Tampereen yliopisto

Rasmus Salla

C++:N MOVE-SEMANTIIKAN ESITTELY

C++11:ssa julkaistu ominaisuus

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Syyskuu 2019

TIIVISTELMÄ

Rasmus Salla: C++:n move-semantiikan esittely
Kandidaatintyö
Tampereen yliopisto
Ohjelmistotekniikka
Syyskuu 2019

Kandidaatintyön päätavoitteena on esitellä lukijalle, mitä C++:n aihe move-semantiikka pitää sisällään ja ennen kaikkea, miten move-semantiikkaa voi hyödyntää omassa C++-koodissa. Esitellyssä on hyödynnetty runsaasti koodiesimerkkejä, ja vaikka esittely kulkee välillä erittäin syvällä move-semantiikan teoriassa, on teksti kirjoitettu niin, että C++:n perusteet osaava henkilö voi sen ymmärtää.

Kandidaatintyö perustuu suurimmilta osin uusimpaan valmiiseen C++17-standardiin. Laaja osa työn sisällöstä pätee kuitenkin myös C++11:een lukuunottamatta muutamia C++17:n ominaisuuksia, jotka esitellään työssä erikseen.

Move-semantiikka on vuonna 2011 C++11-standardin mukana julkaistu ominaisuus. Move-semantiikalla tarkoitetaan käsitteenä kaikkea sitä, minkä avulla kääntäjä pystyy korvaamaan kalliit kopio-operaatiot halvemmilla siirto-operaatioilla. Siirtäminen taas tarkoittaa olion tilan siirtämistä uuden samanlaisen olion haltuun. Usein tämä tehdään varastamalla olion resurssit, eli ottamalla resurssiin osoittava osoitin uuden olion haltuun. Siirtämisellä siis vältetään olion turhaa kopioimista, vaikkakin sen jälkeen siirretty olio jää usein täsmentämättömään tilaan, koska olion data siirretään sen omistuksesta pois. Tämän vuoksi siirtäminen sopii erityisen hyvin väliaikaisille olioille, joiden elinikä on lähes poikkeuksetta lyhyt.

Tekstin luettua lukija osaa erotella rvaluet ja lvaluet sekä rvalue-viitteet, viitteet ja forwardoivat viitteet toisistaan. Lukija myös ymmärtää, mitä olioiden siirtämisellä tarkoitetaan, ja miten rvalue- ja lvalue-oliota siirretään. Lisäksi lukija tietää, miten move-rakentaja ja move-sijoitusoperaattori määritetään luokalle.

Avainsanat: move-semantiikka, C++11, siirtäminen, olio, rvalue, lvalue, rvalue-viite, move-rakentaja, move-sijoitusoperaattori, forwardoiva viite

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. RVALUET JA LVALUET 2
2.1 rvaluet ja lvaluet C++:ssa 2
2.2 C+11:n arvokategoriat 4
2.3 prvalue ja väliaikainen materialisaatio – C++17 6
2.4 Väliaikaisten olioiden ja lvalue-olioiden elinikä 6
3. RVALUE-VIITTEET 8
4. RVALUE-OLIOIDEN SIIRTÄMINEN 10
4.1 Esimerkkiluokka String 10
4.1.1 Move-rakentaja 11
4.1.2 Move-sijoitusoperaattori 12
4.1.3 *noexcept*-määre move-operaatioissa 13
4.2 Kääntäjän määrittelemät move-operaatiot 15
4.3 Rule of five 16
4.4 Taattu kopion välttäminen – C++17 16
5. FORWARDOIVAT VIITTEET 18
5.1 Viiteromahdus 18
5.2 Funktio templaten parametri *T&&* 19
5.3 auto&& 20
6. LVALUE-OLIOIDEN SIIRTÄMINEN 21
6.1 std::move 21
6.1.1 Määritelmä ja toiminta 22
6.1.2 Kutsuminen vakioargumentilla 23
6.2 std::forward 24
7. YHTEENVETO 25
LÄHTEET 26

LYHENTEET JA MERKINNÄT

CPL	Combined Programming Language, ohjelmointikieli
glvalue	generalized lvalue, C++:n arvokategoria
NRVO	named return value optimization, C++:n paluuarvo-optimointi
prvalue	pure rvalue, C++:n arvokategoria
RVO	return value optimization, C++:n paluuarvo-optimointi
STL	Standard Template Library, osa C++:n standardikirjastoa
xvalue	eXpiring value, C++:n arvokategoria

1. JOHDANTO

Elokuun 12. päivä vuonna 2011 julkaistiin C++11-standardi [1]. Standardi tunnettiin ennen sen virallista ratifointia nimellä C++0x [2]. Tämä standardi toi mukanaan lukuisia uusia ominaisuuksia C++-kieleen ja yksi niistä oli move-semantiikka [3].

Käsite move-semantiikka tarkoittaa kaikkea sitä, minkä ansiosta kääntäjä kykenee korvaamaan kalliit kopio-operaatiot halvemmilla siirtämisoperaatioilla [4, s. 157], mutta ennen kaikkea siinä on kysymys olioiden siirtämisestä. Olion siirtämisellä tarkoitetaan sitä, että olion data siirretään toisen samanlaisen olion haltuun. Esimerkiksi vektoria siirrettäessä siirretään sen alkiot toiselle vektorille. C++:n sisäänrakennetut tyypit (esimerkiksi *int, char, bool)* siirrettään aina kopioimalla (assembly-käsky *mov*), minkä vuoksi siirtäminen on mielekästä vain itse määritetyille ja kirjaston tyypeille.

Kandidaatintyössä on tarkoituksena esitellä aihe move-semantiikka perusteellisesti, mikä tehdään tutkimalla aiheen teoriaa syvällä tasolla ja hyödyntämällä runsaasti koodiesimerkkejä. Koodiesimerkkien pohjalta move-semantiikkaa voi soveltaa omassa C++-koodissa.

Teksti olettaa, että lukija tuntee C++:n perusteet ja syntaksin hyvin. Erityisesti funktioiden, luokkien ja muuttujien perusrakenne täytyy hallita. Forwardoivien viitteiden ja *std::moven* toiminnan ymmärtämiseksi templatet sekä templatien tyyppideduktio olisi lukijan myös hyvä osata.

Kandidaatintyö perustuu suurimmilta osin kirjoitushetkellä uusimpaan valmiiseen C++-standardiin C++17. Jos kuitenkin C++11-standardiin viitataan C++17-standardin sijaan, niin se ilmenee lauseen tai luvun kontekstista. Kandidaatintyö sivuaa tarkoituksella forwardoimista, vaikka se on hyvin sidoksissa move-semantiikkaan. Sidosteisuuden myötä tämän työn pohjalta forwardoimiseen helppo jatkaa.

Luvussa 2 tutkitaan C++:n arvokategorioita ja niiden välisiä eroja sekä eri tyyppisten olioiden elinikiä. Tämän jälkeen esitellään luvussa 3 rvalue–viitteet ja miksi tämä uusi viitetyyppi oli tarpeen. Sitten luvussa 4 selvitetään, miten rvalueita siirrettään luokan move-operaatioiden kautta, ja miten *noexcept*-määre ja C++17 vaikuttavat move-operaatioihin. Luvussa 5 tehdään lyhyt katsaus viiteromahdukseen ja forwardoiviin viitteisiin, jonka jälkeen Luvussa 6 tutkitaan lvalue-olioiden siirtämistä. Lopuksi luvussa 7 tehdään yhteenveto.

2. RVALUET JA LVALUET

Historiallisesti termejä lvalue ja rvalue käytti ensimmäisen kerran Christopher Strachey ohjelmointikielessä CPL (engl. Combined Programming Language) vuonna 1963 [5]. CPL:ssä lvaluella tarkoitettiin aina sijoitusoperaatiossa vasemman puolen muuttujaa, johon sijoitetaan oikean puolen rvaluen arvo. lvalue ja rvalue tulivat siis nimistä left-hand value ja right-hand value. [6, s. 136] Ohjelmointikieli C seurasi näitä samoja käytäntöjä, mutta jätti pois termin ”rvalue”, jonka sijasta käytettiin termiä ”ei-lvalue”. C++:n kehittäjä Bjarne Stroustrup toimi samoin C++:n varhaisissa versioissa. [5]

C++:ssa lauseke[1] on jono operaatioita ja operandeja, minkä lopputuloksena evaluoituu jokin arvo [7, s. 93] [8, s. 83]. Esimerkiksi muuttujan nimi, literaali, yhteenlasku, sijoitus ja funktiokutsu ovat lausekkeita, kun taas deklaraatio, *if*- ja *for*-lauseet eivät ole (nämä kaikki voivat kyllä sisältää lausekkeita). Jokaisella lausekkeella on sekä tyyppi että arvokategoria (lvalue tai rvalue) [9], joka luokittelee lausekkeet niistä evaluoituneiden arvojen perusteella.

