Liite1: C++-kieli pähkinänkuoressa

Tässä kirjaan toivotussa liitteessä on käsitelty C++-kielen perusseikkoja lyhyessä ja tiiviissä muodossa, jotta lukija voisi tarpeen tullen palauttaa mieleen keskeiset asiat nopeasti ja vaivattomasti.

Perusteet lyhyesti

C++-ohjelman rakenne:

```
#include <iostream.h>

int main()
{
  cout << "TERVEHDYS\n";
  return 0;
}
```

Ohjelmassa tulee olla main()-funktio. Ohjelmaan voidaan tuoda erilaisia kirjastotiedostoja aina tarpeen mukaan. Nyt esimerkkiohjelmaan tuodaan tiedosto iostream.h, joka mahdollistaa tulostuksen. Otsikkotiedostojen tuominen tehdään esikäsittelijän komennolla #include.
Lähdekoodin tunnus on yleensä .cpp ja otsikkotiedoston .h tai .hpp.

C++-kieli sisältää runsaasti **standardikirjastoja**, joita tulee hyödyntää mahdollisimman paljon. Itse komentojoukko on melko suppea.

Tärkeitä otsikkotiedostoja (ANSI C ja C++) ovat muun muassa: math.h, iostream.h, stdlib.h, string.h, string, time.h, iomanip.h, assert.h, complex.h, ctype.h, fstream.h, except.h. Työkaluissa on lisäksi muita hyödyllisiä kirjastoja.

Ohjelma (.cpp) käännetään (compile; syntyy .obj) ja linkitetään (link; syntyy .exe), jolloin sitä voidaan ajaa.

Muuttuja on muistipaikka, jolle annetaan nimi. Muuttujan tietotyyppi määrää, millaista tietoa muuttujaan voidaan sijoittaa ja kuinka paljon muistia muuttujalle varataan.

Globaali muuttuja esitellään funktioiden ulkopuolella ja se näkyy kaikille ohjelman funktioille.

Paikallinen muuttuja esitellään funktion sisällä (tai lohkon sisällä) ja se näkyy vain esittelyalueellaan.

Perustietotyyppejä ovat: short, int, long, float, double, char, bool. Int on yleensä 32-bittistä kääntäjää käytettäessä 4-tavuinen eli siihen mahtuu etumerkillisenä ± 2 miljardia. Etumerkittömyys merkitään avainsanalla unsigned. Char on yhden tavun kokoinen. Double on tarkempi kuin float ja sen arvoalue on suurempi.

C++-kielessä ei ole tarkkoja määrittelyjä tietotyyppien koolle, joten se kannattaa tarkistaa **sizeof()**-operaattorilla:

```
cout << sizeof(int);
```

Vakio muodostetaan avainsanalla const:

```
const short vuosi = 1900; (suositeltu tapa)
```

Tai

```
#define vuosi 1900;
```

#define on **esikäsittelijä**n komento. Esikäsittelijä käsittelee käännettävän koodin ennen kääntämistä ja etsii koodista lauseita, jotka alkavat risuaitamerkillä (#). Jos sellainen lause löytyy, esikäsittelijä suorittaa merkin jälkeisen komennon. Lopuksi esikäsittelijä antaa koodin kääntäjälle.

Esikäsittelijän komennot ovat seuraavat:

#ifdef	#define	#ifndef	#elif
#include	#else	#line	#endif
#pragma	#error	#undef	#if

Näit komentoja voidaan tarvita esimerkiksi siirrettäessä ohjelmia ympäristöstä toiseen.

Taulukko voidaan muodostaa seuraavasti:

```
int taulu[5];  // 5 kokonaislukua taulukkoon
```

tai

```
int *taulu = new int[5]; // tila vapaasta muistista;
```

Huom! Jos new-operaattorilla varataan muistia, tulee muisti myös vapauttaa samalla tasolla; se tapahtuu operaattorilla delete. Jos tilaa on varattu taulukolle, on delete-syntaksina delete[]. C++-kieli ei sisällä sisäistä roskien keruuta (carbage collection) kuten esimerkiksi Java.

Taulukko voidaan alustaa esittelyssä:

```
int taulu[5] = {3, 4, 5, 3, 2};
```

Tai siihen sijoitetaan tiedot alkio kerrallaan:

```
taulu[0] = 3; taulu[1] = 4; taulu[2] = 5; taulu[3] = 3; taulu[4] =
2;
```

Yleensä taulukon käsittelyyn käytetään silmukoita, joissa kierroslaskuri toimii samalla taulukon indeksinä.

Silmukkarakenteet ovat for, while ja do-while:

```
for (int k = 0; k < 5; k++)
    taulu[k] = 0;
```

tai

```
int k = 0;
while (k < 5)
{
    taulu[k] = 0;
    k++;
}
```

tai

```
int k = 0;
do
{
    taulu[k] = 0;
    k++;
}
while (k < 5);
```

Silmukasta voidaan poistua kesken suorituksen **break**-lauseella. **Continue**-lauseella voidaan ohittaa silmukan rungon alempana olevat lauseet ja palata takaisin silmukan otsikkoon.

Useista ohjelmalauseista muodostuva lausejoukko tehdään **ohjelmalohko**ksi laittamalla lauseet aaltosulkujen sisään.

Ohjelmalausekkeissa käytetään usein **operaattoreita**. Operaation kohteet ovat nimeltään operandeja.

Matemaattiset operaattorit ovat +, -, *, / ja % (jakojäännös).
Vertailuoperaattorit ovat: <, >, <=, >=, !=, ==
Loogiset operaattorit ovat: && (JA), || (TAI), ! (EI)
Sijoitusoperaattori: =

Haarautumisia tiettyjen ehtojen mukaan toteutetaan **if**- ja **if-else-**rakenteilla:

```
if (ehto on tosi)
        tee jotain;
```

Tai

```
if (ehto on tosi)
        tee jotain
    else
        tee jotain muuta;
```

Monivalintarakennetta voidaan käyttää useiden if-lauseiden sijaan:

```
switch(arvo)
{
case arvo1: tehdään jotain; break;
case arvo2: tehdään jotain; break;
default: tehdään jotain;
}
```

Muuttujan *arvo* tulee palauttaa kokonaisluku.
Mikäli usea case toteuttaa saman toiminnon, voidaan case-osat sijoittaa alekkain ja break laitetaan viimeisen samaan joukkoon kuuluvan case-osan jälkeen.

Lueteltu vakio
Lueteltu vakio esitellään avainsanalla enum:

Esimerkkiohjelma: enum:

```
#include <iostream.h>
#include <conio.h>

main()
{
enum VARI {Punainen=1, Sininen, Harmaa, Keltainen};
VARI var;
int vari = Punainen;
cout << "Väri on " << vari << endl;
var = Harmaa;
cout << "Väri on " << var << endl;
var = 3;
cout << "Väri on " << var << endl;
if (var == Harmaa) cout << "Harmaa " << endl;
var = VARI(2);
if (var == Sininen) cout << "Sininen " << endl;


getch();
}
```

Tulos:

Merkkijonot esitetään joko char-taulukkona tai char-tyyppisen osoittimen avulla. Nyt on käytössä myös string-luokka, joka helpottaa merkkijonojen käsittelyä. Perinteisen merkkijonon loppumerkki on ‘\0’.

```
char nimi[] = “Kolehmainen”;   // TAI
char * nimi = “Kolehmainen”;  // TAI
char nimi[] = {‘K’, ‘o’, ‘l’, ‘e’, ‘h’, ‘m’, ‘a’, ‘i’, ‘n’, ‘e’,
‘n’, ‘\0’};
```

TAI

```
string nimi = “Kolehmainen”;  // vaatii: #include <string>
```

Funktiot ovat aliohjelmia, jotka voivat palauttaa jonkun arvon tai eivät palauta mitään arvoa. Funktiot esitellään ja määritellään. Määrittelyssä kirjoitetaan funktion runko. Void-tyyppinen funktio ei palauta arvoa return-lauseella. Kun funktion parametrina on osoitin tai viittaus, voidaan funktiolle vietyjä muuttujia muuttaa funktion sisällä.

Funktion esittely:

```
int summa(int x, int y);
```

Funktion määrittely:

```
int summa(int x, int y)
{
int sum = x + y;
return sum;
}
```

Funktion kutsu:

```
int tulos = summa(a, b);
```

Osoittimet ja viittaukset
Osoitinmuuttujaan tallennetaan jonkun muuttujan osoite tai yleensä muistipaikan osoite. Osoitinoperaattori on * ja osoiteoperaattori taas &:

```
int paino = 10;  // tavallinen muuttuja
int *os_paino = &paino; // osoitinmuuttuja; alustetaan muuttujan
paino
                  // osoitteella
```

Uudelleenviittausoperaattorin (*) avulla voidaan käsitellä osoitteessa olevaa arvoa osoittimen avulla:

```
*os_paino = 20;  // nyt muuttujan paino arvo on 20
```

Meillä voi tietenkin olla myös osoittimen osoitin:

```
int **os_os_paino;
**os_os_paino = *os_paino;
```

Esimerkkiohjelma osoittimista:

```
#include <iostream.h>
#include <conio.h>

main()
{
int paino = 30;
int *os_paino = &paino;

cout << "Paino on " << *os_paino << endl;
int **os_os_paino;
**os_os_paino = *os_paino;

cout << "Paino on " << **os_os_paino << endl;

getch();
}
```

Viittaus on muuttujan sijaisnimi. Viittausoperaattori on &:

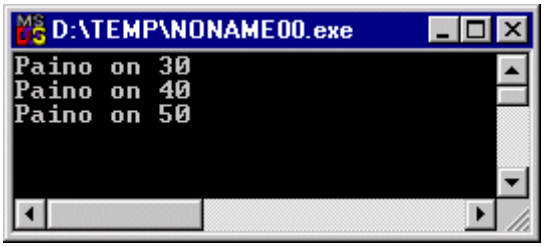
```
int paino = 30;
int & sijaispaino = paino;
```

Kun sijaisnimelle tehdään jotain, sama tehdään myös alkuperäiselle muuttujalle.

