

Jarkko Ahvenniemi

## **C++ VS. RUST**

Vertailua muistinhallinnan helppokäyttöisyydestä

Informaatioteknologian ja viestinnän tiedekunta

Kandidaatintyö

Joulukuu 2025

# TIIVISTELMÄ

Jarkko Ahvenniemi: C++ vs. Rust – Vertailua muistinhallinnan helppokäyttöisyydestä

Kandidaatintyö

Tampereen yliopisto

Tieto- ja sähkötekniikan kandidaattiohjelma, tietotekniikka

Joulukuu 2025

---

Ohjelmointikielen helppokäyttöisyydellä on suora vaikutus ohjelman tuottamisen haastavuuteen, minkä takia se on otettava huomioon kielen valinnassa. Tässä kandidaatintyössä tutkitaan C++- ja Rust-ohjelmointikielten muistinhallinnan helppokäyttöisyyttä, sillä kielille on samanlaisia käyttökohteita ja kielten muistinhallintamallit poikkeavat toisistaan merkittävästi. Työn tavoitteena on selvittää, miten ohjelmointikielen helppokäyttöisyyttä voi mitata, mitä yhtäläisyyksiä ja eroja vertailtujen kielten välillä on ja onko yksi kielistä muistinhallinnaltaan merkittävästi helppokäyttöisempi kuin toinen.

Työ jakaantuu teoria- ja vertailuosioihin. Kirjallisuuskatsauksena toteutetussa teoriaosiossa esitellään ohjelmointikielen helppokäyttöisyyteen vaikuttavia tekijöitä. Lisäksi esitellään vertailtavat ohjelmointikielien ja niiden tärkeimmät erot erityisesti muistinhallinnan näkökulmasta. Vertailuosiossa pohditaan kielten muistinhallinnan helppokäyttöisyyttä esitettyjen helppokäyttöisyysmekanismien mukaisesti.

Ohjelmointikielen helppokäyttöisyyden osa-alueiksi tunnistetaan syntaksin selkeys, kääntäjän tai tulkin ilmoitukset, oppimiskäyrät, ohjelmointikielille saatavilla oleva dokumentaatio sekä kielen ympärille muodostuneen käyttäjäyhteisön tarjoama apu. Modernin C++:n muistinhallinta perustuu älykkäiden osoittimien hyödyntämiseen ja vakiintuneiden ohjelmointikäytäntöjen noudattamiseen. Rustissa muistiturvallisuus saavutetaan omistuksiin pohjautuvan mallin turvallisuussäännöillä, joiden noudattamista valvotaan käännösaikana.

Vertailussa havaitaan, että C++ sallii enemmän implisiittistä käyttäytymistä, kun taas Rust vaatii selkeyttä eksplisiittisellä syntaksilla. Rustin kääntäjän todetaan antavan tarvittaessa yksityiskohtaisempaa tietoa esiintyneestä virheestä. Molempien kielten oppimiskäyriä pidetään haastavina, mutta eri tavoilla. C++:lla kirjoitettu ohjelma kääntyy helpommin, mutta muistinhallintavirheiden välttäminen on vaikeampaa. Vastaavasti Rustilla on haastavampaa saada ohjelma kääntymään, mutta lopputulos on muistiturvallisempi. Dokumentaatiota ja yhteisön apua todetaan olevan hyvin saatavilla molemmille kielille.

Vertailun tuloksista päätellään, että Rust on helppokäyttöisempi vaihtoehto silloin, kun muistiturvallisuus on ehdoton vaatimus ohjelmalle. Rustin turvallisuussääntöjä rikkova ohjelma ei käänny lainkaan, kun taas C++:ssa ajoaikaiset ongelmat ovat yleisempiä, vaikka ohjelma käännyisi ongelmitta. C++:aa pidetään kuitenkin helppokäyttöisempänä yksinkertaisempien ohjelmien kirjoittamiseen, sillä ohjelma on helpompi saada kääntymään.

---

Avainsanat: ohjelmointikielien, C++, Rust, muistinhallinta, helppokäyttöisyys, muistiturvallisuus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

# TEKOÄLYN KÄYTTÖ OPINNÄYTTEESSÄ

Opinnäytteessäni on käytetty tekoälysovelluksia:

- Ei
- Kyllä

Ilmoitukseni mukaan olen käyttänyt opinnäytteessäni tutkielmaprosessin aikana seuraavia tekoälysovelluksia:

Tekoälysovellusten nimet ja versiot: ChatGPT, kielimallin versiot välillä GPT-4 ja GPT-5.1.

Käyttötarkoitus: Tekoälyä on hyödynnetty aiheen ideoinnissa, yleisen tekstipalautteen hankkimisessa, koodiesimerkkien kehittämisessä, vertailtujen ohjelmointikielten yhtäläisyyksien selvittämisessä sekä joidenkin englanninkielisten termien suomentamisessa.

Osiot, joissa tekoälyä on käytetty: Luvut 2-4

Olen tietoinen siitä, että olen täysin vastuussa koko opinnäytteeni sisällöstä, mukaan lukien osat, joissa on hyödynnetty tekoälyä, ja hyväksyn vastuun mahdollisista eettisten ohjeiden rikkomuksista.

# SISÄLLYSLUETTELO

1. JOHDANTO .....	1
2. OHJELMOINTIKIELEN HELPPOKÄYTTÖISYYDEN MITTAAMINEN .....	2
2.1 Syntaksin selkeys .....	2
2.2 Kääntäjän tai tulkin ilmoitukset .....	3
2.3 Oppimiskäyrät .....	4
2.4 Saatavilla oleva dokumentaatio .....	5
2.5 Käyttäjyhteisöjen tarjoama apu .....	6
3. VERTAILTAVAT OHJELMOINTIKIELET .....	7
3.1 C++ .....	7
3.2 Rust .....	8
4. KIELTEN MUISTINHALLINNAN HELPPOKÄYTTÖISYYDEN VERTAILUA .....	11
4.1 Syntaksin selkeys .....	11
4.2 Kääntäjän tai tulkin ilmoitukset .....	13
4.3 Oppimiskäyrät .....	16
4.4 Saatavilla oleva dokumentaatio .....	17
4.5 Käyttäjyhteisöjen tarjoama apu .....	18
5. YHTEENVETO .....	19
LÄHTEET .....	20

# LYHENTEET JA MERKINNÄT

GNU  
GCC  
MSVC

GNU's Not Unix  
GNU Compiler Collection  
Microsoft Visual C++

# 1. JOHDANTO

Ohjelmistotuotannon alalla huomataan jatkuvasti muutoksia päivittyvien ja kokonaan uusien ohjelmointikielten myötä. 2010-luku toi mukanaan muun muassa modernisointia C++-ohjelmointikielen, rinnakkaisuuteen keskittyvän Go-kielen ja muistiturvallisuutta tarjoavan Rust-kielen. Ohjelmointikielen valintaan on siis nykyään monia vaihtoehtoja, myös samankaltaisille käyttötarkoituksille. Miten ohjelmointikieli tulisi siis valita, jos tarkoituksena olisi esimerkiksi tuottaa suorituskykyinen ohjelma mahdollisimman vaivattomasti?

Tämän kandidaatintyön tarkoituksena on verrata C++- ja Rust-ohjelmointikielten helppokäyttöisyyttä selvittämällä, mitä yhtäläisyyksiä ja eroja kielillä on, mitkä asiat vaikuttavat ohjelmointikielen koettuun helppokäyttöisyyteen ja miten nämä asiat näkyvät kielten käytössä. Erityisesti vertailussa tarkastellaan kielten muistinhallintaa, sillä se on vertailtavien kielten keskeinen ero.