2.1 rvaluet ja lvaluet C++:ssa

C++:ssa lvalue kuvaa lauseketta, jonka osoite voidaan ottaa, ja rvalue taas kuvaa lauseketta, jonka osoitetta ei voida ottaa [4, s. 2]. Koska lausekkeiden tuloksena voi syntyä uusi olio (lausekkeen arvo), tai koska olion nimen käyttäminen muodostaa lausekkeen, voidaan ajatella, että myös olioilla on arvokategoria. lvalue-oliot ovat siis nimettyjä olioita, kun taas rvalue-oliot ovat usein nimettömiä väliaikaisia olioita [10, s. 10] (paitsi ei C++17:n jälkeen, josta myöhemmin luvussa 2.3).

Ohjelmassa 1 näkyy esimerkkejä rvalueista ja lvalueista. Kaikki punaisella väritetyt lausekkeet ovat lvalueita ja kaikki sinisellä rvalueita. Ohjelmassa 1 on runsaasti deklaraatioita, joilla ei itsessään ole arvokategoriaa, mutta niistä syntyvistä nimistä voidaan muodostaa hyvin yksinkertaisia lausekkeita, joilla on arvokategoria. Ohjelmassa 1 siis korostuu, että arvokategorioilla luokitellaan vain lausekkeita (ja joskus myös olioita).

[1] engl. expression

```
std::string& returnStringRef();     // Pelkillä deklaraatioilla ei ole arvo-
std::string returnString();         // kategoriaa vain lausekkeilla on.
void takesInt(int number);          // Näistä nimistä voidaan luoda yksinker-
struct Example { int e; };          // taisia lausekkeita (Esim. alla).

returnString;                       // Nimistä muodostuvia lausekkeita,
takesInt;                           // jotka ovat aina lvalueita.
void takesInt(int number)           // Huom. Funktioista käytetään vain
{ number; }                         // nimiä, eikä niitä kutsuta.

returnString();                     // Funtiokutsu on usein eri lauseke kuin
returnStringRef();                  // pelkkä funktion nimi, koska funktio
                                    // palauttaa yleensä väliaikaisen rvalue-
                                    // olion (paitsi C++17:ssa).

int a = 5;                          // Lausekkeilla, joilla initalisoidaan
int b = a;                          // deklaroituja olioita, on arvokategoria.

bool foo1 = true;
int* aPtr = nullptr;
Example example1 = Example();

std::string string3 = "Hello!";     // Merkkijonoliteraalit ovat lvalueita

int c = a + b;                      // Näiden viiden lausekkeen osissa on
bool foo2 = foo1 && true;           // lvalueita, vaikka lausekkeet ovat
bool comp = b > a;                  // kokonaisuudessaan rvalueita.
int* cPtr = &a;
Example* example1Ptr = &example1;

a = 6;                              // Sisäänrakennetut sijoitusoperaatiot
example1.e = 5;                     // ovat lvalue-lausekkeita.

int f = (a += b);                   // Sijoitusoperaatioita, joissa
b = *cPtr;                          // molemmat operandit lvalueita.
b = example1.e;
b = example1Ptr->e;
```

Ohjelma 1. *Esimerkkejä rvalueista ja lvalueista C++:ssa*

Ohjelmasta 1 nähdään, että yleisiä rvalueita ovat literaalit, sisäänrakennetut aritmeettiset ja loogiset operaatiot sekä funktiokutsut, joissa palautetaan ei-viite. lvalueihin kuuluu taas esimerkiksi muuttujien, parametrien ja funktioiden nimet, sekä sisäänrakennetut sijoitusoperaatiot ja sisältöoperaattori. On hyvä huomata, että merkkijonoliteraalit ovat muista literaaleista poiketen lvalueita. [9]

Operaatio- tai funktiokutsujen arvokategoriaa päätellessä voi ajatella joko syntyvän olion arvokategoriaa tai vain lausekkeen arvokategoriaa, koska lausekkeen arvokategoria on sama kuin lausekkeesta syntyvän olion arvokategoria. Esimerkiksi erisuuruusvertailussa *b > a* operaattorin tulos on rvalue-olio, koska sisäänrakennetut vertailuoperaattorit palauttavat ei-viite totuusarvon [7, s. 133]. Toisaalta sisäänrakennettu erisuuruus-operaatio on aina rvalue-lauseke (samasta syystä). Päinvastaisesti sisäänrakennetut

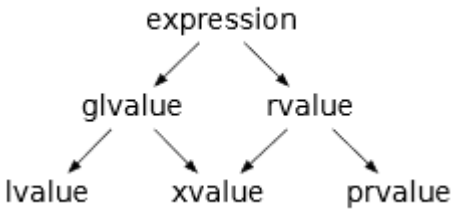
sijoitusoperaatiot palauttavat aina viitteen sijoitettavaan, vasemman puolen olioon, jolloin tulos on aina lvalue-olio [7, s. 138]. Taas toisin ajatellen sisäänrakennettu sijoitusoperaatio on aina lvalue-lauseke, koska se on funktio, joka palauttaa viitteen.

2.2 C+11:n arvokategoriat

Ennen move-semantiikan lisäämistä C++11:een suurin osa C++-komitean Core Working Group -osasta oli sitä mieltä, että rvalue- ja lvalue-terminologiaan oli tultava muutoksia. Komitean osan mukaan nämä muutokset olivat välttämättömiä, jotta saataisiin ratkottua tunnettuja ongelmia ja epäjohdonmukaisuuksia, joita siis move-semantiikan tuominen kieleen olisi aiheuttanut. [5, s. 1]

C++11:ssä lausekkeilla on kaksi toisistaan riippumatonta ominaisuutta: identiteetti eli esimerkiksi muistiosoite tai nimi sekä se, että onko lausekkeesta syntyvä tai sitä kuvaava olio turvallista siirtää [5, s. 2]. Esimerkiksi funktion palauttamalla väliaikaisella olioilla ei ole identiteettiä – nimeä – ja se voidaan turvallisesti siirtää, koska tähän samaan olioon ei tulla viittaamaan tulevaisuudessa. Nimetyllä muuttujalla taas päinvastoin on identiteetti ja sitä ei voida siirtää, koska muuttujan sisältämää tietoa saatetaan tarvita muualla ohjelmassa.

Edellä mainituista ominaisuuksista saadaan koottua lausekkeille viisi eri arvokategoriaa, jotka ovat esillä kuvassa 1. Kuvassa 1 nähtävistä arvokategorioista lvalue, xvalue (eXpiring value [8, s. 72]) ja prvalue (pure rvalue) ovat perustavanlaatuisia ja näitä yhdistämällä muodostetaan kaksi viimeistä: glvalue (generalized lvalue) ja rvalue [5, s. 2–4]:



Kuva 1. *C++11:n kanssa esitetyt arvokategoriat [8, s. 74]*

Lausekkeet jakautuvat mainittuihin kolmeen perustavanlaatuiseen arvokategoriaan seuraavasti [9]:

- lvalue-lausekkeilla on identiteetti, ja niistä syntyvää tai niiden kuvaamaa oliota ei voida siirtää.
- xvalue-lausekkeilla on identiteetti, ja niistä syntyvä tai niiden kuvaama olio voidaan siirtää.

- prvalue-lausekkeilla ei ole identiteettiä. ja niistä syntyvä tai niiden kuvaama olio voidaan siirtää.

Siis lausekkeet, joilla on identiteetti, ovat glvalueita ja lausekkeet, joiden kuvaamia tai synnyttämiä olioita voidaan siirtää, ovat rvalueita.

C++11:n jälkeen termit lvalue ja rvalue eivät seuraa enää niiden historiallista määritelmää, että lvaluet ovat sijoitusoperaation vasemmalla puolella ja rvaluet oikealla, vaikkakin luvussa 2.1 esitetyt määritelmät pätevät yhä: lvalue on yhä lauseke, jonka osoite voidaan ottaa, koska sillä on aina identiteetti, ja rvaluesta ei saa edelleenkään otettua osoitetta.

Jos tutkitaan ohjelman 1 esimerkkiä uudelleen, niin jokainen punaisella merkattu on edelleen lvalue, mutta kaikki sinisellä merkatut ovat prvalueita. Ohjelmassa 1 ei ole yhtäkään xvalueta. C++11:ssa yleisin xvalue-olio on lvalue, joka on muutettu rvalueksi. Tälläisen tyyppimuunnoksen suorittaa esimerkiksi *std::move* [8, s. 475], joka käydään tarkemmin läpi luvussa 6.

C++11-standardin aikaan ei tunnettu mekanismia, joka suorittaisi edellä mainitun tyyppimuunnoksen vastakkaiseen suuntaan eli rvaluesta lvalueksi. Tiettyjen lausekkeiden tuloksena pystyi kuitenkin syntymään xvalue. Yksi mielenkiintoinen esimerkki on jäsenvalintaoperaation tulos, joka määriteltiin C++11-standardissa prvalueksi aina, jos olio, johon jäsenvalinta kohdistetaan on prvalue [8, s. 96]. Kuitenkin hieman yli vuosi C++11-standardin julkaisun jälkeen tämä määritelmä muutettiin muotoon, että jäsenvalintaoperaatio tuottaa aina xvaluen, jos olio on rvalue [11]. Ohjelmaan 2 on kerätty esimerkkejä xvalueista. lvaluet on merkattu punaisella, xvaluet vihreällä, ja prvaluet sinisellä.

```
struct Example
{
  int a;
  int array[2];
};
Example returnsExample();

int b = returnsExample().a              // Lausekkeet ovat kokonaisuudessaan
int c = returnsExample().array[0];      // xvalueita.

Example foo;
Example foo2 = std::move(foo);
```

Ohjelma 2. *Esimerkkejä xvalueista*

Edellä mainitut muutokset huomioiden C++11:ssa ohjelman 2 *returnsExample*-funktion palauttama olio olisi arvokategorialtaan prvalue, kuten kaikki ei–viite paluuarvot, ja

molempien jäsenvalintaoperaatioiden ja taulukon indeksoinnin tulos olisi xvalue. [9] Ohjelman 2 lausekkeet käyttäytyvät C++17:ssä melko erilailla väliaikainen materialisaatio[2] -mekanismin myötä.

