Muistinhallinta Java- ja C++-ohjelmoinnissa

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
TkK-tutkielma
1.5.2019
Elmeri Hyvönen

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos / Luonnontieteiden ja tekniikan tiedekunta

HYVÖNEN, ELMERI: Muistinhallinta Java- ja C++-ohjelmoinnissa

TkK-tutkielma, 27 s.
Tietotekniikka
Helmikuu 2019

Muisti on tärkeä resurssi ohjelmistokehityksessä, ja sitä voidaan tilannekohtaisesti hallita erilaisilla tekniikoilla. Muistinhallinnalla pyritään hyödyntämään muistiresursseja tehokkaasti ja välttämään mahdolliset muistinkäsittelyyn liittyvät virhetilanteet. Ohjelman suorituksen kannalta tarpeettomiksi muodostuneista muistialueista luopuminen on järkevää. Vastuu muistin vapauttamisesta voidaan siirtää automaattiselle mekanismille tai hoitaa itse manuaalisesti.

Java-ohjelmointikieli hyödyntää roskienkerääjää, joka hoitaa muistin vapauttamisen kehittäjän puolesta. Javan roskienkerääjään on kehitetty erilaisia mekanismeja seuraamaan muistialueiden tarpeellisuutta ja siten tunnistamaan, milloin muistialue ei ole enää ohjelmalle tarpeellinen, ja se voidaan vapauttaa. Javan roskienkerääjästä on kehitetty useita versioita, joiden toiminta perustuu erilaisiin algoritmeihin.

C++-ohjelmointikieli mahdollistaa manuaalisen muistinhallinnan new- ja delete-käskyjen avulla. C++-kielen kehitys muistinhallinnan osalta on ollut vähäisempää, vaikka kieleen onkin kehitetty tekniikoita, joilla muistinkäytön vastuuta saadaan siirrettyä enemmän automaattiselle mekanismille. Kieli toimii kuitenkin tässä tutkielmassa hyvänä vertailukohtana Javan automaattiselle muistinhallinnalle.

Tässä tutkielmassa tehdään lyhyt katsaus muistinhallinan merkitykseen ohjelmoinnissa sekä kuvataan keskeisimmät musitinhallintaan liittyvät tekniikat ja muistinkäsittelyn virhetilanteet. Tutkielmassa tarkastellaan Java- ja C++-ohjelmointikielien muistinhallintamenetelmiä, ja lopussa kieliä vertaillaan keskenään. Tarkoituksena on luoda lukijalle ymmärrys molempien kielten keskeisistä muistihallintatekniikoista.

Tarkastelun kohteena olleissa ohjelmointikielissä käytettävät muistinhallintamenetelmät ovat varsin eroavaiset. Javan suurena etuna voidaan pitää turvallisuutta, kun taas C++-kielen suunnitellussa on selkeästi panostettu tehokkuuteen.

Asiasanat: Java, C++, ohjelmointi, muistinhallinta, roskienkerääjä

Sisällys

1 JOHDANTO

Tietokoneen keskusmuisti on tietokoneella suoritettavien ohjelmien työmuisti [1]. Ohjelmat käyttävät muistia suorituksessaan tarvittavien tietojen säilömiseen. Muisti onkin tärkeä resurssi ohjelmistokehityksessä, ja sen hallinta voidaan toteuttaa tilannekohtaisesti monin eri tavoin. Roskienkeruu (engl. *garbage collection*) on muistinhallintamekanismi, joka pyrkii luopumaan muistialueista, joille ei ohjelman suorituksessa ole enää tarvetta [2]. Tarpeettomista muistialueista luopuminen vähentää muistinkäytöstä aiheutuvia virhetilanteita.

Muistinhallinnan ongelmakohdiksi saattaa muodostua erilaiset muistinkäytön ongelmatilanteet, kuten muistivuodot, muistin pirstoutuminen tai muistiviittauksiin liittyvät ongelmat. Ongelmatilanteet vaikuttavat ohjelmiston suoritusaikaan, turvallisuuteen ja luotettavuuteen. [3] Ongelmatilanteet saattavat myös aiheuttaa ohjelman keskeytymisen käyttöjärjestelmän toimesta, millä estetään ongelman eskaloituminen lisää.

Tyypillisesti roskienkeruu jaotellaan manuaaliseen ja automaattiseen. Automaattinen roskienkerääjä on manuaalisen muistinhallinnan rinnalle kehitetty tehokas ja helppo vaihtoehto hoitaa ohjelman muistinkäsittely. Muistinkäyttöön liittyvien ongelmien todennäköisyys pienenee muistinvapauttamisen vastuun siirtyessä kehittäjältä automaattiselle mekanismille. Tällöin kuitenkin muuttujien elinaikaan ei saada samanlaista kontrollia, mikä joissain tilanteissa voi olla tarpeellista. Tutkielmassa tarkasteltu Java-ohjelmointikieli tarjoaa varsin pitkälle kehittyneet automaattiset mekanismit ohjelman tarvitseman muistin hallintaan. Automaattinen roskienkeruu toimii hyödyntämällä erilaisia algoritmeja, joista tässä työssä on esitelty tunnetuimmat. [2]

C++-ohjelmointikielen kehitys muistinhallinnan osalta on ollut Javan kehitystä hitaampaa, ja muistiresurssien manuaalista hallintaa toteutetaankin kielessä edelleen paljon. Kieleen on kuitenkin kehitetty tekniikoita, joilla muistinkäytön vastuuta saadaan siirrettyä enemmän automaattiselle mekanismille.

2 MUISTINHALLINTA

Muistinhallinnalla tarkoitetaan menetelmiä muistin tehokkaan käytön ja monitoroinnin edistämiseksi. Muistiresurssien tehokas hyödyntäminen on muistinhallinnan keskeinen tavoite. Järjestelmän muistinkäyttöä tehostettaessa on keskeistä suunnitella ohjelman sijoittelu muistiin ja miettiä toteutettavan järjestelmän kannalta sopiva ohjelmointikieli tai toteutustyyppi kehitystyölle.

Erilaiset suorituskykyyn vaikuttavat tekijät tulee huomioida ohjelman kehitysvaiheessa, mikäli ohjelman halutaan toimivan virheettömästi. Ohjelmiston suorituskyky voidaan ajatella ohjelmiston ajallisen käyttäytymisen täsmällisyytenä suhteessa sille asetettuihin tavoitteisiin. Vastausaika (engl. *latency*) ja suoritusteho (engl. *troughput*) ovat yleisiä mittareita suorituskyvyn arvioimiseen. Vastausajalla tarkoitetaan esimerkiksi interaktiivisissa käyttöliittymissä aikaa, jossa järjestelmä tuottaa vastauksen käyttäjän tekemään pyyntöön. Suoritusteholla tarkoitetaan järjestelmän suorittamien pyyntöjen määrää tietyssä ajanjaksossa. [4]

Ohjelmoitaessa on suotavaa myös pyrkiä välttämään erilaiset muistinkäyttöön liittyvät virhetilanteet. Erilaiset muistinkäyttöön liittyvät virhetilanteet saattavat saada ohjelman käyttäytymään epätoivotulla tavalla, mutta käyttöjärjestelmän suojamekanismi voi myös käsittelemättömän muistivirheen takia pakottaa prosessin päättymään. Muistia voidaan hallita myös manuaalisesti, jolloin kehittäjä varaa ja vapauttaa muistia erillisillä käskyillä. Muistin varaaminen ja vapauttaminen manuaalisesti lisää virhetilanteiden todennäköisyyttä mahdollisten huolimattomuusvirheiden vuoksi.

2.1 Muistialueet

Ohjelmilla on tyypillisesti käytettävissään kaksi muistialuetta: pino ja keko. Pinoa (engl. *stack*) voidaan käyttää funktiokutsujen parametrien, funktioiden paluuosoitteiden ja paikallisten muuttujien säilömistä varten. Keosta (engl. *heap*) voidaan varata suoritusaikana muistia ilman, että varaus on sidoksissa kontrollivuohon ja aliohjelmien kutsupinoon. Eksplisiittinen muistinhallinta on pitkälti kekomuistin hallintaa. [5]

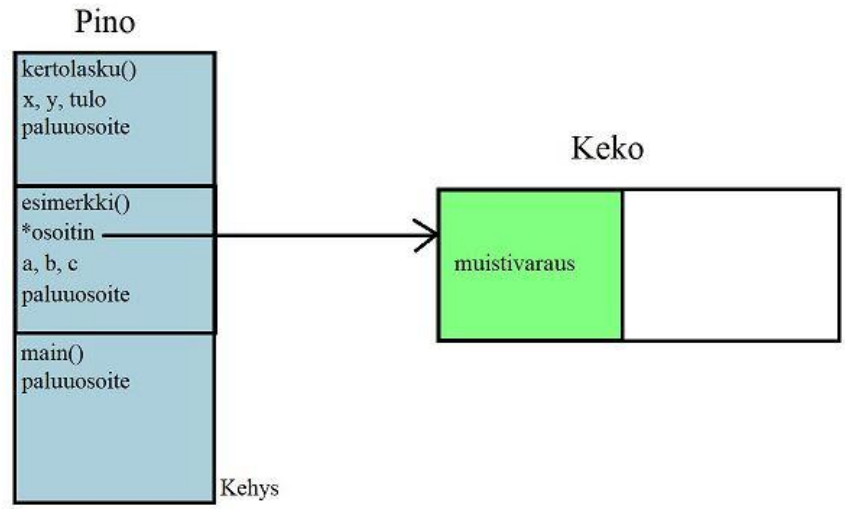
Pino muodostuu kehyksistä (engl. *frame*), jotka kasautuvat päällekkäin muodostaen pinomaisen rakenteen. Esimerkiksi jokaiselle funktiokutsulle voidaan muodostaa pinoon