Nämä kielet on valittu vertailtaviksi, koska niillä on hyvin samanlaiset käyttökohteet. Rust on alusta asti suunniteltu täyttämään aukko matalan tason ohjelmointikielelle, joka on samalla muistiturvallinen (Matsakis & Klock, 2014). Sitä voi siis pitää korvaajana C++:lle, joka on saanut vastaavia ominaisuuksia, kuten älykkäät osoittimet (engl. *smart pointers*) vasta C++11-version myötä (Lucas et al., 2023). C++-kieleen verrattuna Rust on suhteellisen uusi, mutta sillä on jatkuvasti kasvava suosio muun muassa järjestelmäohjelmoinnissa (engl. *systems programming*), minkä voi huomata esimerkiksi sen käyttöönotosta Linux-käyttöjärjestelmäytimen kehityksessä.

Tämä kandidaatintyö on toteutettu kirjallisuuskatsauksena ja vertailuna. Aineistoa on kerätty etsimällä ohjelmointikielten helppokäyttöisyyteen liittyvää tutkimusta sekä teknistä tietoa vertailtavista ohjelmointikielistä.

Luvussa 2 määritellään, mitä ohjelmointikielen helppokäyttöisyys on ja miten sitä mitataan. Luvussa 3 esitellään lyhyesti vertailussa olevat ohjelmointikieliet. Luvussa 4 vertailaan kielten ominaisuuksia luvussa 2 tunnistettujen helppokäyttöisyyteen vaikuttavien tekijöiden mukaisesti. Lopuksi pohditaan tulosten merkitystä ja tarvetta jatkotutkimukselle.

## 2. OHJELMOINTIKIELEN HELPPOKÄYTTÖISYYDEN MITTAAMINEN

Jotta ohjelmointikielten helppokäyttöisyyttä voidaan luotettavasti verrata, on ensin tarpeellista määritellä, mitä ohjelmointikielen helppokäyttöisyys on ja miten sitä mitataan. Tässä luvussa tarkastellaan aikaisempaa tutkimusta ja kirjallisuutta ohjelmointikielten käytettävyydestä.

Helppokäyttöisyys on laaja käsite, ja tässä luvussa esitellään vain joitakin sen mahdollisia osa-alueita. Luvun ei siis ole tarkoitus kattaa ohjelmointikielen helppokäyttöisyyttä täydellisesti, vaan antaa esimerkein konkreettinen pohja myöhemmin suoritettavalle vertailulle.

### 2.1 Syntaksin selkeys

Ohjelmointikielen syntaksi on se osa kieltä, jota noudattamalla ohjelmoija lopulta kykenee ilmaisemaan tietokoneelle haluamansa toiminnallisuuden. Korkean tason ohjelmointikielen syntaksi on helpommin ymmärrettävissä kuin esimerkiksi matalan tason konekieli. Ohjelmointikielen syntaksin selkeys on siis tärkeä osatekijä kielen koetussa helppokäyttöisyydessä, ja sitä tullaan käsittelemään tässä työssä tarkasteltavien ohjelmointikielten vertailussa.

Kärjistettyinä esimerkkeinä syntaksin vaikutuksesta helppokäyttöisyyteen voi pitää esoteerisia ohjelmointikieliä, jotka ovat usein suunnitellusti vaikeakäyttöisiä, vaikka ne olisivatkin Turing-täydellisiä. Tämä tarkoittaa sitä, että niillä voidaan toteuttaa mikä tahansa ohjelmoitavissa oleva algoritmi. (Temkin, 2023)

Esoteeriset kielet ovat monesti luonteeltaan kokeellisia tai humoristisia, ja käytettävyys jää niissä vähemmän merkittäväksi. Kuuluisa esimerkki tällaisesta ohjelmointikielestä on Urban Müllerin luoma Brainfuck, joka on suunniteltu äärimmäisen yksinkertaiseksi. (Temkin, 2023) Kielen toiminta perustuu ainoastaan kahdeksaan komentoon, joita käytetään muistiosoitteen valintaan ja sen sisällön käsittelyyn. Kieli on todistetusti Turing-täydellinen, mutta käytännössä sillä ei voi kirjoittaa kuin erittäin yksinkertaisia ohjelmia sen heikon luettavuuden vuoksi. (Temkin, 2017) Käytettävyyden sivuuttamisella on siten suora vaikutus kielellä kirjoittamisen vaikeuteen.

Ohjelmoinnin opetuksessa joudutaan väistämättä harkitsemaan, mitä ohjelmointikieltä kannattaisi hyödyntää ensimmäisenä kielenä uusille ohjelmoijille. Olennaisinta on keskittyä jo valmiiksi haastaviin ohjelmoinnin ydinperiaatteisiin eikä yksittäisen ohjelmointikielen syntaksin hallitsemiseen. Asiaa tukee Pereran et al. (2021) tutkimus, jossa selvitettiin, miten aloittelijoille suunnatut yksinkertaistetut ohjelmointikielet vaikuttavat ohjelmoinnin oppimiseen. Tuloksissa esitetyn havainnon mukaan yksinkertaistettu syntaksi on ollut tärkeä osatekijä oppimisen edistämiseksi.

Käytännössä monen ohjelmoijan ensimmäistä ohjelmointikokemusta ei kuitenkaan saavuteta yksinkertaistetuilla kielillä, vaan esimerkiksi ohjelmointikurssin kautta jollain työelämässäkin mahdollisesti sovellettavalla kielellä. Lokkila et al. (2023) tutkimuksessa todetaan, että Python on syntaksinsa selkeyden vuoksi suosittu vaihtoehto ensimmäisille ohjelmointikursseille, ja esimerkiksi Java-ohjelmointikieleen verrattuna sen syntaksi soveltuu paremmin aloitteleville ohjelmoijille. Tämä johtuu siitä, että yksinkertaisemmalla syntaksilla ohjelmointivirheitä tulee tehtyä vähemmän. Tutkimuksessa kuitenkin huomattiin, että tässäkin tapauksessa valtaosa virheistä oli syntaksivirheitä, mikä edelleen korostaa syntaksin vaikutusta ohjelmointikielen helppokäyttöisyyteen.

## 2.2 Kääntäjän tai tulkin ilmoitukset

Kääntäjät (engl. *compiler*) tunnetaan parhaiten ohjelmina, jotka nimensä mukaisesti kääntävät ohjelmakoodia tietokoneen ymmärtämään muotoon. Kääntäjällä on kuitenkin ohjelmoinnin kannalta toinenkin tärkeä tehtävä: varoitusten ja virheilmoitusten esittäminen ongelmien tunnistamiseksi. (Cooper & Torczon, 2012) Kääntäjälläkin voi siis olla vaikutusta koettuun ohjelmointikielen helppokäyttöisyyteen, sillä sen tarjoamilla ilmoituksilla on suora yhteys esimerkiksi vianetsinnän vaikeuteen.

Käännetyistä kielistä poiketen tulkatuilla kielillä kirjoitettuja ohjelmia ei käännetä suoritettaviksi binääritiedostoiksi. Sen sijaan tulkiksi (engl. *interpreter*) kutsuttu erillinen ohjelma suorittaa ohjelmakoodia rivi kerrallaan. Tulkki on siis kääntäjän vastine tulkatuissa kielissä. Koska käännösprosessia ei ole, myöskään käännösaikaista virheiden havaitsemista ei tapahdu.