Mainittakoon, että C++11:n arvokategoriat ovat yhä käytössä C++17-standardissa lukuunottamatta seuraavan luvun mainitsemaa muutosta prvaluen määritelmään. Lisäksi vaikka arvokategorioille on näinkin tarkat määritelmät, monessa tapauksessa riittää termien rvalue ja lvalue käyttö. Jokapäiväisessä C++-keskustelussa prvalue, xvalue ja glvalue termejä on harvoin tarpeen käyttää.

2.3 prvalue ja väliaikainen materialisaatio – C++17

C++17:ssa prvaluen määritelmää muuttui, kun taattu kopion välttäminen[3] tuotiin kieleen [12]. C++11:ssä prvalue määriteltiin muun muassa väliaikaisena oliona [8, s. 74]. C++17:ssä prvalue tarkoittaa lauseketta, joka initialisoi olion [7, s. 83]. Väliaikainen olio ei siis enää ole prvalue, mutta prvaluesta voidaan tarpeen vaatiessa materialisoida väliaikainen xvalue-olio. Tätä mekanismia kutsutaan väliaikaiseksi materialisaatioksi [7, s. 88].

Väliaikainen materialisaatio tapahtuu silloin, kun jokin glvaluen vaatima operaatio tehdään prvaluella [7, s. 84]. Tälläisiä operaatiota ovat esimerksi viitteen sitominen, jäsenvalintaoperaatio ja taulukon indeksointi [7, s. 230, 112, 109]. C++17:ssä ohjelmassa 2 kummassakin *returnsExample*-kutsussa funktion palauttama prvalue materialisoidaan xvalueksi, koska prvalueen kohdistuu jäsenvalintaoperaatio, joka siis vaatii glvaluen. Kun toisessa kohdassa materialisoitunutta xvalue-taulukkoa vielä indeksoidaan, niin materialisaatioita ei tarvitse tehdä uudestaan, koska xvalue on glvalue.

Tällä prvaluen määritelmän muutoksella on merkittäviä vaikutuksia prvalueiden siirtämiseen. Nämä vaikutukset käydään tarkemmin läpi luvussa 4.4, mutta tiivistetysti prvalue-lausekkeista muodostuvia olioita ei enää siirrettä, koska prvalue-lausekkeista muodostuu harvoin oliota ollenkaan.

2.4 Väliaikaisten olioiden ja lvalue-olioiden elinikä

Yksi lvalue-olioiden ja väliaikaisten olioiden merkittävimmistä eroista on niiden elinikä. Ilman *static-, thread_local-,* tai *extern*-määritelmää lvalue-oliot jonkin lohkon sisällä

[2] engl. temporary materialization
[3] engl. guaranteed copy elision

säilyvät kunnes lohkosta poistutaan. Dynaamisesti varattu muisti ja jäsenmuuttujat ovat poikkeuksia tähän sääntöön edellä mainittujen määritelmien lisäksi. [7, s. 70–71, 74] Tutkitaan erilaisten lvalue-olioiden elinikiä yksinkertaisen esimerkin avulla:

```
void foo(int k)
{
  if (k > 5) {
    char letter = 'h';
  }
  static int i = 10;
}
```

Esimerkissä lvalue *k* tuhotaan funktiolohkosta poistuessa, ja merkki *letter if*-lohkosta poistuessa. Kokonaisluku *i* elää ohjelman loppuun asti *static*-määritelmän vuoksi.

C++:ssa kaikki väliaikaiset oliot tuhotaan täyden lausekkeen lopuksi ellei olioon sidota viitettä. Taulukoiden initialisointiin ja kopiointiin liittyy myös kaksi erikoistapausta, joissa väliaikaista oliota ei tuhota täyden lausekkeen lopussa, mutta ne sivutaan. [7, s. 284] Täyden lausekkeen päättää yleensä puolipiste, jolloin siis tavallisesti lausekkeesta muodostuneet väliaikaiset oliot tuhotaan. Tutkitaan tätä esimerkin avulla ohjelmassa 3.

```
struct Example { int a; };
Example returnsExample();

int i = returnsExample().a;

Example eObj = Example();
const Example& eRef = Example();
```

Ohjelma 3. *Esimerkkejä väliaikaisista olioista*

Ohjelmassa 3 näkyvä tummennettu väliaikainen olio on ainoa, joka tuhotaan puolipisteen kohdalla, eli kokonaisluvun *i* initialisoinnin jälkeen. Väliaikainen olio materialisoidaan *returnsExample*-kutsun palauttamasta prvaluesta, koska prvalueen kohdistetaan jäsenvalintaoperaatio. Rakentajakutsu *Example(),* jolla olio *eObj* initialisoidaan, ei synnytä C++17:ssa väliaikaista oliota. Tutkitaan tätä tarkemmin seuraavassa kappaleessa. Viimeisellä rivilla materialisoitunut väliaikainen olio sidotaan viitteeseen, jolloin sen elinikä on sama kuin viitteen. Syy siihen, miksi viitteen täytyy olla vakio, selviää luvussa 3.

Koska edellä mainittu rakentajakutsu on prvalue-lauseke, niin luvun 2.3 perusteella väliaikaista oliota ei synny. Ohjelman 3 rakentajakutsu on siis vain lauseke, jolla olio *eObj* initialisoidaan suoraan. Ennen C++17:ää kyseinen rakentajakutsu olisi voinut standardin mukaan tuottaa väliaikaisen prvalue-olion [8, s. 244]. Käytännössä kuitenkin lähes jokainen implementaatio optimoi väliaikaisen olion pois (paluuarvo-optimointi). Suora-initialisointia ja paluuarvo-optimointia tutkitaan tarkemmin luvussa 4.4.

3. RVALUE-VIITTEET

Samassa vuoden 2002 dokumentissa, jossa ehdotettiin move-semantiikan lisäämistä C++:aan, ehdotettiin myös rvalue-vitteen lisäämistä kieleen. Uutta viitetyyppiä ehdotettiin, koska ennen C++11:tä ainoastaan vakio–lvalue-viite (vakio-viite) kykeni sitoutumaan rvalue-olioon, mikä teki rvalue-olioiden erottamisen vakio-oliosta hankalaa. [13]

Edellä mainitusta syystä ei ollut mahdollista luoda rvalueita siirtävää rakentajaa, koska vakio-viite hyväksyisi argumentteina myös lvaluet, joita ei haluta siirtää. Ainoa tapa oli valistaa kopiorakentaja pelkästään siirtämistä varten, jolloin luokkaa ei voinut enää kopioida. Tämän toteutti C++11-standardin vanhentama *std::auto_ptr*, jonka *std::unique_ptr* korvasi [14].

Dokumentti ehdotti rvalue-vitteen syntaksiksi *T&&* [13]. Sama syntaksi otettiin käyttöön C++11-standardissa [3]. Taulukossa 1 on esitetty nykyisen C++17-standardin kaikki viitteet, ja mihin kukin viite kykenee sitoutumaan [7, s. 192–193, 87]. *T* kuvaa taulukossa 1 jotain tiettyä tyyppiä.

Taulukko 1. *Arvokategoriat, johon mikäkin viitetyyppi voi sitoutua.*

Viitetyyppi	*lvalue*	***const** lvalue*	*rvalue*	***const** rvalue*
T&	x			
const T&	x	x	x	x
T&&			x	
const T&&			x	x

rvalue-viitteen avulla vakio-olio kyetään erottamaan rvalue-oliosta, mutta viitetyypi tuo mukanaan uuden ongelman funktiokutsun kohdetta pääteltäessä [13]. Jos jokin funktio on kuormitettu sekä vakio–lvalue-viitteellä että rvalue-viitteellä, funktiokutsu rvaluella on kaksitulkintainen. Tutkitaan ongelmaa esimerkillä:

```
class B {};

void example(const B& t);  // 1
void example(B&& t);       // 2

B lvalue;
example(lvalue);           // kutsuu 1

example(B());              // 1 vai 2?
```

Esimerkissä ilmenevästä ongelmasta johtuen samainen dokumentti esitti myös, että rvaluet suosivat rvalue-viitteellä kuormitettuja funktioita ja lvaluet taas vastaavasti lvalue-viitteellä [13]. Samainen sääntö päätyi myös lopulta C++11-standardiin [7, s. 325]. Esimerkin kaksitulkintaisessa funktiokutsusssa kutsuttaisiin siis funktiota 2, joka on kuormitettu rvalue-viitteellä.

rvalue-viitteet ja edellä mainittu sääntö mahdollistavat move-semantiikan implementoimisen mille tahansa luokalle. Näiden kahden ominaisuuden ansiosta rvalue- ja lvalue-argumenteilla tehdyt funktiokutsut voidaan erotella toisistaan. Erityisesti luokan rakentajakutsuissa tämä mahdollistaa sen, että lvalue-argumentti kutsut välitetään kopiorakentajalle ja rvalue-argumentti kutsut rvalueita siirtävälle rakentajalle, eli luokan lvalue-oliot kopioidaan, ja rvalue-oliot siirretään.

