2

Matka C++-kieleen

Luku alkaa katsauksella tukeen, jota C++-kielellä on tarjota taulukkotyypille — tarkoittaa peräkkäistä kokoelmaa yhdentyyppisiä elementtejä, kuten kokonaislukuarvojen taulukkoa, joka sisältää ehkä testituloksia, tai merkkijonotaulukkoa, joka sisältää ehkä yksittäisiä sanoja tekstitiedostossa, kuten tämä luku. Sitten katsomme sisäisen taulukkotyypin heikkouksia ja parannamme sitä tekemällä oliopohjaisen Array-tyyppiluokan ja sitten laajennamme sitä erityisten Array-alityyppien oliokeskeiseksi hierarkiaksi. Lopuksi vertaamme Array-luokkatyyppiämme C++-vakiokirjaston vektoriluokkaan ja silmäilemme ensimmäistä kertaa geneerisiä algoritmeja. Matkan varrella motivoimme ja kurkistamme ensimmäisen kerran C++:n tukeen poikkeusten käsittelyssä, malleissa ja nimiavaruuksissa.

2.1 Sisäinen Array-tietotyyppi

Kuten näimme luvussa 1, C++:ssa on sisäinen tuki primitiivisille, aritmeettisille tietotyypeille kuten kokonaisluvuille:

```
// esittelee kokonaislukuolion ival
// alustetaan alkuarvoon 1024
int ival = 1024;
```

C++ tukee myös kaksoistarkkuuden ja yhden tarkkuuden liukulukutietotyyppejä:

```
// esittelee kaksoistarkkuuden liukulukuolion dval
// alustetaan alkuarvoon 3.14159
double dval = 3.14159;

// esittelee yhden tarkkuuden liukulukuolion fval
// myös alustetaan alkuarvoon 3.14159
float fval = 3.14159;
```

C++ tukee myös Boolean-tietotyyppiä sekä merkkityyppiä, joka voi pitää sisällään merkistön yksittäisiä elementtejä.

Aritmeettiset tietotyypit tukevat sisäisesti sijoituksia ja tavallisia aritmeettisia operaatioita kuten yhteen-, vähennys-, kerto- ja jakolaskua sekä myös vertailuoperaatioita kuten yhtäsuuruus, erisuuruus, suurempi kuin, pienempi kuin jne. Esimerkiksi:

```
int ival2 = ival + 4096;   // yhteenlasku
int ival3 = ival2 - ival;  // vähennyslasku

dval = fval * ival; // kertolasku
ival = ival3 / 2;   // jakolasku

bool result = ival2 == ival3;   // yhtäsuuruus
result = ival2 + ival != ival3; // erisuuruus
result = fval + ival2 < dval;   // pienempi kuin
result = ival > ival2;          // suurempi kuin
```

Lisäksi vakiokirjasto tukee kantaluokka-abstraktioita kuten merkkijonoa ja kompleksinumeroa. (Kappaleeseen 2.7 saakka aiomme sopivasti unohtaa kaiken vakiokirjaston vektoriluokasta.)

Yhdistetyt (*compound*) tyypit jäävät sisäisten ja vakiokirjaston luokkatyyppien väliin — erityisesti osoitin ja taulukkotyypit. (Katsomme osoitintyyppiä kappaleessa 2.2.)

Taulukko (*array)* on järjestetty säiliö yhdentyyppisiä elementtejä. Esimerkiksi sarja

```
0 1 1 2 3 5 8 13 21
```

edustaa yhdeksää ensimmäistä elementtiä Fibonacci-lukujonosta. (Kun kaksi ensimmäistä annetaan, jokainen seuraava elementti generoidaan laskemalla yhteen kaksi aikaisempaa.)

Kun haluamme määritellä ja alustaa taulukon, joka sisältää nämä elementit, kirjoitamme

```
int fibon[ 9 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

Taulukko-olion nimi on fibon. Se on kokonaislukutaulukko, jonka *ulotteisuus* on yhdeksän elementtiä. Ensimmäinen elementti on 0 ja viimeinen on 21. Käsittelemme elementtejä *indeksoimalla* taulukkoa käyttäen *indeksi*operaattoria. Kun luemme esimerkiksi ensimmäisen elementin, kirjoitamme

```
int first_elem = fibon[ 1 ]; // ei ihan oikein
```

Valitettavasti tämä on väärin, vaikka se ei ole sinänsä kielivirhe.

C++-kielen taulukon indeksoinnissa on eräs asia, jota ei heti ehkä välttämättä tajua: elementtien positiot alkavat kohdasta 0 eikä 1. Position 1 elementti on todellisuudessa toinen elementti. Samalla tavalla elementti positiossa 0 on ensimmäinen elementti. Kun haluamme käsitellä taulukon viimeistä elementtiä, indeksoimme aina: ulotteisuus — 1 elementti.

```
fibon[ 0 ]; // ensimmäinen elementti
fibon[ 1 ]; // toinen elementti
...
fibon[ 8 ]; // viimeinen elementti
fibon[ 9 ]; // hups ...
```

Elementti indeksillä 9 ei ole taulukon elementti. fibon-taulukon yhdeksän elementtiä indek-

soidaan positioilla 0–8. Yleinen vasta-alkajien virhe on indeksoida positiot 1–9. Se on niin yleistä, että sillä on itse asiassa oma nimensä: *ylitys yhdellä* -virhe.

Taulukon elementit käydään läpi tyypillisesti silmukkaa käyttäen. Esimerkiksi seuraava ohjelma alustaa taulukon kymmenen elementtiä arvoilla 0 — 9 ja kirjoittaa ne sitten laskevassa järjestyksessä vakiotulostukseen:

```
int main()
{
    int ia[ 10 ];
    int index;

    for ( index = 0; index < 10; ++index )
        // ia[0] = 0, ia[1] = 1 jne
        ia[ index ] = index;

    for ( index = 9; index >= 0; --index )
    cout  << ia[ index ] << " ";


    cout << endl;
}
```

Molemmat silmukat toistuvat kymmenen kertaa. for-avainsanan jälkeen sulkujen sisällä olevat kolme lausetta tekevät silmukan kaiken työn. Ensimmäinen lause sijoittaa arvon 0 index-muuttujaan

```
index = 0;
```

Se suoritetaan kerran ennen kuin silmukan varsinainen työ alkaa. Toinen lause

```
index < 10;
```

edustaa silmukan *pysähtymisehtoa*. Se aloittaa varsinaisen silmukan suorituksen. Jos sen arvo on tosi, silmukkaan kuuluva lause (tai lauseet) suoritetaan; jos sen arvo on epätosi, silmukka päättyy. Esimerkissämme joka kerta, kun indeksin arvo on pienempi kuin 10, lause

```
ia[ index ] = index;
```

suoritetaan. Kolmas lause

```
++index
```

on lyhennetty merkintätapa aritmeettisen olion kasvattamiselle arvolla 1. Se on yhtä kuin

```
index = index + 1;
```

Se suoritetaan silmukkaan liittyvän lauseen jälkeen (index-arvon sijoitus elementtiin, jota index indeksoi). Sen suoritus päättää for-silmukan yhden toistokerran. Sarja toistuu jälleen kerran ehtoa testaamalla. Kun ehto todetaan epätodeksi, silmukka päättyy. (Katsomme for-silmukkaa yksityiskohtaisemmin luvussa 5.) Toinen silmukka toimii päinvastaisessa järjestyksessä tulostamalla arvot.

Vaikka C++-kielessä on sisäinen tuki taulukkotyypille, on yksittäisten elementtien lukemiseen ja kirjoittamiseen vaadittava mekanismi rajoittunut. C++ ei tue taulukon *abstraktiota* eli se ei tue operaatioita, joita toivottaisiin tehtävän taulukolla, kuten yhden taulukon sijoittaminen toiseen, kahden taulukon yhtäsuuruuden vertailu tai taulukon koon kysyminen. Kun on annettu kaksi taulukkoa, emme voi esimerkiksi kopioida niistä yhtä toiseen yhtenä yksikkönä sijoitusoperaattoria käyttäen:

```
int array0[ 10 ], array1[ 10 ];
...
// virhe: yhtä taulukkoa ei voi kopioida suoraan toiseen
array0 = array1;
```

Jos halutaan sijoittaa yksi taulukko toiseen, pitää ohjelmoida se itse ja kopioida jokainen elementti vuorollaan:

```
for ( int index = 0; index < 10; ++index )
      array0[ index ] = array1[ index ];
```

Lisäksi taulukkotyypillä ei ole tietoa itsestään. Kuten sanoimme, se ei tiedä kokoaan, joten meidän täytyy pitää tietoa yllä itse taulukosta erillään. Tämä muuttuu ongelmalliseksi, kun haluamme välittää taulukoita yleisinä argumentteina funktioille. C++-kielessä sanomme, että toisin kuin kokonaisluku- ja liukulukutyypit, taulukko ei ole kielen *ensimmäisen luokan* kansalainen. Se periytyy C-kielestä ja kuvastaa tiedon ja algoritmien, jotka operoivat tuolla tiedolla, erillään oloa, ja on luonteenomaista proseduraaliselle paradigmalle. Luvun loppuun mennessä katsomme eri strategioita, joilla voimme antaa taulukolle lisästatusta kaupunkilaisuudessaan.

Harjoitus 2.1

Minkä luulet olevan syynä siihen, että sisäinen taulukko ei tue yhden taulukon sijoittamista toiseen? Mitä tietoa tarvitaan tämän operaation tukemiseksi?

Harjoitus 2.2

Millaisia operaatioita tulisi ensimmäisen luokan taulukon tukea?

2.2 Dynaaminen muistinvaraus ja osoittimet

Ennen kuin voimme edetä oliopohjaiseen suunnitteluumme, pitää poiketa hieman asiasta ja keskustella C++-ohjelman muistinvarauksesta. Syy on se, että emme voi toteuttaa realistisesti suunnitelmiamme (ja näyttää todenmukaista C++-koodia) esittelemättä ensin, kuinka muistia hankitaan ja käsitellään ohjelman suorituksen aikana. Se on tämän lyhyen kappaleen tarkoitus.

C++:ssa oliot voidaan varata joko staattisesti — eli kääntäjän toimesta, kun se prosessoi ohjelmamme lähdekoodia — tai dynaamisesti — eli ajonaikaisen kirjaston toimesta, kun se käynnistetään ohjelman suorituksen aikana. Pääerot näiden kahden muistinvarausmenetelmän välillä ovat tehokkuus ja joustavuus: staattista muistinvarausta pidetään paljon tehokkaampana, koska se tehdään juuri ennen ohjelman suorituksen alkamista. Se on kuitenkin joustamattomampi, koska se vaatii, että tiedämme ennen ohjelman suorituksen aloitusta tarvitsemamme muistin tyypin ja määrän. Esimerkiksi emme voi helposti käsitellä ja tallentaa mielivaltaisen tekstitiedoston sanoja käyttämällä staattisesti varattua merkkijonotaulukkoa. Yleensä tuntematon elementtimäärä vaatii dynaamisen muistinvarauksen joustavuutta.

Tähän saakka kaikki muistivarauksemme on ollut staattista. Esimerkiksi määrittely

```
int ival = 1024;
```

käskee kääntäjää varaamaan riittävästi muistia ival-nimisen int-tyypin minkä tahansa arvon tallennusta varten ja sitten sijoittamaan alkuarvoksi 1024 tuohon muistipaikkaan. Tämä kaikki tehdään ennen ohjelman suoritusta.

ival-olioon liittyy kaksi arvoa: arvo, jonka se sisältää — tässä tapauksessa 1024 — ja osoite, johon arvo on tallennettu. C++:ssa on mahdollista käsitellä molempia arvoja. Kun kirjoitamme

```
int ival2 = ival + 1;
```

käsittelemme arvoa, jonka ival sisältää, lisäämme siihen arvon 1 ja alustamme ival2-muuttujan tuolla uudella arvolla. Esimerkissämme ival2-muuttujan alkuarvo on 1025. Kuinka käsittelemme ja talletamme muistiosoitteen?

C++ tukee osoitintyyppiä, johon voidaan tallentaa olioiden muistiosoitteiden arvoja. Kun esimerkiksi haluamme esitellä osoitintyypin, johon voidaan tallentaa ival-muuttujan osoitteen arvo, kirjoitamme