uusi kehys. Paikalliset muuttujat, jotka on määritelty funktion sisällä, voidaan myös lisätä pinoon. Pinoa ohjaa pino-osoitin (engl. *stack pointer*), joka osoittaa pinossa päällimmäisenä olevaan tietoon. Pino toimii automaattisesti, eikä kehittäjän tarvitse ohjata sitä ohjelman sisältä. Funktioon mentäessä pino-osoitinta siirretään pinovarauksen yhteismäärän verran ja funktiosta poistuttaessa vapautetaan kutsuun liittyvät varaukset. Pino-osoitin siirretään funktiosta poistuttaessa seuraavana pinossa olevaan tietoon. Pinorakenteessa päällimmäisenä oleva kehys on siis parhaillaan ohjelman suoritettavana. [6] Päällimmäisen kehyksen päälle voi osassa ohjelmointikielistä varata dataa, mutta kääntäjä osaa pitää kirjaa nykyisestä kehyksestä. Kääntäjä voi myös allokoida pinomuistiin satunnaislukuja (engl. *canary*), joilla voidaan havaita esimerkiksi mahdolliset pinon ylivuodot [7]. Satunnaisluku voidaan hakea pinosta ennen funktion paluuosoitetta ja satunnaislukua vertailtaessa pinoon vietyyn lukuun havaitaan ylivuototilanne, mikäli luvut eroavat toisistaan. Tässä työssä on pinon osalta keskitytty ainoastaan yhden säikeen (engl. *thread*) ohjelmiin. Pino haarautuu tilanteissa, joissa säie jakautuu uusiksi säikeiksi. Tällaista pinoa kutsutaan kaktuspinoksi [8].

Muistinhallinnassa kekorakenne on muistialue, joka sisäisesti rakentuu ohjelman kannalta tärkeistä yksittäisistä muistialueista. Kekomuisti on tarkoitettu dynaamisesti varattujen muuttujien säilömiseen [9]. Keon sisäiset muistialueet kytkeytyvät toisiinsa ajonaikaisen järjestelmän sisäisillä muistiviittauksilla, jotka muodostavat kekorakenteen. Kekoon kohdistuvien muistivarauksien koko voi vaihdella. Kekomuistin käyttöön liittyy ongelmatilanteita, joita käsitellään myöhemmin tässä tutkielmassa.

```cpp
#include <iostream>
using namespace std;

int kertolasku(int x, int y){
    int tulo = x * y;
    return tulo;
}
void esimerkki(){
    int *osoitin = new int;
    int a = 3;
    int b = 2;
    int c = kertolasku(a, b);
    delete osoitin;
}
int main(){
    esimerkki();
    return 0;
}
```



Kuva 1: Pino ja keko.

Kuva 1 havainnollistaa kekoa ja pinoa yksinkertaisen esimerkkiohjelman kanssa. Ohjelma on toteutettu C++-ohjelmointikielellä.

2.1 Sovellustason muistinhallinta

Kehitettävien ohjelmien tulee varata muistia säilöäkseen ja käsitelläkseen ohjelman tarvitsema data tehokkaasti suoritusaikana. Ohjelma itsessään tulee myös säilöä muistiin ajon ajaksi. Ohjelmat varaavat muistia suorituksen aikana, jotta niiden toiminnallisuus saadaan toteutettua, ja muistiresurssien tehokkaan hyödyntämisen kannalta ohjelman tulisi myös vapauttaa muistialueet, joille ei enää ole tarvetta. [10] Ohjelmistokehityksessä puhutaankin erilaisten muuttujien ja tietorakenteiden elinkaaresta, jolla viitataan siihen aikaan, mikä muuttujan luomisesta ja tilanvarauksesta kuluu muuttujan tuhoamiseen ja

muistin vapauttamiseen. Viittauksen osalta elinajalla voidaan tarkoittaa aikaa, jolloin objektiin on viittaus saatavilla. Muistinhallinnan kannalta elinaika viittaa muistivarauksen elinaikaan. Jos muistinkäyttöä halutaan tarkastella muistiresurssien tehokkaan käytön osalta, on suotavaa välttää tarpeettomia muistivarauksia. Muistin täyttyminen voi hidastaa uusien varausten tekemistä kekorakenteen muistinhallinnassa. Turha muistin varattuna pitäminen siis saattaa tehdä ohjelman suorittamisesta raskaampaa, mikä näkyy ohjelman suorituskyvyssä.

Ohjelmointikielet voidaan jaotella tyypillisen ”käännettävät ja tulkattavat” -jaon lisäksi myös muistinhallinnan osalta. Osa kielistä edellyttää kehittäjältä muistin varaamista ja vapautusta manuaalisesti [10]. C++-ohjelmointikielessä muistinhallinta toteutetaan yleisesti ottaen manuaalisesti, vaikka kieleen onkin kehitetty roskienkerääjä vaihtoehtoiseksi toteutustavaksi. Kieli ei pitkään sisältänyt roskienkerääjää, ja siksi kieli onkin valittu esimerkiksi manuaalisesta muistinhallinnasta tässä työssä. Manuaalinen muistinhallinta on jokseenkin riskialtista, koska muistinkäytöstä huolehtiminen jää kehittäjän vastuulle, mikä saattaa tuottaa ongelmia kehittäjälle. [2]

Manuaalisella muistinhallinnalla voidaan kuitenkin saavuttaa tietyissä tapauksissa tehokkaampaa muistiresurssien käyttöä, ja tämän kautta myös parempaa suorituskykyä. Dynaamisesti toteutettu muistinvaraus mahdollistaa sellaisten ohjelmien kirjoittamisen, joiden osalta on mahdotonta etukäteen tietää, miten paljon muistia ohjelma lopulta tarvitsee käyttöönsä. Dynaamisesti toteutetulla muistinvarauksella voidaan myös kirjoittaa ohjelmia, joissa dynaamisesti varatun muistin kokonaismäärä koko suorituksen ajalta ylittää fyysisen muistin määrän, sillä muistialueiden vapauttaminen mahdollistaa uusien varauksien tekemisen. Dynaaminen muistinvaraus siis tarjoaa riskien ohella myös mahdollisuuden kirjoittaa ohjelmia, jotka edellyttävät kehittyneempää ja tehokkaampaa muistinkäyttöä. [11]

Osa kielistä tarjoaa sisäänrakennetun automaattisen muistinhallinnan, joka huolehtii kehittäjän puolesta muistin varaamisesta ja vapauttamisesta [10]. Hyvänä esimerkkinä kielestä, joka hyödyntää automaattista roskienkeruuta, toimii Java. Muistin siivousprosessi toimii ohjelman suorituksen taustalla, ja kun käytössä olevalle muistille ei ole enää tarvetta, vapautetaan tila ohjelman käyttöön automaattisesti [2]. Tämä helpottaa huomattavasti ohjelman kehitystä, sillä kehittäjän ei tarvitse huolehtia itse ohjelman tarvitseman

muistin hallinnoinnista. Automaattisen muistinhallinnan etuina voidaan pitää luotettavuutta ja turvallisuutta, kun muistinkäytön ongelmatilanteita vältetään automaattisuudella. Roskienkerääjän kehittyneet versiot osaavat myös tiivistää muistialueita, mikä voi ehkäistä joidenkin muistiongelmien muodostumista. Javan roskienkeruu ei kuitenkaan ole täydellinen, vaikkakin se estää monien muistinkäytön ongelmakohtien muodostumisen [9].

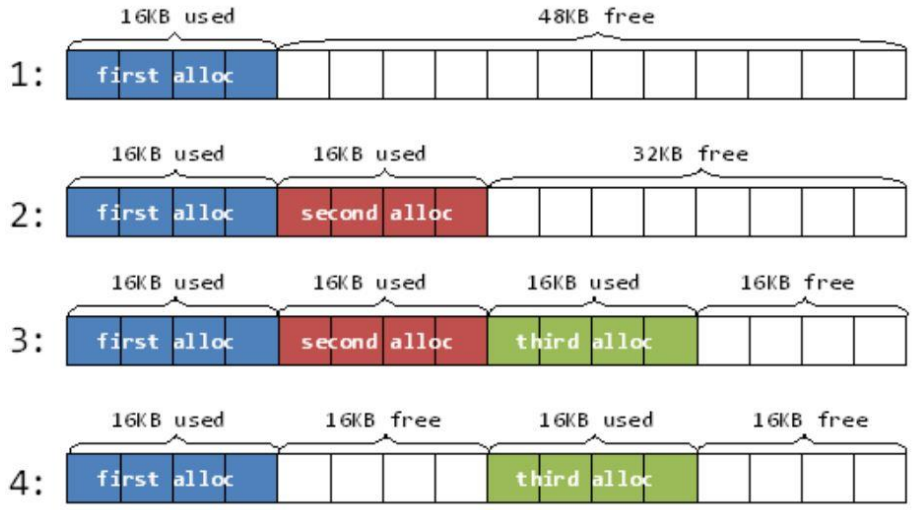
2.2 Muistinkäytön ongelmatilanteet

2.2.1 Muistin roskaantuminen

Muistin roskaantuminen on yksi suurimpia suorituskykyyn liittyviä ongelmia, joita kohdataan ohjelmiston kehityksessä. Muistin roskaantumisella tarkoitetaan tilannetta, jossa muistissa olevan datan säilyttämiselle ei ole enää ohjelman suorituksen kannalta tarvetta, mutta muisti on jäänyt silti vapauttamatta. [12] Turhaan varattuna oleva muisti on pois varattavissa olevista muistiresursseista. Tämä voi johtaa ongelmiin ohjelman suorituksessa, jos muistia ei ole riittävästi jäljellä uusille varauksille. Roskaantumisen välttämiseksi on onneksi kehitetty erilaisia työkaluja ja rutiineja muistin käytön helpottamiseksi.