Kääntäjien ja tulkkien virheilmoitusten hyödyllisyyttä voidaan tarkastella niin varsinaisen ohjelmoinnin kuin ohjelmointikielen opettelun näkökulmasta. Becker et al. (2019) ovat tunnistaneet seuraavia ilmoitusten hyödyllisyyttä parantavia tekijöitä:

- virheilmoituksen ymmärrettävyys

- ohjelmoijan työmuistin kuormittamisen minimointi
- laadukas virheeseen liittyvä tieto
- virheilmoituksen positiivinen äänensävy
- esitetyt esimerkit samankaltaisista virheistä
- esitetyt ratkaisuehdotukset
- kääntäjän mukautuvuus ohjelmoijan taitotasoon
- kielen oppimisen edistäminen tukirakenteilla
- virheilmoituksen argumentoinnin johdonmukaisuus
- virheilmoituksen oikea ajoitus.

Myös Zhu et al. (2022) ovat havainneet, että kääntäjän antaman virheilmoituksen laadulla on ratkaiseva vaikutus virheen ymmärtämiseen ja korjaamiseen. Tässä tapauksessa virheet ovat liittyneet Rustin muistinhallintaan, joka esitellään tässä työssä tarkemmin luvussa 3. Helppokäyttöisyyden kannalta kääntäjä tai tulkki saattaa siis olla ongelmallinen, jos sen virheilmoitukset eivät ole tarpeeksi kuvaavia tai johtavat harhaan. Vastaavasti kuvaavat ja yksityiskohtaiset virheilmoitukset voivat auttaa selvittämään ongelman vaivattomasti.

## 2.3 Oppimiskäyrät

Erilaisten taitojen on todettu edistävän ohjelmoinnin opettelua. Näihin kuuluvat esimerkiksi abstrakti ajattelu, looginen päättely ja kyky soveltaa ohjelmoinnin käsitteitä tositilanteissa. Lisäksi aikaisempi tietämys ohjelmoinnista on nähty merkittävänä vaikuttavana tekijänä. Vastaavasti näiden esitietojen puuttuminen on vaikeuttanut oppimista. (Tahy & Czirkos, 2016) Oppimisen vaikeuteen vaikuttavat siis opettelijan olemassa oleva tieto ja osaaminen.

Yhdellä kielellä ohjelmakoodia kirjoittamaan tottuneelle ohjelmoijalle saattaa olla haastavaa totutella toiseen kieleen, joka on jollain tavalla erilainen aikaisemmin käytettyyn kieleen. Tämä johtuu siitä, että ohjelmoijalla ei välttämättä ole ennalta käsitystä uuden kielen vaatimista ohjelmointikäsitteistä tai -tavoista. Aikaisempi kokemus ohjelmoinnista ei siis takaa täydellistä osaamista kaikilla ohjelmointikielillä, sillä erilaisiin käyttötarkoituks-

siin suunnitellut kielet voivat erota semantiikoiltaan ja paradigmoiltaan merkittävästi. Semantiikalla tarkoitetaan tässä yhteydessä ohjelmointikielen tapaa käsitellä arvoja, ja paradigmalla ohjelmointimallia.

Pythoniin totunut ohjelmoija voi kokea C++:n muistinhallinnan opetteluun vaikeaksi, ja vastaavasti arvo-semantiikkaan totuneelle C++-ohjelmoijalle Pythonissa käytetty viitesemantiikka voi vaikuttaa epäselvältä. Arvo-semantiikassa muuttujan arvosta luodaan uusi kopio sijoittaessa tai välittäessä muuttuja funktiolle parametrina, ja viitesemantiikassa puolestaan uuteen muuttujaan sijoittaessa luodaan uusi viite samaan arvoon. C++:ssa sama toteutetaan viitteiden ja osoittimien avulla. Jos viitesemantiikkaa ei ota huomioon kirjoittaessa ohjelmakoodia, virheitä voi syntyä esimerkiksi listoja käsitellessä. Ohjelmassa 1 yritetään alustaa lista kahdella tyhjällä listalla ja lisätä ensimmäiseen listaan arvo 1. Todellisuudessa on kuitenkin syntynyt kaksi viitettä samaan listaolioon, jolloin ensimmäiseen listaan lisätty arvo näkyy myös toisessa listassa. Odotettu tuloste on siis `[[1], []]`, mutta ohjelma tulostaa `[[1], [1]]`.

```

1     def main():
2
3         listat = [[]] * 2
4         listat[0].append(1)
5         print(listat)
6         return 0
7
8     main()
```

**Ohjelma 1.** *Viitesemantiikkavirhe Python-ohjelmassa*

## 2.4 Saatavilla oleva dokumentaatio

Ohjelmointikielen oppiminen ja tehokas käyttö edellyttää kattavan kielidokumentaation olemassaoloa. Dokumentaatio voi pitää sisällään esimerkiksi ohjeita, esimerkkejä ja korkean tason kuvauksia kielen käsitteistä. (Cogo et al., 2023) Dokumentaation olemassaolo on tärkeää, sillä vaikka kielen syntaksi olisi helposti opittava ja helppolukuinen, esimerkiksi sen standardikirjasto voi olla niin laaja, että sen sisäistäminen kokonaisuudessaan on ohjelmoijalle haastavaa. Ohjelmointikielten dokumentaatiomuotoihin kuuluvat kirjat tai verkkosivut.

Esimerkkinä kirjallisesta dokumentaatiosta voidaan mainita Kernighanin ja Ritchien (1988) kirja ”The C Programming Language”. Verkkodokumentaatioista yksi esimerkki on Pythonin virallinen dokumentaatio. Internetissä sijaitsevien dokumentaatioiden etuna on niiden kyky pysyä jatkuvasti ajan tasalla kielten kehittyessä.

Dokumentaation laatuun vaikuttaa sen vastaavuus ohjelmoijien tiedonhakutarpeisiin. Kaikenkattavan dokumentaation luominen on haastavaa, sillä monet ohjelmoinnin alalajit, kuten verkko- tai peliohjelmointi tietyllä kielellä, vaativat oman ohjeistuksensa. (Cogo et al., 2023)

## 2.5 Käyttäjyhteisöjen tarjoama apu

Aktiivisen yhteisön on todettu edistävän ohjelmointikielen oppimista. Artikkelissaan Shaw (2012) pääättelee, että oppimista varten luodulla verkkofoorumilla on ollut positiivinen vaikutus oppimistuloksiin. Foorumin kautta on ollut mahdollista löytää nopeasti ratkaisuja ohjelmointiongelmiiin, jolloin ohjelmointikieli on vaikuttanut helpommalta ja lähestyttävämältä.

Verkkoyhteisöt ja foorumit ovat monelle ohjelmoijalle tärkeä osa työnkulkua. Etenkin kysymys-vastausyhteisöt (engl. *Q&A communities*), kuten Stack Overflow ovat laajassa käytössä ohjelmoijien keskuudessa (Mustafa et al., 2023). Ohjelmointikielten ympärillä olevat yhteisöt tarjoavat neuvoja ja tukea, millä voi olla positiivinen vaikutus myös ohjelmointikielen koettuun käytettävyyteen. Käyttäjyhteisöjen avun hyödyllisyys korostuu erityisesti silloin, kun ohjelmoija ei löydä tarvitsemaansa tietoa kielen dokumentaatiosta.

Ohjelmointikieli ja sen ympärillä oleva yhteisö linkittyvät toisiinsa siten, että yhteisö nähdään osana kielen käytön kokemusta, mikä vaikuttaa koettuun helppokäyttöisyyteen. Toisin sanoen, kun yhteisö koetaan avuliaksi, niin myös kyseessä olevan ohjelmointikielen saavutettavuus paranee.