Esimerkkiohjelma viittauksen käytöstä:

```
#include <iostream.h>
#include <conio.h>

main()
{
int paino = 30;
int & sijaispaino = paino;

cout << "Paino on " << paino << endl;
sijaispaino = 40;
cout << "Paino on " << paino << endl;
paino = 50;
cout << "Paino on " << sijaispaino << endl;

getch();
}
```

Tulos:



Huom!
Kun funktion parametreina on osoittimia tai viittauksia, voidaan vietyjä muuttujia muuttaa funktion sisällä.

Esimerkki: osoitinparametrit

Funktion esittely:

```
void tuplaa(int *x, int *y);
```

Funktion määrittely:

```
void tuplaa(int *x, int *y)
{
*x = *x * 2;
*y = *y * 2;
}
```

Funktion kutsu:

```
tuplaa(&a, &b);
```

Esimerkki: viittausparametrit

Funktion esittely:

```
void tuplaa(int &x, int &y);
```

Funktion määrittely:

```
void tuplaa(int &x, int &y)
{
x = x * 2;
y = y * 2;
}
```

Funktion kutsu:

```
tuplaa(a, b);
```

Tavallisten osoitinmuuttujien lisäksi voidaan käyttää myös **funktio-osoittimia**. Funktion nimi ilman sulkumerkkejä on osoitin funktion alkuun (samalla lailla kuin taulukon nimi ilman yhtä hakasulkuparia on osoitin).

Esimerkiksi:

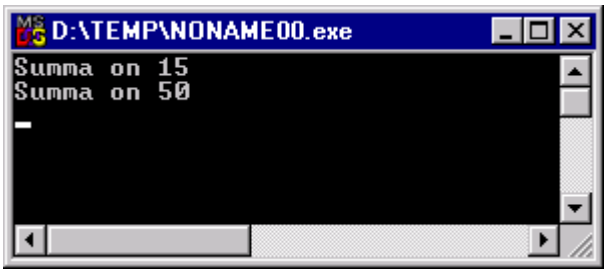
```
int summa(int x, int y);  // tavallinen funktio
int (*os_summa)(int m, int n);  // funktio-osoitin
```

Huomaa sulkumerkit funktio-osoittimessa ((*os_summa)); ne tarvitaan, jotta asteriski ei viittaisi palautusarvoon.

Esimerkkiohjelma funktio-osoittimen käytöstä:

```
#include <iostream.h>
#include <conio.h>

int summa(int x, int y);
int (*os_summa)(int m, int n);

main()
{
int a = 5, b = 10;
int tulos = summa(a, b);
cout << "Summa on " << tulos << endl;

os_summa = summa;

tulos = os_summa(20, 30);
cout << "Summa on " << tulos << endl;
getch();
```

```
}

int summa(int x, int y)
{
int sum = x + y;
return sum;
}
```



Luokka-käsite

Luokka (class) on C++-kielen tärkein tietotyyppi.

Luokka voidaan kuvata kaaviolla:

Luokan nimi
Luokan ominaisuudet tai attribuutit (jäsenmuuttujat)
Luokan toiminnot (jäsenfunktiot eli metodit)

Tietue (struct) muistuttaa luokkaa, mutta yleensä tietueeseen ei sijoiteta toimintoja (funktioita) kuten luokkaan, eli jäseninä on vain muuttujia, jotka vastaavat tietueen kenttiä.

Esimerkki tietueesta

```
#include <conio.h>
#include <fstream.h>

void main()
{

struct nimi
{
int ika;
}hlo1;

nimi hlotaulu[10];

hlo1.ika = 10;


for (int k = 0; k < 10; k++)
hlotaulu[k].ika = 2*k;

for (k = 0; k < 10; k++)
cout << hlotaulu[k].ika << "\n";

getch();
}
```

Reaalimaailman kohde sisältää yleensä
ominaisuuksia (millainen)
toimintoja (mitä osaa tehdä, kuinka palvelee muita)

Luokka kuvaa reaalimaailman kohteen ohjelmaan ja sisältää
Jäsenmuuttujat (myös attribuutti-sanaa käytetään)
Jäsenfunktiot eli metodit

Samaan tietotyyppiin **kapseloidaan** siis sekä jäsenmuuttujat että jäsenfunktiot.

Luokan esittely
Luokka esitellään avainsanalla class, jonka jälkeen tulee aloittava aaltosulku ja luettelo tietojäsenistä sekä metodeista. Esittely päättyy lopettavaan aaltosulkuun ja puolipisteeseen. Seuraavassa on Kana-luokan esittely:

Kana-luokka:

```
class Kana
{
  unsigned int ika;
  unsigned int paino;
  void kotkota();
};
```

Esittely siis vain kuvaa tietotyypin, joka on luokka. Ohjelmassa tulee olla muuttuja, joka on tuota tietotyyppiä, ennen kuin luokkaa voidaan hyödyntää.
Tuota muuttujaa kutsutaan luokan ilmentymäksi tai olioksi (object).

```
Kana kana1;
```

Nyt kana1 on siis Kana-tyyppiä. Se on myös Kana-olio ja se saa kaikki Kana-luokan piirteet.

Olio kana1 on nyt luotu pinoon (staattisesti). Olion jäsenmuuttujia ja –funktioita voidaan käsitellä nyt pisteoperaattorin avulla:

```
kana1.ika = 10;
kana1.paino = 20;
kana1.kotkota();
```

Huomaa, että kotkota()-funktiota ei ole vielä määritelty; se on vain esitelty luokan esittelyssä ja se tuleekin määritellä erikseen, kuten normaalit funktiot.

Voimme luoda toisenkin olion:

```
Kana kana2;
kana2.ika = 10;
kana2.paino = 20;
kana2.kotkota();
```

Seuraavana on koko ohjelma:

Listaus Kana-luokka:

```
#include <iostream.h>

class Kana
{
  unsigned int ika;
  unsigned int paino;
  void kotkota()
};
```

```
void Kana::kotkota()
{
cout << ”Kaakati kaa \n”;
}

void main()
{
kana1.ika = 10;
kana1.paino = 20;
kana1.kotkota();
cout << ”Tämän kanan paino on ” << kana1.paino << ”\n”;

kana2.ika = 4;
kana2.paino = 11;
kana2.kotkota();
cout << ”Tämän kanan paino on ” << kana2.paino << ”\n”;

}
```

Huomaa: tämä ohjelma ei toimi saantimääreiden takia; kaikki jäsenet ovat nyt private-tyyppiä (oletusarvo, jos saantimäärettä ei ole annettu), joten ulkopuolinen main()-funktio ei pääse käsiksi kyseisiin jäseniin!

Saantimääreet
Kaikki luokan jäsenet - tietojäsenet ja metodit - ovat yksityisiä (private) oletusarvoltaan. Yksityisiä jäseniä voidaan käsitellä vain luokan omien metodien avulla. Julkisia (public) jäseniä voidaan käsitellä minkä tahansa luokan olion kautta tai missä ulkopuolisessa funktiossa tahansa.

Huomaa, että myös main() on ulkopuolinen funktio, joka pääsee käsiksi luokan jäseniin saantimääreen mukaan.

Seuraavana saantimääreet ovat mukana luokan esittelyssä:

Kana-luokka:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
public:
  kotkota()
};
```

Nyt jäsenmuuttujat ika ja paino ovat private-tyyppiä ja kotkota() taas public-tyyppiä.

Kokeile nyt ohjelmaa uudelleen: se ei toimi vieläkään, koska main()-funktiossa yritetään päästä suoraan käsiksi private-tyyppisiin jäsenmuuttujiin ika ja paino. Funktio kotkota() on taas public-tyyppiä, joten se ei aiheuta herjauksia.

Luokan esittelyä olisi muutettava seuraavasti, jos main() halutaan pitää alkuperäisenä:

Toimiva Kana-olioiden käsittely:

```
#include <iostream.h>

class Kana
{
public:
  unsigned int ika;
  unsigned int paino;
  void kotkota()
};

// kotkota()-funktion määrittely
void Kana::kotkota()
{
cout << ”Kaakati kaa \n”;
}

void main()
{
kana1.ika = 10;
kana1.paino = 20;
kana1.kotkota();
cout << ”Tämän kanan paino on ” << kana1.paino << ”\n”;

kana2.ika = 4;
kana2.paino = 11;
kana2.kotkota();
cout << ”Tämän kanan paino on ” << kana2.paino << ”\n”;

}
```

Nyt kaikki jäsenet ovat public-tyyppiä, joten main() pääsee niihin suoraan käsiksi. Esittely on kuitenkin hieman huono, koska nyt kaikki ulkopuoliset funktiot ja luokat pääsevät käsiksi Kana-luokan jäsenmuuttujiin suoraan.

Jos suoraa pääsyä ei sallita, tulisi luokan esittelyssä olla metodeita, joiden kautta päästään jäsenmuuttujiin käsiksi. Näitä metodeita kutsutaan saantimetodeiksi.

Nyt Kana-luokassa ei ole esitelty metodeita, jotka voisivat muuttaa jäsenmuuttujia ja palauttaa (tai tulostaa) jäsenmuuttujien arvot. Hyvässä luokassa on aina sellaisia metodeja.

Jäsenfunktioiden määrittely
Jäsenfunktion määrittely alkaa aina luokan nimellä, jota seuraa kaksi kaksoispistettä, funktion nimi sekä parametrit.

Määrittelemme seuraavana Kana-luokan kotkota()-funktion:

```
void Kana::kotkota()
{
cout << ”Kotkoti kot \n”;
}
```

Nyt voisimme lisätä Kana-luokkaan 4 metodia, jotka toteuttavat seuraavat toiminnot:
1 metodi palauttaa Kana-olion iän
1 metodi palauttaa Kana-olion painon
1 metodi asettaa Kana-olion iän
1 metodi asettaa Kana-olion painon

Tietenkin voisimme asettaa iän ja painon samalla metodilla.

Kana-luokan esittelyyn tulee nyt siis lisätä nuo neljän metodin esittelyt. Sitten metodit tulee määritellä.

Päivitetty Kana-luokka:

```
class Kana
{
public:
  unsigned int ika;
  unsigned int paino;
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};
```

Nyt uudet metodit tulee esitellä:

```
int Kana::palautaika()
{
return ika;
}

int Kana::palautapaino()
{
return paino;
}

void Kana:: asetaika(int i)
{
ika = i;
}

void Kana::asetapaino(int p)
{
paino = p;
}
```

Ja koko ohjelma: Kana-luokka metodeineen:

```
#include <iostream.h>

class Kana
{
public:
  unsigned int ika;
  unsigned int paino;
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};

void Kana::kotkota()
{
cout << ”Kotkoti kot \n”;
}


int Kana::palautaika()
{
return ika;
}
```

```
int Kana::palautapaino()
{
return paino;
}

void Kana:: asetaika(int i)
{
ika = i;
}

void Kana::asetapaino(int p)
{
paino = p;
}

void main()
{
Kana kana1;
kana1.paino = 20;
kana1.asetaika(4);
cout << ”Kanan ikä on ” << kana1.palautaika() << ”\n”;
cout << ”Kanan paino on ” << kana1.palautapaino() << ”\n”;
}
```

Nyt luodaan ensin Kana-olio ja asetetaan sen painoksi 20 suoraan jäsenmuuttujaa käsittelemällä. Sen jälkeen asetetaan ikä metodin avulla.
Lopuksi tulostetaan ikä ja paino käyttäen metodeita, jotka palauttavat arvot.

Huomaa, että nyt kaikki Kana-luokan jäsenet ovat public-tyyppiä. Tällöin ulkopuoliset funktiot ja luokat pääsevät suoraan käsiksi jäseniin.
Näet, että main() pääsee suoraan käsiksi esimerkiksi paino-jäsenmuuttujaan lauseessa

```
kana1.paino = 20;
```

Se ei ole hyvä ratkaisu, koska luokan jäsenmuuttujat on hyvä suojata ulkopuolisten käsittelyltä. Jäsenmuuttujien suora käsittely tulisi estää; niitä tulee voida käsitellä vain luokan omien metodien kautta, koska tällöin voimme esimerkiksi tarkistaa, ovatko annetut tiedot oikealta arvoalueelta (esimerkiksi *ikä* > 0) tai oikeaa tyyppiä (esimerkiksi *nimi* on merkkijono).

Teemmekin nyt seuraavat muutokset Kana-luokkaan:

- Laitamme jäsenmuuttujat private-tyyppisiksi.
- Annamme metodien olla public-tyyppisiä.

Kana-luokka:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
public:
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};
```

Nyt esimerkkilistauksen main()-ohjelmaan tulee tehdä seuraava muutos:
lauseen, jossa jäsenmuuttujaa käsitellään suoraan eli

```
kana1.paino = 20;
```

sijaan tulee käyttää metodia seuraavasti:

```
kana1.asetapaino(20);
```

Nyt luokka on hyvä perusesimerkki:
Yleensä jäsenmuuttujat suojataan laittamalla ne private-tyyppisiksi.
Luokkaan sijoitetaan public-metodeja, joiden kautta jäsenmuuttujien käsittely on mahdollista.

Muodostimet ja tuhoajat

Normaalin muuttujan esittelyn yhteydessä voidaan suorittaa myös muuttujan alustus:

```
int massa = 200;
```

Myös luokkien jäsenmuuttujat voidaan alustaa silloin, kun olio luodaan. Siihen käytetään luokan muodostinta. Luokan muodostin ajetaan aina, kun ohjelmassa luodaan luokan olio.

Esimerkkilistauksessa luotiin Kana-luokan olio seuraavasti:

```
Kana kana1;
```

Tällöinkin käytettiin muodostinta, vaikka sitä ei näy missään. Kyseessä on **oletusmuodostin**, jonka C++-kääntäjä antaa käyttöön automaattisesti. Voimme luoda myös oman oletusmuodostimen ja myös muita muodostimia.
Jos esittelemme oman muodostimen luokan esittelyssä, se tulee myös määritellä.

Ajattele muodostinta ikään kuin metodina, joka poikkeaa muista metodeista siten, että se luo olion.

Muodostin voi ottaa parametreja, mutta se ei koskaan palauta arvoa - ei edes void-arvoa. Muodostin on luokan metodi, jolla on sama nimi kuin luokalla itsellään.

Kana-luokan oletusmuodostin olisi nimeltään:

```
Kana();
```

Oletusmuodostin on muodostin, joka ei ota parametreja. Meillä voi siis olla parametrisoituja muodostimia, joille annetaan kutsun yhteydessä parametreja, joilla luokan jäsenmuuttujat alustetaan.

Aina, kun esittelet muodostimen, sinun on esiteltävä myös tuhoaja. Aivan samoin kuin muodostin luo ja alustaa luokan jäseniä, tuhoaja puhdistaa paikat ja vapauttaa varatun muistin. Tuhoajalla on aina luokan nimi, jota edeltää tilde-merkki (~). Tuhoaja ei ota argumentteja eikä palauta arvoa.

Kana-luokan tuhoaja olisi nimeltään

```
~Kana();
```

Laitamme nyt Kana-luokkaan muodostimen ja tuhoajan:

Kana-luokan muodostin ja tuhoaja:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
public:
  Kana();
  ~Kana();
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
```

```
  void asetapaino(int p);
};
```

Nyt muodostin ja tuhoaja tulee määritellä:

```
Kana::Kana()
{
}

Kana::~Kana()
{
}
```

Nyt muodostimen ja tuhoajan rungoissa ei ole mitään koodia; yleensä tuhoaja on aina tyhjä, mutta muodostimeen voimme sijoittaa esimerkiksi jäsenmuuttujien alustaminen oletusarvoilla:

```
Kana::Kana()
{
ika = 5;
paino = 10;
}
```

Huomaa, että vaikka jäsenmuuttujat nyt alustetaankin, ei kyseessä ole parametrisoitu muodostin, koska muodostimen otsikossa ei ole parametreja. Kyseessä on edelleenkin oletusmuodostin.

Nyt oliota luotaessa alustetaan jäsenmuuttujat automaattisesti.

Kana-ohjelma, jossa on muodostimet:

```
#include <iostream.h>

class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
public:
  Kana();
  ~Kana();
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};
```

```
Kana::Kana()
{
ika = 5;
paino = 10;
}

Kana::~Kana()
{
}


void Kana::kotkota()
{
cout << ”Kotkoti kot \n”;
}


int Kana::palautaika()
{
return ika;
}

int Kana::palautapaino()
{
return paino;
}

void Kana:: asetaika(int i)
{
ika = i;
}

void Kana::asetapaino(int p)
{
paino = p;
}

void main()
{
Kana kana1;
cout << ”Kanan ikä on ” << kana1.palautaika() << ”\n”;
cout << ”Kanan paino on ” << kana1.palautapaino() << ”\n”;
}
```

Inline-metodit

Inline-funktiot ovat funktioita, joihin ohjelmassa ei hypätä, vaan funktion koodi puretaan kääntämisen yhteydessä kutsujan koodialueelle.

Inline on siis nopeampi suorittaa kuin tavallinen funktio, jonka koodi on eri muistialueella. Yleensä inline-funktio on lyhyt funktio, joka lisäksi toistetaan useampia kertoja.

Inline-funktioita voidaan käyttää luokan metodien kohdalla. Lyhytrunkoinen metodi voidaan laittaa inline-tyyppiseksi kirjoittamalla funktion runko suoraan luokan esittelyyn. Tällöin metodin erillistä esittelyä ei tarvita.

Muutamme nyt Kana-luokan muodostimen ja tuhoajan inline-tyyppisiksi:

Kanalla inline-metodeita:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
public:
  Kana() { ika = 5; paino = 10; };
  ~Kana() { };
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};
```

Huomaa, että jokin metodi voidaan laittaa inline-tyyppiseksi myös siten, että se esitellään ensin luokan esittelyssä ja määritellään sitten luokan ulkopuolella käyttäen varattua sanaa inline metodin otsikon edessä.

Muut muodostimet
Kääntäjä tarjoaa käyttöön oletusmuodostimen ja -tuhoajan. Jos oma muodostin esitellään, on hyvä ohjelmointitapa esitellä myös tuhoaja.

Oletusmuodostin ei ota parametreja. Sen sijaan voidaan muodostaa parametrisoitu muodostin, jolle annetaan parametrit.

Parametrisoitu muodostin
Luokassa voi olla useita erilaisia muodostimia. Ne tarjoavat käyttäjälle enemmän mahdollisuuksia luoda olioita. Käyttäjä voi haluta luoda olioita ilman mitään alustuksia tai sitten antaa olion luomisen yhteydessä arvot jäsenmuuttujiin.

Lisäksi ohjelmaa ajettaessa voi tulla tilanteita, jolloin tarvitaan nimenomaan oletusmuodostinta, jolloin olioiden jäsenmuuttujien arvoja ei vielä tiedetä tai haluta antaa olion luontivaiheessa.

Parametrisoitu muodostin on muodostin, joka ottaa parametreja. Luomme nyt Kana-luokkaan muodostimen, jolle annetaan kaksi parametria, joiden arvot sijoitetaan Kana-olion jäsenmuuttujiin luomisen yhteydessä.