```
// osoitin olioon, joka on int-tyyppinen
int *pint;
```

C++:ssa on esimääritelty erityinen *osoiteoperaattori* (&), joka olioon käytettynä palauttaa sen osoitteen arvon. Täten, jos sijoitamme pint-osoittimeen ival-muuttujan muistiosoitteen, kirjoitamme

```
int *pint;
pint = &ival; // sijoita pint-osoittimeen ival-muuttujan osoite
```

Kun käsitellään varsinaista oliota, johon pint osoittaa, pitää käyttää *epäsuoraa osoitusta* pint-osoittimeen käänteisoperaattorilla *. Tässä on esimerkki siitä, kuinka lisäisimme epäsuorasti arvon 1 ival-muuttujaan pint-osoittimen kautta:

```
// arvon 1 lisääminen epäsuorasti ival-muuttujaan pint-osoittimen kautta
*pint = *pint + 1;
```

Tämä on täsmälleen sama asia kuin seuraava suora ival-muuttujan käsittely:

```
// arvon 1 lisääminen suoraan ival-muuttujaan
ival = ival + 1;
```

Tässä esimerkissä ei ole todellista etua epäsuoran osoittamisen käytöstä ival-muuttujan

käsittelyssä: se on sekä tehottomampaa että helpommin väärin ohjelmoitavissa kuin suora ival-muuttujan käsittely. Esitimme sen antaaksemme ensimmäisen yksinkertaisen esimerkin osoittimien käytöstä. Eräs pääsyy osoittimien käytölle C++-kielessä on dynaamisesti varattavan muistin hallinta ja käsittely.

Staattisen ja dynaamisen muistinvarauksen välillä on kaksi pääeroa:

1. Staattiset oliot ovat nimettyjä muuttujia, joita käsittelemme suoraan, kun taas dynaamiset oliot ovat nimettömiä muuttujia ja käsittelemme niitä epäsuorasti osoittimien kautta. Näemme esimerkin tästä kohta.
2. Staattisten olioiden muistinvarauksen ja -vapautuksen hoitaa kääntäjä automaattisesti. Ohjelmoijan tulee ymmärtää se, mutta ei tarvitse tehdä sen eteen mitään. Dynaamisten olioiden muistinvaraus ja -vapautus sitä vastoin tulee ohjelmoijan hoitaa eksplisiittisesti, ja se on käytännössä merkittävästi virhealttiimpaa. Se tehdään new- ja delete-lausekkeiden avulla.

Oliot varataan dynaamisesti kahden eri *new*-lausekkeen avulla. Niistä ensimmäinen varaa yksittäisen tietyntyyppisen olion. Esimerkiksi

```
int *pint = new int( 1024 );
```

varaa nimettömän, int-tyyppisen olion alustaen sen alkuarvoksi 1024 ja palauttaa sitten olion muistiosoitteen. Osoitetta käytetään sitten pint-osoitinoliomme alustamiseen. Meidän ainoa pääsymme dynaamisesti varattuun muistiin on epäsuora osoitus osoittimen kautta.

Toinen versio new-lausekkeesta varaa taulukon, jonka elementit ovat tietyn tyyppisiä ja jolla on tietty ulotteisuus. Esimerkiksi

```
int *pia = new int[ 4 ];
```

varaa taulukon neljälle kokonaislukuelementille. Valitettavasti ei ole olemassa tapaa määrittää eksplisiittistä alkuarvoa dynaamisesti varatun taulukon yksittäisille elementeille.

Joskus hämmentäväkin piirre dynaamisen taulukon varaamisessa on, että palautettu arvo on yksinkertaisesti osoitin, joka on samaa tyyppiä kuin yksittäisen olion varauksessa palautettu tyyppi. Esimerkiksi ero pint- ja pia-osoittimien välillä on, että pia sisältää nelielementtisen taulukon ensimmäisen elementin osoitteen, kun taas pint sisältää yksinkertaisesti yksittäisen olion osoitteen.

Kun emme enää tarvitse dynaamisesti varattua oliota tai oliotaulukkoa, meidän pitää eksplisiittisesti vapauttaa muisti käyttäen jompaakumpaa kahdesta *delete*-lausekkeesta. Ohjelma voi käyttää uudelleen vapautetun muistin. Yksittäisen olion delete-lauseke on seuraava:

```
// poista yksittäinen olio
delete pint;
```

delete-lausekkeen taulukkomuoto on seuraava:

```
// poista objektitaulukko
delete [] pia;
```

Mitä jos unohdamme poistaa dynaamisesti varatun muistin? Päädymme *muistivuotoon*. Muistivuoto on alue dynaamisesti varattua muistia, johon meillä ei ole enää osoitinta, emmekä siten voi palauttaa sitä ohjelmalle myöhempää uusiokäyttöä varten. (Useimmissa järjestelmissä on nykyisin työkaluja, jotka tunnistavat ohjelman muistivuotoja. Kysy järjestelmänhoitajaltasi.)

Myönnettäköön, että tämä pikamatka osoitintietotyyppiin ja dynaamiseen muistinvaraukseen nostattaa todennäköisesti enemmän kysymyksiä kuin vastauksia. Dynaaminen muistinvaraus ja osoitinkäsittely ovat kuitenkin perustavaa laatua oleva ominaisuus tosielämän C++-ohjelmoinnissa, emmekä siksi halunneet viivytellä sen esittelemistä. Katsomme niiden käyttöä oliopohjaisen ja oliosuuntautuneen Array-luokan toteutuksissa seuraavissa kappaleissa. Kappaleessa 8.4 katsomme dynaamista muistinvarausta sekä new- ja delete-lausekkeiden käyttöä yksityiskohtaisesti.

Harjoitus 2.3

Selitä seuraavien olioiden määrittelyjen väliset erot:

```
(a) int ival = 1024; (c) int *pi2 = new int( 1024 );
(b) int *pi = &ival; (d) int *pi3 = new int[ 1024 ];
```

Harjoitus 2.4

Mitä seuraava koodikatkelma tekee? Mikä on sen merkittävä virhe? (Huomaa, että seuraavassa olevan pia-osoittimen yhteydessä käytetty indeksioperaattori on oikein; syy siihen, että voimme tehdä näin, on selitetty kappaleessa 3.9.2.)

```
int *pi = new int( 10 );
int *pia = new int[ 10 ];

while ( *pi < 10 ) {
        pia[ *pi ] = *pi;
        *pi = *pi + 1;
}

delete pi;
delete [] pia;
```

2.3 Oliopohjainen suunnittelu

Tässä kappaleessa suunnittelemme ja toteutamme taulukkoabstraktion käyttäen C++-luokkamekanismia. Ensimmäinen toteutuksemme tukee vain kokonaislukutaulukkoa. Myöhemmin, kun käytämme mallimekanismia, laajennamme abstraktion tukemaan rajoittamatonta määrää tietotyyppejä.

Ensimmäinen tehtävä on päättää, mitä operaatioita teemme taulukkoluokallemme. Vaikka haluaisimme tehdä kaiken mahdollisen, emme voi toteuttaa kaikkea kerralla. Seuraavassa on ensimmäinen joukko tuetuista operaatioista:

1. Taulukkoluokkamme tulee tietää jotain itsestään ja se rakennetaan sen sisäiseen toteutukseen. Ensimmäisessä vaiheessa se tietää kokonsa.
2. Taulukkoluokkamme tulee tukea yhden taulukon sijoitusta toiseen ja kahden taulukon vertailua sekä yhtä- että erisuuruudessa.
3. Taulukkoluokkamme tulee tukea seuraavia kyselyitä arvoistaan, joita se sisältää. Mikä on taulukon pienin arvo? Mikä on suurin arvo? Esiintyykö tietty arvo taulukossa, ja jos esiintyy, mikä on sen ensimmäinen positio?
4. Taulukkoluokkamme tule tukea mahdollisuutta lajitella itsensä. Puolustukseksi väittäkäämme, että eräs joukko potentiaalisia käyttäjiä puhui siitä, kuinka tärkeää sellaisten taulukoiden tuki on, jotka on lajiteltu. Toiset eivät ilmaisseet voimakasta mielipidettään suuntaan taikka toiseen.

Taulukko-operaatioiden lisäksi meidän pitää tukea taulukkomekanismeja. Näihin kuuluvat seuraavat:

5. Mahdollisuus määrittää koko, jollaiseksi taulukko luodaan. (Emme vaadi, että tämä arvo pitää tietää käännöksen aikana.)
6. Mahdollisuus, että taulukko voidaan alustaa arvojoukolla.
7. Mahdollisuus käsitellä taulukon yksittäisiä elementtejä indeksin kautta. Puolustukseksi väitettäköön, että käyttäjämme ilmaisivat voimakkaan halun käyttää indeksioperaattoria tähän.
8. Mahdollisuus keskeyttää ja ilmaista kelpaamattomat indeksiarvot. Puolustukseksi olettakoon, että pidämme tätä erittäin tärkeänä emmekä ole kysyneet käyttäjiltämme, mitä mieltä he ovat siitä. Olemme päättäneet, että se on välttämätön piirre hyvin suunnitellulle taulukkototeutukselle.

Puheemme potentiaalisista käyttäjistä on luonut paljon innostusta. Nyt meidän pitää toimittaa varsinainen toteutus. Mutta miten muunnamme suunnitelman C++-koodiksi? Yleinen muoto luokalle, joka tukee oliopohjaista suunnittelua, näyttää tältä:

```
class luokan_nimi {
public:
    // joukko julkisia operaatioita
```

```
private:
    // yksityinen toteutus
};
```

Tässä class, public ja private edustavat kielen avainsanoja ja luokan_nimi on käyttäjän määrittämä tunnus, jota käytetään luokkaa nimettäessä jatkoviittauksissa. Annamme luokallemme nimen IntArray, kunnes yleistämme tietotyypit, joita se voi tukea; siinä vaiheessa nimeämme luokkamme uudelleen Array-nimiseksi.

Luokan nimi edustaa uutta tietotyyppiä. Voimme käyttää sitä määritelläksemme olioita luokkatyypistämme samaan tapaan kuin käytämme sisäisiä tyyppejä määritelläksemme niiden tyyppejä. Esimerkiksi:

```
// yksittäinen IntArray-luokan olio
IntArray myArray;

// osoitin yksittäiseen IntArray-luokan olioon
IntArray *pArray = new IntArray;
```

Luokan määrittely koostuu kahdesta osasta: *luokan otsikosta*, joka puolestaan muodostuu avainsanasta class ja siihen liitetystä luokan nimestä ja *luokan rungosta*, joka on aaltosulkujen sisällä ja päättyy puolipisteeseen. Itse luokan otsikko toimii luokan esittelynä. Esimerkiksi:

```
// esittelee IntArray-luokan ohjelmalle,
// mutta ei sisällä määrittelyä
class IntArray;
```

Luokan runko sisältää jäsenten määrittelyjä ja pääsyn luokan eri osiin kuten public ja private. Luokan jäsenet muodostuvat operaatioista, joita luokka voi suorittaa, ja tiedoista, jotka edustavat luokan abstraktiota. Operaatioita kutsutaan *jäsenfunktioiksi* eli *metodeiksi*. IntArray-luokassamme ne muodostuvat seuraavasti:

```
class IntArray {
public:
    // yhtä- ja erisuuruus-operaatiot: #2b
    bool operator==( const IntArray& ) const;
    bool operator!=( const IntArray& ) const;

    // sijoitusoperaattori: #2a
    IntArray& operator=( const IntArray& );

    int  size() const; // #1
    void sort();       // #4

    int  min()  const; // #3a
    int  max()  const; // #3b

    // jos arvo löytyy taulukosta,
    // palauta sen ensimmäisen esiintymän indeksi
    // palauta muussa tapauksessa -1
```