## 3. VERTAILTAVAT OHJELMOINTIKIELET

Tässä luvussa esitellään vertailtavat ohjelmointikielet suppeasti. Tarkasteltavia asioita ovat kielten käyttökohteet, työkalut, muistimallit ja semanttiset ominaisuudet. Esittelyistä jätetään pois yksityiskohtia, sillä tavoitteena on antaa ainoastaan vertailun kannalta oleelliset esitiedot kielistä.

### 3.1 C++

Vertailluista kielistä C++ on vanhempi, ja siitä löytyy selkeästi enemmän kirjallisuutta kuin Rustista. C++ on alkujaan laajennus C-kielelle, ja se on kasvanut laajasti käytetyksi yleiskäyttöiseksi ohjelmointikieleksi, jonka käyttökohteita ovat muun muassa järjestelmä- ja sovellusohjelmointi (Dmitrović, 2023).

Tärkeänä ominaisuutenaan C++ mahdollistaa olio-ohjelmoinnin, joka toteutetaan luokilla. Kieli tarjoaa siis mahdollisuudet abstrahointiin, kapselointiin, luokkien periytymiseen ja polymorfismiin. (Stroustrup, 2013)

Kieli toimii kääntämällä käännösyksiköitä (engl. *translation unit*) eli lähdekooditiedostoja objektitiedostoiksi, jotka käännösprosessin jälkeen linkitetään suoritettavaksi ohjelmaksi (Stroustrup, 2013). Tämä mahdollistaa ohjelman jakamisen useaan moduuliin projektin rakenteen selkeyttämiseksi ja koodin uudelleenkäyttöä varten. Useita käännösyksiköitä sisältäviä projekteja varten on olemassa käännösautomaatiotyökaluja, kuten CMake.

Usein on toivottavaa tarjota moduulin käyttäjälle vain käytön kannalta olennainen tieto, mitä kutsutaan abstrahoinniksi (Rintala & Jokinen, 2005). Tätä varten voidaan luoda lähdekooditiedostoihin sisällytettäviä otsikkotiedostoja (engl. *header file*), joihin kirjoitetaan moduulin julkisen rajapinnan esittelyt ilman toteutusta. Näin käyttäjän näkökulmasta turhat asiat saadaan piilotettua varsinaiseen lähdekooditiedostoon.

C++:ssa muistin käyttö on jaettu osiin, joista ohjelmoijan näkökulmasta tärkeimpiä ovat kutsupino (engl. *call stack*) ja keko (engl. *heap*). Pinomuistia varataan automaattisesti luomalla pinokehys (engl. *stack frame*) funktioiden paikallisille muuttujille, parametreille ja paluusoitteille. Myös muistin vapautus on automaattista ja tapahtuu funktion suorituksen päättyessä. Varattu kekomuisti puolestaan ei liity suorituksessa olevan funktion

pinokehyykseen, joten se säilyy varattuna myös funktion suorituksen päätyttyä. (Solter & Kleper, 2005) Tästä syystä sen käyttö ei ole yhtä automaattista kuin pinomuistin käyttö.

Perinteisesti C++:ssa kekomuistia hyödynnetään manuaalisesti varaamalla ja vapauttamalla muistia ohjelmakoodissa. Tätä kutsutaan dynaamiseksi muistinhallinnaksi, sillä muisti varataan ajoaikana. Dynaamisesti varatun muistin osoite voidaan tallentaa osoittimeen, jonka kautta muistiosoitteessa sijaitsevaa arvoa voidaan käyttää. (Solter & Kleper, 2005) Muistin tehokas ajoaikainen hyödyntäminen C++:ssa vaatiikin osoittimien toiminnan ymmärtämistä.

C++:ssa sijoittaessa muuttuja toiseen muuttujaan tai välittäessä muuttuja parametrina funktiolle sen arvo kopioidaan uudeksi, ellei viittauksia tai siirtoja erikseen ohjelmakoodissa käytetä. (Stroustrup, 2013) Tästä päätellen C++ käyttää muuttujien käsittelyssä oletuksena arvosemantiikkaa.

Modernin C++:n voi katsoa saaneen alkunsa C++11-versiosta, joka toi kieleen tärkeitä uusia ominaisuuksia, kuten tietotyypin automaattisen päättelyn, säiekirjaston ja älykkäät osoittimet (Lucas et al., 2023). Sittemmin kieleen on tullut paljon lisäominaisuuksia helpottamaan ohjelmointia kolmen vuoden välein valmistuneiden versiopäivitysten myötä.

Esimerkiksi dynaaminen muistinhallinta on perinteisesti ollut manuaalista, jolloin muistinhallintavirheet ovat olleet yleisempiä, mutta modernin C++:n älykkäät osoittimet tarjoavat automaattisen elinkaaren päättämisen osoitetulle datalle. Lisäksi datan omistus on määritettävissä uniikiksi tai jaetuksi. Nämä osoittimet eivät kuitenkaan poista muita ongelmia, kuten kilpailutilanteita (engl. *race condition*), roikkuvia viittauksia (engl. *dangling reference*) tai jaettuun omistukseen liittyviä virheitä.

## 3.2 Rust

Rust on suhteellisen uusi, viime vuosikymmenellä kehitetty ohjelmointikieli, jonka suunnittelun tärkeitä tavoitteita ovat olleet samanaikaisesti sekä laitteistoläheisyys että turvallinen staattinen tyyppijärjestelmä ilman automaattista roskienkeräystä (engl. *garbage collection*). (Matsakis & Klock, 2014) Kielen käyttö on yleistynyt etenkin järjestelmäohjelmoinnissa, johon se soveltuu hyvin näistä syistä.

Rustilla kehitettävän ohjelman luominen ja hallinnoiminen tapahtuu Cargo-järjestelmän avulla. Projekti koostuu itse luoduista lähdekooditiedostoista ja valinnaisesti ulkoisista Cargo-paketeista, jotka liitetään projektiin tällä järjestelmällä. Cargo tarjoaa siis saman-

laisia ominaisuuksia kuin esimerkiksi Pythonin pip tai Node.js:n npm. Cargon muihin tehtäviin kuuluu lisäksi muun muassa projektin kääntäminen suoritettavaksi ohjelmaksi sekä testien ajaminen. (Klabnik et al., 2025)

Erillisiä otsikkotiedostoja ei Rustissa käytetä, eli jokainen lähdekooditiedosto itsessään on kokonainen käännösyksikkö. Lähdekoodin voi kuitenkin jaotella moduuleiksi kutsuttuihin loogisiin kokonaisuuksiin, joihin saadaan määriteltyä moduulin julkiset osat ja mahdollisia alimoduuleja. (Klabnik et al., 2025)

Kuten C++:ssa, Rustissa on käytössä pino- ja kekomuistien käsitteet. Myös Rustissa kekomuistia varataan ja vapautetaan dynaamisesti, ja älykkäille osoittimille löytyvät omat vastineensa. Ajoaikaista elinkaarten seuranta tai roskienkeruuta ei kuitenkaan tarvita. Kieli on suunniteltu siten, että muisti vapautuu aina automaattisesti. (Klabnik et al., 2025)

Automaattisuuden mahdollistaa Rustin omistajuuteen perustuva muistinhallintamalli, joka poikkeaa merkittävästi muista ohjelmointikielistä ja on siten Rustin tärkeimpiä ominaisuuksia. Malli pohjautuu datan omistukseen tietyillä säännöillä, joiden noudattamista kääntäjä valvoo. Säännöt ovat:

- Jokaisella arvolla on tasan yksi omistaja.
- Kun arvolla ei enää ole omistajaa, se vapautetaan muistista. (Klabnik et al., 2025)

Rustin tarjoamat ominaisuudet, kuten taattu muistiturvallisuus ja ”peloton” rinnakkaisuus (engl. *fearless concurrency*) perustuvat näihin sääntöihin. Jos yhtäkään sääntöä rikotaan ohjelmakoodissa, ohjelma ei käänny. (Klabnik et al., 2025)

Sääntöjen merkityksen kannalta on määriteltävä omistajan ja omistajuuden käsitteet. Omistaja tarkoittaa yksinkertaisesti muuttujaa, johon on tallennettu muistissa sijaitsevaa dataa, ja omistajuus tarkoittaa yksinoikeutta kyseiseen dataan. (Klabnik et al., 2025) Aikaisemmin mainitut säännöt voidaan siis ilmaista toisin: muistissa ei saa olla dataa, joka ei kuulu millekään muuttujalle, ja samaan dataan ei voi viitata useasta muuttujasta.