2.2.2 Muistin pirstoutuminen

Muistin pirstoutumisella tarkoitetaan tilannetta, jossa muistitilaa ei ole hyödynnetty tehokkaasti, mikä johtaa siihen, ettei kaikkea prosessille varattua muistitilaa saada hyödynnetyksi. Ohjelman käynnistymisen jälkeen sen kekomuistiavaruus muodostuu pitkästä vierekkäisten muistipaikkojen yhtäjaksoisesta muistialueesta, mutta ajan kuluessa muistin jatkuva varaus ja vapautus muokkaavat yhtenäiset muistialueet pienemmiksi. Tämä muodostuu varatessa myöhemmin ongelmaksi, vaikka varattavan objektin koko olisi pienempi kuin vapaan muistin kokonaismäärä, sillä muistivarauksen tulee muodostua palasta keskenään peräkkäisiä muistipaikkoja, jota ei ole jäljellä pirstoutumisen takia.



Kuva 2: Muistin pirstoutuminen. [13]

Muistin pirstoutuminen voidaan jakaa sisäiseen pirstoutumiseen (engl. *internal fragmentation*) ja ulkoiseen pirstoutumiseen (engl. *external fragmentation*). Sisäisellä pirstoutumisella tarkoitetaan tilannetta, jossa varausalgoritmi on myöntänyt varaukselle pyydettyä muistialuetta suuremman alueen. Muistinvaraus voi tehokkuuden vuoksi varata suurempia muistialueita kuin olisi tarpeellista. Ulkoisella pirstoutumisella tarkoitetaan tilannetta, jossa käytössä olevien muistilohkojen väliin on muodostunut muistin vapautuksen myötä lyhyitä vapaana olevia muistialueita. Muisti jää tällöin pahimmassa tapauksessa käyttämättä, kun mistään ei löydy riittävän suurta yhtenäistä muistialuetta. [3]

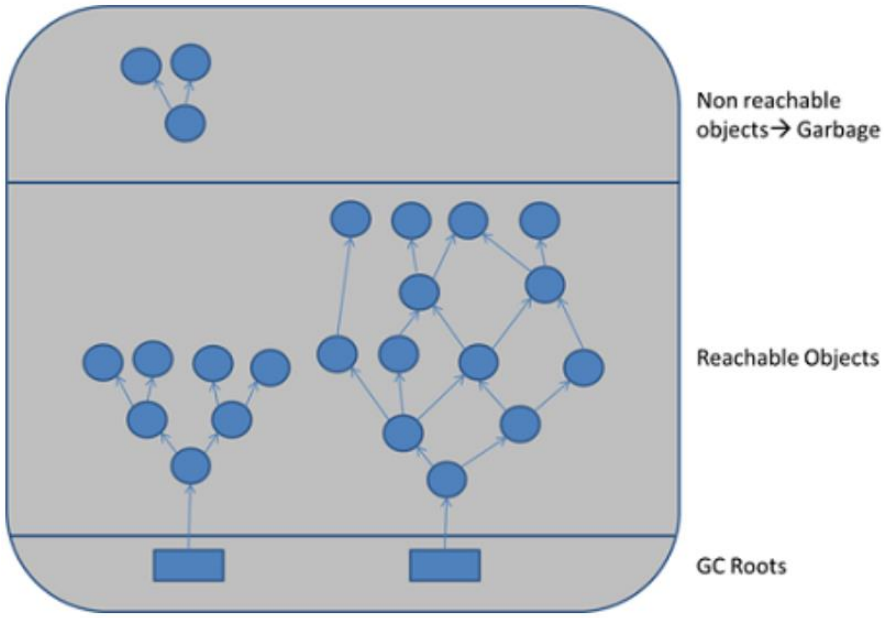
2.2.3 Muistiviittauksiin liittyvät ongelmat

Muistiviittauksiin liittyvät ongelmat ovat yleisiä ohjelmointikielissä, joissa ei hyödynnetä automaattista muistinhallintaa. Roikkuva viittaus (engl. *dangling reference*) tarkoittaa tilannetta, jossa ohjelma vapauttaa muistialueen, jota pystytään vielä käyttämään muistiviittauksen eli osoittimen kautta jostain muusta ohjelman alueesta. Roikkuva viittaus kuvaa siis tilannetta, jossa ohjelma on tuhonnut sille tarpeellisen muistiresurssin liian aikaisin. Kun myöhemmin tätä muistialuetta pyritään käyttämään viittauksen kautta, aiheuttaa viittaus ongelmatilanteen, minkä seurauksena ohjelman suoritus saattaa keskeytyä käyttöjärjestelmän toimesta. [3]

2.3 Roskienkerääjä

Kuten luvussa 2.1 todettiin, ohjelmoitaessa Java-ohjelmointikielellä muistinhallinta on toteutettu automaattisesti, eikä ohjelmoijan tarvitse huolehtia siitä. Automaattinen muistinhallinta perustuu erilaisten roskienkerääjien toimintaan. Javalla tehdyt ohjelmat roskaantuvat kuten muutkin ohjelmat, jos tarpeettomia muistin osia ei vapauteta. Javan roskienkerääjään on kehitetty erilaisia mekanismeja seuraamaan muistialueiden tarpeellisuutta ja siten tunnistamaan, milloin muistialue ei ole enää ohjelmalle tarpeellinen, ja se voidaan vapauttaa. Kehittyneet versiot roskienkerääjästä toimivat ohjelman taustalla, eivätkä aiheuta ohjelmalle näkyvää viivettä [14].

Automaattinen roskienkeräys perustuu objektien määrittämiseen aktiiviseksi tai saavuttamattomaksi. Objekti on saavuttamaton, jos siihen ei ole yhtäkään aktiivista viittausta suoritettavassa ohjelmassa. Myös Javan roskienkerääjä seuraa ohjelman kannalta aktiivisia objekteja ja huomaa, jos jokin objekti ei ole enää tarpeellinen ja siitä voidaan luopua. [14] Java käyttää roskienkeruussa lähtökohtana roskienkeruun juuria eli viittauksia olioihin, jotka ovat aina saavutettavissa. Ohjelman kannalta tarpeelliset objektit muodostavat juuresta alkaen kekoa muistuttavan tietorakenteen. Saavuttamattomat objektit eivät kuulu tähän tietorakenteeseen.

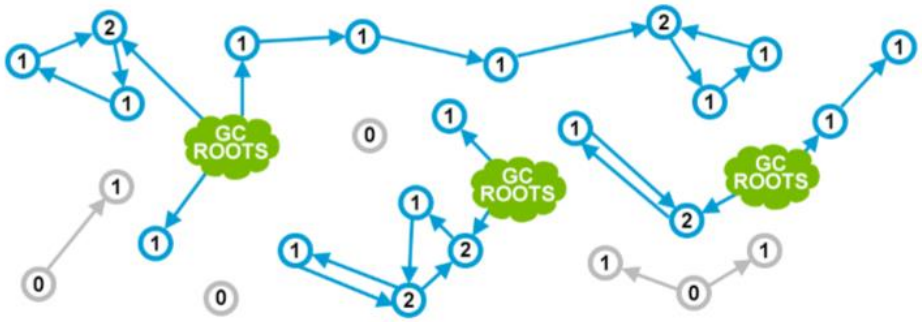


Kuva 3: Roskienkeruun juuret. [15]

Kuva 3 havainnollistaa roskienkeruun juurien roolia ja ohjelman hyödyntämien objektien muodostamaa tietorakennetta. Kuvan yläosassa ovat myös saavuttamattomat objektit, jotka ovat irrallaan tietorakenteesta.

2.4 Viittausten laskenta

Viittausten laskenta (engl. *reference counting*) on menetelmä, jossa lasketaan aktiivisten viittausten määrä tiettyyn objektiin tai muistialueeseen. Menetelmää hyödynnetään useissa ohjelmointikielissä. Esimerkiksi C++-ohjelmointikieli sisältää osoittimia, jotka hyödyntävät viittaustenlaskentamenetelmää. Automatisoidussa toteutuksessa algoritmi pitää kirjaa jokaisen objektin aktiivisten viittausten lukumäärästä. Jos tämä lukumäärä on nolla, luokitellaan objekti roskaksi. Silloin objekti on saavuttamaton ohjelmassa, sillä siihen ei ole aktiivista muistiviittausta, eikä sitä täten voi muokata tai hyödyntää ohjelmassa. Ohjelman tulee siis tarkistaa ja päivittää objektin viittausten lukumäärä jokaisella kerralla, kun sen muistiviittauksia muutetaan. Viittausten laskeminen kuormittaakin ohjelmaa jatkuvalla viittausten lukumäärän tarkastuksella. [16]



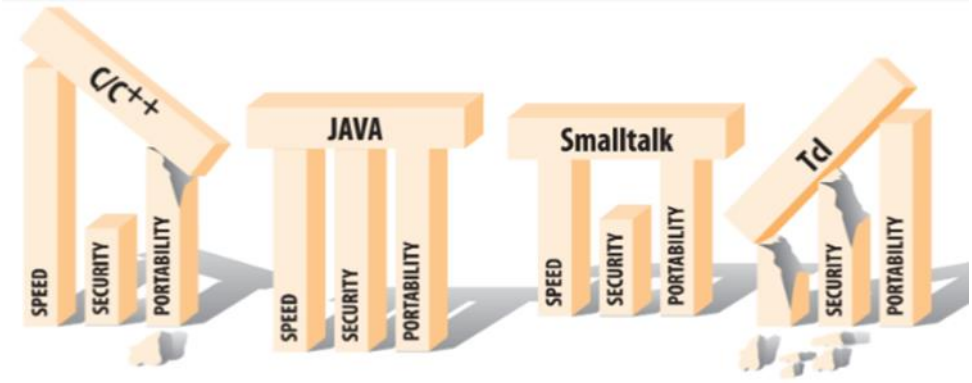
Kuva 4: Viittausten laskenta. [2]

Kuvassa 4 vihreät pilvet kuvastavat roskienkeruun juuria, ja luvut objekteihin kohdistuvien viittausten lukumäärää. Harmaat objektit ovat saavuttamattomia ohjelmassa ja täten roskaa, joka voitaisiin vapauttaa ohjelman käyttöön uudelleen.