Kanalla parametrisoitu muodostin:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
public:
  Kana() { ika = 5; paino = 10; };
  Kana(int ik, int pa);
  ~Kana() { };
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};

Kana::Kana(int ik, int pa)
{
ika = ik;
paino = pa;
}
```

Kutsu main()-funktiossa olisi nyt:

```
Kana kana1(8, 12);  // parametrisoitua muodostinta käytetään
Kana kana2;  // oletusmuodostinta käytetään
```

Kanalla voi olla vielä **kopiomuodostin**, jolle viedään usein vakioviittaus kopioitavaan olioon parametrina. Viittaus on vakio siksi, että kopioitavalle oliolle ei tehdä mitään muutoksia.

Oletusmuodostimen ja tuhoajafunktion lisäksi kääntäjällä on käytössään myös oletuskopiomuodostin. Kopiomuodostinta kutsutaan joka kerta, kun oliokopio tehdään. Kun olio viedään arvona, joko funktioon tai funktion palautuksena, luodaan tilapäinen kopio oliosta. Jos olio on käyttäjän määrittelemä, kutsutaan luokan

kopiomuodostinta. Kaikki kopiomuodostimet ottavat yhden parametrin: viittauksen saman luokan olioon. On hyvä laatia vakioviittaus, koska muodostimen ei tule muuttaa sille vietyä oliota. Esimerkiksi:

```
Kana(const Kana &kana);
```

Oletuskopiomuodostin yksinkertaisesti kopioi sille viedyn olion jäsenmuuttujat uuden olion jäsenmuuttujiksi. Kyseessä on jäsenkohtainen kopio ja vaikka tämä menettely toimiikin useimpien jäsenmuuttujien kohdalla, se kaatuu helposti sellaisten jäsenmuuttujien kohdalla, jotka ovat osoittimia vapaaseen muistitilaan.

Pinnallinen ja tarkka kopio:
Jäsenkohtainen kopiointimenettely kopioi olion jäsenmuuttujien tiedot toiseen olioon. Tarkka (syvä) kopio sen sijaan kopioi keossa olevat arvot uudelleen varattuun muistiin.
Oma kopiomuodostin on siis tarpeellinen silloin, kun jäsenmuuttujina on osoittimia; tällöin voidaan taata, että kopion jäsenmuuttujille varataan omat muistialueensa eivätkä osoittimet osoita kahdessa oliossa samaan muistialueeseen.

Luokan esittelyjen ja metodien määrittelyjen organisointi
Yleensä kaikki C++-työkalut käyttävät projektimuotoa. Projektiin liittyy useita erilaisia tiedostoja ja kääntämisen ja linkittämisen lopputuloksena syntyy suoritettava tiedosto, jolla on projektin nimi.
Projektin avulla on helppo hallita laajaa tiedostomäärää ja tutkia tiedostoja.

Yleensä luokan esittely sijoitetaan omaan otsikkotiedostoonsa, jonka tunnus on .h tai .hpp ja nimenä luokan nimi. Metodien määrittelyt sijoitetaan sitten omaan lähdekooditiedostoonsa, jolla on luokan nimi ja tunnuksena .cpp.

Lopuksi itse pääohjelma on omassa tiedostossaan.

Tällainen järjestely on looginen ja helpommin hallittavissa. Nyt tulee aina muistaa #include-lauseet ja niiden oikea järjestys.

Toteutamme nyt Kana-luokan organisoinnin edellä kuvatulla tavalla.

Kana.h sisältää seuraavan koodin:

```
class Kana
{
private:
```

```
  unsigned int ika;
  unsigned int paino;
public:
  Kana() { ika = 5; paino = 10; };
  Kana(int ik, int pa);
  ~Kana() { };
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
};
```

Kana.cpp on seuraavanlainen:

```
include <iostream.h>
include ”Kana.h”

Kana::Kana(int ik, int pa)
{
ika = ik;
paino = pa;
}

void Kana::kotkota()
{
cout << ”Kotkoti kot \n”;
}


int Kana::palautaika()
{
return ika;
}

int Kana::palautapaino()
{
return paino;
}

void Kana:: asetaika(int i)
{
ika = i;
}

void Kana::asetapaino(int p)
{
paino = p;
}
```

Lopuksi pääohjelma kanamain.cpp:

```
include ”Kana.cpp”
void main()
{
Kana kana1(8, 12);  // parametrisoitua muodostinta käytetään
Kana kana2;  // oletusmuodostinta käytetään
cout << ”Kanan ikä on ” << kana1.palautaika() << ”\n”;
cout << ”Kanan paino on ” << kana1.palautapaino() << ”\n”;
cout << ”Kanan ikä on ” << kana2.palautaika() << ”\n”;
cout << ”Kanan paino on ” << kana2.palautapaino() << ”\n”;

}
```

Huomaa seuraavat lauseet edellisissä listauksissa:

```
#include <iostream.h>
#include ”Kana.cpp”
```

Jos #include-lauseessa käytetään kulmasulkumerkkejä, haetaan tiedostoa standardista kansiosta eli kansiosta, johon asetukset viittaavat. Yleensä kyseessä on työkalun asennuksen yhteydessä luotu kansio.
Jos #include-lauseessa käytetään lainausmerkkejä, haetaan tiedostoa oletuskansiosta eli yleensä siitä kansiosta, jossa itse ohjelmakin on. Muutoin lainausmerkkien sisälle voi tarvittaessa kirjoittaa koko hakemistopolun. Voit kirjoittaa lisäpolkuja myös asetuksiin.

Metodien ylimäärittely ja oletusparametrit
Tavallisia funktioita voidaan ylimääritellä ja niiden parametreille voidaan antaa oletusarvoja.

Ylimäärittely tarkoittaa sitä, että ohjelmassa on useita samannimisiä funktioita, joilla kuitenkin voi olla erilaiset palautustyypit ja parametrit.

Oletusparametrien käyttö tarkoittaa sitä, että jollakin funktion parametrilla (tai useallakin) voi olla oletusarvo, jota käytetään silloin, kun funktion kutsun yhteydessä ei parametrille anneta mitään arvoa. Oletusarvot annetaan esittelyssä ja aina kaikille parametriluettelon koko loppuosan parametreille.

Ylimäärittelyä ja oletusparametreja voidaan käyttää myös luokkien metodien kohdalla.

Lisäämme nyt Kana-luokkaan jäsenmuuttujan vari ja metodin asetavari(). Oletetaan, että suurin osa kanoista on valkoisia, jolloin asetavari()-metodin oletusarvona voi olla merkkijono “valkoinen”. Myös muodostimiin voidaan tehdä vastaavat muutokset.

Oletusparametrit:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
  char * vari;
public:
  Kana() { ika = 5; paino = 10; };
```

```
  Kana(int ik, int pa);
  ~Kana() { };
  void kotkota()
  int palautaika();
  int palautapaino();
  void asetaika(int i);
  void asetapaino(int p);
  void asetavari(char * va);
};

void Kana::asetavari(char * va = ”valkoinen”)
{
vari = va;
}
```

Jos kutsu on seuraavanlainen:

```
kana1.asetavari();
```

käytetään oletusarvoa “valkoinen”, joka sijoitetaan jäsenmuuttujan vari arvoksi.

Jos kutsu on seuraavanlainen:

```
kana1.asetavari(“harmaa”);
```

käytetään arvoa “harmaa”, joka sijoitetaan jäsenmuuttujan vari arvoksi.

Metodin ylimäärittelyn esimerkkinä voisimme käyttää tulosta()-metodia, joka tulostaa iän, painon tai kaikki arvot riippuen kutsun yhteydessä annetuista parametreista.

Kana-luokassa ylimääritelty metodi:

```
class Kana
{
private:
  unsigned int ika;
  unsigned int paino;
  char * vari;
public:
  Kana() { ika = 5; paino = 10; };
  Kana(int ik, int pa);
  ~Kana() { };
  void kotkota()
  int palautaika();
  int palautapaino();
```

```
  void asetaika(int i);
  void asetapaino(int p);
  void asetavari(char * va);
  void tulosta();
  void tulosta(int x);
};

void Kana::tulosta()
{
cout << "Ikä on " << ika << " paino on " << paino << "\n";
}

void Kana::tulosta(int i)
{
cout << "Ikä on " << ika << " paino on " << paino << " ja väri on "
<< vari << "\n";
}
```

Nyt kutsu

```
kana1.tulosta();
```

tulostaa iän ja painon.

Kutsu

```
kana1.tulosta(1);
```

tulostaa kaikki arvot.

Metodi tulosta() on siis ylimääritelty. Huomaa, että ylimäärittely (overload) on eri asia kuin korvaaminen (override), jota käytetään johdettaessa luokasta uusia luokkia. Korvaamisessa kantaluokan metodi kirjoitetaan uudelleen aliluokissa, jolloin aliluokka käyttää kantaluokan metodin sijaan omaa metodiaan, vaikka metodit ovatkin saman nimisiä.

