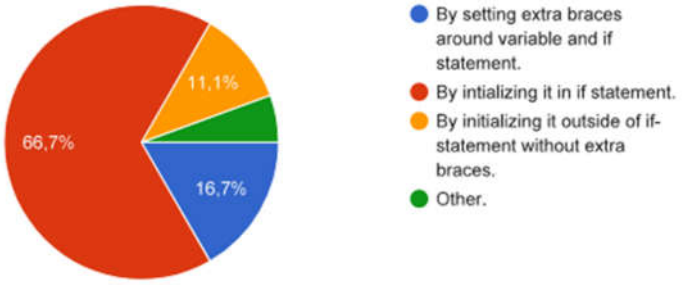



Which of loops would you prefer to use to iterate through class with different type of begin and end:

(18 vastust)



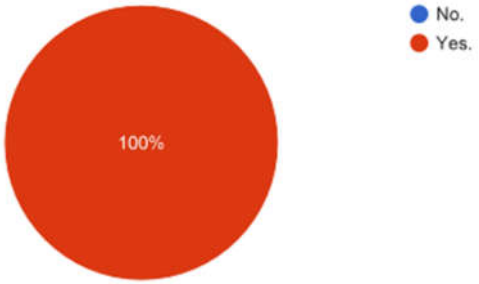
How would you prefer to initialize variable, which you need during if-statement?




## Aggregates and constant expressions

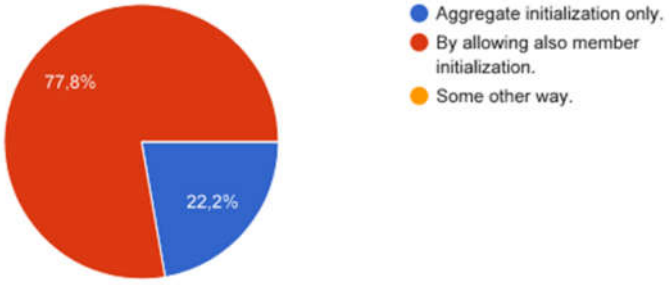
Would you prefer to have ability to write multi-line constant expression function?

(18 vastust)

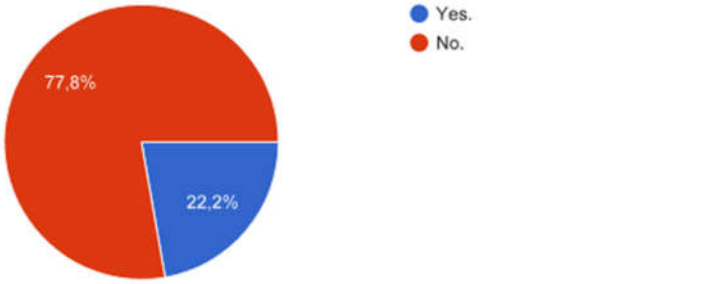



Which way would you prefer to initialize aggregates most of time?

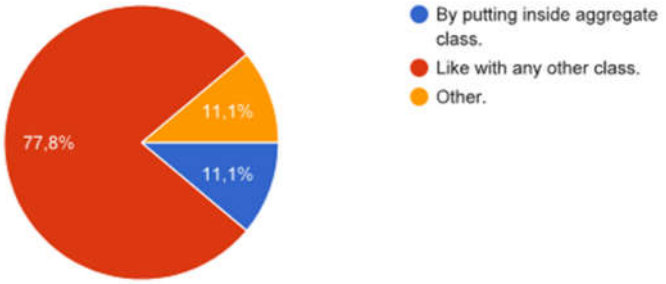
(18 vastust)



Should be static member redeclared in translation unit? (18 vastust)

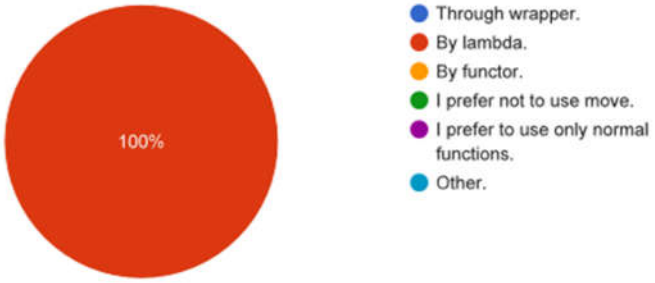



How would you prefer to add base class to aggregate class?
(18 vastust)