Viittaustenlaskentamenetelmä on kuitenkin altis virheille. Algoritmi ei välttämättä tunnista kaikkia tarpeettomia objekteja ja osaa poistaa niitä. Esimerkkinä tästä toimii tilanne, jossa ohjelman kannalta tarpeettomat objektit ovat muodostaneet viittausten avulla kehän. Algoritmi tarkistaa viittaukset, muttei tunnista objekteja roskaksi, sillä jokainen tarpeettomista objekteista on viittauksen kohteena kehärakenteen myötä. Kehä on kuitenkin irrallaan juuresta rakentuvasta tietorakenteesta, ja kehän objektit voitaisiin vapauttaa. [2]

3 JAVA MUISTINHALLINTA

Javalla on paljon yhtäläisyyksiä C++-kielen kanssa, mikä ei ole yllätys, sillä alun perin Javan kehitys lähti liikkeelle C++:n ominaisuuksien laajentamisesta. Javasta on kuitenkin kehittynyt täysin oma kieli, jolla on nykyisin huomattavia eroavaisuuksia C++-kieleen verrattuna. Kielet ovat syntaksiltaan varsin samanlaiset, mutta ominaisuuksiltaan Javalla on enemmän yhtäläisyyksiä muiden korkean tason kielten, kuten Smalltalkin ja Lispin kanssa [14, s. 10]. Ensimmäinen versio roskienkerääjästä nähtiinkin Lisp-ohjelmointikielessä John McCarthyn kehittämänä vuonna 1959. Java ei mahdollista suoran muistiosoituksen käyttöä ja sisältää automaattisia roskienkerääjiä poiketen C++-kielestä [17]. Kieli mielletäänkin C++-kieleen verrattuna yleisesti turvallisempana. Java on suunniteltu olemaan turvallinen työkalu kehittäjälle. Javan tapa huolehtia muistin käytöstä toimii tästä erinomaisena esimerkkinä.



Kuva 5: Java vertailussa muihin kieliin. [14]

Roskienkeruu on toteutettu Javassa erilaisia algoritmeja hyödyntämällä. Java-ohjelmia kirjoittaessa kehittäjän ei pääsääntöisesti siis tarvitse huolehtia ohjelman tarvitseman muistin varaamisesta tai vapauttamisesta. Roskienkerääjä helpottaa siis kehittäjän tekemää työtä ja pienentää muistinkäyttöön liittyvien virheiden todennäköisyyttä.

3.1 Algoritmit

Automaattisessa muistinhallinnassa hyödynnettäviä perusstrategioita ovat viittausten laskeminen (engl. *reference counting*), merkitse ja pyyhi (engl. *mark and sweep*), kopiointi tai pysähdy ja kopioi (engl. *copying/stop and copy*) ja merkitse ja tiivistä (engl. *mark and compact*). Osa algoritmeista kykenee toimimaan vähitellen ohjelman aikana, mikä pie-

nentää roskienkeruun aiheuttamia taukoja ohjelman suorituksessa. Tällöin siis roskienkerääjän ei tarvitse vapauttaa koko kekomuistia yhdellä kerralla. Kehittyneet algoritmit pystyvät toimimaan ohjelman suorituksen aikana taustalla. Jotkin algoritmit ovat toiminnaltaan näitä kankeampia ja vaativat koko kekomuistin läpikäymistä kerralla. Tällöin roskienkeruun aikana ohjelma on pysäytettynä, mikä ei tietenkään ole kovin toivottava asia. Roskienkeruun uusimmissa versioissa kuitenkin tyypillisesti hyödynnetään hybriditoteutusta, jolloin roskienkerääjä osaa hyödyntää eri algoritmeja keon eri alueille. [16] Uusimmat roskienkerääjät osaavat myös hyödyntää useampaa säiettä.

3.1.1 Merkitse ja pyyhi

Merkitse ja pyyhi -algoritmi oli ensimmäisenä käytössä John McCarthyn kehittämässä Lisp-ohjelmointikielessä jo vuonna 1960 [16]. Merkitse ja pyyhi -algoritmin toiminta koostuu kahdesta vaiheesta:

1. Merkintävaihe (engl. *mark phase*)
2. Pyyhkimisvaihe (engl. *sweep phase*)

Ensimmäisessä vaiheessa roskienkerääjä lähtee liikkeelle juurista ja käy tietorakenteen jokaisen solmun läpi. Eteneminen tapahtuu muistiviittausten avulla aina niihin solmuihin asti, joista ei ole enää viittausta seuraavaan, jolloin tietorakenne on käyty kokonaan läpi. Roskienkerääjä merkitsee jokaisen solmun, jossa se on käynyt, ja näin pidetään kirjaa ohjelmalle tarpeellisista muistialueista. [16]

Toisessa vaiheessa kekomuisti tyhjennetään eli ”pyyhitään”. Kaikki objektit, joita ei ole merkitty roskienkerääjän toimesta, luokitellaan roskaksi ja vapautetaan [16]. Kuvat 3 ja 4 toimivat havainnollistavana tässäkin tapauksessa. Jokainen juuresta muistiviittauksien kautta etenemällä löydetty solmu vain merkitään erikseen.

Merkitse ja pyyhi -algoritmilla on myös ongelmia muistinhallintaan liittyen. Algoritmi käy pyyhkimisvaiheessakin jokaisen kekomuistissa olevan objektin läpi riippumatta siitä, onko objekti merkitty vai ei. Olettaen, että ohjelmoitaessa usein suuri osa kekomuistissa sijaitsevista objekteista on merkitsemättömiä eli roskaa, tekee roskienkerääjä huomattavan määrän turhaa työtä käydessään kaikki solmut läpi. Tämä ei tietenkään ole suoritus-

kyvyn kannalta toivottava asia. Merkitse ja pyyhi -menetelmään perustuvat roskienkerääjät saattavat myös aiheuttaa kekomuistin pirstoutumista, jonka aiheuttamat ongelmatilanteet käsiteltiin luvussa 2.2.2. [16]

3.1.2 Pysähdy ja kopioi

Kopiointiin perustuva roskienkerääjä jakaa kekomuistin kahteen alueeseen, joista toinen sisältää kaiken aktiivisen datan, ja toinen alue jää täysin tyhjäksi. Roskienkerääjä odottaa aktiivisen alueen täyttymistä, ja kun alue on täynnä, ohjelman suoritus pysäytetään. Kaikki aktiiviset objektit kopioidaan tyhjäksi jääneeseen muistitilaan. Tämän jälkeen kaikki aktiiviseen tilaan jääneet objektit ovat roskaa, ja ne voidaan vapauttaa. Operaatio voidaan ajatella myös muistialueiden roolien vaihtamisella. Vanhasta aktiivisesta alueesta tulee uusi tyhjä muistialue, kun taas vanha tyhjä muistialue sisältää nyt ohjelman kannalta aktiiviset objektit. [16]

Kopiointimenetelmän selvä etu on se, että roskienkerääjän tarvitsee käydä vain aktiiviset objektit läpi. Roskaksi luokitellut objektit jäävät ilman tarkastusta, mikä nopeuttaa prosessia. Kopioinnin heikkous on objektien siirtely muistitilasta toiseen ja viittausten päivittäminen uuteen alueeseen jokaisen kopioinnin jälkeen. Ohjelmassa pitkään hyödynnettävät objektit kopioidaan siis edestakaisin jatkuvasti aina roskienkeruuta tehdessä. Kopiointimenetelmä vaatii myös enemmän muistia kuin merkitse ja pyyhi -menetelmä, mikä johtuu datan kopioinnista kahden muistialueen välillä. [16]

3.1.3 Merkitse ja tiivistä

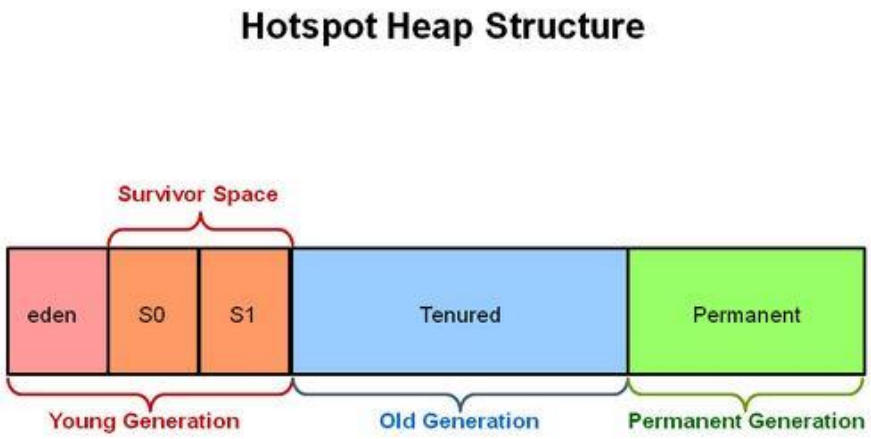
Merkitse ja tiivistä -algoritmi yhdistää kopiointialgoritmin suorituskyvyn sekä merkitse ja pyyhi -menetelmän pienemmän muistiresurssien tarpeen. Merkitse ja tiivistä -menetelmä noudattaa merkitse ja pyyhi -algoritmin tavoin kahta vaihetta roskienkeruuprosessissa. Menetelmän ensimmäisessä vaiheessa käydään kaikki kekomuistin tietorakenteen solmut läpi samalla tavalla kuin merkitse ja pyyhi -algoritmi tekee. Jokainen aktiivinen solmu merkitään tulevaa muistin pyyhkimistä varten. Toisessa vaiheessa jokainen aktiivinen objekti kopioidaan kekomuistin pohjalle eli tiivistetään. Jos algoritmi toimii jokaisella roskienkeruun suorituskerralla toivotusti, kekomuistista voidaan selkeästi hahmot-

taa raja aktiivisen alueen ja vapaan alueen välillä. Pitkäkestoiset objektit kerääntyvät tällöin kekomuistin pohjimmaiseksi, ja niitä ei tarvitse kopioida toistuvasti siivouksen yhteydessä kuten kopiointimenetelmässä. [16]

3.2 Roskienkerääjät Javan kehitysympäristössä

Javan roskienkerääjästä on kehitetty useampia versioita, sillä roskienkeräyksen tarve voi vaihdella hyvinkin paljon kehitettävän ohjelman mukaan. Eri toteutukset hyödyntävät aikaisemmin esiteltyjä algoritmeja eri kekomuistin alueisiin. Näin pystytään kontrolloimaan, millaisiin objekteihin ja muistialueisiin siivousoperaatio kohdistuu. Tässä kappaleessa on esitelty roskienkerääjistä yleisimmät. Lopussa tarkastellaan hieman tarkemmin Java SE 11 -version myötä ajankohtaista roskienkerääjän toteutusta.