Jos dataa kuitenkin halutaan käyttää muualla, omistettu data tulee lainata väliaikaiseen käyttöön viitteen kautta, siirtää kokonaan toiselle muuttujalle tai sen tietotyyppi tulee olla määritelty kopioinnin toteuttavaksi. Esimerkiksi sijoittaessa merkkijonomuuttuja toiseen muuttujaan tapahtuu automaattinen omistuksen siirto, jolloin alkuperäisellä muuttujalla ei enää ole dataan käyttöoikeutta, mutta sijoittaessa kokonaislukumuuttuja toiseen tapahtuu kopiointi, sillä kyseinen tietotyyppi toteuttaa kopioinnin. (Klabnik et al., 2025) Rust käyttää siis oletuksena siirtosemantiikkaa, eli arvon kopiointi tai siihen viittaus täytyy lähtökohtaisesti tehdä manuaalisesti.

Rustin omistusjärjestelmällä ja C++:n älykkäillä osoittimilla on siis yhteisiä ominaisuuksia, joihin kuuluvat automaattinen muistinhallinta ja omistusten määrittäminen. Tärkein ero näiden välillä kuitenkin on se, että Rustissa muistiturvallisuutta valvotaan käännösaikana, kun taas C++:ssa olion omistavan ohjelman osan vastuut, kuten olion elinkaaren hallinta ovat ohjelmoijan tai älykkäiden osoittimien vastuulla. Kunhan ohjelma siis kääntyy, Rustilla kirjoitetussa ohjelmassa ei ole mahdollista esiintyä C++:lle tyypillisiä muistinhallintaan liittyviä ohjelmointivirheitä, kuten roikkuvia osoittimia tai muistin vapauttamista useaan kertaan, vaikka C++:ssa älykkäät osoittimet olisivatkin käytössä.

Jung et al. (2018) ovat tutkineet Rustin turvallisuuslupauksien paikkansapitävyyttä Rust-Belt-projektinsa myötä. Tutkimuksessa todettiin, että Rustin tyyppijärjestelmä ja standardikirjasto todellakin ovat turvallisia. Rust kykenee siis todistetusti estämään ohjelmointivirheitä käännösaikana.

## 4. KIELTEN MUISTINHALLINNAN HELPPOKÄYTTÖISYYDEN VERTAILUA

Tässä luvussa vertaillaan tarkasteltuihin ohjelmointikieliin liittyviä ominaisuuksia ja piirteitä sekä pohditaan niiden helppokäyttöisyyteen liittyviä asioita luvussa 2 tunnistettujen osa-alueiden mukaisesti. Aliluvuissa tarkastellaan kieliä ensin yleisesti kyseisen osa-alueen näkökulmasta, minkä jälkeen keskitytään tarkemmin muistinhallintaan.

### 4.1 Syntaksin selkeys

C++:n ja Rustin syntakseilla on paljon yhteistä, minkä takia Rust-koodi voi vaikuttaa C++-ohjelmoijalle ulkonäöllisesti tutulta. Joitain tyylieroja lukuun ottamatta molemmista kielistä löytyvät pitkälti samankaltaiset operaattorit, suorituksenohjausrakenteet ja tavat määritellä funktioita. Kielissä on kuitenkin piileviä eroja: C++-koodi on enemmän tyyliohjeiden ja vakiintuneiden konventioiden varassa, kun taas Rust-koodissa enemmän asioita on ilmaistavissa varsinaisen syntaksin avulla. Esimerkiksi omistussuhteet voidaan mallintaa C++:ssa älykkäiden osoittimien (eli kirjasto-olioiden) avulla, mutta omistajuus käsitteenä ei ole osa kieltä. Sitä vastoin Rustissa omistajuus on kääntäjän tarkasti valvoma kieliominaisuus.

C++ sallii lähdekoodissa implisiittisyyttä, eli epäsuorasti johdettua käyttäytymistä, enemmän kuin Rust. Esimerkiksi C++:ssa yksi kokonaislukutyyppi on mahdollista muuntaa implisiittisesti toiseksi, mutta Rustissa ei. Tämä on havainnollistettu ohjelmissa 2 ja 3.

```
1 int main() {
2     int i = 1;
3
4     // Int-arvon implisiittinen muunnos long-arvoksi:
5     long j = i;
6
7     // Sama muunnos eksplisiittisesti:
8     j = static_cast<long>(i);
9
10    return 0;
11 }
```

**Ohjelma 2.** Tyypimuunnokset C++:ssa

```

1 fn main() {
2     let i: i32 = 1;
3
4     // Väärin: implisiittinen tyyppimuunnosyritys aiheuttaa virheen.
5     // let j: i64 = i;
6
7     // Oikein: i32-arvo muunnetaan eksplisiittisesti i64-arvoksi.
8     let j: i64 = i as i64;
9 }

```

### **Ohjelma 3.** *Tyyppimuunnokset Rustissa*

Implisiittinen käyttäytyminen on oikein hyödynnettynä tehokas työkalu koodin yksinkertaistamiseen, mutta se piilottaa ohjelmassa todellisuudessa tapahtuvia asioita. Tämä voi heikentää syntaksin selkeyttä varsinkin, jos implisiittisesti tapahtuva operaatio ei ole ilmiselvää. Rustin syntaksia voi siis tältä osin pitää selkeämpänä, sillä ohjelmoijan aiomukset näkyvät lähdekoodissa tarkemmin. Implisiittisyys on sallittu suhteellisen yksikäsitteisissä tapauksissa, kuten tyyppin päättelyssä.

Rustin syntaksin eksplisiittisempi luonne auttaa esimerkiksi elinkaarien selkeyttämisessä. Havainnollistetaan tätä luomalla funktio, jonka tarkoituksena on valita kahdesta merkkijonosta pidempi.

Ohjelmassa 4 nähdään C++:lla kirjoitettu funktio, joka ottaa parametreikseen kaksi viitettä merkkijono-olioihin ja palauttaa samantyyppisen viitteen. Suoraan funktion määrittelystä ei ole nähtävissä, kumman viitteen elinkaarta palautetun viitteen on noudatettava, eikä kielestä löydy syntaksimekanismia tämän ilmaisemiseksi. Nyt roikkuvien viittausten välttäminen on ohjelmoijan vastuulla.

```

1 const std::string& longest(const std::string& x,
2                           const std::string& y) {
3     return (x.length() > y.length()) ? x : y;
4 }

```

### **Ohjelma 4.** *Kahdesta merkkijonosta pidemmän valitseva funktio C++:lla*

Rustissa puolestaan on mahdollista kertoa kääntäjälle elinikäparametreilla (engl. *lifetime parameters*), miten useiden lainausten eliniät rajoittavat toisiaan, mikä on nähtävissä ohjelmassa 5. Funktiolle on asetettu elinikäparametri 'a, jota jokaisen määrittelyssä esiintyvän viitteen on noudatettava. Kääntäjä valitsee 'a:n siten, että se ei ylitä kumpaakaan

x- tai y-lainauksen elinikä, jolloin myöskään palautettu viite ei ylitä niitä. Kääntäjä valvoo, että palautettua lainausta ei käytetä sen eliniän päättymisen jälkeen. Roikkuvaa viittausta ei siis synny.

```

1 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2     if x.len() > y.len() { x } else { y }
3 }
```

### **Ohjelma 5.** *Rust-vastine ohjelmalle 4 (Klabnik et al., 2025)*

Esimerkin tapauksessa elinikäparametrin käyttö on pakollista, sillä kääntäjä ei kykene pääättelemään palautetun viitteen elinikä implisiittisesti. Tämä on esimerkki siitä, miten Rust pakottaa selkeyteen eksplisiittisemmällä syntaksilla kuin C++.