```
    int find( int value ) const; // #3c

private:
    // yksityinen toteutus
};
```

Jäsenfunktioiden oikealla puolella olevat numerot viittaavat aikaisemman määrityslistauksemme kohtiin. Emme aio juuri nyt selittää const -määrettä, joka on argumenttiluettelossa ja sen perässä. Se ei ole välttämätöntä tässä vaiheessa, mutta se on välttämätön koodille, jota käytetään tosielämän ohjelmissa.

Nimetty jäsenfunktio kuten min() käynnistetään käyttämällä kahta *jäsenen käsittelyoperaattoria*. Itse asiassa on olemassa kaksi operaattoria: pisteoperaattori (.) luokan olioille ja nuoli-operaattori (->) osoittimille luokkaolioihin. Etsiäksemme esimerkiksi pienimmän arvon myArray-luokkaoliostamme, kirjoitamme

```
// alusta min_val myArray:n pienimmällä arvolla
int min_val = myArray.min();
```

Kun haluamme löytää suurimman arvon dynaamisesti varatusta IntArray-oliostamme, kirjoitamme

```
int max_val = pArray->max();
```

(Niin, emme ole vielä esitelleet, kuinka alustamme IntArray-luokkamme oliot koolla ja muilla arvoilla. Luokassa on olemassa erityinen jäsenfunktio, *muodostaja* (*constructor*), tuon tekemiseen. Me esittelemme sen lyhyesti.)

Operaattoreita käytetään luokan olioihin täsmälleen samalla tavalla kuin niitä käytetään sisäisiin tietotyyppeihin. Kun on olemassa kaksi IntArray-oliota

```
IntArray myArray0, myArray1;
```

niin sijoitusoperaattoria käytetään seuraavasti:

```
// käynnistää jäsenfunktion, joka sijoittaa ja kopioi:
// myArray0.operator=( myArray1 )
myArray0 = myArray1;
```

Yhtäsuuruusoperaattorin käynnistys näyttää tältä:

```
// käynnistää yhtäsuuruus-jäsenfunktion:
// myArray0.operator==( myArray1 )
if ( myArray0 == myArray1 )
      cout << "!!sijoitusoperaattorimme toimii!\n";
```

Avainsanat private ja public kontrolloivat luokan jäseniin pääsyä. Jäseniä, jotka ovat luokan rungon julkisessa (public) osassa, voidaan käsitellä mistä tahansa yleisestä ohjelmasta. Jäseniä, jotka ovat yksityisessä (private) osassa, voidaan käsitellä vain luokan jäsenfunktiosta (ja ystävistä. Emme aio selittää ystäviä (friends) vasta kuin kohdassa 15.2).

Yleensä julkiset jäsenet muodostavat luokan *julkisen rajapinnan* (*public interface*) — joka tarkoittaa joukkoa operaatioita, jotka toteuttavat luokan käyttäytymistä. Tämä muodostuu luokan kaikista tai osasta sen jäsenfunktioista. Yksityiset jäsenet muodostavat *yksityisen toteutuksen* (*private implementation*) — eli tiedon, johon tieto on talletettu.

Tätä luokan julkisen ja yksityisen rajapinnan jakoa kutsutaan *tiedon piilottamiseksi* (*information hiding*). Tiedon piilottaminen on tärkeä käsite ohjelmistojen ohjelmoijille ja katsommekin sitä tarkemmin myöhemmissä luvuissa. Lyhyesti sanottuna se tuo kaksi etua ohjelmillemme:

1. Jos luokan yksityistä toteutusta pitää muokata tai laajentaa, vain suhteellisen pientä määrää jäsenfunktiota, joita tarvitaan tuon toteutuksen käsittelyyn, pitää muokata; monet käyttäjän ohjelmat, jotka käyttävät luokkaamme, eivät yleensä tarvitse muokkausta, vaikka ne saattavat vaatia uudelleenkääntämisen (käsittelemme tätä kohdassa 6.18).
2. Jos luokan yksityisessä toteutuksessa on virhe, ohjelmakoodin määrä, joka pitää tutkia, on yleensä paikallistettavissa suhteellisen pieneen määrään jäsenfunktioita, jotka käsittelevät tuota toteutusta; koko ohjelmaa ei tarvitse tutkia.

Mitä ovat välttämättömät tietojäsenet, jotka edustavat IntArray-luokkaamme? Kun IntArray-olio esitellään, käyttäjä määrittää koon. Meidän pitää tallentaa se. Määrittelemme tietojäsenen tekemään juuri tuon. Lisäksi meidän pitää varata ja tallentaa muodostuva taulukko. Tämä tehdään käyttäen `new`-lauseketta. Määrittelemme osoitintyyppisen tietojäsenen, johon tallennetaan osoitteen arvo, jonka `new`-lauseke palauttaa:

```
class IntArray {
public:
    // ...
    int size() const { return _size; }
private:
    //  tiedon sisäinen esitystapa
    int _size;
    int *ia;
};
```

Koska olemme sijoittaneet `_size`-tietojäsenen luokan yksityiseen osaan, meidän pitää esitellä julkinen jäsenfunktio, jotta käyttäjät pääsisivät käsittelemään sen arvoa. Koska C++ ei salli, että tietojäsenellä ja jäsenfunktiolla on sama nimi, on yleinen käytäntömme tällaisissa tapauksissa lisätä alaviiva tietojäseneen. Joten nyt meillä on julkinen `size()`-käsittelyfunktio ja yksityinen `_size`-tietojäsen. Tämän tekstin aikaisemmissa painoksissa lisäsimme avainsanat *get* ja *set* käsittelyfunktioihin, mutta tämä osoittautui käytännössä kömpelöksi.

Vaikka julkisen käsittelyfunktion käyttö sallii käyttäjälle arvon lukemisen, jokin näyttää olevan perusteellisesti pielessä tässä toteutuksessa, ainakin ensisilmäyksellä. Huomaatko, mikä se on? Pane merkille, että sekä

```
IntArray array;
int array_size = array.size();
```

että

```
// olettaen, että _size olisi julkinen
int array_size = array._size;
```

alustavat array_size-muuttujan arvoksi taulukon ulotteisuuden. Kuitenkin ensimmäinen ilmentymä näyttää vaativan funktiokutsun, kun taas toinen tekee suoraa muistikäsittelyä. Yleensä funktiokutsu on merkittävästi “kalliimpi” kuin suora muistin käsittely. Tuleeko tiedon piilottaminen siis merkittävästi kalliimmaksi — ehkäpä liian kalliiksi — ohjelman suorituksenaikaisessa tehokkuudessa? Onneksi yleensä vastaus on ei.

Kielen antama ratkaisu tähän on *välittömän* (*inline*) *funktion mekanismi*. Välitön funktio laajennetaan paikalleen sen kutsukohdalla. Yleensä se tarkoittaa, että välitön funktio ei tee yhtään funktiokutsua[1]. Esimerkiksi size()-funktion kutsua for-silmukan ehtolauseessa

```
for ( int index = 0; index < array.size(); ++index )
    // ...
```

ei todellisuudessa käynnistetä _size kertaa, vaan laajennetaan välittömästi käännöksen aikana seuraavaan yleiseen muotoon:

```
// array.size()-funktion yleinen välitön laajennus()
for ( int index = 0; index < array._size; ++index )
    // ...
```

Jäsenfunktiota, joka on määritelty luokan määrittelyn yhteydessä kuten size(), kohdellaan automaattisesti välittömänä funktiona. Vaihtoehtoisesti voidaan käyttää inline-avainsanaa pyyntöön, että funktiota kohdeltaisiin välittömänä (meillä on lisää sanottavaa välittömistä funktioista kohdassa 7.6).

Tähän mennessä olemme tehneet IntArray-luokan vaatimat operaatiot (kohdat 1— 4), mutta emme taulukon yksittäisten elementtien alustamista ja käsittelyä (kohdat 5—8).

Eräs yleisimmistä ohjelmavirheistä on käyttää oliota ilman, että sitä olisi kunnolla alustettu. Tämä on itse asiassa niin yleinen virhe, että C++:ssa on automaattinen alustusmekanismi käyttäjän määrittelemille luokille: luokan *muodostaja* (*constructor*).

1. Tämä ei kuitenkaan aina pidä paikkaansa. Välitön funktio on pyyntö kääntäjälle eikä tae. Katso kohdasta 7.6 lisätietoja.

Muodostaja on luokan erityinen jäsenfunktio, jota käytetään alustamiseen. Jos se on määritelty, sitä käytetään automaattisesti luokan jokaiselle oliolle ennen niiden ensimmäistä käyttöä. Kuka määrittelee muodostajan? Luokan tekijä — tarkoittaa siis meitä. Luokalle tarvittavien muodostajien yksilöiminen on luokan suunnittelukokonaisuuteen kuuluva osa.

Muodostaja määritellään antamalla sille luokan nimi. Lisäksi muodostajalle ei voida määrittää palautustyyppiä. On mahdollista määritellä luokalle useita muodostajia, vaikka niillä kaikilla on sama nimi — edellyttäen, että kääntäjä voi erottaa ne toisistaan niiden parametriluetteloiden perusteella.

Yleisemmin sanottuna C++ tukee piirrettä, jota kutsutaan *funktion ylikuormitukseksi* (*function overloading*). Funktion ylikuormitus sallii saman nimen käyttämisen yhdelle tai useammalle funktiolle — sillä erotuksella, että jokaisella pitää olla yksilöllinen parametriluettelo, joko määrältään tai parametriensa tyypiltä. (Yksilöllinen parametriluettelo sallii kääntäjän päätellä, minkä ylikuormitetuista funktioista se valitsee tietyssä käynnistyksessä.) Esimerkiksi seuraavassa on kelvollinen joukko ylikuormitettuja `min()`-funktioita (nämä voisivat olla myös luokan jäsenfunktioita):

```
// joukko ylikuormitettuja min()-funktiota
// jokaisella pitää olla yksilöllinen parametriluettelo
#include <string>;

int    min( const int *pia, int size );
int    min( int, int );
int    min( const char *str );
char   min( string );
string min( string, string );
```

Ylikuormitetun funktion käyttäytyminen suorituksen aikana on samanlaista kuin ylikuormittamattoman funktion. Pääasiallinen kuormitus tapahtuu käännöksen aikana, kun päätellään, mikä monista ilmentymistä käynnistetään. Ilman funktion ylikuormitusta meitä vaadittaisiin nimeämään ohjelmamme jokainen funktio yksilöllisesti. (Funktion ylikuormitus käsitellään yksityiskohtaisesti luvussa 9.)

Olemme yksilöineet seuraavat kolme muodostajaa IntArray-luokallemme:

```
class IntArray {
public:
    explicit IntArray( int sz = DefaultArraySize );
    IntArray( int *array, int array_size  );
    IntArray( const IntArray &rhs );
    // ...

private:
    static const int DefaultArraySize = 12;
      //  ...
};
```

Muodostajaa

```
IntArray( int sz = DefaultArraySize );
```

pidetään *oletusmuodostajana*, koska se ei vaadi käyttäjää välittämään argumenttia. (Emme aio tällä kertaa selittää avainsanaa explicit, joka on mukana oletusmuodostajan esittelyssä. Näytämme sen vain yksinkertaisesti täydellisyyden takia.) Jos ohjelmoija välittää argumentin, välitetään tuo arvo muodostajalle. Esimerkiksi

```
IntArray array1( 1024 );
```

välittää argumentin 1024 muodostajalle. Jos toisaalta käyttäjä ei välitä määrittää kokoa, käytetään sen sijaan arvoa DefaultArraySize. Esimerkiksi

```
IntArray array2;
```

saa aikaan oletusmuodostajan käynnistämisen DefaultArraySize-arvolla. (Tietojäsen, joka on esitelty static-tyyppisenä, on erityinen jaettu tietojäsen, joka esiintyy vain kerran ohjelmassa huolimatta siitä, kuinka monta luokan oliota on määritelty. Se on tapa jakaa tietoa luokan kaikkien olioiden kesken — katso koko esitys kohdasta 13.5.)

Tässä on hieman yksinkertaistettu toteutus oletusmuodostajastamme — yksinkertaistettu siinä mielessä, että versiomme ei ole huolissaan mahdollisuudesta, että asiat menevät pieleen. (Mitä voisi mahdollisesti mennä pieleen? Kaksi asiaa tässä esimerkissä: ensiksikin, ohjelmallemme käytettävissä oleva muisti ei ole loputon; on mahdollista, että new-lauseke epäonnistuu. (Kohdassa 2.6 katsomme, kuinka voisimme käsitellä sen.) Toiseksi, sz-argumentti saatetaan välittää kelpaamattomalla arvolla kuten negatiivisena, nollana tai liian suurena arvona, joka voidaan tallettaa int-tyyppiseen muuttujaan.)