Javan roskienkerääjien toiminnan ymmärtämiseksi on syytä tutustua aluksi tarkemmin Javan virtuaalikoneen tapaan jaotella kekomuisti sukupolviin (engl. *JVM Generations*). Kekomuisti jaetaan nuoreen sukupolveen (engl. *Young Generation*), vanhaan sukupolveen (engl. *Old Generation or Tenured Generation*) ja pysyvään sukupolveen (engl. *Permanent Generation*). [18]



Kuva 6: Kekomuistin jaottelu sukupolviin. [18]

Nuori sukupolvi toimii muistialueena, jonne kaikki uudet objektit tallennetaan. Kun nuoren sukupolven muistialue täyttyy, aiheuttaa tämä pienen roskienkeruun (engl. *minor garbage collection*) kyseiselle muistialueelle. Pienempi roskienkeruu on tyypillisesti nopea

toimenpide, mikä johtuu suuresta roskien määrästä suhteessa pidempiaikaisiin objekteihin. Ajan kuluessa nuoressa sukupolvessa ”ikääntyneet” objektit siirretään vanhan sukupolven muistialueelle. [18]

Vanha sukupolvi toimii muistialueena, jota hyödynnetään pitkään käytössä olevien objektien tallentamiseen. Kun vanhan sukupolven alue kekomuistista täyttyy, suoritetaan tähän muistin osaan suuri roskienkeräys (engl. *major garbage collection*). Suurempi roskienkeräys on tyypillisesti pidempään kestävä toimenpide kuin pieni roskienkeräys, koska suuri roskienkeräys kattaa kaikki aktiiviset objektit. Tämän vuoksi suurten roskienkeräysten määrä olisi hyvä pitää mahdollisimman pienenä. Sekä pieni roskienkeruu että suuri roskienkeruu keskeyttävät kaikki ohjelman osat suorituksen ajaksi (engl. *Stop the World Event*). [18]

Pysyvän sukupolven muistialue sisältää metadatan, jota Javan virtuaalikone hyödyntää luokkien ja metodien määrittelyissä. Tämä muistialue täydentyy ohjelman toteutuksessa hyödynnetyillä luokilla. Myös luokkien varaama muistitila voidaan vapauttaa, mikäli virtuaalikone havaitsee, että niitä ei enää tarvita ohjelman toteutuksessa. [18]

3.2.1 The Serial Garbage Collector

Serial on roskienkerääjän toteutuksista yksinkertaisin. Nimensä mukaisesti *Serial* toimii jaksoittaisesti ja hyödyntää vain yhtä virtuaalista prosessoria. *Serial* hyödyntää merkitse ja tiivistä algoritmia. Merkitse ja tiivistä -menetelmä siirtää vanhemmat objektit kekomuistin alkuun, jolloin uudet muistinvaraukset tehdään yhteen jatkuvaan muistialueeseen kekomuistin loppupäässä. Muistin tiivistys nopeuttaa uuden muistialueen allokointia kekomuistista. *Serial* sopii roskienkerääjäksi, kun ohjelmalta ei odoteta erityisen lyhyitä roskienkeruun aiheuttamia pysähdysaikoja. *Serial*-roskienkeruu voidaan ottaa ohjelman toteutuksessa käyttöön komentoriviltä kutsulla `-XX:+UseSerialGC`. [18]

3.2.2 The Parallel Garbage Collector

Parallel-roskienkerääjä hyödyntää siivoamisoperaatiossa useaa säiettä nuoren sukupolven muistialueen siivoukseen. Toteutus on siis nimensä mukaisesti rinnakkainen. Säikeiden lukumäärä voidaan määrittää komentoriviltä syötteellä –

`XX:ParallelGCThreads=<desired number>`. *Parallel* osaa hyödyntää useampaa prosessoria nopeuttaakseen ohjelman toimintaa. *Parallel*-roskienkerääjä sopii tilanteisiin, joissa roskaantuneita muistin osia voidaan olettaa olevan paljon ja suuret pysähdykset ohjelmassa on sallittuja. [18]

Parallel-roskienkerääjä jakautuu vielä kahteen erilaiseen toteutukseen. Komentorivisyötteellä `-XX:+UseParallelGC` saadaan useaa säiettä hyödyntävä nuoren sukupolven roskienkerääjä ja yhtä säiettä käyttävä vanhan sukupolven roskienkerääjä. Tämä toiminto osaa myös tiivistää siivoamansa muistialueen. Komentorivisyötteellä `-XX:+UseParallelOldGC` saadaan useaa säiettä hyödyntävä, nuoren -ja vanhan sukupolven muistialueiden roskienkerääjä. Tämäkin toteutus osaa tiivistää muistialueet siivouksen jälkeen. [18]

3.2.3 The Concurrent Mark Sweep Collector (CMS)

The Concurrent Mark Sweep Collector (CMS) -roskienkerääjän siivousoperaatio kohdistuu vanhan sukupolven muistialueeseen. *CMS*:llä pyritään minimoimaan roskienkeruun aiheuttamat ohjelman pysähdykset suorittamalla roskienkeruu samanaikaisesti ohjelman suorituksen kanssa. *CMS*-roskienkerääjä ei tiivistä siivottuja muistialueita, mikä aiheuttaa muistin pirstoutumista. *CMS*-kerääjä sopii tilanteisiin, joissa kehitettävän ohjelmiston pysähdykset halutaan minimoida. *CMS*-kerääjä voidaan ottaa käyttöön komentorivisyötteellä `-XX:+UseConcMarkSweepGC`. [18]

3.2.4 The G1 Garbage Collector

G1-roskienkerääjä on roskienkerääjä, joka yhdistää aikeisempien toteutuksien hyviä ominaisuuksia. *G1* on ollut saatavilla Javan 7-versiosta lähtien. *G1*-roskienkerääjä osaa hyödyntää useaa säiettä ja se työskentelee ohjelman suorituksen taustalla. Se osaa myös vähitellen tiivistää siivottuja muistialueita. *G1*-roskienkerääjä voidaan ottaa käyttöön komentorivisyötteellä `-XX:+UseG1GC`. [18]

3.2.5 The Z Garbage Collector

Z-roskienkerääjä on kehitetty matalaa viiveaikaa vaativia ohjelmia varten. *Z* toimii ohjelman suorituksen kanssa rinnakkaisesti aiheuttamatta suuria pysähdyksiä ohjelman osien suorituksessa. *Z*-kerääjän kehityksessä tavoitteena oli kehittää roskienkerääjä, jonka aiheuttamat pysähdykset olisivat maksimissaan 10 millisekuntia. Tavoitteena oli myös mahdollistaa peräti teratavujen kokoinen kekomuistin alue ohjelman käyttöön ilman vaikutuksia roskienkeruun pysähdyksiin. *Z*-roskienkerääjä tarjoaa kehittäjälle mahdollisuuden asettaa kekomuistin maksimikoko. Toinen suorituskyvyn optimointiin tarjottu keino on rinnakkaisten säikeiden lukumäärän määrittäminen. Tämä voidaan tehdä komentorivisyötteellä `-XX:ConcGCThreads`. *Z*-kerääjän kokeiluversio tuli yleisesti saataville 25.9.2018 Java SE 11 -version myötä. *Z*-roskienkerääjä saadaan käyttöön syötteillä `-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`. [19]

```
ZGC
               avg: 1.091ms (+/-0.215ms)
    95th percentile: 1.380ms
    99th percentile: 1.512ms
  99.9th percentile: 1.663ms
 99.99th percentile: 1.681ms
               max: 1.681ms

G1
               avg: 156.806ms (+/-71.126ms)
    95th percentile: 316.672ms
    99th percentile: 428.095ms
  99.9th percentile: 543.846ms
 99.99th percentile: 543.846ms
               max: 543.846ms
```

Kuva 7: *Z*- ja *G1*-roskienkerääjät vertailussa pysähdysaikojen osalta. [19]

Kuvassa 7 on vertailtu *Z*- ja *G1*-roskienkerääjien pysähdysaikoja käytettäessä 128 gigatavun kekomuistia. Testit on toteutettu *SPECjbb® 2015* -suorituskykytestillä. Testituloksista huomataan, että *Z*-kerääjän pysähdysajat ovat selkeästi 10 millisekunnin alle ja pysähdysaikojen olevan selkeästi *G1*-kerääjän pysähdysaikoja pienempiä. [19]

3.3 Työkalut

Javaan on kehitetty erilaisia työkaluja suorituskyvyn mittaamiseen ja seurantaan. Javan työkaluilla pystytään myös tarkastelemaan muistinkäyttöä tarkemmin. Javan muistinkäytön mittaamisella pyritään optimoimaan sovelluksien vastausaikaa ja suorittimien kuormitusta. Muistinkäyttöä tarkasteltaessa on hyödyllistä tutkia, miten roskienkerääjän toiminta vaikuttaa suoraan ohjelman vastausaikaan. Roskienkerääjän aiheuttamat pysähdykset ohjelman suorituksessa näkyvät suoraan ohjelman vastausajassa. [20]