Lambda

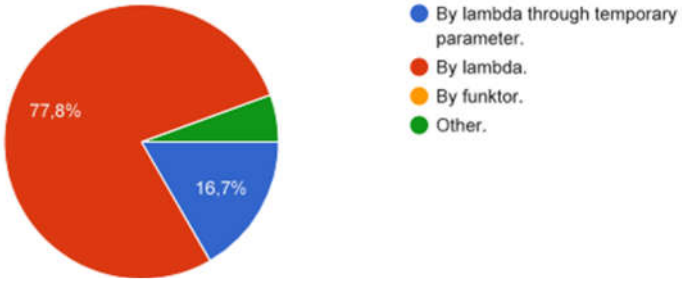
How do you prefer to move closure parameters? (18 vastust)



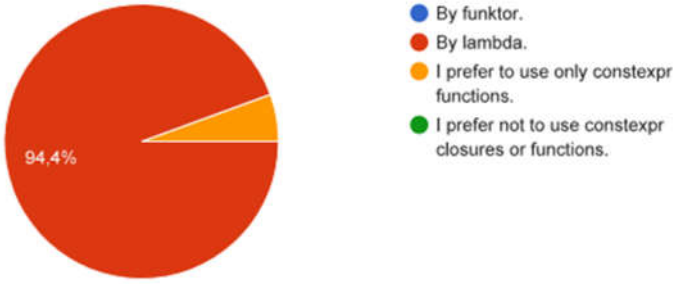

How do you prefer to deal with generalized closure? (18 vastust)

How do you prefer to use/call owning class in closure, if by reference is not an option.

(18 vastust)



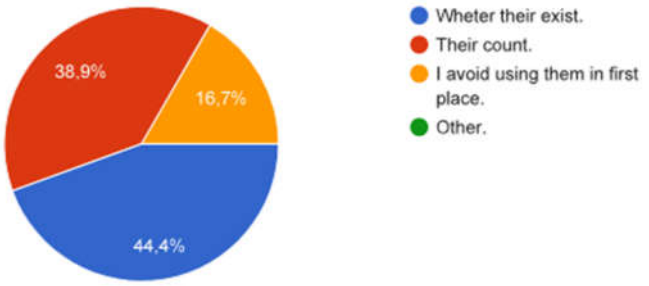

How do you prefer to use constant expression closures? (18 vastust)



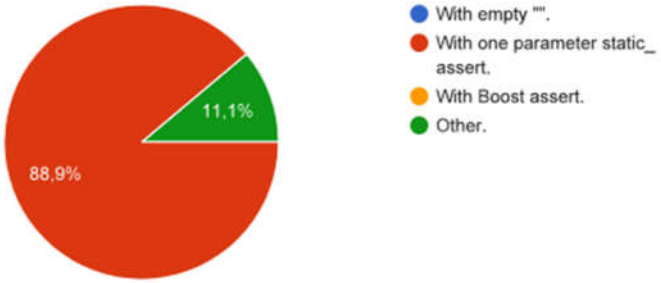

## Exceptions and assert

When you ask for uncaught exceptions, should you be given back whether they are found or their count?

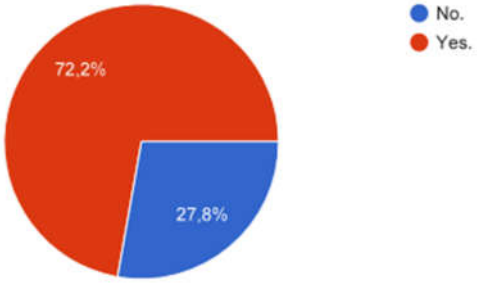
(18 vastust)




How would you prefer to create assertion during compile-time, which does not require custom message?
(18 vastust)