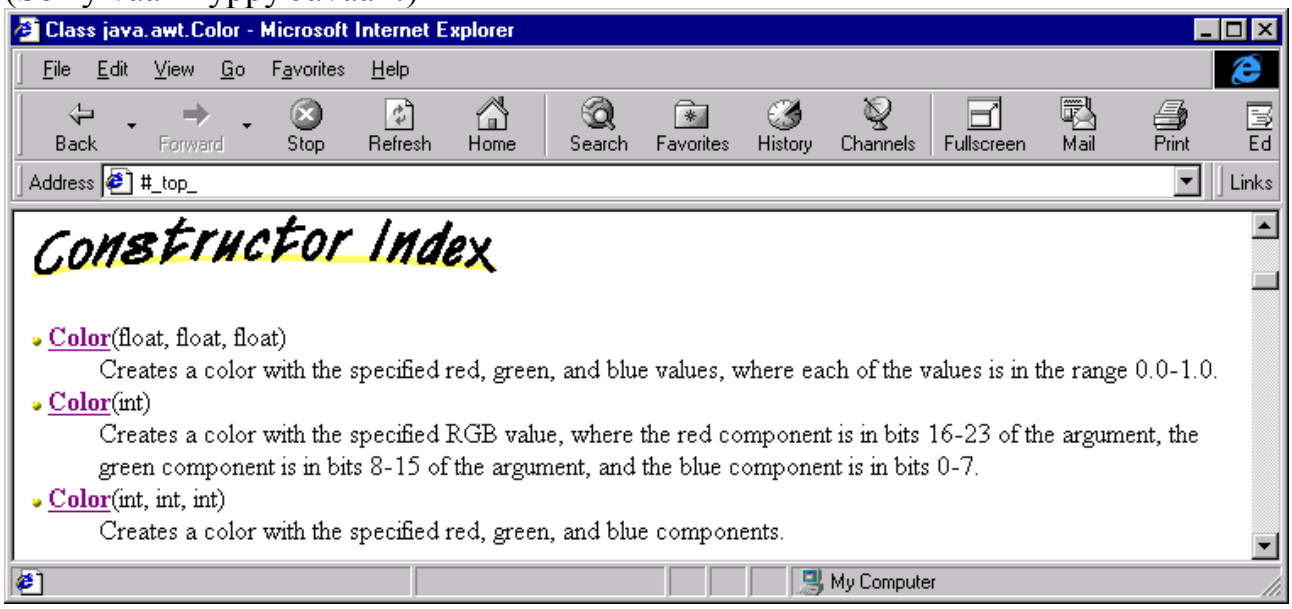
Muodostin ylimääritellään hyvin usein: luokassa on oletusmuodostin, joka ei ota parametreja sekä yksi tai useampi parametrisoitu muodostin. Näin luokan olio voidaan luoda usealla eri tavalla: joskus annetaan parametreja, joiden arvot sijoitetaan olion jäsenmuuttujiin ja joskus taas muodostetaan olio ilman mitään parametreja (eli käytetään oletusmuodostinta).

Ajattele esimerkiksi luokkaa nimeltä Väri:
Ohjelmassa käyttäjän tulisi pystyä luomaan Väri-olio esimerkiksi seuraavilla tavoilla:

Annetaan muodostimen parametrina jokin tietty vakioväri.
Annetaan parametrina RGB-arvot kokonaislukuina.

Annetaan väri heksadesimaalimuodossa.
Luodaan Väri-olio antamatta aluksi mitään väriarvoja.

Esimerkiksi Java-kielessä on Color-luokalla useita erilaisia muodostimia:
(Sorry vaan hyppy Javaan!)



Olioiden luonti vapaaseen muistiin
Samalla lailla kuin int-osoittimen voit luoda minkä tahansa muunkin tyyppisen osoittimen. Jos olet esitellyt tyyppiä Kana olevan olion, voit luoda tuohon luokkaan osoittavan osoittimen ja luoda Kana-olion vapaalle muistialueelle samalla lailla kuin pinoonkin.
Syntaksi on samanlainen kuin int-osoittimien kohdalla:

```
Kana *pKana = new Kana;
```

Huom! Perinteisesti on osoitinmuuttujan nimen ensimmäinen kirjain p tai esimerkiksi o, jotta muuttuja erotettaisiin osoitinmuuttujaksi.

Lause kutsuu oletusmuodostinta - muodostinta, joka ei ota parametreja. Muodostinta kutsutaan aina, kun olio luodaan - joko pinoon tai vapaaseen muistiin.

Olioiden jäsenmuuttujia käsitellään nuolioperaattoria käyttäen (muistanet, että luotaessa olio pinoon käytettiin pisteoperaattoria jäsenten käsittelyssä).

Osoitinoperaattori (->) on siis väliviivan ja suurempi-merkin yhdistelmä.
C++ käsittelee merkkejä yhtenä symbolina.

Esimerkiksi:

```
Kana *pKana = new Kana;
pKana->ika = 20;
```

this-osoitin
Jokaisella luokan jäsenfunktiolla on kätketty parametri: this-osoitin. Se osoittaa yksittäiseen, nykyiseen (aktiiviseen) olioon. Siksi jokaisen metodin kutsussa on mukana this-osoitin kätkettynä parametrina.

this-osoittimen tehtävänä on osoittaa yksittäistä oliota, jonka metodia käytetään. Yleensä sitä ei tarvita; voit vain kutsua metodia ja asettaa jäsenmuuttujia. Joskus on kuitenkin voitava käsitellä itse oliota (esimerkiksi nykyisen olion osoittimen käyttöä varten). Juuri tällaisessa tilanteessa on this-osoitin tarpeellinen.
this-osoitinta ei tarvitse itse luoda tai tuhota; kääntäjä huolehtii noista toimenpiteistä.

const-osoitin ja metodi
Jäsenfunktio voi olla const-tyyppinen. Kun funktio esitellään const-tyyppisenä, kääntäjä antaa virheilmoituksen, jos tuon funktion sisällä yritetään muuttaa kohteen tietoa.

Jos osoitin on const-tyyppinen, voidaan vain const-tyyppisiä metodeita kutsua tuota osoitinta käyttäen.

const-määrittelyä käyttämällä varmistetaan, ettei olion tietoja muuteta vahingossa.

Olioiden vieminen funktioille
Aina, kun kohde viedään funktiolle arvona, tehdään kopio kohteesta. Ja aina, kun kohde palautetaan funktiosta arvona, tehdään myöskin kopio.

Käytettäessä laajoja, käyttäjien määrittelemiä kohteita, aiheuttaa kopiointi menetyksiä. Tällöin käytetään enemmän muistia kuin tarvitaan ja ohjelma on hitaampi.

Käyttäjän luoman oliokohteen koko pinossa on jäsenmuuttujien summa. Nuo jäsenmuuttujat voivat vuorostaan olla käyttäjän luomia olioita. Kun tuollainen massiivinen rakenne kopioidaan pinoon, on selvää, että suorituskyky huononee ja muistia tuhlaantuu runsaasti.

Tärkeitä perusasioita koskien olioiden vientiä funktioihin

Seuraavana palautetaan mieliin kopiomuodostin ja hyödynnetään samalla string-luokkaa.

Olion vienti kopiomuodostimelle:

```
#include <iostream.h>
#include <conio.h>
#include <string>

class Koira
{
public:
Koira();
Koira(const Koira &k);
void toimi(){cout << "hauku\n";}
int ika;
string merkki;
};

Koira::Koira(const Koira &k)
{
ika = k.ika;
merkki = k.merkki;
}

Koira::Koira()
{
ika = 10;
merkki = "Matala";
}


main()
{
Koira Teppo;
Teppo.toimi();
Koira Patu = Teppo;
Patu.toimi();
Patu.merkki ="Karvakas";
cout << Patu.merkki << endl;
Patu = Teppo;
cout << Patu.merkki<< endl;

Koira Erkki(Patu);
Koira Minna(Patu);
Koira Pinna(Patu);
// Tarkistetaan, että tilaa varataan muistista uusille jäsenille

cout << &(Erkki.merkki) << endl;
```

```
cout << &(Minna.merkki) << endl;
cout << &(Pinna.merkki) << endl;
getch();
}
```

Huomaa, että ohjelmaan tuodaan string-luokka lauseella

```
#include <string>
```

Kopiomuodostimen parametrina on viittaus vakioon olioon. Tuo olio on se olio, josta kopio tehdään. Siksi sitä ei tietenkään saa muuttaa. Kun kopiomuodostimelle viedään viittaus, ei viennin yhteydessä tehdä vietävästä oliosta kopiota (kuten tehdään vietäessä olio arvona). Olisikin jopa ihmeellistä, että kopiomuodostimeen vietävästä oliosta tehtäisiin kopio; kopiomuodostinhan on itse juuri kopioimista varten. Lisäksi turhien kopioiden tekeminen vie aina tehokkuutta ja muistia.

Olioiden vienti funktioihin

Funktion parametri voi olla

tavallinen arvoparametri (olion nimi)
viittaus olioon
osoitin olioon

Olion vienti funktioon

Olion vienti funktioon viittausparametrien kautta:

```
#include <iostream.h>
#include <conio.h>
#include <cstring.h>


class Koira
{
public:
Koira();
Koira(const Koira &k);
void toimi(){cout << "hauku\n";}
int ika;
char * merkki;
};
```

```
Koira::Koira(const Koira &k)
{
ika = k.ika;
merkki = new char;
strcpy(merkki,k.merkki);
}

Koira::Koira()
{
ika = 10;
merkki = new char;
strcpy(merkki,"Tavallinen");
}

void Vanhene(Koira & kk);

main()
{
Koira Teppo;
Teppo.toimi();
cout << Teppo.ika << endl;
Vanhene(Teppo);
cout << Teppo.ika << endl;
getch();
}

void Vanhene(Koira & kk)
{
kk.ika++;
}
```

Olion vienti funktioon osoitinparametrien kautta:

```
#include <iostream.h>
#include <conio.h>
#include <cstring.h>


class Koira
{
public:
Koira();
Koira(const Koira &k);
void toimi(){cout << "hauku\n";}
int ika;
char * merkki;
};
```

```
Koira::Koira(const Koira &k)
{
ika = k.ika;
merkki = new char;
strcpy(merkki,k.merkki);
}

Koira::Koira()
{
ika = 10;
merkki = new char;
strcpy(merkki,"Tavallinen");
}

void Vanhene(Koira * kk);

main()
{
Koira Teppo;
Teppo.toimi();
cout << Teppo.ika << endl;
Vanhene(&Teppo);
cout << Teppo.ika << endl;
getch();
}

void Vanhene(Koira * kk)
{
kk->ika++;
}
```