```
IntArray::
IntArray( int sz )
{
    // aseta tietojäsenet
    _size = sz;
    ia = new int[ _size ];

    // alusta muisti
    for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = 0;
}
```

Tämä on ensimmäinen kerta, kun olemme määritelleet luokan jäsenfunktion rungon ulkopuolelle. Ainoa syntaktinen ero on, että pitää yksilöidä luokka, johon jäsenfunktio kuuluu. Tämä tehdään käyttäen *luokan viittausalueoperaattoria* (*class scope operator*):

```
IntArray::
```

Tuplakaksoispisteoperaattoria (::) kutsutaan *viittausalueoperaattoriksi*. Kun se liitetään luokan nimeen kuten edellä, siitä tulee luokan viittausalueoperaattori. Viittausaluetta voidaan epävirallisesti ajatella näköalaikkunaksi. Olio, jolla on globaali viittausalue, on näkyvissä sen tiedoston loppuun, jossa se on määritelty. Oliolla, joka on määritelty funktiossa, sanotaan ole-

van *paikallinen viittausalue*; se on näkyvissä vain funktion rungossa, jossa se on määritelty. Jokainen luokka säilyttää viittausalueensa, paitsi että sen jäsenet eivät ole näkyviä. Luokan viittausalueoperaattori kertoo kääntäjälle, että operaattorin jälkeen tuleva tunniste löytyy tuon luokan näkyvyysalueelta. Meidän tapauksessamme

```
IntArray::
IntArray( int sz )
```

kertoo kääntäjälle, että määritelty IntArray()-funktio on IntArray-luokan jäsen. (Vaikka ohjelman viittausalue ei ole sitä, johon meidän pitäisi keskittyä tässä vaiheessa, se on kuitenkin jotain sellaista, mikä meidän tulee joka tapauksessa ymmärtää. Palaamme ohjelman viittausalueeseen yksityiskohtaisemmin luvussa 8. Luokan näkyvyyttä käsitellään erityisesti kohdassa 13.9.)

IntArray-luokan toinen muodostaja alustaa uuden olion sisäisellä kokonaislukutaulukolla. Se vaatii kaksi argumenttia: varsinaisen taulukon ja toisen argumentin, joka ilmaisee taulukon koon. Esimerkiksi:

```
int ia[10] = {0,1,2,3,4,5,6,7,8,9};
IntArray iA3(ia,10);
```

Tämän muodostajan toteutus on melkein identtinen ensimmäisen muodostajan kanssa. (Jälleen kerran, tässä vaiheessa emme suojaa koodiamme tilanteilta, jolloin asiat voivat mennä pieleen.)

```
IntArray::
IntArray( int *array, int sz )
{
    // aseta tietojäsenet
    _size = sz;
    ia = new int[ _size ];

    // kopioi tietojäsenet
    for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = array[ ix ];
}
```

Viimeinen IntArray-luokan muodostaja käsittelee yhden taulukon olion alustamista toisella. Se käynnistetään automaattisesti aina, kun jompikumpi seuraavista kahdesta määrittelymuodosta esiintyy:

```
IntArray array;

// samanlaiset alustukset
IntArray ia1 = array;
IntArray ia2( array );
```

Tätä muodostajaa kutsutaan *kopiointimuodostajaksi* (*copy constructor*) ja kerromme siitä paljon lisää myöhemmissä luvuissa. Tässä on toteutuksemme, ja jälleen kerran: emme välitä ohjelman mahdollisista suorituksenaikaisista poikkeustilanteista:

```
IntArray::
IntArray( const IntArray &rhs )
{ // kopiointimuodostaja
    _size = rhs._size;
    ia = new int[ _size ];

    for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = rhs.ia[ ix ];
}
```

Tässä esimerkissä esitellään uusi yhdistetty tyyppi, nimittäin *viittaus* (*reference*) (`IntArray &rhs`). Viittaus on eräänlainen osoitin ilman erityistä osoitinsyntaksia. (Täten kirjoitamme `rhs._size` emmekä `rhs->_size`.) Viittauksen avulla voidaan kuten osoittimellakin käsitellä oliota epäsuorasti. (Meillä on lisää sanottavaa viittauksista ja osoittimista kohdassa 3.6.)

Huomaa, että kaikki kolme muodostajaa on toteutettu samankaltaisella tavalla. Yleensä, kun kahdessa tai useammassa funktiossa on samanlainen yleinen ohjelmakoodi, on järkevää laittaa tuo koodi erilliseen funktioon, joka voidaan jakaa niiden kesken. Myöhemmin, jos toteutukseen vaaditaan muutosta, täytyy se tehdä vain kerran. Lisäksi ihmisen on helpompi tajuta jaetun toteutuksen luonnetta.

Kuinka voisimme viedä muodostajiemme koodin itsenäiseen, jaettuun funktioon? Tässä on yksi mahdollinen toteutustapa:

```
class IntArray {
public:
    // ...
private:
    void init( int sz, int *array );
    // ...
};

void
IntArray::
init( int sz, int *array )
{
    _size = sz;
    ia = new int[ _size ];

    for ( int ix=0; ix < _size; ++ix )
        if ( ! array )
            ia[ ix ] = 0;
        else ia[ ix ] = array[ ix ];
}
```

Kolme uudelleenkirjoitettua muodostajaamme ovat seuraavassa:

```
IntArray::IntArray( int sz ){ init( sz, 0 ); }
IntArray::IntArray( int *array, int sz )
        { init( sz, array ); }
```

```
IntArray::IntArray( const IntArray &rhs )
        { init( rhs.size, rhs.ia ); }
```

Luokkamekanismi tukee myös jäsenfunktiota *tuhoaja* (*destructor*). Tuhoaja käynnistetään automaattisesti luokan jokaisen olion kohdalla, kun sitä viimeisen kerran käytetään ohjelmassamme. Yksilöimme tuhoajan antamalla sille luokan nimen ja laittamalla sen nimen eteen tilden (~). Yleensä tuhoaja vapauttaa resurssit, joita vaadittiin muodostamisen ja olion käytön aikana. Esimerkiksi IntArray-tuhoaja poistaa muodostamisen aikana varatun muistin. Tässä on toteutuksemme (muodostajia ja tuhoajia käsitellään yksityiskohtaisesti luvussa 14):

```
class IntArray {
public:
        // muodostajat
        explicit IntArray( int size = DefaultArraySize );
        IntArray( int *array, int array_size  );
        IntArray( const IntArray &rhs );

        // tuhoaja
        ~IntArray() { delete [] ia; }

        // ...

private:
      //  ...
};
```

Taulukkoluokallamme ei olisi paljoakaan merkitystä, elleivät käyttäjät voisi helposti indeksoida luokkaoliota jä käsitellä niiden yksittäisiä elementtejä. Luokkamme pitää tukea esimerkiksi seuraavia yleisiä käyttötapoja:

```
IntArray array;
int last_pos = array.size()-1;

int temp = array[ 0 ];
array[ 0 ] = array[ last_pos ];
array[ last_pos ] = temp;
```

Tuemme indeksointia IntArray-luokkaoliolle antamalla luokkakohtaisen ilmentymän indeksioperaattorista. Tässä on toteutuksemme, joka tukee vaadittua käyttötapaa:

```
#include <cassert>
int&
IntArray::
operator[]( int index )
{
     assert( index >= 0 && index < size );
  return ia[ index ];
}
```

Kieli tukee yleensä *operaattorin ylikuormitusta* (*operator overloading*), jossa operaattorin

uusi ilmentymä voidaan määritellä tietyn luokan tyyppiseksi. Tyypillisesti luokalla on yksi tai useampi sijoitusoperaattorin ilmentymä: yhtäsuuruusoperaattori, ehkä yksi tai useampi vertailuoperaattori sekä iostream-toimintojen syöttö- ja tulostusoperaattorit. (Meillä on lisäesimerkkejä ylikuormitetuista operaattoreista kohdassa 3.15. Luvussa 15 käsittelemme operaattorin ylikuormitusta yksityiskohtaisemmin.)

Tyypillisesti luokan määrittely ja kaikki siihen liittyvät vakioarvot tai typedef-nimet tallennetaan otsikkotiedostoon. Otsikkotiedostolle annetaan luokan nimi. Joten me voisimme luoda esimerkiksi parin IntArray.h ja Matrix.h otsikkotiedostoiksi. Kaikki ohjelmat, jotka haluavat käyttää joko IntArray- tai Matrix-luokkaa, ottavat vastaavan otsikkotiedoston mukaan ohjelmaan.

Samalla tavalla luokan jäsenfunktiot, joita ei ole määritelty luokkamäärittelyn yhteydessä, tallennetaan tyypillisesti ohjelman tekstitiedostoon ja sille annetaan sama nimi kuin luokalla on. Voisimme esimerkiksi luoda ohjelman tekstitiedostoparin IntArray.C ja Matrix.C, johon voisimme tallentaa luokkaan liittyvät jäsenfunktiot. (Muista, että loppuliite, jota käytetään ilmaisemaan ohjelman tekstitiedostoa, vaihtelee eri käännösjärjestelmissä; sinun tulisi tarkistaa järjestelmässäsi käytetty ilmaisutapa.) Sen sijaan, että vaatisimme, että nämä funktiot käännettäisiin uudelleen jokaisen ohjelman kanssa, joka haluaa käyttää niihin liittyviä luokkia, jäsenfunktiot esikäännetään ja tallennetaan luokkakirjastoon. Iostream-kirjasto on yksi sellainen esimerkki.

Harjoitus 2.5

C++-luokan avainpiirre on rajapinnan ja toteutuksen erottaminen toisistaan. Rajapinta on joukko operaatioita, joita käyttäjät voivat käyttää luokkaolioihin. Se muodostuu kolmesta osasta: operaatioiden nimistä, paluuarvoista ja niiden parametriluetteloista. Yleensä tämä on kaikki, mitä käyttäjän edellytetään luokasta tietävän. Yksityinen toteutus muodostuu algoritmeista ja tarvittavasta tiedosta julkisen rajapinnan tukemiseksi. Vaikka ihannetapauksessa luokan rajapinta kasvaisi, se ei muutu yhteensopimattomaksi aikaisempien versioiden kanssa luokan elinaikana. Toisaalta toteutus on vapaa kehittymään luokan elinaikana. Valitse yksi seuraavista abstraktioista ja kirjoita julkinen rajapinta tuolle luokalle:

(a) Matrix (c) Person (e) Pointer
(b) Boolean (d) Date (f) Point

Harjoitus 2.6

Sanat *muodostaja* ja *tuhoaja* ovat jollakin tavalla harhaanjohtavia siinä mielessä, että nämä käyttäjän tekemät funktiot eivät muodosta tai tuhoa luokkaolioita, joihin kääntäjä käyttää niitä automaattisesti. Kun kirjoitamme

```
int main() {
    IntArray myArray( 1024 );
    // ...
    return 0;
}
```

myArray-olion tietojäsenten tarvitsema muisti on varattu ennen muodostajaa. Itse asiassa kääntäjä muuntaa sisäisesti ohjelmamme seuraavanlaiseksi (huomaa, että tämä ei ole kelvollista C++-koodia[2]):

```
int main() {
    IntArray myArray;