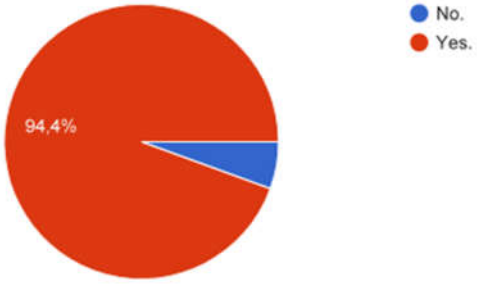

In your opinion should exception be part of type system?
(18 vastust)



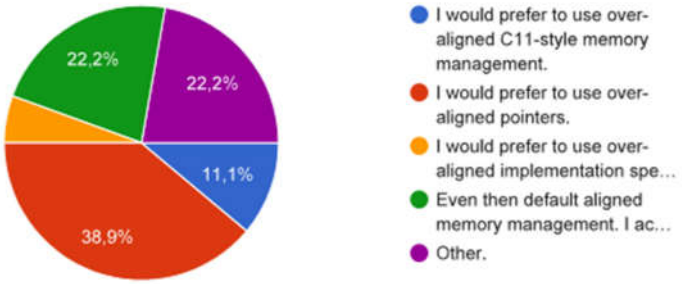

Memory management

Which would you prefer to use in case sized allocation? (18 vastust)

Should pointers be allowed to be optimized out? (18 vastust)

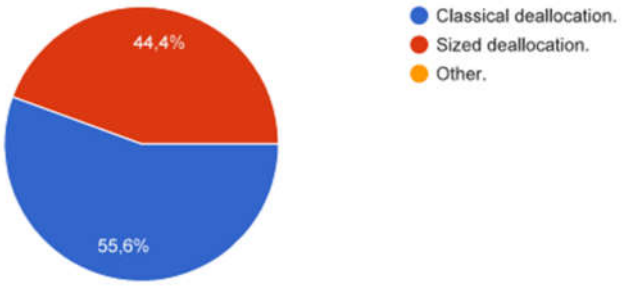



How would you like to deal with pointer/alloc, which keeps having cache misses due false sharing?
(18 vastust)



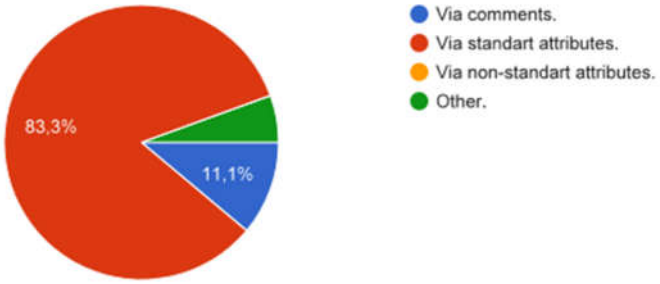

Attributes and namespaces

How would you prefer to have deprecated features marked?
(18 vastust)



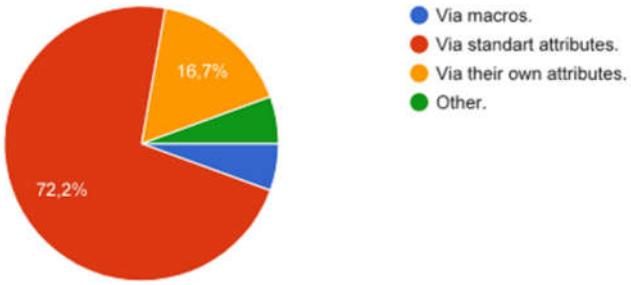

How would you prefer write down your intent, when writing code which usually is considered source of bugs.

(18 vastust)



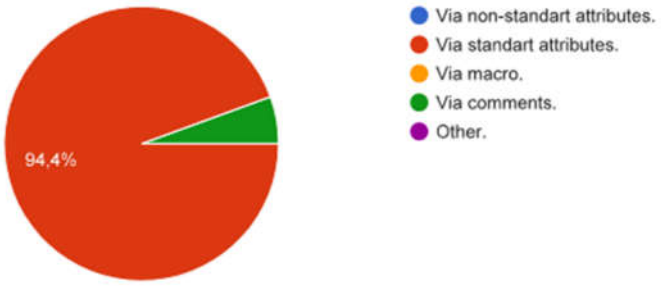

How would you prefer to see implementation creators mark their entities to allow their own functionality on those entities?