Entä, jos viedään olio arvona:

(Laita tulostusrivi kopiomuodostimeen ja tutki, montako kopiota tehdään.)

```
#include <iostream.h>
#include <conio.h>
#include <cstring.h>


class Koira
{
public:
Koira();
Koira(const Koira &k);
void toimi(){cout << "hauku\n";}
int ika;
char * merkki;
};
```

```
Koira::Koira(const Koira &k)
{
ika = k.ika;
merkki = new char;
strcpy(merkki,k.merkki);
}

Koira::Koira()
{
ika = 10;
merkki = new char;
strcpy(merkki,"Tavallinen");
}

void Vanhene(Koira kk);

main()
{
Koira Teppo;
Teppo.toimi();
cout << Teppo.ika << endl;
Vanhene(Teppo);
cout << Teppo.ika << endl;
getch();
}

void Vanhene(Koira kk)
{
kk.ika++;
}
```

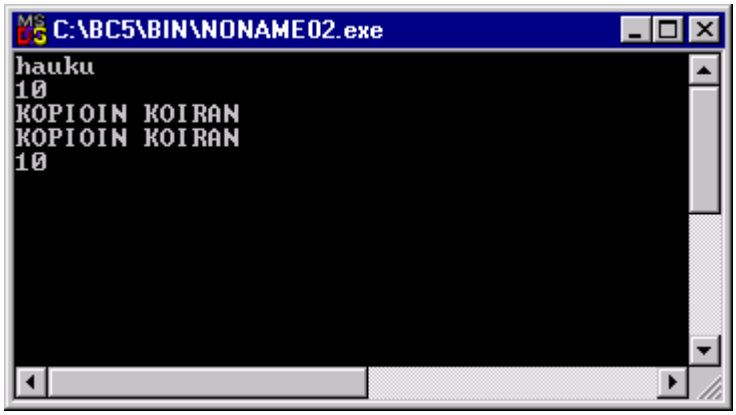
Entä, jos palautetaan Koira-olio:

(Laita tulostusrivi kopiomuodostimeen ja tutki, montako kopiota tehdään.)

```
#include <iostream.h>
#include <conio.h>
#include <cstring.h>


class Koira
{
public:
Koira();
Koira(const Koira &k);
void toimi(){cout << "hauku\n";}
int ika;
char * merkki;
```

```
};

Koira::Koira(const Koira &k)
{
cout << "KOPIOIN KOIRAN \n";
ika = k.ika;
merkki = new char;
strcpy(merkki,k.merkki);
}

Koira::Koira()
{
ika = 10;
merkki = new char;
strcpy(merkki,"Tavallinen");
}

Koira Vanhene(Koira kk);

main()
{
Koira Teppo;
Teppo.toimi();
cout << Teppo.ika << endl;
Vanhene(Teppo);
cout << Teppo.ika << endl;
getch();
}

Koira Vanhene(Koira kk)
{
kk.ika++;
return kk;
}
```

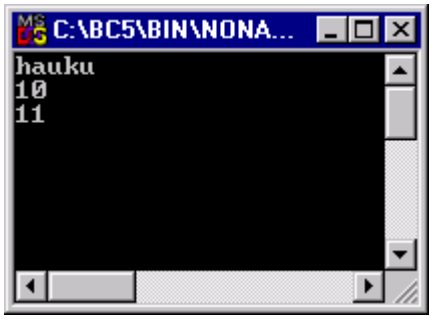
Kun laitat tulostusrivin kopiomuodostimeen, saat selville, kuinka monta kertaa (ja missä vaiheissa) tehdään kopioita olioista:



Näit samalla, että viety olio ei todellisuudessa muuttunut.

Kun funktion palautusarvona on osoitin olioon, ei kopioita tehdä:

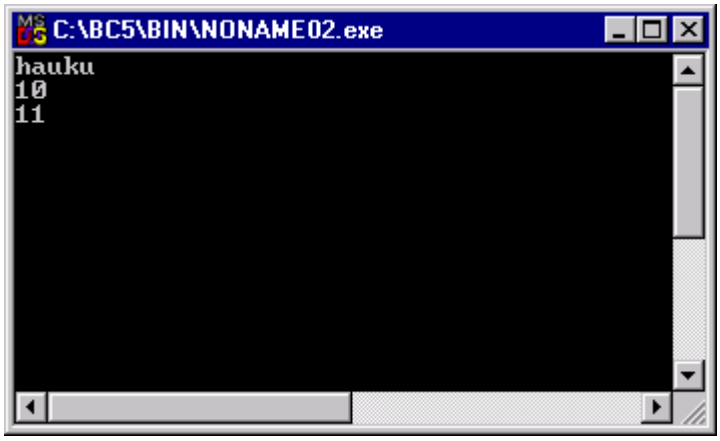
```
Koira* Vanhene(Koira *kk);

main()
{
Koira Teppo;
Teppo.toimi();
cout << Teppo.ika << endl;
Vanhene(&Teppo);
cout << Teppo.ika << endl;
getch();
}

Koira* Vanhene(Koira *kk)
{
kk->ika++;
return kk;
}
```



Kun funktion palautusarvona on osoitin olioon, ei kopioita myöskään tehdä:

```
Koira& Vanhene(Koira &kk);

main()
{
Koira Teppo;
Teppo.toimi();
cout << Teppo.ika << endl;
Vanhene(Teppo);
cout << Teppo.ika << endl;
getch();
}

Koira& Vanhene(Koira &kk)
{
```

```
kk.ika++;
return kk;
}
```



Olio käsitteenä, olioiden väliset suhteet ja mallintaminen

Luokkien käsitteitä

Kapselointi

Samaan tietotyyppiin kapseloidaan sisälle jäsenmuuttujat (ominaisuudet) ja jäsenfunktiot (metodit, toiminnot). Samalla jäsenet voidaan suojata halutulla tavalla.

Periytyminen, sisältyminen, kommunikointi

Jostakin pääluokasta johdetaan (periytetään) uusia luokkia (esimerkiksi eläin-luokasta voidaan johtaa kotieläin- ja villieläin-luokat. Johdettu luokka saa kaikki kantaluokkansa jäsenet ja voi lisätä uusia jäseniä tai muuttaa aiempia jäseniä. Sisältyminen tarkoittaa sitä, että toinen olio sisältyy toiseen olioon eli on tuon toisen olion jäsenmuuttujana.
Olioiden välistä kommunikointia tapahtuu tietenkin useissa ohjelmissa. Tällöin toisen olion jäsenmuuttujana on usein osoitin toiseen olioon.

Abstrakti luokka

Abstrakti luokka toimii vain kantaluokkana. Siitä ei johdeta omia olioita, vaan se toimii vain mallina ja pohjana muille luokille. Esimerkiksi Kuvio-luokka voisi toimia mallina, josta johdetaan Suorakaide- ja Ympyrä-luokat, jotka taas ovat konkreettisia (niillä on ala ja piiri jne). Abstrakti luokka syntyy, kun jokin luokan metodeista on puhdas virtuaalimetodi: se on määritelty luokan esittelyssä avainsanalla virtual ja asetettu arvoksi nolla, esimerkiksi:

```
virtual tulosta() = 0;
```

Oliopohjaisen ohjelman suunnittelusta:
Ohjelmiston tuotantoprosessissa on yleensä vaiheet:

- Vaatimukset ja tavoitteet selville
- Analyysi: haetaan kohteita ja kohteiden välisiä riippuvuuksia
- Suunnittelu: kohteista kehitetään luokat ja riippuvuudet (suhteet)
- Koodaus
- Testaus

Kaikki vaiheet dokumentoidaan. Analyysi ja suunnittelu tapahtuvat usein iteroivasti. Olioiden ja niiden suhteiden mallinnus on tärkeää ja toimii myös pohjana kokonaisuuden tarkastelussa. Protoilu on myös jatkuvasti lisääntynyt määrittelyjen tarkentumiskeinona.

Analyysi ja suunnittelu
Haetaan ohjelmiston ongelmakentästä kohteita (malleja), joista voidaan sitten luoda luokkia. Kuvataan kohteiden ominaisuudet, toiminnot ja kohteiden väliset riippuvuudet. Voidaan kuvata kaavioina ja/tai taulukkoina ja käyttää kuvaavaa tekstiä. (Pyritään myös aina hyödyntämään valmiita malleja.)

Keinoja löytää kohteita
Jos selviä kohteita ei löydy, niin voidaan kirjoittaa kuvaus ohjelman toiminnasta. Toimintakuvauksesta saadaan sitten poimittua selviä kohteita (substantiivit, verbit). Kohteita saattaa tulla liikaakin: voisiko joistakin kohteista tehdä jäsenmuuttujia.

Voidaan myös kirjoittaa käyttötilanteita, jotka ovat siis usein konkreettisessa käytössä ilmeneviä käyttötapauksia. Myös skenaarioita (mitä, jos?) voidaan tällöin kehittää. Tässä tulee selvemmin esille se, ovatko tietyt palvelut saatavilla eli mitä toimintoja luokan tulee tarjota asiakkailleen.

Luokkien väliset suhteet
Tähän mennessä olemme käsitelleet vain yksittäisiä luokkia. Kuitenkin ohjelmissa voi olla useita erilaisia luokkia, jotka lisäksi palvelevat ja tarvitsevat toisiaan. Näitä luokkien välisiä perussuhteita ovat:

Sisältyminen (has): jokin (tai useampi) luokka sisältyy toiseen luokkaan.
Esimerkiksi: auto-luokkaan sisältyy moottori-luokka.