Huomion arvoista on, että vaikka rvalue-viite sitoutuu rvalueen, on rvalue-viite itsessään ainakin glvalue. Tämä johtuu siitä, että viitteellä on aina identiteetti – muistiosoite ja nimi, joka tekee siitä vähintään glvaluen, mutta se että onko rvalue-viite lvalue vai xvalue rippuu tapauksesta. Esimerkiksi viitemuuttujana rvalue-viite on aina lvalue, kun taas funktion palauttama rvalue-viite on aina xvalue (*std::move)*.

4. RVALUE-OLIOIDEN SIIRTÄMINEN

rvalue-olioiden siirtäminen vaatii C++:ssa vain, että käytetylle luokalle on määritelty siirtoja tekevä rakentaja ja/tai sijoitusoperaattori. Jos ohjelmoija on itse määrittänyt luokan, niin hän myös vastuussa siitä, että siirtäminen tehdään oikein, kuten kopio-operaatioillakin. Kääntäjä huolehtii loppukädessä, että rvalue-argumenteilla tehdyt initialisoinnit ja sijoitukset tehdään kopioimisen sijaan siirtämällä.

Yleinen tapa ”siirtää” olio on varastaa sen resurssit, koska tiedetään, että siirtävän operaation rvalue-viitteesen on sitoutunut rvalue. Tähän samaan rvalueen ei tulla viittaamaan ohjelmassa myöhemmin, jolloin sen resurssit voidaan huoletta riistää. Tässä luvussa käydään läpi tavallinen tapa siirtää olio, joka varaa dynaamista muistia. Tavassa ”varastetaan” (kopiointi ja nollaus) osoitin, joka siis osoittaa olion dynaamisesti varaamaan muistialueeseen.

Jos move-operaatiot on unohdettu määrittää tai kääntäjän määrittelemät operaatiot eivät ole saatavilla, saattavat toivottavat siirto-operaatiot korvautua usein kalliimmilla kopio-operaatioilla. Tämä tapahtuu siksi, että rvaluet voivat sitoutua myös vakio–lvalue-viitteisiin. Tälläinen vakioviite löytyy tunnetusti luokan kopiorakentajasta.

4.1 Esimerkkiluokka String

Koska siirtäminen on helpointa esittää koodiesimerkeillä, käytän tämän luvun esimerkkeihin *String*-luokkaa, joka kuvaa implementaatiota merkkijonolle. Siirtämisen tehokkuus tulee erityisen hyvin esille, kun luokka varaa muistia dynaamisesti, jolloin merkkijono toimii esimerkkiluokkana hyvin. Ohjelmassa 4 on esitelty *String*-luokan header-tiedosto.

```
#include <cstring>
#include <algorithm>

class String
{
public:
    explicit String(const char* data);
    ~String();
    String(const String& other);

private:
    char* _data;
};
```

Ohjelma 4. *String-luokan header-tiedosto*

String-luokalle on tässä vaiheessa deklaroitu vasta rakentaja, purkaja ja kopiorakentaja. Luokalla on lisäksi yksi jäsenmuuttuja, joka on osoitin luokan dynaamisesti varaamaan muistilohkoon, jossa merkkijonon kirjaimet säilytetään. Ohjelmassa 5 on esillä rakentajan, purkajan ja kopiorakentajan implementaatio.

```
#include "String.h"

String::String(const char* data)
{
    std::size_t size = strlen(data) + 1;          // Huom. nollabitti (+1)
    _data = new char[size];
    memcpy(_data, data, size);
}

String::~String()
{
    delete[] _data;
}

String::String(const String& other)
{
    std::size_t size = strlen(other._data) + 1;    // ks. yllä
    _data = new char[size];
    memcpy(_data, other._data, size);
}
```

Ohjelma 5. *String-luokan rakentajan, purkajan ja kopiorakentajan implementaatio*

Ohjelmasta 5 nähdään, että *String*-luokan rakentaja varaa parametrin merkkijonon pituisen muistilohkon (huomioiden nollabitin). Parametrin osoittama *char*-taulukko kopioidaan sen jälkeen tähän muistilohkoon. Purkaja vapauttaa rakentajan varaaman muistilohkon uudelleen käytettäväksi.

Kopiorakentaja ottaa parametrina vakio–lvalue-vitteen kopioitavaan *String*-olioon. Muuten kopiorakentaja toimii aivan kuten rakentajakin paitsi, että kopioitava data ja merkkijonon pituus saadaan toiselta oliolta. Tässä vaiheessa on hyvä huomata, että sekä rakentaminen että kopioiminen ovat dynaamisen muistivarauksen myötä melko raskaita operaatioita.

4.1.1 Move-rakentaja

C++11:sta lähtien luokalle voidaan määrittää move-rakentaja hyödyntämällä rvalue-viitettä [7, s. 254]. Move-rakentajan tehtävä on initialisoida olio argumentin viitteeseen sitoutuneesta rvaluesta. String-luokan move-rakentaja on helppo määritellä, ja sen deklaraatio ja implementaatio ovat esillä koodiesimerkissä.

```
// header
String(String&& other) noexcept;

// implementaatio
String::String(String&& other) noexcept
{
    _data = other._data;
    other._data = nullptr;
}
```

String-luokan move-rakentaja ottaa parametrina rvalue-viitteen siirrettävään String-rvalueen ja kopioi siirrettävän olion osoittimen tälle olioille. Siirrettävän olion osoitin asetetaan nollaksi, jotta sen purkaja ei vapauta osoittimen takana olevaa muistilohkoa. Tässä siis nähdään luvun alussa mainittu ”resurssien varastaminen”. Dynaamisen muistinvarauksen sekä muistinkopioinnin sijaan siis siirretään vain osoitin, mikä on toimenpiteenä huomattavasti nopeampi.

Move-rakentajan implementaatiosta voi myös huomata C++11:n kanssa mukana tulleen *noexcept*-määreen, joka kertoo move-rakentajan nothrow-poikkeustakuun [3]. Koska move-rakentajassa käytetään vain osoittimen sijoitusoperaattoria (assembly-käsky *mov*), pitää tämä poikkeustakuu paikkansa. *noexcept*-määreestä ja sen hyödyistä kerrotaan myöhemmin tässä luvussa, mutta lyhyesti mainittuna sen myötä esimerkiksi *std::vector* voi laajentuessaan siirtää alkioita kopioimisen sijaan.

4.1.2 Move-sijoitusoperaattori

Ennen C++11:tä neljä erityisjäsenfunktiota, jotka kääntäjä tai ohjelmoija määritteli luokalle, olivat rakentaja, kopiorakentaja, purkaja sekä sijoitusoperaattori. Kun move-semantiikan myötä kopiorakentaja sai parikseen move-rakentajan, niin myös sijoitusoperaatiolle voidaan määritellä vastine rvalue-argumenteille. Tätä sijoitusoperaattoria kutsutaan move-sijoitusoperaattoriksi. *String*-luokan sijoitusoperaattori ja move-sijoitusoperaattori on määritelty ohjelmassa 6.

```
// header
String& operator=(const String& rhs);
String& operator=(String&& rhs) noexcept;

// implementaatio
String& String::operator=(const String& rhs)
{
    if (this == &rhs) return *this;

    delete[] _data;

    std::size_t size = strlen(rhs._data) + 1;
    _data = new char[size];
    memcpy(_data, rhs._data, size);

    return *this;
}

String& String::operator=(String&& rhs) noexcept
{
    if (this == &rhs) return *this;

    delete[] _data;

    _data = rhs._data;
    rhs._data = nullptr;

    return *this;
}
```

Ohjelma 6. *String-luokan sijoitusoperaattori ja move-sijoitusoperaattori.*

Ohjelmassa 6 esillä oleva sijoitusoperaattori muistuttaa toiminnaltaan kopiorakentajaa paitsi, että sijoitusoperaatiossa pitää myös huolehtia korvattavan resurssin vapauttamisesta. Move-sijoitusoperaattori vapauttaa myös korvattavan resurssin, mutta toimii muuten kuin move-rakentaja. Molemmat sijoitusoperaatiot suojautuvat itsesijoitukselta ja palauttavat käytännön mukaan viitteen sijoitettuun, sijoitusoperaation vasemman puolen olioon.

Move-sijoitusoperaatiossa voidaan myös nähdä *noexcept*-määre. *delete* on standardin mukaan *noexcept* [7, s. 496] ja osoittimen sijoitus todettin aikaisemmin nothrowksi, joten *String*-luokan move-sijoitusoperaattorilla todella on nothrow-poikkeustakuu.