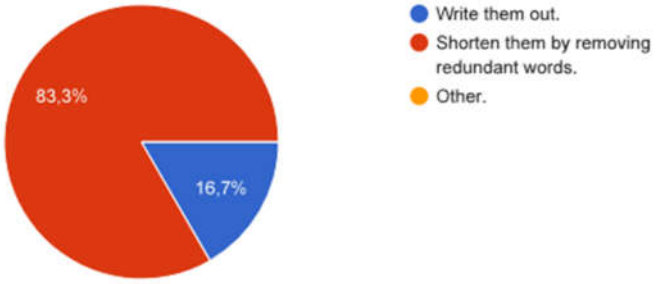
(18 vastust)




How would you prefer to mark multiple attributes from same namespace for same entity.

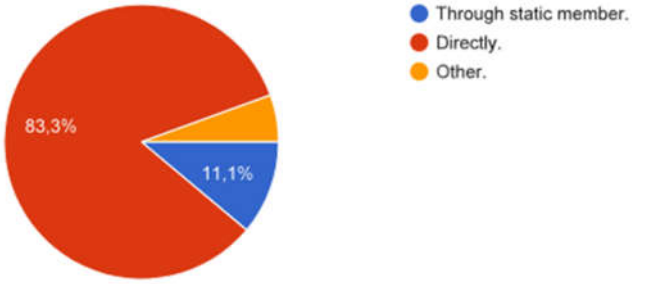
(18 vastust)



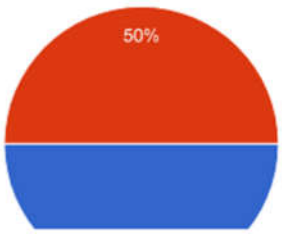




Templates

How would you prefer to write generalized constants? (18 vastust)

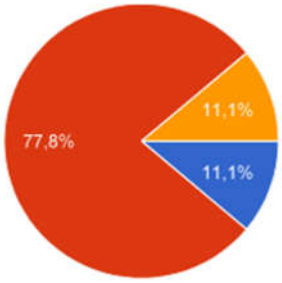



What do you prefer to use as template parameter? (18 vastust)




- By naming each attribute.
- By setting common namespace and after naming each attribute.
- Other.

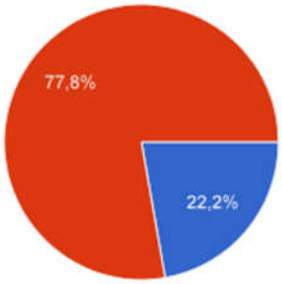
s?




- By specifing its arguments.
- By letting its arguments to be deduced.
- By make functions.
- Other.

Would you like to be able to write values as template arguments?

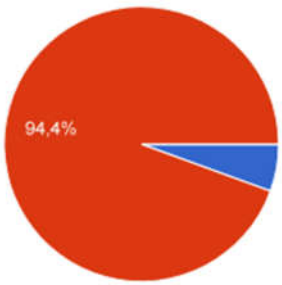
(18 vastust)



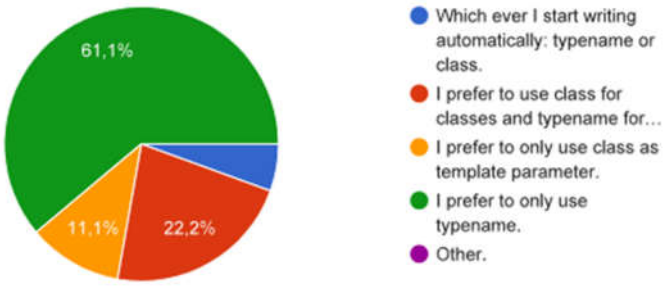

- No.
- Yes.

How would you prefer to calculate with parameter packs?

(18 vastust)




- With overloading.
- With folding.
- Other.




yes/no/other.

This
got a little tedious once I realized you're simply asking that or nay for selected modern C++ features

The
questions are pretty tough even if they are also for beginners 😅

Results
would be nice :)

Igapäevaste vastuste arv