Periytyminen (Kuuluminen (is)): Luokka kuuluu johonkin yleisempään luokkaan. Ylempi luokka on yliluokka (tai kantaluokka, perusluokka) ja alempi luokka aliluokka.
Esimerkiksi: Eläin-luokkaan kuuluu kotieläin-luokka. Kotieläin-luokka voi taas jakaantua aliluokkiin -> syntyy luokkahierarkia.

Tunteminen (knows, communicates, serves): Jokin luokka käyttää toisen luokan palveluja. Luokka ei kuitenkaan periydy toisesta luokasta eikä sisälly toiseen luokkaan. Luokat vain tarvitsevat toisiansa. Toteutetaan usein osoittimien avulla.

Luokkien kuvaustekniikka
Kun kyseessä on suurempi ohjelmisto, tulee analyysissä ja suunnittelussa käyttää kuvaamistekniikoita, joita on useita erilaisia.

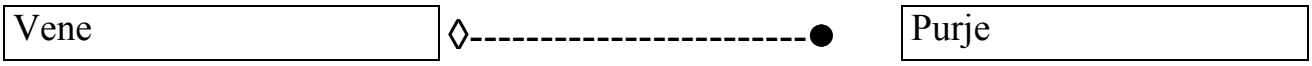
Esimerkki kuvaustekniikasta: OMT

Symbolit:
◊ ⭘ (nolla tai 1) ● (nolla tai usea) ∧

Sisältyminen (has) ◊

LUE: Tietokoneessa on suoritin. Suoritin on osa tietokonetta.
Tai: Tietokoneoliolla on suoritinkomponentti.
Tai: Tietokoneoliolla on jäsenmuuttuja (ominaisuus) -suoritin.

Täytetty pallo (●) merkitsee 'nolla tai enemmän'.



LUE: Veneessä on 0 tai enemmän purjetta.

Ympyrä (⭘) merkitsee (nolla tai 1)

LUE: Veneessä on 0 tai 1 moottoria.

****

Mitä sisältyminen (koosteluokka) tarkoittaa?

Kun luokka A sisältyy luokkaan B, on luokan B jäsenmuuttujina luokan A olioita (eli muuttujia, jotka ovat tyyppiä A).

Tällöin muodostettaessa luokan B olio, on luokan A olio automaattisesti sen osaoliona.

Luokan B metodit voivat kutsua luokan A metodeita ja näin tuleekin yleensä hyödyntää luokkaa A, koskapa useinkaan luokka B ei pääse suoraan käsiksi luokan A jäsenmuuttujiin.

Esimerkki sisältymisestä:
Luokkaan HIIHTO sisältyy KELLO-luokka, jonka avulla asetetaan hiihdon lähtö- ja saapumisajat.

Seuraavana ensin supistettu KELLO-luokka.

Tiedosto kello.hpp sisältää luokan KELLO määrittelyn:

```
class KELLO {
 public:
    void aseta(int tun, int min);
    void tulostaAika();
 private:
    int t, m;
 };
```

Tiedosto kello.cpp sisältää metodien määrittelyt:

```
void KELLO::aseta(int tun, int min)
{
  t = tun; m = min;
}
```

```
void KELLO::tulostaAika()
{
    cout << t << ':'  << m;
}
```

HIIHTO-luokka (ja pääohjelma) voisi olla seuraavanlainen:

```
#include <iostream.h>
#include <conio.h>
```

```
#include <string.h>
#include "kello.cpp"

class HIIHTO
{
public:
void alusta(char *tunnus, int Ltun, int Lmin, int Stun, int Smin);
void tulostaTiedot();
private:
KELLO lahto, saapu;
char hiihtotunnus[5];
};

void HIIHTO::alusta(char *tunnus, int Ltun, int Lmin, int Stun, int
Smin)
{
strcpy(hiihtotunnus,tunnus);
lahto.aseta(Ltun, Lmin);
saapu.aseta(Stun, Smin);
}

void HIIHTO::tulostaTiedot()
{
cout << hiihtotunnus << "\n";
cout << "LAHTÖAIKA on \n";
lahto.tulostaAika();
cout << "SAAPUMISAIKA on \n";
saapu.tulostaAika();
}

main()
 {
   HIIHTO reissu1;
   char tunn[] = "A20";
   int tt1, mm1, tt2, mm2;
   cout << "Anna lähtöaika" << endl;
   cin >> tt1 >> mm1;
   cout << "Anna tuloaika" << endl;
   cin >> tt2 >> mm2;

   reissu1.alusta(tunn,tt1, mm1, tt2, mm2);
   reissu1.tulostaTiedot();
   getch();
 }
```

Tunteminen (assosiatiivisuus) ●

Vene	omistaa ●---------- omistaja	Henkilö

Kuvassa kaksisuuntainen assosiaatio: "Veneellä on omistaja" sekä "Henkilö omistaa nolla tai useamman veneen".

Viivalla esiintyvät nimet ovat ns. roolinimiä.

Ohjelmassa suhde kuvataan osoittimin tai viittauksin.
Luokassa Vene voi olla jäsenmuuttujana osoitin olioon, joka kuuluu Henkilö-luokkaan.

Esimerkki: Tili-luokassa on viittaus Henkilö-luokkaan, joka sisältää tilinomistajan perustiedot.

Henkilö-luokka:

```
#include <iostream.h>

class hlo
{
public:
hlo();
hlo(char n[]){strcpy(nimi,n);};
char * tiedot() {return nimi;};

private:
char nimi[20];
};
```

Tili-luokka:

```
#include "hlo.cpp"

class Tili {
 public:
   Tili() : tilinro(0), saldo(0) {};
   void avaatili(long int t, hlo *pp);
   double asaldo() {return saldo;}
   void info();
   void tapahtuma (double summa);
   double korko() {return korkokanta;}
   void muutakorko(double uuskorko);
 private:
```

```
  long int tilinro;
  double saldo;
  hlo* omistaja;
  static double korkokanta;
};
```

Tilin määrittely:

```
#include "tili.h"
 #include <iostream.h>


 double Tili::korkokanta = 0.0; // staattinen-> määrittely

 void  Tili::avaatili(long int t, hlo* pp)
      {
      tilinro = t;
         omistaja = pp;
      }


 void Tili::info()
 {
 cout << "Saldo on " << saldo << "\n";
 cout << "Korkokanta on " << korkokanta << "\n";
 cout << "Omistaja on " << omistaja->tiedot() << "\n";
}

 void Tili::muutakorko(double uuskorko)
 {
   korkokanta = uuskorko;
 }


 void  Tili:: tapahtuma(double summa)
 {
   if (summa < 0 && saldo + summa < 0)
     cout << "TYHJÄ JO!" << endl;
   else
     saldo += summa;
 }
```

Pääohjelma:

```
#include <iostream.h>
#include "tili.cpp"
#include <conio.h>

main()
```

```
{
hlo o("kk");
hlo p("ii");
Tili k1;
k1.avaatili(555, &o);
k1.muutakorko(10.0);
k1. tapahtuma (10000);
cout << k1.korko() << "\n";
cout << k1.asaldo() << "\n";
k1. tapahtuma (10000);
cout << k1.korko() << "\n";
cout << k1.asaldo() << "\n";

Tili k2;
k2.avaatili(333, &p);
k2.muutakorko(12.0);
k2.tapahtuma(5000);
cout << k2.korko() << "\n";
cout << k2.asaldo() << "\n";
cout << k1.korko() << "\n";

k1.info();
k2.info();


getch();
}
```

Huomaa ohjelmassa myös Tili-luokan staattinen jäsenmuuttuja korkokanta. Staattisesta jäsenmuuttujasta syntyy ohjelman ajossa vain yksi ilmentymä, jonka kaikki luodut Tili-oliot jakavat. Tämä onkin järkevää, koska sama korkokanta koskee nyt kaikkia tilejä.

Esittelyt tapahtuivat näin:

Luokan esittelyssä:

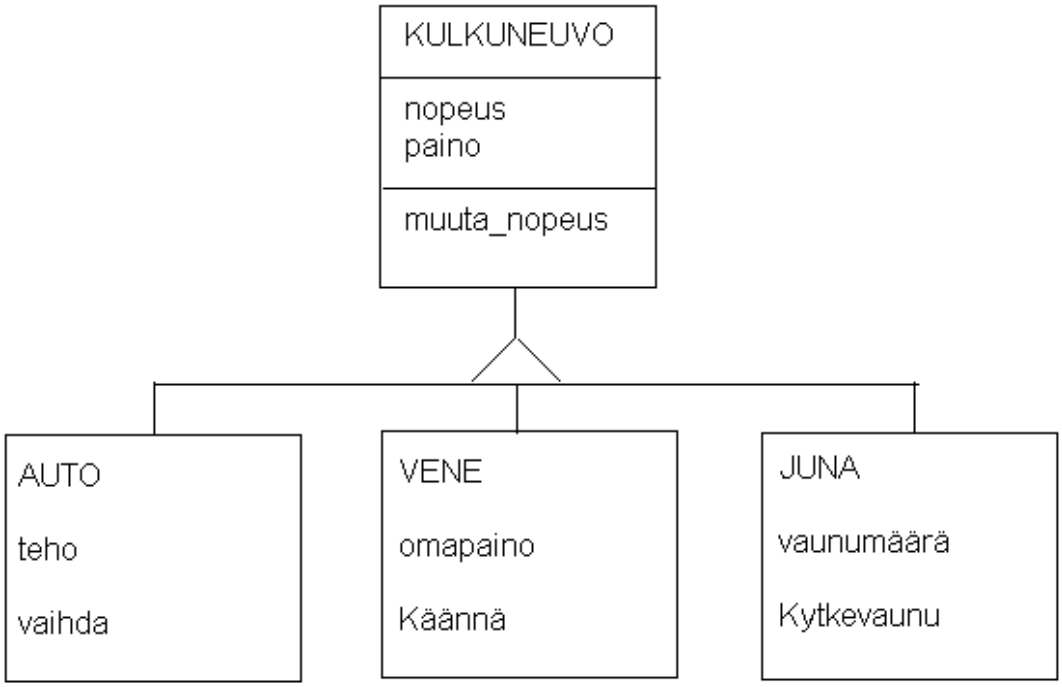
```
static double korkokanta;
```

Luokan metodien määrittelytiedostossa:

```
double Tili::korkokanta = 0.0; // staattinen muuttuja: määrittely
```

Periytyminen/Kuuluminen (is) ∧

Luokka periytyy toisesta luokasta.
Johdettu luokka saa kantaluokkansa jäsenet ja voi muokata jäseniä mieleisikseen sekä lisätä itselleen ominaisia jäseniä.