Javan viidennestä versiosta lähtien Javan kehitystyökalut on tarjonnut *JConsole*-seurantatyökalun. *JConsole* tarjoaa graafisen käyttöliittymän ja hyödyntää Javan virtuaalikonetta ohjelmien suorituskyvyn ja resurssienkäytön seuraamisessa. [21]

Muistinhallinnan kannalta keskeisempi seurantatyökalu on kuitenkin *jStat*, joka mahdollistaa muistinkäytön ja roskienkerääjän seurannan. Se hyödyntää *JConsolen* tavoin Javan virtuaalikonetta, ja sitä voidaan hyödyntää kekomuistiin ja roskienkerääjään liittyvien suorituskykyongelmien tarkasteluun. [22]

Javan kuudennesta versiosta lähtien ohjelmissa ilmeneviä ongelmia on pystytty kartoittamaan *VisualVM*-työkalulla. Aikaisemmin mainitut työkalut *JConsole* ja *jStat* ovat osa *VisualVM*:n työkaluja. *VisualVM* kerää tietoa virtuaalikoneelta ja esittää kerätyt tiedot graafisesti. [23]

4 C++:N MUISTINHALLINTA

C++-ohjelmointikielen kehitys alkoi vuonna 1979 Bjarne Stroustrupin toimesta. Kieli on kehitetty C-kielestä parantelemalla sen ominaisuuksia. Aluksi kieli tunnettiin nimellä ”C with Classes” ja nimeksi vaihdettiin ”C++” vasta vuonna 1983. Ensimmäiset kehitysaskeleet kielessä tapahtuivatkin olio-ohjelmoinnin konseptissa. Myöhemmin kieltä paranneltiin esimerkiksi lisäämällä geneerisyyteen ja poikkeusten käsittelyn liittyviä ominaisuuksia. Kielestä kehittyi varsin suosittu, ja useissa kielissä onkin vaikutteita C++-kielestä. Kieli on yleisesti ottaen suunniteltu olemaan tehokas ja toimimaan lähellä laitteistoa. [11] GitHubin tekemän listauksen mukaan C++ on edelleen yksi käytetyimmistä ohjelmointikielistä maailmanlaajuisesti [24].

C++-ohjelmien muistinhallinta toteutetaan tyypillisesti käsin, vaikka kieleen onkin jo kehitetty konservatiivinen roskienkerääjäkirjasto *Boehm*, joka hyödyntää merkitse ja pyyhi -algoritmia. Konservatiivisella roskienkerääjällä tarkoitetaan roskienkerääjää, joka toimii ilman yhteistyötä kääntäjän kanssa, jolloin kerääjä ei pysty erottamaan varmuudella, onko jokin muistissa oleva arvo viittaus vai ei. Kerääjä sisältää toiminnon, jolla voidaan tarkistaa ohjelma mahdollisilta muistivuodoilta, vaikka muistinhallinta olisi toteutettu ohjelmassa manuaalisesti. [25]

Kuten aikaisemmin todettiin, C++-kieli eroaa Javasta esimerkiksi mahdollistamalla osoittimien käytön. Osoittimilla on merkittävä rooli C++-ohjelmoinnissa, ja kieli tarjoaakin useampia erilaisia osoitintyyppejä kehittäjän käyttöön. Tässä osiossa tarkastellaan osoittimia muistinhallinnan näkökulmasta ja perehdytään hieman tarkemmin C++-kielessä käytettävään RAII-muistinhallintakonseptiin (engl. *Resource acquisition is initialization*). Aluksi kuitenkin käsitellään muistia C++-ohjelmoinnissa yleisesti.

4.1 Muuttujan näkyvyysalue ja elinikä

Muuttujan näkyvyysalueella (engl. *scope*) viitataan siihen osaan ohjelmaa, jossa muuttuja on käytettävissä. Muuttujan varauksen yhteydessä tilanvaraus tehdään tälle ”muistialueelle”. Yleisesti ottaen muuttuja voi olla käytettävissä funktion, luokan, tiedoston tai ohjelman sisällä. Muuttujan määritys sitoo sen olemaan käytettävissä alueella, jossa se on

määritelty ja tätä sisemmillä näkyvyysalueilla. Muuttuja voidaan esimerkiksi luoda funktion sisällä, jolloin se on käytettävissä ainoastaan kyseisen funktion sisällä. Tällaista lohkon sisällä luotua muuttujaa kutsutaan paikalliseksi muuttujaksi (engl. *local variable*). Luokkarakenteeseen kuuluvia muuttujia kutsutaan jäsenmuuttujiksi (engl. *member variables*), ja niitä voidaan käyttää luokan jäsenfunktioissa. Tietyn moduulin sisäinen yksityinen muuttuja (engl. *private variable*) on käytettävissä ainoastaan yhden tiedoston sisällä. Kaikkialla ohjelmassa käytettävissä olevaa muuttujaa kutsutaan globaaliksi muuttujaksi. Globaali muuttuja määritellään funktioiden ja luokkien ulkopuolella. [26]

Tarkemmin määriteltynä muuttujan elinikä voi olla automaattinen, staattinen tai dynaaminen. Paikallisten muuttujien elinikä on automaattinen, sillä ne tuhotaan näkyvyysalueen lopussa automaattisesti. Automaattinen muuttuja määritellään aina uudestaan näkyvyysalueelle tultaessa, ja tämän myötä myös muisti varataan uudelleen jokaisella kerralla. Näkyvyysalue ja muuttujan elinikä eivät kuitenkaan aina ole samat, vaan tämä pätee ainoastaan automaattisille muuttujille. [26]

Muuttujan staattinen elinikä tarkoittaa, että muuttujan elinikä on koko ohjelman suoritusaika. Ohjelmointikielissä staattiset muuttujat määritellään tyypillisesti `static`-määreellä. Staattinen muuttuja voi olla myös paikallinen, vaikka sen elinikä onkin näkyvyysalueen suoritusaikaa huomattavasti pidempi. Lohkosta poistuttaessa staattinen paikallinen muuttuja jää eloon ja palattaessa lohkoon myöhemmin se muistaa arvonsa edelliseltä kerralta. [26]

Muuttuja voi olla myös dynaaminen, jolloin ohjelmoija päättää erillisillä käskyillä, milloin muuttuja luodaan ja tuhotaan. Tämän tutkielman osalta keskitytään dynaamisiin muuttujiin, sillä tarkastelussa on ohjelmoijan toteuttama muistinhallinta.

4.2 Muistin varaaminen ja vapauttaminen

C++-kielessä on kaksi peruskäskyä muistinhallintaa varten. Uusi dynaaminen muuttuja varataan `new`-käskyllä. Dynaamisen muuttujan hyödyntäminen ohjelmassa vaatii muuttujalle varatun muistin osoitteen tallentamista osoitinmuuttujaan. Varauskäsky `new` palauttaakin varaamansa muistialueen osoitteen. Muistialueesta voidaan luopua `delete`-käskyllä. Jos ohjelmassa on tarvetta suurelle muistilohkolle, voidaan `new`-käskyllä varata

myös iso muistialue taulukkotyyppiselle muuttujalle. Tällöin `new`-käsky palauttaa osoitteen taulukolle varatun muistialueen alkuun. Dynaamisen muuttujan elinkaari alkaa, kun `new`-käsky on suoritettu, ja muistialue on sen myötä varattu. [26]

Muistivarauksiin käytettävät `new`-käskyt ovat syntaksiltaan seuraavanlaisia: `osoitinmuuttuja = new tietotyyppi;` tai `osoitinmuuttuja = new tietotyyppi[muistilohkojen lukumäärä];`. Ohjelman suorittaessa `new`-käskyn se kutsuu `operator new` -funktiota ja vastaavasti `delete`−käskyn kohdalla kutsutaan `operator delete` -funktiota. `delete`-käskyt ovat syntaksiltaan seuraavanlaisia: `delete osoitinmuuttuja;` tai `delete[] osoitinmuuttuja;`. Käsky vapauttaa koko muuttujan osoittaman muistialueen. [27]

C++-ohjelmointikieli tarjoaa myös tekniikoita muistinvarauksen onnistumisen seurantaan. Muistinvarauksesta saattaa aiheutua `std::bad_alloc`-poikkeus, jos vapaata muistia ei ole riittävästi jäljellä. Ohjelman suoritus keskeytyy, jos ohjelmassa aiheutunutta `std::bad_alloc`-poikkeusta ei käsitellä. Toinen metodi on `nothrow`, jota käytetään `new`-käskyn yhteydessä. Metodia käytettäessä palauttaa kutsu `bad_alloc`-poikkeuksen sijaan tyhjän osoittimen (engl. *null pointer*), jos muistia ei ollut varaushetkellä tarpeeksi. [26] Tällöin muistinvarauksen onnistuminen voidaan tarkistaa osoitinmuuttujan tarkastuksella, eikä ohjelman suoritus keskeydy.

```
1  #include <iostream>
2  using namespace std;
3
4  void esimerkki(){
5
6      int *osoitin1 = new int;
7      int *osoitin2 = new(nothrow) int;
8      int *osoitin3 = new int[3];
9
10     if(osoitin2 == nullptr){
11         cout << "Muistinvaraus epäonnistui!" << endl;
12     }
13
14     delete osoitin1;
15     delete osoitin2;
16     delete[] osoitin3;
17 }
18
19 int main(){
20     esimerkki();
21     return 0;
22 }
```

Kuva 8: `new` ja `delete` operaattoreiden käyttö.