4.1.3 *noexcept*-määre move-operaatioissa

Kuten aikaisemmin ollaan lyhyesti mainittu, *noexcept*-määre kertoo funktion määritelmässä, että funktio on poikkeustakuultaan nothrow. *noexcept*-funktio ei siis tule tuottamaan poikkeuksia. Jos *noexcept*-määreen unohtaa lisätä, vaikka move-operaatio olisikin nothrow, niin sillä voi vaikuttaa esimerkiksi muiden luokkien operaatioiden *noxecept*-määreeseen tai suoraan *std::vectorin* suorituskykyyn.

Tutkitaan ensin miten move-operaatioista unohdetut *noexcept*-määre voivat vaikuttaa standardikirjaston *swap*-algoritmin *noexcept*-määreeseen. C++17-standardin deklaraatio *swap*-algoritmille on näkyvillä ohjelmassa 7. Varsinainen määritelmä riippuu implementaatiosta.

```
template <class T>
void swap(T& a, T& b) noexcept(is_nothrow_move_constructible_v<T>
                               && is_nothrow_move_assignable_v<T>
```

Ohjelma 7. *C++17-standardin deklaraatio swap-algoritmille [7, s. 534].*

Ohjelmasta 7 nähdään, että *swap* on ehdollisesti *noexcept*. Ehtona toimii se, että onko tyypillä, jonka oliot *swap* vaihtaa, *noexcept* move-rakentaja ja move-sijoitusoperaattori. Standardikirjastossa on myös tietorakenteita, joiden move-operaatiot ovat ehdollisesti *noexcept*: esimerkiksi *std::pair* tai *std::tuple*. Näiden operaatioiden *noexcept*-määre riippuu siitä, onko kyseisen parin tai tuplen sisältämillä tyypeillä *noexcept* move-sijoitusoperaattori. [7, s. 541, 549]

std::vector toimii esimerkkinä siitä, että *noexcept*-määreen jättäminen pois move-rakentajasta voi heikentää ohjelman suorituskykyä. *std::vector*<String> on suorituskyvyltään heikompi kuin *std::vector*<*String*>, jossa *String-luokan* move-rakentajassa on tämä *noexcept*-määre. Tutkitaan *std::vectorin* toimintaa alkioita lisätessä, mistä myös ilmenee tämä suorituskykymenetys.

Kun *std::vectoriin* lisätään alkioita *std::vector::push_back* metodin kautta, niin jossain vaiheessa vektorin muistialuetta täytyy laajentaa. *std::vector* laajentuu aina, kun alkion lisääminen *push_back*-metodin kautta ylittää vektorin koon [7, s. 897]. *push_back* lupaa lisäksi vahvan poikkeustakuun [4, s. 92], eli jos *push_backin* aikana tapahtuu poikkeus, niin vektorin tila säilyy muuttumattomana. Vahva poikkeustakuu saavutetaan laajennuksessa kopioimalla vektorin alkiot uuteen muistialueeseen, ja tuhoamalla vanhat alkiot vasta, kun kaikki alkiot ovat kopioitu [4, s. 92].

Olisi ilmeistä, että alkiot siirrettäisiin uuteen muistialueeseen kopioimisen sijasta, koska olioiden siirtäminen on lähes aina tehokkaampaa kuin kopioiminen. Siirtäminen kuitenkin uhkaa *push_back*-metodin vahvaa poikkeustakuuta. Kuvitellaan tapaus, jossa puolivälissä alkioiden siirtämistä uuteen muistialueeseen move-rakentajassa tapahtuu poikkeus. *std::vectorin* laajentuminen keskeytyy poikkeuksen seurauksena, ja koska alkiot siirrettiin kopioimisen sijaan, nyt vanhaa muistialuetta on muokattu. Esimerkiksi *String*-vektorin tapauksessa tämä tarkoittaisi, että puolilta vektorin *String*-olioilta puuttuu dataa, koska siirrettäessä *String*-olioita niiden osoitin nollataan. *push_back*-metodi ei siis olisi enää poikkeustakuultaan vahva.

Siirtäminen voidaan kuitenkin tehdä kopioimisen sijaan vektorin laajentuessa, jos vektorin sisältämä tyyppi lupaa move-rakentajalle nothrow-poikkeustakuun, eli jos move-rakentajalla on *noexcept*-määre. Tällöin *push_back* säilyttää vahvan poikkeustakuunsa, koska uuden muistialueen alkiot voidaan initialisoida move-rakentajalla, joka siis lupaa, että poikkeuksia ei synny sen toiminnan myötä.

On hyvä huomioida, että vaikka *noexcept*-määreestä on paljon hyötyä *std::vector* kanssa, se tulisi laittaa move-rakentajaan, vain jos move-rakentajalla todella on nothrow-poikkeustakuu. Jos *noexcept* määritetyssä *move*-rakentajassa saattuukin poikkeus kesken vektorin laajentumisen, niin standardi ei määrittele seurauksia [7, s. 897]. Vektorin sisältö siis voi olla melkein mitä vain.

4.2 Kääntäjän määrittelemät move-operaatiot

C++11:ssa ja sen jälkeen kääntäjä voi rakentajan, tuhoajan ja kopio-operaatioiden lisäksi määritellä move-rakentajan ja move-sijoitusoperaattorin tiettyjen ehtojen täyttyessä. Pääasiassa kopio- ja move-operaatiolle kääntäjä katsoo, että sekä kantaluokilla että jäsenillä on saatavilla vastaavat operaatiot [7, s. 305; 307]. Move-operaatiot ovat tässä suhteessa poikkeuksellisia, koska kantaluokat ja jäsenet voidaan myös siirtää kopioimalla [4, s. 110] (rvaluet sitoutuvat myös vakio–lvalue-viitteisiin).

Kääntäjä tutkii myös, onko ohjelmoija määrittänyt tiettyjä erityisjäsenfunktioita. Kopio-operaatiot ovat kääntäjän suhteen itsenäisiä: sijoitusoperaation määritteleminen ei estä kääntäjää määrittämästä kopiorakentajaa ja päinvastoin. Siirtämisoperaatiot taas eivät ole, koska jommankumman move-operaation määrittäminen estää kääntäjää määrittämästä toista. [7, s. 304]

Move- ja kopio-operaatiot eivät ole keskenään myöskään itsenäisiä: määrittämällä move-operaatiot estää kääntäjää määrittämästä kopio-operaatioita, mikä pätee myös päinvastoin. Kaiken tämän lisäksi purkajan määrittäminen estää kääntäjää määrittämästä kumpaakaan siirtämisoperaatiota. [7, s. 304, 306]

Kääntäjän määrittelemät erityisjäsenfunktiot ovat *noexcept*, jos jäsenfunktion kutsumat operaatiot jäsenille ja kantaluokille ovat *noexcept* [7, s. 430–431]. Esimerkiksi jos luokan kaikilla jäsenillä ja kantaluokilla on *noexcept*-kopiorakentaja, niin kääntäjä pystyy määrittelemään *noexcept*-kopiorakentajan. Kääntäjän määrittämän erityisjäsenfunktion poikkeustakuu määritetään vain tarvittaessa, eli esimerkiksi jos funktion *noexcept*-määrettä kysytään [7, s. 432].

4.3 Rule of five

Ennen C++11:tä tunnettiin sääntö ”Rule of three”, joka sanoi, että jos luokka tarvitsee ohjelmoijan määrittämän kopiorakentajan, sijoitusoperaattorin tai purkajan, ohjelmoijan tulisi määrittää kaikki kolme [15]. Säännön idea oli se, että jos jonkin näiden kolmen erityisjäsenfunktion tulee tehdä jotain, jota kääntäjän määrittelemä funktio ei tee, niin silloin kaiken kolmen erityisjäsenfunktion tulisi myös osata tehdä se. [4, s. 111]

C++11 ja move-operaatiot laajentavat tämän säännön viiteen, koska minkä tahansa edellä mainittujen kolmen operaation määrittäminen estää kääntäjää määrittämästä move-rakentajaa tai move-sijoitusoperaattoria. Siis jos täytyy määritellä mikä tahansa näistä viidestä operaatiosta, tulisi määritellä kaikki viisi [15], josta nimi ”Rule of five” tuleekin.

Luokilla, joilla kaikki viisi kääntäjän määrittelemää erityisjäsenfunktiota ovat suotavia, tulisi noudattaa sääntöä ”Rule of zero”, jossa vain luokan rakentaja määritellään [15].

4.4 Taattu kopion välttäminen – C++17

C++17:n prvaluen määritelmän muutos johtaa siihen, että prvalue-lausekkeesta ei muodostu oliota, joka siirrettäisiin [9]. Tästä aikaisemmin mainittu termi taattu kopion välttäminen tulee [12]. Käydään esimerkkien avulla läpi kaksi tapausta, jossa C++17-standardi takaa, että olio initialisoidaan prvalue-lausekkeesta suoraan. Ohjelmassa 8 on esillä molemmat tapaukset: ensimmäisessä tapauksessa olio initialisoidaan saman tyypin rakentajakutsusta ja toisessa funktio palauttaa paluutyyppinsä rakentajakutsun [16].

```
String string = String("Hello, World!");     // 1.tapaus

String createString(const char* data)        // 2.tapaus
{
    return String(data);
}
String string2(createString("Hello, World!"));
```

Ohjelma 8. *Esimerkki, jossa funktio palauttaa paluutyyppinsä rakentajakutsun.*

Kuten luvussa 2.4 huomattiin, ennen C++17:ää kääntäjä optimoi väliaikaisen olion lähes aina pois tälläisissä initialisoinneissa. Tämä optimointi tunnettiin RVO:na (return value optimization) [16]. RVO:n kytkemiseksi pois tarvittiin kääntäjäasetus ”*-fno-elide-constructors*”, koska se oli oletuksena aina päällä.