    // C++-pseudokoodia -- käytä muodostajaa
    myArray.IntArray::IntArray( 1024 );
    // ...
    // C++-pseudokoodia -- käytä tuhoajaa
    myArray.IntArray::~IntArray();

    return 0;
}
```

Luokan muodostajat toimivat pääasiallisesti luokkaolioiden tietojäsenten alustajina. Tuhoaja etupäässä vapauttaa resursseja, joita luokkaoliot ovat tarvinneet elinaikanaan. Määrittele joukko muodostajia, joita tarvitaan harjoituksesta 2.5 valitsemallesi luokalle. Vaatiiko luokkasi tuhoajan?

Harjoitus 2.7

Harjoituksissa 2.5 ja 2.6 olet määrittänyt melkein täydellisen julkisen rajapinnan luokan käytölle. (Meidän pitäisi vielä määritellä sijoitusoperaattori kopiointiin, mutta jätämme sen huomiotta tässä vaiheessa — C++:ssa on oletuksena tuki luokan yhden olion sijoitukselle toiseen. Ongelmana on, että tämä oletuskäyttäytyminen on usein riittämätön. Katso kohdasta 14.6 sen käsittely.) Kirjoita ohjelma kahdessa edellisessä harjoituksessa määrittelemällesi luokan julkiselle rajapinnalle. Onko sitä helppo vai hankala käyttää? Haluaisitko uudistaa määrityksiä? Voitko tehdä sen ja säilyttää yhteensopivuuden?

2.4 Oliokeskeinen suunnittelu

Toteutuksemme min()- ja max()-funktioista eivät tee mitään erityisiä oletuksia taulukkoelementtien tallentamisesta, joten siitä syystä vaaditaan, että tutkimme jokaisen elementin. Jos vaatisimme elementtien olevan lajiteltuna, molempien operaatioiden toteutuksesta tulisi yksinkertainen indeksi ensimmäiseen ja viimeiseen elementtiin. Lisäksi tietyn elementin ole-

2. Tästä kiinnostuneille asiaa on käsitelty oheislukemisessamme *Inside the C++ Object Model*.

massaolon etsinnälle on merkittävästi tehokkaampaa, jos tiedetään elementtien olevan lajiteltuna. Tuki elementtien lajittelulle kuitenkin lisää Array-luokan toteutuksen monimutkaisuutta. Olemmeko tehneet virheen suunnitelmassamme?

Emme ole tehneet virhettä, vaan valinnan. Lajiteltu taulukko on erikoislaatuinen toteutus: kun se on tarpeen, se on ehdottomasti välttämätöntä. Muussa tapauksessa tuen aiheuttama kuormitus on rasite. Toteutuksemme on yleisempi ja useimmissa tapauksissa riittävä, sillä se tukee laajaa käyttäjäkuntaa. Valitettavasti, jos käyttäjä ehdottomasti haluaa lajiteltua taulukkoa, toteutuksemme ei voi tukea sitä. Ei ole olemassa käyttäjälle tapaa korvata yleisempiä toteutuksia min()-, max()- ja find()-funktioista. Olemme itse asiassa valinneet yleisesti hyödyllisen toteutuksen, joka on riittämätön tietyissä erikoistilanteissa.

Toisaalta, eräälle toiselle käyttäjäkunnalle toteutuksemme on liian erikoislaatuinen: indeksin raja-arvojen tarkistus lisää jokaisen elementin käsittelyn kuormaa. Vähennämme kuormitusta suunnitelmassamme (katso vaihe 8 kohdassa 2.3) olettamuksella, että nopeudesta ei ole apua, jos olemme väärässä. Tämä olettamus ei kuitenkaan pidä sisällään koko totuutta ainakaan yhdelle käyttäjiemme pääosasta: reaaliaikaista virtuaali-immersiota eli virtuaalitodellisuutta. Taulukot edustavat tässä tilanteessa monimutkaisen 3D-geometrian jaettuja ulottuvuuksia. Tilanne menee satunnaisessa virheessä ohitse niin nopeasti, että se ei ole yleensä havaittavissa, mutta jos yleinen käsittely on liian hidasta, immersioefekti tunkee läpi. Toteutuksemme on, vaikkakin huomattavasti turvallisempi kuin raja-arvoja tutkimaton taulukkoluokka, epäkäytännöllinen tälle sovellusalueelle.

Kuinka voisimme tukea näiden kolmen eri käyttäjäkunnan tarpeita? Ratkaisut ovat jo läsnä koodissa; enemmän tai vähemmän. Esimerkiksi raja-arvojen tarkistuksemme on lokalisoitu indeksioperaattoriin. Poista check_range()-funktion käynnistys, nimeä taulukko uudelleen, niin meillä on kaksi toteutusta: yksi raja-arvotarkistuksella ja yksi ilman. Kopioi lisää koodia, muokkaa se käsittelemään taulukkoa lajiteltuna, ja niin meillä on tuki lajitellulle taulukolle:

```
// Lajittelematon, ilman raja-arvojen tarkistusta
class IntArray{ ... };

// Lajittelematon, raja-arvojen tarkistuksella
class IntArrayRC{ ... };

// Lajiteltu, ilman raja-arvojen tarkistusta
class IntSortedArray{ ... };
```

Mitkä ovat tämän ratkaisun varjopuolet?

1. Meidän pitää ylläpitää kolmea taulukkototeutusta, jotka sisältävät huomattavasti samanlaista koodia. Pitäisimme parempana ratkaisua, jossa meillä olisi yksi yhteinen kopio koodista, jonka kolme taulukkoluokkaa jakaisivat kuten myös kaikki muut taulukkoluokat, joita myöhemmin tuemme (ehkäpä lajiteltu taulukko raja-arvotarkistuksin).

2. Koska kolme taulukkototeutusta ovat toisistaan poikkeavia tyyppejä, täytyy kirjoittaa erilliset funktiot niiden operoimiseksi, vaikka funktioiden yleiset operaatiot voivat olla samanlaisia. Esimerkiksi:

```
void process_array( IntArray& );
void process_array( IntArrayRC& );
void process_array( IntSortedArray& );
```

   Pitäisimme parempana kirjoittaa yksi funktio, joka ei ainoastaan hyväksyisi kaikkia olemassaolevia taulukkoluokkiamme, vaan myös kaikki tulevaisuuden taulukkoluokat edellyttäen, että sama operaatiojoukko käy jokaiselle luokalle.

Oliosuuntautunut paradigma antaa meille täsmälleen nämä mahdollisuudet. Kohta 1 tehdään mekanismilla nimeltään *periytyminen* (*inheritance*). Kun IntArrayRC-luokka (tarkoittaa IntArray-luokkaa raja-arvotarkistuksin) periytyy IntArray-luokasta, sillä on pääsy IntArray-luokan tietojäseniin ja jäsenfunktioihin ilman, että meidän pitäisi ylläpitää koodista kahta eri versiota. Uuden luokan pitää sisältää vain ne tietojäsenet ja jäsenfunktiot, jotka ovat välttämättömiä sen lisämerkityksen toteuttamiseksi.

C++:ssa kutsutaan *kantaluokaksi* (*base class*) luokkaa, josta periydytään, ja joka tässä tapauksessa on IntArray. Uuden luokan sanotaan *johdetun* (*derived*) kantaluokasta. Me kutsumme sitä kantaluokasta *johdetuksi luokaksi* eli *alityypiksi* (*subtype*). Sanomme, että IntArrayRC on eräänlainen IntArray, jolla on erikoistuki indeksin raja-arvojen tarkistukselle. Alityyppi jakaa *yhteisen rajapinnan* (*common interface*) kantaluokkansa kanssa — tarkoittaa yhteistä joukkoa julkisia operaatioita. Tämä yhteisen rajapinnan jakaminen sallii kantaluokan ja alityypin käytön ohjelmassa samanarvoisesti välittämättä siitä, mikä olion varsinainen tyyppi on. Tietyssä mielessä yhteinen rajapinta kapseloi yksittäisten alityyppien tyyppikohtaiset yksityiskohdat. Tyyppi/alityyppi-suhde luokkien välillä muodostaa *periytymis- eli johdannaishierarkian*. Tässä on esimerkiksi toteutus `swap()`-funktiosta, joka ei ole jäsen, mutta joka saa pääargumenttinaan viittauksen IntArray-kantaluokkaolioon. Funktio vaihtaa keskenään kaksi elementtiä, joita indeksoidaan arvoilla `i` ja `j`.

```
#include <IntArray.h>

void swap( IntArray &ia, int i, int j )
{
    int tmp = ia[ i ];
    ia[ i ] = ia[ j ];
    ia[ j ] = tmp;
}
```

Tässä on kolme sallittua `swap()`-funktion käynnistystä:

```
IntArray      ia;
IntArrayRC    iarc;
IntSortedArray ias;
```

```
// ok: ia on IntArray
swap( ia, 0, 10 );

// ok: iarc IntArray:n alityyppi
swap( iarc, 0, 10 );

// ok: ias on myös IntArray:n alityyppi
swap( ias, 0, 10 );

// virhe: string ei ole alityyppi ...
string str( "ei ole IntArray!" );
swap( str, 0, 10 );
```

Jokaisessa näissä kolmessa taulukkoluokassa on oma toteutuksensa indeksioperaattorista. Me vaadimme tietysti, että kun

```
swap( iarc, 0, 10 );
```

käynnistetään, IntArrayRC:n indeksioperaattori käynnistetään, ja kun

```
swap( ias, 0, 10 );
```

käynnistetään, IntSortedArray:n indeksioperaattori käynnistetään jne. swap()-funktion käynnistämä indeksioperaattori pitää mahdollisesti vaihtaa joka kerta ja sen pitää päättyä sen taulukon todelliseen tyyppiin, jonka elementtejä vaihdetaan. Tämä tehdään automaattisesti C++:ssa mekanismilla nimeltään *virtuaalifunktio* (*virtual function)*.

Syntaktiset muutokset, jotka ovat välttämättömiä valmistellessamme IntArray-luokkaamme periytymistä varten, ovat vähäisiä: meidän pitää (valinnaisesti) vähentää kapseloinnin tasoa salliaksemme johdettujen luokkien pääsyn muuhun kuin julkiseen toteutukseen ja meidän pitää eksplisiittisesti yksilöidä, mitkä funktiot haluamme ratkaistavan virtuaalimekanismin kautta. Merkittävä muutos on tavassamme suunnitella luokka, jonka tarkoituksena on toimia kantaluokkana.

Oliopohjaisessa suunnittelussa luokalle on yleensä yksi tekijä ja monta käyttäjää. Luokan tekijä suunnittelee ja usein toteuttaa luokan. Käyttäjät opettelevat julkisen rajapinnan, jonka tekijä on heille antanut käytettäväksi. Tämä toimien erottelu heijastaa luokan jakamista yksityiseen ja julkiseen käsittelytasoon.

Periytymisessä on nyt useita luokan tekijöitä: yksi, joka on tehnyt kantaluokan toteutuksen (ja ehkä muutaman johdetun luokan) ja yksi tai useampi, jotka ovat tehneet johdettuja luokkia läpi koko periytymishierarkian elinajan. Tämä toiminta on myös toteutustoimintaa; alityypin tekijän pitää usein (mutta ei aina) käsitellä kantaluokan toteutusta. Jotta se onnistuisi ja yhä estettäisiin yleinen pääsy toteutukseen, on pääsylle olemassa yksi lisätaso: protected. Luokan suojatun (protected) osan tietojäsenet ja jäsenfunktiot, joita tavalliset ohjelmat eivät edelleenkään pääse käsittelemään, ovat yhä johdetun luokan käytettävissä. (Kaikki, mitä on sijoitettu kantaluokan yksityiseen osaan, on vain itse luokan käytettävissä eivätkä yhdenkään johdetun luokan.) Tässä on uudistettu IntArray-luokkamme:

```
class IntArray {
```

```
public:
    // muodostajat
    explicit IntArray( int size = DefaultArraySize );
    IntArray( int *array, int array_size  );
    IntArray( const IntArray &rhs );

    // virtuaalinen muodostaja!
    virtual ~IntArray() { delete [] ia; }

    // yhtä- ja erisuuruusoperaatiot:
    bool operator==( const IntArray& ) const;
    bool operator!=( const IntArray& ) const;