Tietenkin luokkasuhteet voivat mennä ristiin: mukana voi olla johdettuja luokkia, joista jotkut sisältävät muita luokkia ja jotkut kommunikoivat joidenkin luokkien kanssa.

Periytymisessä johdetaan kantaluokasta (perusluokasta) uusi luokka, aliluokka (johdettu luokka).

Aliluokka saa kaikki kantaluokan jäsenet ja voi esitellä lisäjäseniä tai kirjoittaa uusiksi kantaluokan metodeita.

Jos kantaluokan jäsenet ovat protected-tyyppisiä, voidaan kantaluokan jäseniä käsitellä suoraan aliluokassa.

Luotaessa johdetun luokan olio, kutsutaan sekä kantaluokan että aliluokan muodostinta ja vastaavasti tuhoajaa.

Jos kantaluokan jäseniä on alustettava johdetun luokan muodostimen kautta, voidaan johdetun luokan muodostimessa kutsua kantaluokan alustavaa muodostinta tai sijoittaa arvot suoraan kantaluokan jäsenmuuttujiin.

Kun johdettu luokka luo funktion, jolla on sama palautustyyppi ja koko nimi kuin perusluokan jäsenfunktiolla, mutta uusi toteutus, puhutaan metodin korvaamisesta. Tällöin aliluokassa kirjoitettu metodi kätkee perusluokan metodin.

Jos aliluokassa halutaan kuitenkin kutsua perusluokan metodia, tulee kantaluokan nimi sijoittaa metodin kutsun eteen yhdessä näkyvyysalueoperaattorin (::) kanssa.

Esimerkki periytymisestä:

```
#include <iostream.h>
#include <conio.h>
#include <string>

class Kotielukka
{
public:

void asetaIka(int i) { ika = i; }
void asetaPaino(int p) { paino = p; }
void tulostaIka() const { cout << "kotielukan ikä on " << ika <<
"\n"; }
void tulostaPaino() const { cout << "kotielukan paino on " << paino
<< "\n"; }

protected:
int paino;
int ika;
};


class Kana : public Kotielukka
{
public:
void tulostaIka() const { cout << "Kanan ikä on " << ika <<"\n"; }
void tulostaVari() const { cout << "Kanan vari on " << vari <<"\n";
}
void asetaVari(string v){ vari = v; }
private:
string vari;
int muniaPaivassa;
};


main()
{
Kotielukka molli;
molli.asetaIka(20);
molli.asetaPaino(20);
molli.tulostaIka();
Kana matti;
matti.asetaIka(10);
matti.tulostaIka();
```

```
matti.asetaVari("Harmaa");
matti.tulostaVari();
getch();

}
```

Operaattoreiden ylikuormittaminen
Luokan olioilla suoritetaan hyvin usein erilaisia toimenpiteitä: olioita lasketaan yhteen, olioon sijoitetaan toinen olio jne.

Normaalisti operaattorit on suunniteltu toimimaan yksinkertaisilla muuttujilla. Ne eivät sen sijaan osaa käsitellä olioita, jotka ovat monimutkaisia tietotyyppejä.
Siksi operaattorit tulee usein määritellä uudelleen toimimaan haluamallamme tavalla, kun operandeina on olioita.

Esimerkiksi yhteenlaskuoperaattorin tulee laskea yhteen kahden olion jäsenmuuttujat. Samoin sijoitusoperaattorin tulee sijoittaa toisen olion jäsenmuuttujien arvot toisen olion jäsenmuuttujiin.
Nuo toiminnot eivät onnistu ilman operaattoreiden ylimäärittelyä.

Voit ajatella operaattoreiden ylimäärittelyjä ikään kuin erikoismetodeina, jotka toteuttavat tällä kertaa erilaisia operaatioita.

Yleensä ylimäärittelymetodin parametrina on viittaus vakioon olioon. Kun ylimäärittely on mukana luokan esittelyssä normaalina metodina, viedään ylimäärittelyyn vain operaation oikea puoli.
Jos ylimäärittely on ulkopuolinen friend-funktio, viedään ylimäärittelyyn usein molemmat operandit.

Kirjan luvuissa on ylimäärittelyt käsitelty tarkemmin, joten emme paneudu aiheeseen tässä enempää.

Lisäksi const-avainsana estää vahingossa tehtävät muutokset operaation oikeaan operandiin.

Sisäisten tietotyyppien (kuten int) operaattoreita ei voida ylikuormittaa. Niiden suoritusjärjestystä ei voida muuttaa eikä myöskään sitä, kuinka moneen operandiin (yhteen tai useampaan) ne kohdistuvat. Uusia operaattoreita ei voida luoda eli et voi luoda esimerkiksi operaattoria ** olemaan 'potenssiin korotus' -operaattori.

Sijoitusoperaattori Operator=

Kääntäjä antaa käyttöön oletusmuodostimen, tuhoajafunktion ja kopiomuodostimen. Neljäs ja viimeinen kääntäjän tarjoama toiminto on sijoitusoperaattori (operator=()).

Tätä operaattoria kutsutaan aina, kun sijoitat olioon. Esimerkiksi:

```
Kana kana1(5,7);
Kana kana2(3,4);
// muuta koodia…
kana1 = kana2;
```

Aluksi luodaan oliot kana1 ja kana2. Ne alustetaan alkuarvoin. Huomaa, että tässä tapauksessa ei kutsuta kopiomuodostinta. kana2 on jo olemassa; sitä ei tarvitse muodostaa.

Muista: Pinnallinen kopio kopioi vain jäsenet, jolloin molemmat oliot päätyvät osoittamaan samaan muistialueeseen. **Tarkka kopio** varaa tarvittavan muistin. Sama ajattelutapa koskee sijoitusta ja kopiomuodostinta. Sijoitusoperaattoria koskee kuitenkin vielä yksi lisäseikka. Olio kana2 on jo olemassa ja muisti on jo varattu. Tuo muisti täytyy vapauttaa haluttaessa välttää muistikato.

Oliotaulukot

Taulukon jäseninä voi olla olioita:

```
Kana kanala[10]; // 10 kanan kanala
```

Taulukko kanala sisältää 10 Kana-oliota; taulukko laitetaan pinoon.

Osoitintaulukko:
Osoitintaulukkoa käyttäen voidaan itse oliot sijoittaa vapaaseen muistiin pinon sijaan.

Esimerkki osoitintaulukosta ja olioiden luomisesta:

```
Kana * kanala[500];
  int i;
  Kana * osKana;
  for (i = 0; i < 500; i++)
  {
    osKana = new Kana;
    osKana->asetaIka(2*i +1);
    kanala[i] = osKana;
  }
```

Myös koko **taulukko** voidaan sijoittaa **vapaaseen muistiin**:

Koko taulukko on mahdollista sijoittaa vapaaseen muistiin, jota kutsutaan myös nimellä keko. Se toteutetaan kutsumalla operaattoria new. Tuloksena on osoitin vapaaseen muistitilaan, jossa taulukko on. Esimerkiksi:

```
Kana *kanala = new Kana[100];
```

Taulukon tilan vapautus: muisti vapautetaan komennolla delete[]. Hakasulut kertovat kääntäjälle, että koko tuo taulukko tuhotaan.

Dynaaminen ja staattinen sidonta
Staattisessa sidonnassa kääntäjä tietää, mitä metodia kutsutaan.

Virtual-avainsanalla voidaan määrittää kantaluokan metodi virtuaaliseksi. Tällöin johdetussa luokassa kirjoitetaan metodi uudelleen ja johdetun luokan oliot kutsuvat omaa metodiaan kantaluokan metodin sijaan. Tätä kutsutaan myöhäiseksi sidonnaksi (dynaamiseksi sidonnaksi). Siinä metodi kytketään olioon myöhään, ajon aikana, oliota luotaessa. Myöhäistä sidontaa käytetään usein, kun olio luodaan vapaaseen muistiin. Tällöin osoitin voi osoittaa kantaluokkaan ja samaa kantaluokan osoitinta käytetään luotaessa aliluokkien olioita ja kutsuttaessa niiden metodeja.

Pääsääntö: kun kantaluokan metodi laitetaan virtuaaliseksi avainsanalla virtual ja johdetussa luokassa esitellään samanlainen metodi kuin kantaluokassa, ohjelmassa osataan kutsua oikeaa metodia eli kantaluokan olion kohdalla kantaluokan omaa metodia ja johdetun luokan olion kohdalla sen vastaavaa metodia. Näin voidaan esimerkiksi luoda erilaisia olioita vapaaseen muistiin käyttäen pelkästään kantaluokkaan osoittavaa osoitinta. Tuon osoittimen avulla voidaan sitten kutsua metodeita, jolloin virtuaaliset metodit tulevat oikein kutsutuiksi.
Jos kantaluokan osoittimen avulla halutaan kutsua johdetun luokan omaa erikoismetodia, tulee osoitin pakkomuuntaa osoittamaan kyseiseen olioon ja tarkistaa vielä tilanne.

Poikkeukset ja tiedostot
Poikkeukset ja tiedostot on käsitelty kirjan pääsisällössä, joten emme ota niitä tässä esille.