Kuva 8 havainnollistaa `new`- ja `delete`-operaattoreiden käyttöä. Osoitinmuuttujat `osoitin1`, `osoitin2` ja `osoitin3` on määritelty eri tavoin hyödyntämällä `new`-käskyä. Ohjelmassa rivillä 6 suoritettava `osoitin1`-muuttujan varaus aiheuttaa `bad_alloc`-poikkeuksen, ja ohjelman suoritus keskeytyy, mikäli muistitila ei riitä. Rivillä 7 suoritettava `osoitin2`-muuttujan määrittely käsittelee vastaavassa tilanteessa poikkeuksen `nothrow`-metodilla. Myöhemmin ohjelmassa `osoitin2`-muuttujan muistinvarauksen onnistuminen tarkistetaan if-lausekkeessa tarkastamalla, onko muuttuja tyhjä osoitin. Rivillä 8 määritellään `osoitinmuuttuja3`, joka osoittaa kokonaislukutaulukkoon, jossa on tilaa kolmelle kokonaisluvulle. Kaikki osoitinmuuttujat tuhotaan, ja niiden osoittamat muistialueet vapautetaan `delete`-käskyllä.

4.3 C++-osoittimet

C++-ohjelmointikielessä osoitinmuuttujia käytetään dynaamisten muuttujien muistiosoitteiden tallentamiseen. Osoitinmuuttuja on tietotyypiltään sellainen arvo, johon voidaan

tallentaa osoittimen osoittaman tietotyypin mukainen muistiosoite. Osoitin mahdollistaa tietoalkioon viittaamisen sen osoitteen avulla. C++-kielessä osoittimet määritellään seuraavasti: `kohdetyyppi *osoitinmuuttujan_nimi;`. Muuttujan muistiosoite voidaan tarkistaa &-operaattorilla asettamalla operaattori muuttujan nimen eteen. Muistipaikan sisältö saadaan käyttöön *-operaattorilla asettamalla operaattori muuttujan nimen eteen samoin kuin &-operaattorin kanssa. [11]

4.3.1 Älykkäät osoittimet

C++-ohjelmointikieli tarjoaa käyttäjälle `memory`-kirjaston, joka pitää sisällään useita osoitintyyppejä. Näitä *memory*-kirjaston osoittimia kutsutaan älykkäiksi osoittimiksi (engl. *smart pointer*). *Memory*-kirjaston tarjoamia älykkäitä osoittimia ovat esimerkiksi heikko osoitin (engl. *weak shared pointer*, `weak_ptr`), jaettu osoitin (engl *shared pointer*, `shared_ptr`) ja yhden omistajan osoitin (engl. *unique pointer*, `unique_ptr`). Vaikka ohjelman muistinkäsittely voitaisiin järjestää pelkästään `new`- ja `delete`-operaattoreiden avulla, on älykkäiden osoittimien hyödyntäminen todennäköisesti helpompi ratkaisu. Älykkäät osoittimet hoitavat dynaamisesti varatun muistin vapauttamisen automaattisesti. Ne toimivat samankaltaisesti normaalien osoittimien kanssa ja helpottavat ohjelmoijan työtä, kun muistia ei tarvitse itse vapauttaa. Tämä vähentää siis myös virheiden todennäköisyyttä. Älykkäitä osoittimia voidaan hyödyntää ohjelmassa ottamalla `memory`-kirjasto käyttöön ohjelman alussa rivillä: `#include <memory>`. [11]

Älykkäiden osoittimien kyky huolehtia muistin vapauttamisesta perustuu viittausten laskentatekniikkaan. Jaettu osoitin ylläpitää viitelaskuria, joka sisältää tiedon siitä, kuinka monta jaettua osoitinta osoittaa samaan muistialueeseen, johon kyseinen osoitin osoittaa. Muistialue vapautuu automaattisesti, kun viitelaskurin arvoksi tulee nolla. Muistin vapauttamiseen liittyen puhutaan ”omistajasta”, jolla tarkoitetaan sitä osoitinta, jonka vastuulla `new`-käskyllä varatun muistin vapautus on. Älykkäitä osoittimia käytettäessä tämä vastuu siirretään jaetuille osoittimille. [26]

4.4 Resource acquisition is initialization (RAII)

RAII-konseptilla tarkoitetaan toimintamallia, jossa ohjelmassa tarvittavat resurssit varataan konstruktorissa ja vapautetaan destruktorissa. Toimintamallia hyödynnetään useissa olio-orientoituneissa ohjelmointikielissä, ja ensimmäisenä se nähtiin C++-kielessä Bjarne

Stroustupin ja Andrew Koenigin kehittämänä. RAII-konseptin merkittävimmät hyödyt ovat resurssien varaus näkyvyysalueiden mukaisesti ja poikkeustilanteisiin liittyvien uhkien välttäminen. RAII-toimintamallin mukaisesti resurssit voidaan sitoa pinoon luotavan olion elinaikaan, jolloin ne vapautetaan olion tuhoutuessa. [11]

Kuvan 8 esimerkkiohjelmassa osoitinmuuttujat menetetään, kun `esimerkki()`-funktiosta poistutaan. Mikäli funktiosta poistuttaisiin ennen muistin vapauttamista, jäisivät varatut muistialueet vapauttamatta. RAII-konseptilla ongelma voidaan välttää sitomalla resurssin elinaika olion elinaikaan.

```
1  #include <iostream>
2  using namespace std;
3
4  class Muistiolio1 {
5      public:
6          Muistiolio1(){ osoitin1 = new int; } //Konstruktori
7          ~Muistiolio1(){ delete osoitin1; } //Destruktori
8      private:
9          int *osoitin1;
10 };
11 class Muistiolio2 {
12     public:
13         Muistiolio2(){ osoitin2 = new int[3]; } //Konstruktori
14         ~Muistiolio2(){ delete[] osoitin2; } //Destruktori
15     private:
16         int *osoitin2;
17 };
18 void esimerkki(){
19     Muistiolio1 olio1;
20     Muistiolio2 olio2;
21 }
22 int main() {
23     esimerkki();
24     return 0;
25 }
```

Kuva 9: RAII esimerkkiohjelma.

Kuvan 9 esimerkkiohjelmassa on muutettu kuvan 8 ohjelma RAII-toimintamallin mukaiseksi. Kuvan 9 ohjelmassa resurssit vapautetaan olion destruktoria kutsuttaessa. Destruktoria kutsutaan poistuttaessa `esimerkki()`-funktiosta tai poikkeustapauksessa.

5 VERTAILU

Tässä tutkielmassa tarkastelun kohteena olleet ohjelmointikielet, Java ja C++, ovat syntaksiltaan hyvin samanlaisia. Java onkin saanut vaikutteita C++-ohjelmointikielestä, mikä on huomattavissa. Kielet eroavat toisistaan kuitenkin esimerkiksi osoittimien, perinnän ja muistinhallinnan osalta. [17] Tässä tutkielmassa esiteltiin molempien kielten tunnetuimpia muistinhallintamenetelmiä. Javan kanssa ohjelmia kirjoittaessa siirretään vastuu muistin vapauttamisesta automaattiselle roskienkerääjälle, kun taas C++-kieltä käytettäessä voidaan turvautua konservatiiviseen kerääjään, älykkäisiin osoittimiin tai manuaaliseen vapauttamiseen. Ohjelmistojen suorituskykyvaatimukset ovat kuitenkin harvoin niin tiukat, että muistin manuaalinen varaus ja vapautus olisi välttämätöntä.

Javan tarjoama automaattinen roskienkeräys on pitkälle kehitetty ja mahdollisesti turvallisempi vaihtoehto C++-kielen menetelmiin verrattuna. Java onkin yleisesti tunnettu turvallisuudestaan, kun taas C++-kielen suunnittelussa on panostettu tehokkuuteen. [14] Muistinhallintaan liittyvien virhetilanteiden todennäköisyys on kuitenkin C++-kielellä työskenneltäessä mahdollisesti suurempi. Javan roskienkeräysmekanismi hoitaa muistinsiivouksen samalla ehkäisten muistinkäyttöön liittyvien virhetilanteiden muodostumista [14]. Javan roskienkerääjän uusin versio *Z* suoriutuu suorituskykytesteissä hyvin roskienkerääjän aiheuttamien pysähdysaikojen ollessa aikaisempaa lyhyempiä [19].

C++-kielen tarjoamat älykkäät osoittimet ja konservatiivinen roskienkerääjä ovat pienentäneet ohjelman kehittäjän vastuuta muistin vapauttamisesta. Älykkäiden osoittimien käyttäminen vaatii kuitenkin kehittäjältä vähintään jonkinasteista ymmärrystä osoittimien käytöstä. C- ja C++-kielten muistinvapauttamista helpottamaan kehitetty konservatiivinen roskienkerääjäkirjasto *Boehm* tarjoaa kehittäjälle mahdollisuuden siirtää vastuu vapautuksista kerääjälle sekä tarkistaa ohjelma muistinkäyttöön liittyviltä virhetilanteilta. [25] Javan roskienkeräärät ovat kuitenkin huomattavasti tehokkaampia kuin konservatiivinen kerääjä *Boehm*.

Kielissä käytettävät muistinhallintamenetelmät ovat varsin erilaiset, minkä vuoksi kumpikaan kielistä ei ole yksiselitteisesti toista parempi muistinhallinnan osalta. C++-kieli vaatii mahdollisesti muistinkäytön syvällisempää osaamista, mutta tarjoaa tehokkaan työ-

kalun ohjelmiston sitä vaatiessa. Javan roskienkeräyksen huonona puolena pidetään mahdollisia roskienkeruun aiheuttamia pysähdyksiä. Javan *JNI*-rajapinnan (engl. *Java Native Interface*) avulla voidaan kuitenkin tehdä varauksia siten, ettei roskienkerääjällä ole kontrollia dynaamisen muistin käyttöön [28]. Tässä tutkielmassa keskityttiin Javan automaattiseen muistinhallintaan, joten *JNI*-kutsuja ei käsitelty tarkemmin. Javan muistinhallinnan suurena etuna voidaan pitää turvallisuutta.