    IntArray& operator=( const IntArray& );
    int size() const { return _size; }

    // olemme poistaneet indeksin tarkistuksen ...
    virtual int& operator[](int index) { return ia[index]; }
    virtual void sort();

    virtual int min() const;
    virtual int max() const;
    virtual int find( int value ) const;

protected:
    // katso selitys kappaleesta 13.5
    static const int DefaultArraySize = 12;
    void init( int sz, int *array );

    int _size;
    int *ia;
};
```

Kriteerit sille, määritelläänkö public-jäsentä luokalle, eivät muutu oliopohjaisen ja oliokeskeisen suunnittelun välillä. Uudelleensuunnittelemamme IntArray-luokka on tarkoitettu toimimaan kantaluokkana ja siinä esitellään yhä muodostajat, tuhoaja, indeksioperaattori, min(), max() jne. julkisina jäseninä. Nämä jäsenet jatkavat julkisen rajapinnan muodostajina, mutta nyt rajapinta toimii ei vain IntArray-luokalle, vaan myös periytymishierarkialle, joka siitä on johdettu.

Uusi suunnittelukriteeri on esitellä luokan muut kuin julkiset jäsenet joko protected- tai private-jäseninä. Jäsenestä tehdään luokalle private, jos haluamme estää siitä edelleen johdettujen luokkien suoran pääsyn tuohon jäseneen. Jäsenestä tehdään protected, jos uskomme, että sillä tavoin saamme siitä johdetuille luokille suoran pääsyn operaatioon tai tallennettuun tietoon tehokkaasti. IntArray-luokassamme olemme tehneet kaikista tietojäsenistä suojattuja (protected), jotta sallisimme tulevien johdettujen luokkien pääsyn IntArray-luokan toteutuksen yksityiskohtiin.

Toinen suunnittelunäkökohta luokalle, jonka halutaan toimivan kantaluokkana, on tyyppiriippuvaisten jäsenfunktioiden yksilöiminen. Otsikoimme nämä funktiot virtuaalisiksi nimellä

virtual.

Tyyppiriippuvainen jäsenfunktio on sellainen, jonka algoritmin päättää toteutus tai tietyn kantaluokan tai johdetun luokan käyttäytyminen. Esimerkiksi indeksioperaattorin toteutus on yksilöllinen jokaiselle taulukkotyypillemme. Tästä syystä esittelemme sen virtuaaliseksi.

Kahden yhtäsuuruusoperaattorin ja size()-jäsenfunktion toteutukset ovat riippumattomia taulukkotyypistä, jonka kanssa niitä käytetään. Tästä syystä emme esittele niitä virtuaalisiksi.

Käynnistyskutsun muuhun kuin virtuaalifunktioon kääntäjä valitsee käännöksen aikana. Virtuaalifunktion kutsun ratkaisua viivytetään suoritukseen saakka. Suoritettavassa ohjelmassa virtuaalifunktion ilmentymä, joka perustuu varsinaisen kantaluokan tai johdetun luokan tyyppiin, valitaan käynnistettävän funktion perusteella kutsuhetkellä. Katsotaanpa seuraavaa esimerkkiä:

```
void init( IntArray &ia )
{
    for ( int ix = 0; ix < ia.size(); ++ix )
        ia[ ix ] = ix;
}
```

Muodollinen parametri ia voi viitata IntSortedArray-, IntArrayRC- tai IntArray-luokan olioon (esittelemme luokkamme johdannaiset lyhyesti pian). size(), joka ei ole virtuaalinen funktio, ratkaistaan ja laajennetaan välittömästi kääntäjän toimesta. Indeksioperaattoria ei kuitenkaan voida yleensä ratkaista, ennen kuin jokainen for-silmukan toisto on suoritettu, koska käännöksen aikana ei tiedetä taulukon todellista tyyppiä, johon ia osoittaa.

(Katsomme virtuaalifunktioita huomattavasti tarkemmin luvussa 17. Siinä käsitellään myös virtuaalisia tuhoajia ja mahdollista tehottomuutta virtuaalifunktioiden suunnittelussa. Julkaisussa [LIPPMAN96a] käsitellään toteutus- ja suorituskykyaiheita yksityiskohtaisesti.)

Kun olemme päättäneet suunnitelmastamme, sen toteutus C++:lla on suoraviivaista. Esimerkiksi tässä on täydellinen IntArrayRC-luokka, joka on johdettu määrittely. Se on sijoitettu riippumattomaan otsikkotiedostoon IntArrayRC.h ja sisältää IntArray.h-otsikkotiedoston, joka puolestaan sisältää IntArray-luokan määrittelyn:

```
#ifndef IntArrayRC_H
#define IntArrayRC_H

#include "IntArray.h"

class IntArrayRC : public IntArray {
public:
  IntArrayRC( int sz = DefaultArraySize );
  IntArrayRC( int *array, int array_size );
  IntArrayRC( const IntArrayRC &rhs );

  virtual int& operator[]( int );

private:
  void check_range( int );
```

```
};

#endif
```

IntArrayRC-luokassa pitää määritellä vain ne toteutuksen osat, jotka eroavat tai ovat lisänä IntArray-luokan toteutukselle.

1. Siinä pitää olla oma ilmentymä indeksioperaattorista — sellainen, jossa on raja-arvojen tarkistus.
2. Siinä pitää olla operaatio, joka tekee varsinaisen raja-arvotarkistuksen. (Koska se ei ole osa julkista rajapintaa, esittelemme sen yksityiseksi.)
3. Siinä pitää olla oma joukko automaattisia alustusfunktioita — eli oma muodostajajoukko.

IntArray-luokan tietojäsenet ja jäsenfunktiot ovat kaikki IntArrayRC-luokan käytettävissä aivan kuin se olisi ne itse määritellyt. Seuraava merkitsee juuri tätä:

```
class IntArrayRC : public IntArray
```

Kaksoispiste määrittelee, että IntArrayRC on johdettu IntArray-luokasta. (public-avainsana ilmaisee, että johdettu luokka jakaa kantaluokan julkisen rajapinnan. IntArrayRC-luokan olion tyyppiä voidaan käyttää missä tahansa tilanteessa, jossa se on ohjelmoitu käyttämään kantaluokan oliotyyppejä kuten swap()-esimerkissämme. Tämä selitetään yksityiskohtaisesti luvussa 18.) IntArrayRC-luokan voidaan ajatella laajentavan IntArray-luokkaa antamalla indeksin raja-arvotarkistuksen lisäpiirteen. Tässä on indeksioperaattorin toteutus:

```
inline int&
IntArrayRC::operator[]( int index )
{
    check_range( index );
    return ia[ index ];
}
```

Tässä check_range() on toteutettu välittömänä jäsenfunktiona, joka käynnistää assert()-makron (katso kappaleesta 1.3 assert()-makron käsittely):

```
#include <cassert>

inline void
IntArrayRC::check_range( int index )
{
    assert( index >= 0 && index < size );
}
```

(Teemme check_range()-funktiosta erillisen esitelläksemme yksityisen jäsenfunktion ja kapseloidaksemme raja-arvojen tarkistuksen siltä varalta, että haluamme myöhemmin muuttaa tapaa, jolla raja-arvovirheet hoidetaan; ehkäpä korvaamme assert()-makron poikkeusten käsittelyn tuella.)

Johdetun luokan olio muodostuu itse asiassa useista osista: jokainen kantaluokka on *alioliluokka* vastikään määritellylle johdetulle luokalle. Johdetun luokkaolion alustus tehdään jokaisella kantaluokan muodostajan automaattisella käynnistyksellä, jotka alustavat kantaluokkaan liittyvät alioliot, ja sen jälkeen käynnistetään johdetun luokan muodostaja. Suunnittelun kannalta johdetun luokan muodostajan tulisi alustaa vain ne tietojäsenet, jotka luokka itse on määritellyt, eikä sen kantaluokan tietojäseniä.

Vaikka esittelemme luokkakohtaisen version indeksioperaattorista ja yksityisestä check_range()-apufunktiosta, emme esittele lisää tietojäseniä, jotka vaativat alustusta. Siitä syystä voimme kohtuudella olettaa, että kantaluokan perityt muodostajat ovat riittäviä ja että meidän ei tarvitse tehdä muodostajia IntArrayRC-luokalle — loppujen lopuksi niillä ei ole mitään tehtävää!

Itse asiassa tarvitsemme kuitenkin muodostajia IntArrayRC-luokalle, koska johdettu luokka ei koskaan peri kantaluokan muodostajia (eikä myöskään tuhoajaa tai kopioinnin sijoitusoperaattoria) ja koska tarvitsemme jonkun rajapinnan, jonka kautta voimme välittää tarvittavat argumentit IntArray-kantaluokan muodostajille.

Oletetaan esimerkiksi, että määrittelemme IntArrayRC-olion kuten seuraavassa:

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13 };
IntArrayRC iarc( ia, 8 );
```

Kuinka voimme välittää ia:n ja arvon 8 IntArray-kantaluokan muodostajalle? (Myönnettäköön, jos IntArray-luokan muodostaja perittäisiin, ei meillä olisi tätä ongelmaa. Itse asiassa meillä olisi muita vakavampia ongelmia, mutta meillä ei ole tässä tilaa saada sinua vakuuttuneeksi siitä.) Joka tapauksessa johdetun luokan muodostajan syntaksilla on rajapinta argumenttien välitykseen kantaluokan muodostajalle. Esimerkiksi tässä on kaksi meidän tarvitsemaamme IntArrayRC-luokan muodostajaa (kerromme paljon enemmän muodostajista luvuissa 14 ja 17 mukaan lukien selityksen siitä, miksi emme tarvitse IntArrayRC-luokalle kopiointimuodostajaa):

```
inline IntArrayRC::IntArrayRC( int sz)
    : IntArray( sz ) {}

inline IntArrayRC::IntArrayRC( const int *iar, int sz )
    : IntArray( iar, sz ) {}
```

Sitä muodostajan osaa, joka on merkitty kaksoispisteellä, kutsutaan *jäsenen alustusluetteloksi* (*member initialization list*). Se on mekanismi, jolla IntArray-muodostajalle välitetään sen argumentit. Molempien IntArrayRC-muodostajien rungot ovat tyhjiä, koska niiden työ on vain välittää argumenttinsa vastaavalle IntArray-muodostajalle. Emme tee eksplisiittistä tuhoajaa IntArrayRC-luokalle, koska johdettu luokka ei esittele yhtään tietojäsentä, joka vaatisi tuhoamista. Perityn IntArray-luokan jäsenten vaatiman tuhoamisen hoitaa IntArray-luokan tuhoaja.

Kantaluokan tuhoajaa käytetään automaattisesti kantaluokan alioliolle johdetun luokkaolion sisällä huolimatta siitä, onko johdetulle luokalle itselleen määritelty tuhoajaa.

IntArrayRC.h-otsikkotiedosto pitää sisällään kaksi IntArrayRC-luokan määrittelyä ja kaikkien niiden välittömien jäsenfunktioiden määrittelyn, jotka on määritelty luokan ulkopuolella. Jos olisimme määritelleet muita kuin välittömiä jäsenfunktioita, olisimme sijoittaneet ne niihin liittyvään ohjelmakoodin tekstitiedostoon IntArrayRC.C.