## **4.2 Kääntäjän tai tulkin ilmoitukset**

Ennen vertailua on valittava kummallekin kielelle tarkasteltavat kääntäjät. C++:lle on olemassa useita yleisessä käytössä olevia kääntäjiä, kuten g++ ja MSVC. Näistä ensimmäinen on GNU-projektin GCC-kääntäjäkokoelmaan kuuluva avoimen lähdekoodin ohjelma, kun taas jälkimmäinen on Microsoftin yksityisesti kehittämä Windows-käyttöjärjestelmää varten. Vertailua varten tarkasteltavaksi valitaan g++, sillä se on kehitystyökaluja tarjoavan JetBrainsin (2023) tuottaman kyselyn mukaan käytetyin C++-kääntäjä. Rustc-kääntäjän kehitys on osa itse kielen kehitystä, eikä Rustille ole toistaiseksi olemassa muita tuotantovalmiita kääntäjiä, joten vertailtavaksi valitaan rustc.

G++ kykenee antamaan varsinaisten virheilmoitusten lisäksi usean tyyppisiä varoituksia. Varoitukset liittyvät esimerkiksi tilanteisiin, jotka altistavat ohjelman määrittelemättömälle käyttäytymiselle (engl. *undefined behavior*). Tällaisissa tilanteissa C++-spesifikaatio ei määrittele, miten ohjelman voi odottaa käyttäytyvän, jolloin ohjelmassa voi sallitusti tapahtua myös epätoivottuja asioita (Stroustrup, 2013). Määrittelemätön käyttäytyminen ei siis varsinaisesti ole virhe, mutta kääntäjä voi joissain tapauksissa varoittaa siitä.

Rust puolestaan on suunniteltu siten, että määrittelemätön käyttäytyminen on mahdollista ainoastaan *unsafe*-operaatioilla, joiden muistiturvallisuutta kääntäjä ei valvo yhtä tarkasti kuin tavanomaisia operaatioita (Klabnik et al., 2025). Tämän seurauksena rustc tuottaa virheilmoituksia myös sellaisissa tilanteissa, joissa g++ antaisi vain varoituksen. On kuitenkin huomattava, että projektiin liitetyt ulkoiset moduulit voivat sisältää piileviä,

mahdollisesti väärin käytettyjä *unsafe*-operaatioita, jolloin kääntäjä ei välttämättä havaitse määrittelemätöntä käyttäytymistä ilman erillisiä työkaluja.

Tutkitaan kääntäjien antamia varoituksia ja virheilmoituksia luvussa 2.2 esiteltyjen tekijöiden mukaisesti. Luodaan tätä varten keskenään vertaiset virhetilanteet jatkamalla luvun 4.1 esimerkkejä. Ohjelmassa 6 tulos sijoitetaan ensin paikalliseen apumuuttujaan, minkä jälkeen palautetaan viittaus siihen. Palautettu viite jää roikkuvaksi, sillä viitatus muuttujan elinikä on rajoitettu funktioon, jossa se on luotu.

```

1  #include <iostream>
2  #include <string>
3
4  const std::string& longest(const std::string& x,
5                          const std::string& y) {
6      std::string tmp = (x.size() >= y.size()) ? x : y;
7      return tmp; // Muuttujan tmp elinikä päättyy
8  }
9
10 int main() {
11     std::string a = "yksi";
12     std::string b = "kaksi";
13
14     // Virhe: merkkijonon muodostus roikkuvasta viittauksesta
15     std::string result = longest(a, b);
16
17     std::cout << result << std::endl;
18
19     return 0;
20 }
```

**Ohjelma 6.**     *Roikkuva viittaus C++-ohjelmassa*

Vaikka ohjelma 6 kääntyy normaalisti, sen käyttäytyminen on määrittelemätöntä. Seurauksena voi esimerkiksi olla ohjelman kaatuminen. Vastaava ongelma Rust-koodissa on esitetty ohjelmassa 7.

```

1 fn longest<'a>(x: &'a String, y: &'a String) -> &'a String {
2     let tmp = if x.len() >= y.len() {x.clone()} else {y.clone()};
3     &tmp // Virhe: muuttujan tmp elinikä päättyy
4 }
5
6 fn main() {
7     let a = String::from("yksi");
8     let b = String::from("kaksi");
9
10    let result = longest(&a, &b);
11    println!("{result}");
12 }

```

### **Ohjelma 7.**      *Roikkuva viittaus Rust-ohjelmassa*

Nyt ohjelma ei käänny, sillä funktiossa yritetään palauttaa lainaus paikalliseen muuttajaan. Roikkuvaa viittausta ei siis pääse syntymään.

G++:n antama varoitus näkyy kuvassa 1 ja rustc:n antama virheilmoitus kuvassa 2. Molemmista ilmoituksista löytyy luvussa 2.2 listattuja asioita. Virheet ovat ymmärrettävissä, kunhan tuntee niissä esiintyvät käsitteet. Lukijan työmuistia ei kuormiteta tarpeettomasti liian monimutkaisilla lauseilla, tosin rustc:n ilmoituksessa esiintyy toistoa, jonka Becker et al. (2019) ovat maininneet kuormittavana tekijänä. Molemmat kääntäjät tarjoavat tarpeellisen tiedon virheen ymmärtämiseksi, mutta rustc antaa vielä lisäksi pyydettyä esimerkkejä kyseisestä virheestä, ratkaisuehdotteita sekä yksityiskohtaisempaa tietoa kuin g++. Rustc on siis havaintojen perusteella joiltain osin helppokäyttöisempi kuin g++.

```

virhe1.cc: In function 'const std::string& longest(const std::string&, const std::string&)':
virhe1.cc:7:12: warning: reference to local variable 'tmp' returned [-Wreturn-local-addr]
   7 |     return tmp;
     |           ^~~
virhe1.cc:6:10: note: declared here
   6 |     auto tmp = (x.size() >= y.size()) ? x : y;
     |           ^~~

```

### **Kuva 1.** *Ohjelman 6 aiheuttama varoitus*

```

    Compiling virhe v0.1.0 (/home/jarkkoa/src/kandi/rs/virhe)
error[E0515]: cannot return reference to local variable `tmp`
  → src/main.rs:3:5
   |
3 |     &tmp // Virhe: muuttujan tmp elinikä päättyy
   |     ^^^^ returns a reference to data owned by the current function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `virhe` (bin "virhe")_due to 1 previous error

```

**Kuva 2.** Ohjelman 7 aiheuttama käännösvirhe

### 4.3 Oppimiskäyrät

Muistinhallinta ja osoittimien käyttö ovat todettu haastaviksi asioiksi C++:n opettelussa, sillä kokematon ohjelmoija ei välttämättä ymmärrä määrittelemätöntä käyttäytymistä ja sen seurauksia tai osaa tulkita muistinhallintaan liittyviä virheilmoituksia (Allevalo & Edwards, 2014). Vaikka ohjelmoija hyödyntäisikin älykkäitä osoittimia ja ohjelma kääntyisi, jälkikäteen ilmenevät virheet ovat silti mahdollisia. Esimerkiksi kielessä on mahdollista tallentaa älykkään osoittimen omistama muistiosoite uuteen raakaosoittimeen, jolloin älykkään osoittimen tuhoamisen jälkeen käytettäväksi jää roikkuva osoitin (engl. *dangling pointer*). Opittavana on siis myös älykkäiden osoittimien väärinkäytön ja huolimattomuusvirheiden välttäminen.