C++17:ssä ohjelmassa 8 ei synny milloinkaan väliaikaista oliota, vaikka käytettäisiinkin edellä mainittua kääntäjäasetusta, koska rakentajakutsu ja *createString* ovat prvalue-

lausekkeita, jotka initialisoivat *string- ja string2*-oliot suoraan. RVO:n tai C++17:n kanssa ohjelma 8 menisi siis kääntäjän näkökulmasta seuraavanlaisesti:

```
String string("Hello, World!");        // 1.tapaus

String createString(const char* data)  // 2.tapaus
{
    return String(data);
}
String string2(data);
```

Tämä on myös suorituskyvyn kannalta paras ratkaisu, koska väliaikaisen olion kanssa tarvittaisiin yksi rakentajakutsu, kaksi move-rakentajakutsua sekä kaksi purkajakutsuja. Ensimmäinen siirto tapahtuisi funktiosta kutsupaikalle ja toinen siirto kutsupaikalta *string-*/*string2*-olion haltuun. Taatun kopion välttämisen (tai ennen C++17:ää RVO:n) ansiosta tehdään vain yksi rakentajakutsu.

C++17:ssä RVO ei ole enää kääntäjän tekemää optimointia, vaan standardin vaatimaa käyttäytymistä prvalue-lausekkeiden suhteen. Samasta syystä taattu kopion välttäminen pätee vain prvalue-lausekkeilla. C++17:ssa RVO on siis pakollista. [16]

NRVO (Named Return Value Optimatization) on toisenlainen kääntäjän tekemä paluuarvo-optimointi. Kääntäjä tekee NRVO:ta, kun funktiosta palautetaan nimellinen arvo, josta lisänimi ”Named” tuleekin. [16] NRVO ei ole taattua C++17-standardissa siksi, koska nimetyt arvot ovat glvalueita, ja C++17:n muutokset koskivat vain prvalueita. Tutkitaan NRVO:ta esimerkin avulla:

```
String returnNamedString(const char* data)
{
   String named(data);

   return named;
}
String string2(createString("Hello World!"));
```

Ilman edellä mainittua “*-fno-elide-constructors*”-kääntäjäasetusta esimerkki käyttäytyy täysin samalla tavalla kuin aikaisempikin: *string2*-olion initialisoimiseksi kutsutaan rakentajaa vain kerran, koska kääntäjäasetus vaikuttaa myös NRVO:hon.

Jos kääntäjäasetus kytketään päälle, niin koska *string2* initialisoidaan prvalue-lausekkeesta, tässäkin hyödytään taatusta kopion välttämisestä. Siis C++17:ssa *string2*-olio initialisoidaan yhden move-rakentaja kutsun avulla, jota ei voida välttää, koska *named*-olio rakennettaan ennen *return*-käskyä. Ennen C++17:ää kääntäjäasetus päällä *named*-olion data siirretään ensin kutsupaikalle väliaikaisen olion haltuun ja sitten vasta *string2*-olion haltuun. Tässä kontektissa kääntäjä saa siirtää *named*-lvaluen, koska tämä olio tuhottaisiin funktiosta palatessa [7, s. 309] [16].

5. FORWARDOIVAT VIITTEET

Ennen kuin voidaan siirtyä tutkimaan lvalue-olioiden siirtämistä on käytävä läpi rvalue-viitteiden yhteydessä kehittynyt termi forwardoiva viite[4]. Forwardoiva viite esiintyy pelkästään kontekstissa, jossa tapahtuu tyyppideduktiota, eli esimerkiksi template-parametrissa tai *auto*-tyypin kanssa [4, s. 164–165]. Forwardoiva viite on joko lvalue-viite tai rvalue-viite riippuen onko olio, johon se sidotaan, lvalue vai rvalue [4, s. 164] [17].

Forwardoiva viite on C++-standardin käyttämä termi [7, s. 410]. C++11-standardissa termiä ei vielä ollut, vaan se tuotiin standardiin vasta C++17:ssä [18]. Useita vuosia ennen termin standardisointia Scott Meyers kehitti viitteelle termin universaali viite [17]. Universaali viite ja forwardoiva viite tarkoittavat siis täysin samaa asiaa, mutta dokumentin N4164, jonka myötä termi ”forwarding reference” päätyi C++17-standardiin, mielestä ”universal reference” on liian laajakäsitteinen [18].

Forwardoivan viitteen käyttökohteet ei-template koodissa ovat melko rajatut, mutta templateissa sen kenties tärkein käyttökohde on termin nimen mukaisesti ns. ”perfect forwarding”, jota tämä kandidaatintyö sivuaa. Termi on kuitenkin niin vahvasti sidoksissa rvalue-viitteisiin, ja se esiintyy *std::moven* implementaatiossa, niin käydään se tässä läpi.

5.1 Viiteromahdus[5]

Oikeastaan forwardoiva viite on vain abstraktio viiteromahdukselle. Ennen kuin tutkitaan viiteromahdusta, on tärkeää muistaa, että C++:ssa ei voi ottaa viitettä viitteestä [7, s. 207].

```
int a = 5;
int& &ref = a;        // kääntäjävirhe  ”reference to a reference”
```

Kuitenkin jos hyödyntää esimerkiksi *typedef* kautta määrättyä nimeä jollekkin viitteelle, antaa se käsittää, että viite viitteestä olisi sallittu.

```
typedef int& intRef
int i = 0;

intRef &ref = i;      // Vastaa int& &ref? Kääntäjä ei kuitenkaa valita.
```

Lihavoitu kysymys onkin väärässä. Tämä ei ole kääntäjän näkökulmasta *int& &ref*, koska tässä tapahtuu viiteromahdus. Viiteromahdus tapahtuu aina kontekstissa, jossa yritetään

[4] engl. forwarding reference
[5] engl. reference collapsing

yhdistää viite sekä template-parametri, *typedef* tai *decltype*, jotka ovat viitteitä johonkin tyyppiin [7, s. 207]. Tutkivaan vielä aikaisempaa esimerkkiä tarkemmin ohjelmassa 9.

```
typedef int& intRef;
typedef int&& intRvalRef;
int i = 0;

intRef &ref = i;           // int& & = int&
decltype(ref) &ref2 = i;   // sama kuin yllä eli int&

intRef &&ref3 = i;         // int& && = int&
intRvalRef &ref4 = i;      // int&& & = int&

intRvalRef &&ref5 = 5;     // int&& && = int&&
                           // Huom. ei voida sitoa lvalueen i
```

Ohjelma 9. *Esimerkkejä viiteromahduksesta*

Ohjelmasta 9 huomataan, että aina jos jompikumpi vitteistä on lvalue-viite, tuloksena on aina lvalue-viite. Viiteromahduksesta muodostuu rvalue-viite vain ja ainoastaan silloin, jos kumpikin viitteistä on rvalue-viite. [4, s. 199] [10, s. 16]

5.2 Funktio templaten parametri *T&&*

Forwardoivasta viitteestä on kyse aina, kun funktio templatessa esiintyvä parametri on tyypiltään rvalue-viite template-parametrista. Template-parametri ei saa olla luokan templatesta, eikä forwardoivassa viitteessä saa olla *const*- tai *volatile*-etuliitteitä. [7, s. 410] Ohjelmassa 10 on esimerkkejä forwardoivista viitteistä.

```
template <typename T>
void test(T&& fRef)              // fRef on forwardoiva viite

template <typename T>
void function(const T&& rRef);   // rRef on vakio-rvalue-viite
                                 // const-etuliitteen takia.
template <typename V>
class Example
{
   template <typename T>
   void function(T&& fRef,       // fRef on forwardoiva viite
                 V&& notFRef);   // notFRef ei ole forwardoiva viite
                                 // Huom. V on luokan template-parametri
};
```

Ohjelma 10. *Esimerkkejä forwardoivista ja ei-forwardoivista viitteistä*

Kuten viiteromahdusta käsitellessä tuli ilmi, viiteromahdus tapahtuu myös template-parametreilla. Tutkitaan ohjelman 10 funktio templaten *test* toimintaa kutsumalla sitä ensin lvalue-argumentilla:

```
int lval = 5;
test(lval);                    // template-parametri T on int&
```

Kun kääntäjä johtaa templaten-tyypiksi *int&*, funktion parametrissa tapahtuu viiteromahdus.

```
void test<int&>(int& && fRef);   // ennen viiteromahdusta
```

Viiteromahduksen säännöistä tiedetään, että kun toinen kahdesta viittestä on lvalue-viite, niin tuloksena on aina lvalue-viite. Funktion pararametrin tyypiksi tulee siis *int&.*

```
void test<int&>(int& fRef);      // viiteromahduksen jälkeen
```

rvalue-tapaus on hieman yksinkertaisempi, koska tässä ei tapahdu viiteromahdusta, funktion parametrin tyypiksi tulee kuitenkin rvalue-viite.

```
test(4);                         // T on int

void test<int>(int&& fRef);      // ei viiteromahdusta
```

Siis nähdään, että forwardoivan viitteen tyyppi on aina joko lvalue tai rvalue-viite riippuen argumentin arvokategoriasta.