Tässä on pieni ohjelma, joka käyttää IntArray-luokan ja kahden IntArrayRC-luokan hierarkiaa:

```
#include <iostream>
#include <IntArray.h>
#include <IntArrayRC.h>

extern void swap(IntArray&,int,int);

int main()
{
    int array[ 4 ] = { 0, 1, 2, 3 };
    IntArray ia1( array, 4 );
    IntArrayRC ia2( array, 4 );

    // virhe: ylitys yhdellä: tulisi olla size-1
    // tätä ei IntArray-olio saa siepattua
    cout << "swap() ja IntArray ia1\n";
    swap( ia1, 1, ia1.size() );

    // ok: IntArrayRC-olio sieppasi
    cout << "swap() ja IntArrayRC ia2\n";
    swap( ia2, 1, ia2.size() );

    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tuloksen:

```
swap() ja IntArray ia1
swap() ja IntArrayRC ia2
Assertion failed:  index >= 0 && index < size
```

C++ tukee kahta periytymisen lisämuotoa: moniperiytyminen, jossa luokka on johdettu kahdesta tai useammasta kantaluokasta sekä virtuaaliperiytyminen, jossa yksittäistä kantaluokan ilmentymää jakavat useat johdetut luokat. Viivytämme tämän käsittelyä lukuun 18 saakka. Lisäksi oliosuuntautuneessa ohjelmoinnissa on mahdollisuus kysyä milloin tahansa ohjelman suorituksen aikana kantaluokan viittaukselta tai osoittimelta, mihin todelliseen tyyppiin se viittaa. Tämä tehdään suorituksenaikaisen tunnistuspiirteen (RTTI) avulla. RTTI-piirrettä käsitellään kohdassa 19.1.

Harjoitus 2.8

Tyypin/alityypin periytyminen kuvaa yleensä suhdetta *on_eräänlainen* (*isA_kindOf*): raja-arvojen tarkistusfunktio ArrayRC on eräänlainen Array, Book on eräänlainen LibraryRentalMaterial, AudioBook on eräänlainen Book jne. Mitkä seuraavista pareista kuvaavat on_eräänlainen-suhdetta?

(a) jäsenfunktio on_eräänlainen funktio
(b) jäsenfunktio on_eräänlainen luokka
(c) muodostaja on_eräänlainen jäsenfunktio
(d) lentokone on_eräänlainen kulkuneuvo
(e) moottori on_eräänlainen kuorma-auto
(f) ympyrä on_eräänlaista geometriaa
(g) neliö on_eräänlainen suorakulmio
(h) auto on_eräänlainen lentokone
(i) lainaaja on_eräänlainen kirjasto

Harjoitus 2.9

Yksilöi, mitkä seuraavista operaatioista ovat todennäköisesti tyyppiriippuvaisia ja siten ehdolla virtuaalifunktioiksi ja mitkä ovat joko kaikkien luokkien yhteisiä tai yksilöllisiä yksittäiselle kantaluokalle tai johdetulle luokalle.

(a) rotate(); (b) print();
(c) size(); (d) dateBorrowed();
(e) rewind(); (f) borrower();
(g) is_late(); (h) is_on_loan();

Harjoitus 2.10

On ollut joitakin väittelyitä suojatun käsittelytason käytöstä. Jotkut väittävät, että koska voidaan päästä suoraan kantaluokan jäseniin, suojatun käsittelytason käyttö johdetuille luokille rikkoo kapseloinnin merkitystä, ja ovat sitä mieltä, että kantaluokan toteutuksen yksityiskohtien pitäisi olla yksityisiä. Toiset taas väittävät, että ilman suoraa pääsyä kantaluokan jäseniin ei johdetun luokan toteutusta voi tehdä riittävän tehokkaaksi ja ilman protected-avainsanaa luokan suunnittelijan olisi pakko tehdä kantaluokan jäsenistä julkisia. Mitä mieltä sinä olet?

Harjoitus 2.11

Toinen kiistelynaihe liittyy tarpeeseen esitellä jäsenfunktio eksplisiittisesti virtuaaliseksi. Jotkut väittävät sen merkitsevän, että jos luokan suunnittelija ei huomaa funktion tarvetta olla virtuaalinen, johdetun luokan suunnittelija on neuvoton ohittamaan välttämättömän funktion. He neuvovat tekemään kaikista jäsenfunktioista virtuaalisia. Toisaalta virtuaalifunktiot ovat tehottomampia kuin

funktiot, jotka eivät ole virtuaalisia[3]. Koska niitä ei voi tehdä välittömiksi (välittömäksi muuntaminen tapahtuu käännöksen aikana ja virtuaalifunktiot ratkaistaan suorituksen aikana), ne voivat olla ohjelman suorituksenaikaisen tehottomuuden lähde etenkin pienissä, usein käynnistettävissä funktioissa kuten Array-luokkamme koon kysely. Mitä mieltä olet nyt?

Harjoitus 2.12

Jokainen seuraavista abstraktioista muodostuu implisiittisesti abstraktien alityyppien perheestä. Esimerkiksi abstraktio LibraryRentalMaterial sisältää implisiittisesti Books, Puppets, Videos jne. Valitse yksi seuraavista ja yksilöi tuon abstraktion alityyppien hierarkia. Määritä pieni julkinen rajapinta tuolle hierarkialle muodostajat mukaan lukien. Yksilöi, mitkä funktiot ovat virtuaalisia, jos sellaisia on, ja kirjoita lyhyt pseudokoodi ohjelmasta kokeillaksesi julkista rajapintaa.

(a) Points	(b) Employees
(c) Shapes	(d) TelephoneNumbers
(e) BankAccounts	(f) CourseOfferings

2.5 Geneerinen suunnittelu

IntArray-luokkamme tarjoaa hyödyllisen vaihtoehdon esimääritellylle kokonaislukujen taulukkotyypille. Mutta mitä sanomme käyttäjille, jotka haluaisivat käyttää Array-luokkaa double- tai string-tyypille? double-tyyppisen Array-luokan ja IntArray-luokan välisen toteutuksen ero on yksinkertaisesti vain elementin tyyppi, jonka näiden pitää sisältää; itse koodi pysyy muuttumattomana.

C++:n mallipiirre antaa keinon *parametroida* tyyppejä ja arvoja, joita luokan tai funktion määrityksessä käytetään (jätämme arvoparametrien käsittelyn kohtaan 10.1). Nämä parametrit toimivat paikanpitäjinä muutoin muuttumattomassa koodissa. Myöhemmin parametrit sidotaan todellisiin tyyppeihin, jotka ovat joko sisäisiä tai käyttäjän määrittelemiä. Esimerkiksi Array-luokkamallissa parametroimme sen sisältämien elementtien tyypin. Myöhemmin *instantioimme* tyyppikohtaiset ilmentymät, kuten int-, double- ja string-Array-luokan. Voimme käyttää näitä kolmea ilmentymää ohjelmassamme aivan kuin olisimme koodanneet ne käsin eksplisiittisesti. Katsotaanpa nyt, kuinka IntArray-luokka muutetaan Array-luokan malliksi. Tässä on sen määrittely:

```
template < class elemType >
class Array {
public:
    // parametroi elementtityyppi
    explicit Array( int size = DefaultArraySize );
    Array( elemType *array, int array_size  );
    Array( const Array &rhs );
```

3. Katso kirjasta [LIPPMAN96a] yksityiskohtainen käsittely virtuaalifunktioiden suorituskykyyn liittyvistä aiheista.

```
    virtual ~Array() { delete [] ia; }

    bool operator==( const Array& ) const;
    bool operator!=( const Array& ) const;

    Array& operator=( const Array& );
    int size() const { return _size; }

    virtual elemType& operator[](int index){ return ia[index]; }
    virtual void sort();

    virtual elemType min() const;
    virtual elemType max() const;
    virtual int      find( const elemType &value ) const;

protected:
    static const int DefaultArraySize = 12;

    int _size;
    elemType *ia;
};
```

Avainsana template esittelee mallin. Parametrit ovat kulmasulkujen (<,>) sisällä — tässä tapauksessa yksi parametri, elemType. Avainsana class ilmaisee, että parametri edustaa tyyppiä. Tunnus elemType toimii todellisena parametrin nimenä. Sen seitsemän esiintymää Array-luokassamme toimivat kuin paikanpitäjinä todelliselle tyypille.

Jokaisessa Array-luokan instantioinnissa tyypiksi int, double, string jne. elemType-parametri korvataan todellisella tyypillä. Tässä on esimerkki, kuinka Array-luokan mallia voitaisiin käyttää:

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

int main()
{
    const int array_size = 4;

    // elemType-parametrista tulee int
    Array<int> ia(array_size);

    // elemType-parametrista tulee double
    Array<double> da(array_size);

    // elemType-parametrista tulee char
    Array<char> ca(array_size);

  int ix;
```

```
    for ( ix = 0; ix < array_size; ++ix ) {
            ia[ix] = ix;
            da[ix] = ix * 1.75;
            ca[ix] = ix + 'a';
    }

    for ( ix = 0; ix < array_size; ++ix )
            cout << "[ " << ix << " ]  ia: "  << ia[ix]
                    << "\tca: " << ca[ix]
                    << "\tda: " << da[ix] << endl;

      return 0;
}
```

Tässä esimerkissä määrittelemme kolme itsenäistä Array-luokkamallin ilmentymää:

```
Array<int>    ia(array_size);
Array<double> da(array_size);
Array<char>   ca(array_size);
```

Nämä ilmentymät on esitelty niin, että luokkamallin nimen jälkeen tulee luettelo todellisista tyypeistä kulmasulkujen sisällä. Mitä tapahtuu, kun määrittelemme luokan malliolion kuten ia, da tai ca? Kääntäjän pitää varata muistia siihen liittyvälle oliolle. Jotta niin voitaisiin tehdä, muodolliset malliparametrit sidotaan määritettyihin todellisiin parametreihin. Oliolle ia Array-luokan mallin instantiointi tuottaa seuraavat luokan tietojäsenet, jossa elemType on sidottu int-tyyppiin:

```
// Array<int> ia(array_size);
int _size;
int *ia;
```

Tuloksena on luokka, joka on yhtä kuin aikaisemmin käsin toteuttamamme IntArray-luokka. Olion da jäsenistä tulee

```
// Array<double> da(array_size);
int _size;
double *ia;
```

jossa elemType on sidottu double-tyyppin. Samoin olion ca jäsenistä tulee

```
// Array<char> ca(array_size);
int _size;
char *ia;
```

jossa elemType on sidottu char-tyyppiin.

Mitä sanottavaa on malliluokan jäsenfunktioista? Niitä kaikkia ei instantioida automaattisesti. Sen sijaan vain ne jäsenfunktiot instantioidaan, joita ohjelma todellisuudessa käyttää, ja tämä tapahtuu yleensä ohjelman rakentamisen eri vaiheessa. (Käsittelemme tätä yksityiskohtaisesti kohdassa 16.8.)

Kun ohjelma käännetään ja suoritetaan, se tuottaa seuraavan tulostuksen:

```
[ 0 ]  ia: 0 ca: a   da: 0
[ 1 ]  ia: 1 ca: b     da: 1.75
[ 2 ]  ia: 2 ca: c     da: 3.5
[ 3 ]  ia: 3 ca: d     da: 5.25
```

Mallimekanismi tukee myös oliokeskeistä ohjelmointia. Luokkamalli voi toimia sekä kantaluokkana että johdettuna luokkana. Tässä on määrittely raja-arvoja tarkistavasta Array-luokan mallista:

```
#include <cassert>
#include "Array.h"

template <class elemType>
class ArrayRC : public Array<elemType> {
public:
    ArrayRC( int sz = Array<elemType>::DefaultArraySize )
            : Array< elemType >( sz ){};

    ArrayRC( elemType *ia, int sz )
            : Array< elemType >( ia, sz ) {}

    ArrayRC( const ArrayRC &rhs )
            : Array< elemType >( rhs ) {}

    virtual elemType&
    operator[]( int index )
  {
             assert( index >= 0 && index < Array<elemType>::size() );
           return ia[ index ];
    }

private:
    // ...
};
```

Jokainen ArrayRC-luokan instantiointi saa aikaan siihen liittyvän Array-luokan malli-ilmentymän. Esimerkiksi määrittely

```
ArrayRC<int> ia_rc( 10 );
```

saa aikaan int-ilmentymän instantioinnin sekä Array- että ArrayRC-luokalle. ia_rc käyttäytyy täsmälleen samalla tavalla kuin edellisen kappaleen malliton ilmentymä. Tätä kuvataksemme kirjoittakaamme uudelleen aikaisempi ohjelma ja kokeilkaamme Array- ja ArrayRC-luokkien mallityyppejä. Ensiksi, jotta voimme tukea lausetta

```
// swap()-funktion pitää nyt olla myös malli
swap( ia1, 1, ia1.size() );
```

pitää määritellä swap() funktiomalliksi. Esimerkiksi:

```
#include "Array.h"
```

```
template <class elemType>
void swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}
```

Jokainen swap()-kutsu generoi vastaavan ilmentymän, joka riippuu array-tyypistä. Tässä on uudelleenkirjoitettu ilmentymä main()-funktiosta, joka käyttää Array- ja ArrayRC-luokkamalleja:

```
#include <iostream>

#include "Array.h"
#include "ArrayRC.h"

template <class elemType>
inline void
swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}

int main()
{
    Array<int>   ia1;
    ArrayRC<int> ia2;

    cout << "swap() ja Array<int> ia1\n";
    int size = ia1.size();
    swap( ia1, 1, size );

    cout << "swap() ja ArrayRC<int> ia2\n";
    size = ia2.size();
    swap( ia2, 1, size );

    return 0;
}
```

Tämän ohjelman tulokset ovat samoja kuin mallittoman IntArray-luokan toteutuksessa.