Kuten aiemmin on todettu, Rustin omistamiseen ja lainaamiseen perustuva muistinhallintamalli ratkaisee C++:ssa esiintyviä muistinhallintavirheitä. Malli kuitenkin vaatii opettelua ja erilaista ajattelua verrattuna C++-kieleen, mikä voi tuoda esteitä ohjelmakoodin kirjoittamiseen. Ohjelmoijan aikaisempi tietämys C++-tyylisestä dynaamisesta muistinhallinnasta voi siis aiheuttaa vaikeuksia Rust-tyylisen muistinhallinnan opettelussa.

Zhu et al. (2022) tutkivat, mitkä Rustin turvallisuussäännöt koetaan hankaliksi ja missä tilanteissa niiden soveltaminen on vaikeaa. Tutkimuksessa havaittiin, että virheitä tehdään muuttujien eliniän arvioimisessa ja omistajuussääntöjen noudattamisessa. Tutkimuksessa huomattiin myös, että teoriassa yksinkertainen turvallisuusmekanismi voi olla vaikeasti sovellettavissa oikeissa tilanteissa. Erityisen haastavaa muistinhallinnasta tekeekin ehdottomuus kääntäjän valvomisessa turvallisuussäännöissä.

Kummallakin kielellä on siis vaativa oppimiskäyrä, ja molempien kohdalla muistinhallintaa voi pitää erityisen haastavana osa-alueelta. Muistinhallinnan helppokäyttöisyys riippuu kuitenkin tarkastelukulmasta. C++:lla on vaikeampaa oppia kirjoittamaan muistiturvallisia ohjelmia, mutta helpompaa saada ohjelma kääntymään. Vastaavasti Rustilla on

vaikeampaa oppia kirjoittamaan ohjelmia, jotka saadaan käännettyä, mutta lopputuloksena on muistiturvallisempi ohjelma.

#### 4.4 Saatavilla oleva dokumentaatio

C++:n viralliseen dokumentaatioon kuuluvat ainoastaan valmiit kielen spesifikaatiot ja niiden luonnokset. Monimutkaisen standardin lukeminen ohjelmoinnin tukena on kuitenkin tarpeettoman työlästä, joten kääntäjien kehittäjien vastuulla on tarjota toteutuskohdainen dokumentaatio, ja standardia yksinkertaisempia oppaita on luotu yhteisön toimesta.

Rust-projekti on laajempi kuin pelkkä kielispesifikaatio, ja siihen sisältyy kattavampi virallinen dokumentaatio. Tähän kuuluu muun muassa opaskirjoja ja standardikirjaston kuvaus.

Dokumentaation luominen omasta projektista on tehty Rustilla vaivattomaksi ilman erillisiä työkaluja; dokumentointi on suunniteltu osaksi kieltä, jolloin Cargo-järjestelmällä voi yhdellä komennolla tuottaa verkkoselaimella luettavan dokumentaation sekä itse kirjoitetusta koodista että projektin riippuvuuksista. C++:lla vastaavanlaista dokumentaatiota voi tuottaa esimerkiksi doxygen-työkalulla, mutta koska dokumentaatiotyö ei ole osa kieltä eikä standardoitu, ei ole takuuta siitä, että jokainen projektiin liitetty ulkoinen moduuli toimisi käytetyn dokumentaatiotyökalun kanssa.

C++:aa käsittelevää kirjallisuutta on paljon, josta osa keskittyy erityisesti muistinhallintaan. Lisäksi *cppreference.com*-verkkosivuilta löytyy muistinhallintakirjaston kuvaus, josta pääsee lukemaan käyttöohjeita ja yksityiskohtaisia selityksiä kirjastotyypeille ja -funktioille.

Molempien kielten käytöstä vaikuttaa löytyvän paljon tietoa joko virallisista tai epävirallisista lähteistä. Rustin dokumentaatio on koottu yhteen paikkaan ilmaiseksi saataville, jolloin tieto on saavutettavampaa ja sen etsiminen on helpompaa. C++ on kuitenkin huomattavasti vanhempi kieli kuin Rust, joten siitä löytyy paljon enemmän kirjallisuutta myös erityisiin käyttökohteisiin, kuten peliohjelmointiin. Kummallekin kielelle löytyy dokumentaatio muistinhallintaa varten, joten sen osalta kumpaakaan kieltä ei voi pitää toistaan helppokäyttöisempänä.

## 4.5 Käyttäjyhteisöjen tarjoama apu

Molemmille kielille on olemassa aktiiviset käyttäjyhteisöt. C++-yhteisön jäsenet ovat luoneet muun muassa aiemmin mainitun *cppreference.com*-sivuston, ja Rustin yhteisö on auttanut virallisen dokumentaation kirjoittamisessa. Rustin verkkosivut tarjoavat myös kielen käyttäjille verkkofoorumia, jossa voi keskustella kielestä ja sillä luoduista projekteista sekä kysyä neuvoa muilta käyttäjiltä.

Kysymys-vastausyhteisö Stack Overflow'n (2025) sekä JetBrainsin (2025) tuottamien kehittäjäkyselyjen mukaan C++ on tällä hetkellä merkittävästi käytetympi kuin Rust. Mitä enemmän kielellä on käyttäjiä, sitä helpompaa on teoriassa löytää apua, sillä osaajia on enemmän. Vaikka Rustin käyttäjäkunta on kyselyjen perusteella pienempi, se ei kuitenkaan ole merkityksetön. Käytännössä siis molemmille kielille on olemassa osaajia, jolloin käyttäjyhteisön apua on suhteellisen helposti saatavilla.

C++:n vanhuuden vuoksi internetistä löytyy monia menneinä vuosina siitä kysytyjä kysymyksiä ja niiden vastauksia. On siis mahdollista, että kun ohjelmoija kaipaa yhteisöltä apua, joku toinen on jo kysynyt saman kysymyksen, jolloin vastauksen saa vaivattomammin. Tässä on kuitenkin ongelma: ohjelmoijan löytämä tieto saattaa olla vanhentunutta, sillä ohjelmointikäytännöt muuttuvat ja kieliominaisuuksia tulee lisää. Jos esimerkiksi etsitään ratkaisua johonkin muistinhallintaongelmaan, löydetty vastaus voi suositella vanhentuneita tapoja varata ja vapauttaa muistia manuaalisesti, sillä älykkäitä osoittimia ei ole kirjoitusaikaan ollut olemassa.

Jos ohjelmoija on kokenut ja osaa arvioida löytämäänsä tietoa kriittisesti, C++:aa voi käyttäjyhteisön tarjoaman avun kannalta pitää helppokäyttöisempänä jo olemassa olevan tiedon ansiosta. Aloitteleva ohjelmoija ei kuitenkaan välttämättä tunnista vanhentuneita käytäntöjä, jolloin uudempi Rust voi olla helppokäyttöisempi. Tämän perusteella ei ole tehtävissä johtopäätöstä siitä, onko kumpikaan kieli objektiivisesti toistaan helppokäyttöisempi.