On myös tärkeää huomioida, että koska funktion *test* parametrilla *T&& fRef* ei ole *const*- tai *volatile*-etuliitettä, niin template-parametriin *T* johdetaan kaikki argumentin *const- ja volatile*-etuliitteet [7, s. 410]. Erityisesti jos templatea käytetään olioiden siirtämisen, vakioargumenttien kanssa täytyy olla tarkkana, koska luokan move-operaatiot hyväksyvät yleensä vain ei–vakio-rvalueita. Yksi tälläinen tapaus käydään läpi *std::move*-funktion kanssa luvussa 6.1.2.

```
const int j = 5;
test(j);                         // T on nyt const int&
```

5.3 auto&&

C++11:n kanssa esiteltiin myös *auto*-tyyppimääritelmä [3]. *auto*-määritelmän tyyppi johdetaan lähes identtisesti template-parametreihin verrattuna [7, s. 173]. Tästä syystä myös *auto&&*-tyypillä deklaroidut muuttujat ovat forwardoivia viitteitä [4, s. 167] [18]. Tutkitaan *auto&&:n* toimintaa esimerkillä:

```
int lval = 5;

auto&& lRef = lval;  // lRef tyyppi on int&
auto&& rRef = 5;     // rRef tyyppi on int&&
```

auto&&:n käyttökohteet eivät olleet C++11:ssä läheskään yhtä laajat kuin templatien forwardoivilla viitteillä, mutta C++14:ssä ja sen jälkeen *auto&&:n* kanssa voi tehdä esimerkiksi forwardoivia lambdoja [4, s. 167].

6. LVALUE-OLIOIDEN SIIRTÄMINEN

rvalue-olioiden siirtämisen verrattuna lvalue-olioiden siirtäminen on huomattavasti ilmaisevampaa. rvalue-olioilla riitti, että luokalle oli määritelty move-operaatiot (kääntäjän tai ohjelmoijan toimesta), jonka jälkeen kääntäjä osaa kutsua niitä tarvittaessa. lvalue-olioiden siirtäminen on taas ohjelmoijan tekemä päätös, ja kääntäjä siirtää niitä vain käskettäessä. Kun lvalueita siirretään, sen arvokategoria muutetaan aina ensin xvalueksi, koska olihan yksi lvaluen määritelmistä, että siitä ei voida siirtää.

lvalue-olioita siirretään siis muuntamalla lvalue rvalue-vitteeksi (xvalueksi). Tämän myötä kääntäjä kutsuu siirtävää operaatiota, oli kyseessä sitten funktio, joka ottaa rvalue-viitteen parametrina, tai yleisemmin move-rakentaja tai move-sijoitusoperaattori.

Standardikirjastossa on kaksi funktio templatea, jotka tekevät muunnoksia lvaluesta rvalue-viitteeksi: *std::move* tekee muunnoksen aina ja *std::forward* vain tietyn ehdon täyttyessä. On tärkeää huomata, että *std::move* ja *std::forward* eivät kumpikaan itsessään siirrä mitään. Molemmat funktiot suorittavat vain tyyppimuunnoksia. [4, s. 158]

6.1 std::move

Move-semantiikan lisäämistä C++:aan ehdottanneessa dokumentissa N1377 toivottiin myös lisätä kieleen ominaisuus muuntaa lvalue suoraan rvalueksi [13]:

```
String lvalue("Hello");                 // String-luokan lvalue-olio.

String rvalue =                         // Muutetaan lvalue rvalueksi.
   static_cast<String&&>(lvalue);       // Tuloshan on xvalue.
```

Samainen dokumentti myös totesi, että tälläinen *static_cast* syntaksi on ruma, jonka myötä se esitti tyyppimuunnoksen suorittavan funktio templaten *move* [13]:

```
template <class T>
inline
T&&
move(T&& x)
{
    return static_cast<T&&>(x);
}
```

Tämä määritelmä ei päätynyt suoraan standardikirjastoon, koska edelle näkyvä *static_cast* ei muunna aivan kaikkia argumenttityyppejä rvalue-viitteeksi (viiteromahduksen säännöt). Tarkka syy selviää seuraavassa luvussa, jossa tutkitaan *std::moven* määritelmää.

6.1.1 Määritelmä ja toiminta

std::moven määritelmä on sen verran lyhyt, että käydään sen esimerkki-implementaatio läpi. Implementaatiossa käytetään C++14:n alias-templatea *std::remove_reference_t*.

```
// std namespacen sisällä

template<typename T>
remove_reference_t<T>&&                // paluutyyppi (ks. alla)
move(T&& param)
{
    using ReturnType =                 // Muunnetaan T rvalue-viitteeksi
          remove_reference_t<T>&&;     // Huom. viiteromahduksen säännöt

    return static_cast<ReturnType>(param);
}
```

Ohjelma 11. *std::moven esimerkki-implementatio [4, s. 158]*

Ensin huomataan, että ohjelman 11 funktio template ottaa parametrina forwardoivan viitteen eli T&& tyyppimuunnettavaan olioon. Tyyppimuunnoksessaan *std::move* poistaa ensin viitteet template-parametrista *T std::remove_reference_t:n* avulla ja lisää siihen sitten ”*&&*”. *std::remove_reference_t:tä* käytetään siksi, että jos templaten tyyppi *T* sattuisi olemaan lvalue-viite, niin viiteromahduksesta *T& &&* seuraisi myös lvalue-viite. *std::remove_reference_t*:n käyttö siis varmistaa, että *ReturnTypen* tuloksena on aina rvalue-viite. Lopuksi *std::move* palauttaa parametrin tyyppimuunnettuna tähän rvalue-viitteeseen. [4, s. 159] Tutkitaan seuraavaksi ohjelmassa 12, miten *std::movea* voi käyttää siirtämään lvalueita.

```
#include <String.h>                    // ks. luku 4.1
#include <vector>

String hi("Hello");
String hello = std::move(hi);          // Kutsuu String-luokan
                                       // move-rakentajaa.
// hi-olio sisältää nyt vain nolla-osoittimen

std::vector<String> strings;
strings.push_back(std::move(hello));   // Kutsuu siirtävää push_backia.

// hello-olio sisältää nyt vain nolla-osoittimen

class TextBox
{
public:
      TextBox(String text)             // Huom. arvoparametri
      : _string(std::move(text))       // Kutsuu String-luokan
      { ... }                          // move-rakentajaa
private:
      String _string;
};
```

Ohjelma 12. *std::moven käyttöesimerkkejä*

Ohjelmassa 12 on erityistä huomioida, että *std::move*-kutsun jälkeen, siirrettyä olioita ei tulisi milloinkaan käyttää. Vaikka olion nimi on vielä käytettävissä, ei olio välttämättä ole enää käyttökelpoisessa tilassa. Esimerkiksi luvun 4 *String*-luokan olio sisältäisi move-rakentaja kutsun jälkeen vain nolla-osoittimen. Tälläisen olion käyttäminen kontektissä, joka olettaa, että olio sisältää merkkijonon, voi synnyttää vaikeasti debuggattavan ajonaikaisen virheen.

6.1.2 Kutsuminen vakioargumentilla

std::moven toimintaan liittyy eräs olennainen erityistapaus, jossa kyseistä funktiota kutsutaan vakioargumentilla. Tutkitaan erityistapausta ohjelman 12 *TextBox*-luokan avulla. Käytetään nyt vain *String*-luokan sijasta *std::stringiä*, jotta tapauksen yleisluontoisuus on helpompi nähdä.

```
#include <string>

class TextBox
{
public:
      TextBox(std::string text)
      : _string(std::move(text))          // kutsuu std::string
      { ... }                             // move-rakentajaa

private:
      std::string _string;
};
```

Jos oletetaan, että *text*-parametria ei muokata, niin ei olisi täysin outoa lisätä rakentajan parametriin *const*-etuliite:

```
   class TextBox
   {
   public:
      TextBox(const std::string text)
      : _string(std::move(text))          // Tässä ei kutsuta std::string
      { ... }                             // move-rakentajaa vaan
    private:                              // kopiorakentajaa.
      std::string _string;
   };
```

Tämä koodi tulee toimimaan virheettä, mutta sen sijaan että *text* siirrettäisiin jäsenmuuttujaan *_string*, se kopioidaan. Tämä tapahtuu siksi, että *std::move*-funktiolle annetaan vakioargumentti, jolloin *std::moven* template-parametri *T* tulee olemaan *const std::string&*, jonka myötä *std::move* tyyppimuuntaa argumentin muotoon *const std::string&&*. Ohjelmassa 13 esillä olevasta *std::stringin* kopiorakentajasta ja move-rakentajasta nähdään, kuinka kääntäjä ei voi kutsua move-rakentajaa *const std::string&&*-tyypille.

```
using string = basic_string<char>; // std::string on typedef tyypille
…                                  // std::basic_string<char>
class basic_string {
  …
  basic_string(const basic_string& str);     // kopiorakentaja
  basic_string(basic_string&& str) noexcept; // move-rakentaja
  …                                          // Huom. noexcept
};
```

Ohjelma 13. *std::stringin kopiorakentaja ja move-rakentaja [7, s. 735; 736–737]*

Koska *std::string* move-rakentaja ottaa parametrina ei-vakion rvalue-viitteen, niin *std::moven* palauttama vakio–rvalue-viite (vakio-xvalue) ei käy. Tämä vakio-xvalue voidaan kuitenkin välittää kopiorakentajalle, koska vakio–lvalue-viite voi sitoutua siihen, kuten luvun 3 taulukosta 1 nähdään (vakio xvalue on *const rvalue)*.