6 YHTEENVETO

Tutkielmassa tarkasteltiin ohjelmistojen muistinhallintaa, sekä vertailtiin Java- ja C++-ohjelmointikielien muistinhallinnan tekniikoita. Tutkielman alussa tarkasteltiin yleisemmin muistinhallinnan merkitystä ohjelmoinnissa ja muistia ohjelmistojen resurssina. Vertailussa olleet ohjelmointikielet osoittautuivat muistinhallinnaltaan varsin erilaisiksi. C++-ohjelmointikielen osalta tutkielmassa keskityttiin selvittämään muistin manuaalisen hallinnan keinoja, kun taas Java-ohjelmointikielen osalta keskityttiin automaattisen roskienkeräysmekanismin toimintaan.

Ohjelmiston muistinhallinnan menetelmät voidaan karkeasti jakaa automaattiseen ja manuaaliseen lähestymistapaan. Tutkielmassa löydettiin molemmista menetelmistä hyötyjä ja haittoja. Menetelmistä kumpikaan ei siis yksiselitteisesti ole toista parempi, joten ohjelmistoa kehitettäessä on tärkeää tunnistaa mahdolliset muistinkäsittelyä koskevat vaatimukset sopivan lähestymistavan löytämiseksi. Ohjelmistot voivat myös asettaa rajoituksia ohjelmointikielen valinnalle. Käytettävä ohjelmointikieli kuitenkin määrittää käytettävissä olevat menetelmät muistin hallitsemiseksi.

Tutkielmassa käsiteltiin myös yleisimmät muistinkäsittelyyn liittyvät virhetilanteet. Virhetilanteet ovat haitallisia ohjelman suoritukselle, ja muistinhallinnalla pyritään välttämään virhetilanteiden muodostumista. Manuaalisen muistinhallinnan tapauksessa riski todettiin suuremmaksi muistinhallinnan vastuun ollessa kehittäjällä, mikä mahdollistaa huolimattomuusvirheet muistinkäsittelyssä. Roskienkerääjää hyödyntävä Java-ohjelmointikieli välttää ongelmatilanteet automaatioon nojaten. Roskienkeruussa käytettävä merkitse ja tiivistä -algoritmi osaa jopa tiivistää muistialueen siivouksen jälkeen. Näin roskienkerääjä voi samalla myös välttää muistin pirstoutumista. Molemmat ohjelmointikielet tarjoavat myös työkaluja muistinkäytön mittaamiseen ja tarkasteluun. Seurantatyökaluilla voidaan pyrkiä optimoimaan ohjelman vastausaikoja ja tutkia suorituskykyongelmia tarkemmin.

Automaattiset muistinsiivoustekniikat ovat kehittyneet merkittävästi. Javan kehityksessä roskienkerääjän heikkouksia on pystytty parantamaan huomattavasti. Java-ohjelmointikielen roskienkerääjä kehittyy edelleen, ja viimeisimpänä uudistuksena nähtiin *Z*-roskienkerääjä. C++-ohjelmointikielien muistinhallinta ei ole kokenut samanlaista kehitystä

viime vuosina, ja kielessä hyödynnetään usein perinteisempiä muistinkäsittelytekniikoita. C++-ohjelmointikielen kanssa on kuitenkin nykyään mahdollista hyödyntää konservatiivista roskienkerääjää. Toisinaan kuitenkin ohjelmiston vaatimukset edellyttävät tarkempaa kontrollia muistin vapautukseen, jolloin manuaalisesta muistinkäsittelystä on hyötyä. Automaattisen siivousoperaation ajankohta voi kuitenkin olla arvaamaton.

Tutkielman aihealue osoittautui hyvin laajaksi, ja muistinhallinnan tekniikoita käsiteltiin vain teoreettisesti. Kirjallisuuskatsaus muistinhallintatekniikoihin palvelee kuitenkin mahdollisen lisätutkimuksen suunnittelua. Tutkimusta olisi mahdollista syventää esimerkiksi kokeellisella tutkimuksella. Laajempaa tutkimusta olisi mielenkiintoista tehdä tutkielmassa esitellyn Java-ohjelmointikielen automaattisesta muistinsiivouksesta. Roskienkerääjän kehitys on ollut jatkuvaa, mikä myös tarjoaa mahdollisesti lisää tutkimuksen aiheita tulevaisuudessa.

LÄHTEET

[1] Mikä on keskusmuisti. Web-opas. Saatavilla: <http://www.webopas.net/keskusmuisti.html> (Viitattu 1.5.2019)

[2] Salnikov-Tarnovski, Nikita – Smirnov, Gleb. Java Garbage Collection. 2015. Saatavilla: <http://downloads.plumbr.io/Plumbr%20Handbook%20Java%20Garbage%20Collection.pdf> (Viitattu 1.5.2019)

[3] Kaijanaho, Antti-Juhani. Muistinhallinta ohjelmointikielissä. 2007. Saatavilla: <http://users.jyu.fi/~antkaij/opetus/okp/2007/luennot/08/muisti.pdf> (Viitattu 1.5.2019)

[4] Tuovinen, Antti-Pekka. Ohjelmistojen suorituskyky. Luento 1. 2014. Saatavilla:

<https://www.cs.helsinki.fi/u/aptuovin/spe/k14/luento_1.pdf> (Viitattu 10.11.2018)

[5] Varga, Zoltán. Muistin Siivous. 2006. Saatavilla: <https://tampub.uta.fi/bitstream/handle/10024/93432/gradu01065.pdf?sequence=1> (Viitattu 12.11.2018)

[6] Muistin varaukset. 2019. Saatavilla: <https://wiki.metropolia.fi/display/koneautomaatio/Muistin+varaukset> (Viitattu 1.5.2019)

[7] Ohjelmien turvallisuus. 2011. Saatavilla: <https://www.cs.helsinki.fi/u/karvi/perusteet-luku3-bea_11.pdf> (Viitattu 1.5.2019)

[8] Sanguthevar, Rajasekaran – Lance, Fiondella – Mohamed, Ahmed - Reda A. Ammar. Multicore Computing: Algorithms, Architectures, and Applications. 2013 (Viitattu 17.1.2019)

[9] Juustila, Antti – Kettunen, Harri – Kilpi, Teemu – Räisänen, Toni – Vesanen, Ari. C++-ohjelmointikurssin oheismateriaali. Saatavilla: <http://jultika.oulu.fi/files/isbn9514279425.pdf> (Viitattu 17.1.2019)

[10] Gilmore, Stephen. Advances in Programming Languages: Memory management. 2007. Edinburghin yliopisto. Saatavilla: <http://homepages.inf.ed.ac.uk/stg/teaching/apl/handouts/memory.pdf> (Viitattu 10.11.2018)

[11] C++ Programming. Wikibooks.org. 2012. Saatavilla: <https://upload.wikimedia.org/wikipedia/commons/e/e9/CPlusPlusProgramming.pdf> (Viitattu 10.11.2018)

[12] Järvinen, Hannu-Matti – Mikkonen, Tommi. Sulautettu ohjelmointi. 2010. Saatavilla: <https://docplayer.fi/4728518-Sulautettu-ohjelmointi.html> (Viitattu 10.11.2018)

[13] Memory Fraqmentation. Technology Guides. Saatavilla: <https://developer.brewmp.com/resources/tech-guides/memory-and-heap-technology-guide/high-level-architecture/memory-fragmentation> (Viitattu 10.11.2018)

[14] Niemeyer, Patrick – Knudsen, Jonathan. Learning Java. 2005. Saatavilla: <http://it.guldstadsgymnasiet.se/java/Learning%20Java,%204th%20Edition.pdf> (Viitattu 1.5.2019)

[15] Java Memory Management. Saatavilla: <https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/> (Viitattu 10.11.2018)

[16] Goetz, Brian. Java theory and practice: A brief history of garbage collection. 2003. Saatavilla: <https://www.ibm.com/developerworks/library/j-jtp10283/j-jtp10283-pdf.pdf> (Viitattu 10.11.2018)

[17] Kopra, Otto. Java. Ohjelmistotekniikan seminaari. 1996. Saatavilla: <http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/java/#Chapter2> (Viitattu 1.5.2019)

[18] Java Garbage Collection Basics. Oracle. Saatavilla:

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html> (Viitattu 13.11.2018)

[19] JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental). Saatavilla: <https://openjdk.java.net/jeps/333> (Viitattu 1.5.2019)

[20] Analyzing Java Memory. Saatavilla: <https://www.dynatrace.com/resources/ebooks/javabook/analyzing-java-memory/> (Viitattu 10.11.2018)

[21] Using JConsole. Java Documentation. Saatavilla: <https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html> (Viitattu 10.11.2018)

[22] The jstat Utility. Java Documentation. Saatavilla: <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr017.html> (Viitattu. 10.11.2018)

[23] Java VisualVM. Java Documentation. Saatavilla: <https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/> (Viitattu 27.11.2018)

[24] Chan, Rosalie. The 10 most popular programming languages, according to the 'Facebook for programmers'. 2018. Saatavilla: <https://www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10?r=US&IR=T> (Viitattu 27.11.2018)

[25] A garbage collector for C and C++. Saatavilla: <https://www.hboehm.info/gc/> (Viitattu 27.11.2018)

[26] Suntioinen, Ari. OHJ-1150 Ohjelmointi II. 2013.

[27] New and delete Operators. Microsoft docs. Saatavilla: <https://docs.microsoft.com/en-us/cpp/cpp/new-and-delete-operators?view=vs-2017> (Viitattu 8.1.2019)

[28] Sorensen, Jeffrey – Bikel, Daniel. Improved JNI memory management using allocations from the java heap. 2007. Saatavilla: <https://www.usenix.org/legacy/events/usenix07/posters/sorensen.pdf> (Viitattu 1.5.2019)