## 5. YHTEENVETO

Tässä kandidaatintyössä vertailtiin C++- ja Rust-ohjelmointikielten muistinhallinnan helppokäyttöisyyttä. Ohjelmointikielen helppokäyttöisyyteen vaikuttaviksi asioiksi tunnistettiin syntaksin selkeys, kääntäjän tai tulkin ilmoitukset, kielten oppimiskäyrät, saatavilla oleva dokumentaatio ja käyttäjäyhteisöjen tarjoama apu.

Työssä pääteltiin, että Rust pakottaa selkeämmän koodin kirjoittamiseen syntaksinsa avulla, ja että Rustin rustc-kääntäjä on hyödyllisempi muistinhallintavirheiden ratkaisemisessa kuin C++:n g++-kääntäjä. Molemmilla kielillä todettiin olevan haastavat oppimiskäyrät, mutta Rustilla on helpompaa kirjoittaa muistiturvallisia ohjelmia, sillä kielen haasteet keskittyvät kääntäjän valvomiin turvallisuussääntöihin, kun taas C++-ohjelmissa esiintyy enemmän ajoaikaisia muistinhallintaongelmia. Kielten helppokäyttöisyydellä ei todettu olevan merkittävää eroa saatavilla olevan dokumentaation ja kielten käyttäjäyhteisöjen tarjoaman avun suhteen.

Rustia voi pitää lopulta helppokäyttöisempänä esimerkiksi silloin, kun tavoitteena on muistiturvallisuus toiminnan kannalta kriittisessä sovelluksessa. C++ puolestaan on helppokäyttöisempi yksinkertaisempaa muistinhallintaa sisältävien ohjelmien kirjoittamisessa, sillä g++ ei vaadi tarkkojen muistiturvallisuuksääntöjen noudattamista.

Työssä ei muistinhallinnan lisäksi pohdittu muiden kieliominaisuuksien helppokäyttöisyyksiä tarkemmin. Muita tarkasteltavia asioita voisivat olla olio-ohjelmointi tai rinnakkaisuusominaisuudet, sillä kielet eroavat myös näiltä osin merkittävästi; Rustissa ei esimerkiksi käytetä luokkia, ja kääntäjän turvallisuustarkastukset estävät muun muassa jaetun datan kilpailutilanteet. Tarkastelusta on jätetty pitkälti pois myös Rustin *unsafe*-ominaisuudet.

Yksi mahdollinen Rustin käyttökohde on myös mikrokontrollereiden ohjelmointi, jossa C on yksi käytetyistä kielistä. Nopeus ja muistiturvallisuus ovat tärkeitä tavoitteita mikrokontrollereissa, joten vertailu myös Rustin ja C:n välillä tämän käyttökohteen näkökulmasta olisi perusteltua.

# LÄHTEET

Allevato, A. & Edwards, S. 2014. Dereferree: instrumenting C++ pointers with meaningful runtime diagnostics. *Software: Practice and Experience*, 44, 8, s. 973–997. <https://doi.org/10.1002/spe.2184>

Becker, B., Denny, P., Pettit, R., Bouchard, D., Bouvier, D., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P., Pearce, J. & Prather, J. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. *ITiCSE '19: Innovation and Technology in Computer Science Education*, Aberdeen, s. 177–210. <https://doi.org/10.1145/3344429.3372508>

Cogo, F., Xia, X. & Hassan, A. 2023. Assessing the Alignment between the Information Needs of Developers and the Documentation of Programming Languages: A Case Study on Rust. *ACM Transactions on Software Engineering and Methodology*, 32, 2, s. 1–48. <https://www.doi.org/10.1145/3546945>

Cooper, K. & Troczon, L. 2012. *Engineering a Compiler*. 2. painos. Elsevier. Amsterdam.

JetBrains. 2023. C++ Programming - The State of Developer Ecosystem in 2023 Infographic. Luettavissa: <https://www.jetbrains.com/lp/devecosystem-2023/cpp/>. Luettu: 2.12.2025.

JetBrains. 2025. Tools and Trends - The State of Developer Ecosystem in 2025. Luettavissa: <https://devecosystem-2025.jetbrains.com/tools-and-trends>. Luettu: 4.12.2025.

Jung, R., Jourdan, J., Krebbers, R. & Dreyer, D. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages*, 2, POPL, s. 1–34. <https://doi.org/10.1145/3158154>

Klabnik, S., Nichols, C. & Krycho, C. *The Rust Programming Language*. Luettavissa: <https://doc.rust-lang.org/stable/book/>. Luettu: 2.12.2025.

Lokkila, E., Christopoulos, A. & Laakso, M. 2023. A Data-Driven Approach to Compare the Syntactic Difficulty of Programming Languages. *Journal of Information Systems Education*, 34, 1, s. 84–93.

Lucas, W., Carvalho, F., Nunes, R., Bonifácio, R., Saraiva, J. & Accioly, P. 2023. Embracing Modern C++ Features: An Empirical Assessment on the KDE Community. *Journal of Software: Evolution and Process*, 36, 5. <https://doi.org/10.1002/smr.2605>

Matsakis, N. & Klock, F. 2014. The Rust Language. *HILT '14: High Integrity Language Technology ACM SIGAda Annual Conference*, Portland, s. 103–104. <https://doi.org/10.1145/2663171.2663188>

Mustafa, S., Zhang, S. & Naveed, M. 2023. What Motivates Online Community Contributors to Contribute Consistently? A Case Study on Stackoverflow Netizens. *Current Psychology*, 42, 13, s. 10468–10481. <https://doi.org/10.1007/s12144-022-03307-4>

- Perera, P., Tennakoon, G., Ahangama, S., Panditharathna, R. & Chathuranga, B. 2021. A Systematic Mapping of Introductory Programming Languages for Novice Learners. IEEE Access, 9, s. 88121–88136. <https://doi.org/10.1109/ACCESS.2021.3089560>
- Rintala, M. & Jokinen, J. 2005. Olioiden ohjelmointi C++:lla. 4. painos. Talentum Media Oy. Helsinki.
- Shaw, R. 2013. The Relationships among Group Size, Participation, and Performance of Programming Language Learning Supported with Online Forums. Computers & Education, 62, s. 196–207. <https://doi.org/10.1016/j.compedu.2012.11.001>
- Solter, N. & Kleper, S. 2005. Professional C++. Wrox Press. Birmingham.
- Stack Overflow. 2025. 2025 Stack Overflow Developer Survey. Luettavissa: <https://survey.stackoverflow.co/2025/>. Luettu: 4.12.2025.
- Stroustrup, B. 2013. The C++ Programming Language. 4. painos. Addison-Wesley. Boston.
- Tahy, Z & Czirkos, Z. 2016. “Why Can’t I Learn Programming?” The Learning and Teaching Environment of Programming. Informatics in Schools: Improvement of Informatics Knowledge and Perception (ISSEP 2016), Münster, s. 199–204. [https://doi.org/10.1007/978-3-319-46747-4\\_17](https://doi.org/10.1007/978-3-319-46747-4_17)
- Temkin, D. 2017. Language Without Code: Intentionally Unusable, Uncomputable, or Conceptual Programming Languages. Journal of Science and Technology of the Arts, 9, 3, s. 83. <https://doi.org/10.7559/citarj.v9i3.432>
- Temkin, D. 2023. The Less Humble Programmer. Digital humanities quarterly, 17, 2.
- Zhu, S., Zhang, Z., Qin, B., Xiong, A. & Song, L. 2022. Learning and Programming Challenges of Rust: A Mixed-Methods Study. 44th International Conference on Software Engineering (ICSE), Pittsburgh, s. 1269–1281. <https://doi.org/10.1145/3510003.3510164>