TextBox-luokan rakentajassa kutsutaan siis jäsenmuuttujan *_string* initialisoimiseksi *std::stringin* kopiorakentajaa, vaikka argumenttina annettu *text* muutettiinkin rvalue-viitteeksi *std::moven* toimesta. Luvussa 4 huomattiin, että siirrettävää oliota tarvitsee yleensä muokata. On siis loogista, että vakio-olioiden siirtäminen ei ole sallittua C++:ssa.

Vakiodeklaraatiota tulisi siis välttää olioilla, joita halutaan siirtää. Tästä erityistapauksesta ilmenee myös, että *std::move* ei lupaa, että olio siirrettäisiin. *std::moven* suhteen voi olla varma vain siitä, että tuloksena on rvalue-viite. [4, s. 161]

6.2 std::forward

Kun *std::move* muuntaa argumenttinsa aina rvalue-viitteeksi, niin *std::forward* tekee sen vain tietyllä ehdolla. Tänä ehtona toimii se, että onko *std::forwardille* annettu argumentti rakennettu rvaluesta vai lvaluesta. Vain rvalue-tapauksessa std::forward muuntaa argumentin rvalue-viitteeksi, muutoin funktio palauttaa lvaluen. [4, s. 161]

std::forward liittyy nimensäkin enemmän puolesta forwardoimiseen eikä move-semantiikkaan, joten tässä kandidaatintyössä sitä ei käsitellä enempää. Kandidaatintyön pohjalta *std::forwardin* toiminta ja käyttö on kuitenkin helppo omaksua.

7. YHTEENVETO

Työn tarkoituksena oli esitellä vuonna 2011 julkaistun C++11-standardin ominaisuus move-semantiikka. Työssä pyrittiin erittäin perusteelliseen esittelyyn lähtien aivan move-semantiikan perustasta – arvokategorioista ja viitteistä – siirtyen vähitellen näyttämään, miten move-semantiikan ydinideaa toteutetaan, eli miten olioita siirretään C++-koodissa. Näiden lisäksi työssä tehtiin katsaus hieman move-semantiikan ulkopuolelle forwardoiviin viitteisiin, ja miten ne toimivat abstraktioina viiteromahdukselle.

Esittely alkoi määrittämällä, mitä termit rvalue ja lvalue merkitsivät C++:ssa. Huomattiin, että termien vanhat määritelmät eivät olleet tarpeeksi tarkkoja C++11:een, koska move-semantiikka haluttiin tuoda kieleen. Tämän johdosta C++11:een kehitettiin uudet arvokategoriat: rvalue, prvalue, xvalue, lvalue ja glvalue, joista vielä prvaluen määritelmä muuttui C++17:ssä. Luvun 2 lopussa huomattiin myös, kuinka lyhyt väliaikaisten olioiden elinikä on.

Seuraavaksi esiteltiin move-semantiikan ohella julkaistu uusi viitetyyppi rvalue-viite, joka mahdollisti rvalueihin erikoistuvien funktioiden määrittelemisen. Arvokategorioiden ja rvalue-viitteiden pohjalta työssä lähdettiin tutkimaan rvalueiden siirtämistä, jossa keskeisintä oli move-rakentajan ja move-sijoitusoperaattorin määritteleminen. Samassa luvussa nähtiin myös *noexcept*-määreen merkitys move-operaatioille, missä tärkeintä oli *std::vectorille* saatava optimointi. Luvun lopussa tutkittiin, miksi C++17:ssä prvalueita ei enää siirrettä, ja miten kääntäjän paluuarvo-optimointi toimivat (RVO ja NRVO).

Sitten työssä tehtiin pikakatsaus forwardoiviin viitteiseen, jonka jälkeen siirryttiin tutkimaan lvalueiden siirtämistä. Huomattiin, että lvalueiden siirtäminen vaatii move-operaatioiden määrittämisen lisäksi tyyppimuunnoksen rvalueeksi (*std::move*) ja valvomisen, että siirrettyä oliota ei käytetä. Luvussa tutkittiin myös, kuinka kutsumalla *std::move*-funktiota vakioargumentilla johtaa siihen, että argumentti kopioidaan siirtämisen sijaan.

Tämän työn pohjalta ehdottomasti selkein jatkotutkimuskohde olisi laajentaa työ käsittämään myös forwardoimista. Kandidaatintyön puitteissa työstä tulisi kuitenkin tarpeettoman laaja, koska forwardoimiseen liittyy lukuisia erityistapauksia, joiden tutkiminen veisi aiheen tarpeettoman kauas move-semantiikasta. Forwardoiminen on kaiken lisäksi lähes yksinomaan template-koodissa esiintyvä toimintatapa aivan kuin forwardoivat viitteetkin. Move-semantiikalla on selkeästi enemmän käyttökohteita ei-template koodissa.

LÄHTEET

[1] H. Sutter, ”We have an international standard: C++0x is unanimously approved”, 12.8.2011.
Saatavissa (viitattu 2.6.2019):
https://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/

[2] B. Stroustrup, ”C++11 - the new ISO C++ standard”,19.8.2016.
Saatavissa (viitattu 2.6.2019):
http://www.stroustrup.com/C++11FAQ.html

[3] Standard C++ Foundation, ”C++11 Language Extensions — General Features”.
Saatavissa (viitattu 2.6.2019):
https://isocpp.org/wiki/faq/cpp11-language

[4] S. Meyers, Effective Modern C++, O'Reilly Media, Inc, 11/2014, p. 334.

[5] B. Stroustrup, ”“New” Value Terminology”, 04/2010.
Saatavissa (viitattu 2.7.2019):
http://www.stroustrup.com/terminology.pdf

[6] D. W. B. J, N. Buxton, D. F. Hartley, E. N. C ja Strachey, ”The Main Features of CPL,” *The Computer Journal,* osa/vuosik. 6, nro 2, p. 134–143, 1963.

[7] ISO/IEC., ”Working Draft, Standard for Programming Language C++ (N4659) (C++17 final working draft)”, 21.3.2017.
Saatavissa (viitattu 2.6.2019):
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

[8] Standard C++ Foundation, ”N3337 Working Draft (C++11 first post-publication draft), Standard for Programming Language C++”, 16.1.2012.
Saatavissa (viitattu 16.7.2019):
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[9] cppreference.com, ”Value categories”, 5.2.2019.
Saatavissa (viitattu 2 7 2019):
https://en.cppreference.com/w/cpp/language/value_category

[10] J. Jokinen, ”C++11 tyyppisälää,” 2012.
Saatavissa (viitattu 2.7.2019):
http://www.cs.tut.fi/~jyke/cpp11/cpp11seminaari.pdf

[11] C++ Standard Core Language, ”C++ Standard Core Language Defect Reports and Accepted Issues, Revision 85”, 3.9.2013.
Saatavissa (viitattu 16.7.2019):
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3714.html

[12] R. Smith, ”Wording for guaranteed copy elision through simplified value categories”, 20.6.2016.
Saatavissa (viitattu 16.7.2019):
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0135r1.html

[13] H. E. Hinnant, P. Dimov ja D. Abrahams, ”A Proposal to Add Move Semantics Support to the C++ Language”, 10.9.2002.
Saatavissa (viitattu 2.6.2019):
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm

[14] cppreference.com, ”std::auto_ptr”, 14.7.2017.
Saatavissa (viitattu 15.7.2019):
https://en.cppreference.com/w/cpp/memory/auto_ptr

[15] cppreference.com, ”The rule of three/five/zero”, 3.5.2019.
Saatavissa (viitattu 27.7.2019):
https://en.cppreference.com/w/cpp/language/rule_of_three.

[16] cppreference.com, ”Copy elision”, 6.4.2019.
Saatavissa (viitattu 17.7.2019):
https://en.cppreference.com/w/cpp/language/copy_elision

[17] S. Meyers, ”Universal References in C++11 – Scott Meyers,” Standard C++ Foundation, 1.9.2012.
Saatavissa (viitattu 2.6.2019):
https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers.

[18] H. Sutter ja G. D. R. Bjarne Stroustrup, ”N4164 Forwarding References”, 6.10. 2014.
Saatavissa (viitattu 2.6.2019):
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4164.pdf