Harjoitus 2.13

Kun on annettu seuraavat tyyppiesittelyt:

```
template <class elemType> class Array;
enum Status { ... };
typedef string *Pstring;
```

niin mitkä, seuraavista oliomäärittelyistä ovat virheellisiä, vai onko yksikään?

```
(a) Array< int*& > pri( 1024 );
(b) Array< Array<int> > aai( 1024 );
(c) Array< complex< double > > acd( 1024 );
(d) Array< Status > as( 1024 );
(e) Array< Pstring > aps( 1024 );
```

Harjoitus 2.14

Kirjoita seuraava luokka uudelleen ja tee siitä luokkamalli:

```
class example1 {
public:
    example1( double min, double max );
    example1( const double *array, int size );

    double& operator[]( int index );
    bool operator==( const example1& ) const;

    bool insert( const double*, int );
    bool insert( double );

    double min() const { return _min; };
    double max() const { return _max; };

    void min( double );
    void max( double );

    int count( double value ) const;

private:
    int size;
    double *parray;
    double _min;
    double _max;
};
```

Harjoitus 2.15

Kun on olemassa seuraava luokkamalli:

```
template <class elemType>
class Example2 {
public:
    explicit Example2( elemType val = 0 )
        : _val( val ){}

    bool min( elemType value )     { return _val < value; }
    void value( elemType new_val ) { _val = new_val;      }
    void print( ostream &os )      { os << _val;          }
```

```
private:
    elemType _val;
};

template<class elemType>
ostream& operator<<( ostream &os, const Example2<elemType> &ex )
      { ex.print( os ); return os; }
```

mitä tapahtuu, kun kirjoitamme seuraavan?

```
(a) Example2< Array<int>* > ex1;
(b) ex1.min( &ex1 );
(c) Example2< int > sa( 1024 ), sb;
(d) sa = sb;
(e) Example2< string > exs( "Walden" );
(f) cout << "exs: " << exs << endl;
```

Harjoitus 2.16

Example2-määrittelyssämme kirjoitamme

```
explicit Example2( elemType val = 0 )
      : _val( val ){}
```

Aikomuksemme on määrittää oletusarvo niin, että käyttäjä voi kirjoittaa jommankumman:

```
Example2< Tyyppi > ex1( arvo );
Example2< Tyyppi > ex2;
```

Toteutuksemme kuitenkin rajoittaa Tyyppiä niin, että sen pitää olla sellaisten tyyppien alijoukko, joiden alustaminen sallitaan arvolla 0. (Esimerkiksi merkkijono-olion alustaminen arvolla 0 on virhe[4].) Samalla tavalla, jos Tyyppi ei tue tulostusoperaattoria, print()-funktion käynnistys (ja siksi myös Example2:n tulostusoperaattori) epäonnistuu. Ellei Tyyppi tue pienempi kuin -operaattoria, min()-funktion käynnistys epäonnistuu.

Kielellä ei ole keinoja ilmaista implisiittistä rajoitusta Tyypille, jonka mallin kanssa se voidaan instantioida. Ohjelmoija huomaa nämä rajoitukset pahimmassa tapauksessa ohjelman käännöksen epäonnistuttua. Tulisiko sinun mielestäsi kielen tukea syntaksia tyyppirajoituksille? Jos tulisi tukea, ilmaise syntaksi ja kirjoita Example2-määrittely sitä käyttäen. Ellei tulisi tukea, perustele, miksi ei.

Harjoitus 2.17

Edellisessä harjoituksessa sanoimme, että jos Tyyppi ei tue tulostus- tai pienempi kuin -operaattoria, yritys käynnistää joko funktio print() tai min() saa aikaan virheen. C++-standardin mukaan virheitä ei generoida, kun malliluokka luodaan, vaan kun (tai jos) print()- tai min()-funktio käyn-

4. Yleinen ohjelmaratkaisu tähän ongelmaan on tehdä seuraavasti:
   Example2(elemType nval = elemType()) : _val(nval){}

nistetään. Onko tämä sinun mielestäsi oikeaa kielen semantiikkaa? Tulisiko virheestä ilmoittaa mallin määrittelyvaiheessa? Miksi tai miksi ei?

2.6 Poikkeuspohjainen suunnittelu

Poikkeukset (exceptions) ovat ohjelman suorituksenaikaisia poikkeustapauksia kuten taulukon indeksin rajojen ylitys, kyvyttömyys avata määrättyä tiedostoa, ohjelman dynaamisen muistin katoaminen jne. Yleensä ohjelmoijat kehittävät oman tyylinsä poikkeusten käsittelyyn, mikä johtaa erilaisiin koodauskäytäntöihin, joita on todennäköisesti hankalaa integroida yhteen sovellukseen.

Poikkeusten käsittely tarjoaa standardin kielitasoisen piirteen ohjelman suorituksenaikaisiin poikkeustapauksien reagointiin. Se tukee yhdenmukaista syntaksia ja tyyliä, joka puolestaan tukee yksilöllisten ohjelmoijien loppusilausta. Poikkeusten käsittelypiirre voi merkittävästi vähentää ohjelmakoodin kokoa ja monimutkaisuutta eliminoimalla tarvetta, että joka paikassa testataan eksplisiittisesti epänormaaleja tiloja ja tuotetaan koodia testaamaan poikkeustapauksia tiettyihin otsikoituihin koodiosiin.

Poikkeusten käsittelypiirteen pääkomponentit ovat seuraavat:

1. Ohjelmanosa, jossa poikkeus tapahtuu. Ohjelman havaitsema poikkeavuus johtaa poikkeuksen *aiheuttamiseen* (*raise*). Kun poikkeus aiheutetaan, normaali ohjelmansuoritus keskeytetään, kunnes poikkeus on *käsitelty.* C++:ssa poikkeuksen aiheuttaminen toteutetaan (*heitetään*) `throw`-lausekkeella. Esimerkiksi seuraavassa koodikatkelmassa heitetään string-tyyppinen poikkeus epäonnistuneen tiedostonavauksen takia:

```
if ( ! infile ) {
    string errMsg( "tiedostoa ei voi avata: " );
    errMsg += fileName;
    throw errMsg;
}
```

2. Ohjelmanosa, jossa poikkeus käsitellään. Tyypillisesti ohjelman poikkeukset aiheutetaan ja käsitellään eri funktioissa tai jäsenfunktioiden käynnistyksissä. Käsittelyn löytämiseen liittyy “takaisinkelausta”, jota kutsutaan *ohjelman kutsupinoksi*. Kun poikkeus on käsitelty, ohjelman normaali suoritus jatkuu. Suoritus ei jatku sieltä, missä poikkeus tapahtui, vaan sieltä, missä se käsiteltiin. C++:ssa poikkeuksen käsittely toteutetaan (*siepataan*) `catch`-lauseella. Esimerkiksi seuraavassa `catch`-lause käsittelee poikkeuksen, joka heitettiin kohdassa 1:

```
catch( string exceptionMsg ) {
        log_message( exceptionMsg );
        return false;
}
```

try-lohkot liittyvät catch-lauseisiin. try-lohkoon ryhmitetään ohjelman yksi tai useampi ohjelmalause ja catch-lause. Esimerkiksi tässä on stats()-funktio:

```
int*
stats( const int *ia, int size )
{
        int *pstats = new int[ 4 ];
        try {
                pstats[ 0 ] = sum_it(  ia, size );
                pstats[ 1 ] = min_val( ia, size );
                pstats[ 2 ] = max_val( ia, size );
        }
        catch( string exceptionMsg )
            { /* koodi, joka hoitaa poikkeuksen */ }

        catch( const statsException &statsExcp )
            { /* koodi, joka hoitaa poikkeuksen */  }

        pstats[ 3 ] = pstats[ 0 ]/size;
        do_something( pstats );

        return pstats;
}
```

stats()-funktiossa on neljä lausetta, jotka ovat try-lohkon ulkopuolella. Näistä lauseista kahdessa voidaan poikkeus mahdollisesti aiheuttaa ennen niiden suorittamista loppuun:

```
(1) int *pstats = new int[ 4 ];
(2) do_something( pstats );
```

Lauseessa (1) new-lauseke saattaa epäonnistua. Jos se tapahtuu, vakiokirjasto aiheuttaa bad_alloc-vakiopoikkeuksen. Koska bad_alloc aiheutetaan try-lohkon ulkopuolella, ei sitä yritetä käsitellä stats()-funktiossa. Sen sijaan funktio päättyy: pstats jää alustamatta ja seuraavia stats()-funktion ohjelmalauseita ei koskaan suoriteta. Poikkeusmekanismi ottaa kontrollin hoitaakseen ja pitää sen, kunnes poikkeus on käsitelty.

Lauseessa (2) jokin lause do_something()-funktiossa, jokin funktio, joka käynnistetään do_something()-funktiossa tai tuossa käynnistetyssä funktiossa oleva lause jne. voi aiheuttaa poikkeuksen. Tuo poikkeus saatetaan siepata tai sitten ei tultaessa takaisinpäin funktiokutsujen ketjua, joka alkoi kutsulla do_something(). Jos poikkeus käsitellään, stats() jatkuu kuin mitään ei olisi tapahtunut. Ellei poikkeusta käsitellä ennen do_something()-funktiokutsun päättymistä, stats() puolestaan päättyy, koska poikkeus tapahtuu try-lohkon ulkopuolella.

(Huomaa, että jos size on yhtä kuin 0, lauseke

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

johtaa nollalla jakamiseen. Vaikka tämä johtaa tuntemattoman tiedon sijoitukseen muuttujaan pstats[3], ei nollalla jakamiseen aiheuteta mitään vakiopoikkeusta.)

Mitä sanottavaa on kolmesta lauseesta, jotka ovat try-lohkossa? Käyttäytymisen ero on

seuraava: jos aiheutettu poikkeus on aktiivisena stats()-funktiossa ja se on seurausta funktioiden sum_it(), min_val() tai max_val() kutsusta, niin sen sijaan, että yksinkertaisesti pysäytettäisiin stats(), vastaavat catch-lauseet tutkitaan try-lohkossa ja yritetään käsitellä aiheutettu poikkeus. Sanotaan, että sum_it() aiheutti seuraavan poikkeuksen:

```
throw string( "sisäinen virhe: dumppi27832" );
```

pstats[0]-muuttujaa ei koskaan alusteta eikä seuraavia kahta lausetta try-lohkossa suoriteta. Sen sijaan poikkeusmekanismi huomaa, että sum_it() käynnistettiin try-lohkossa, jolloin se tutkii kaksi siihen liittyvää catch-lausetta.

catch-lause valitaan sen tyypin perusteella, joka vastaa poikkeuksen tyyppiä ja catch-lauseen tyyppiä. Meidän tapauksessamme poikkeuksen tyyppi on merkkijono (string) ja se vastaa catch-lausetta.

```
catch( string exceptionMsg )
    { /* koodi, joka käsittelee poikkeuksen */ }
```

Kontrolli välittyy valitun catch-lauseen rungolle ja puolestaan sen sisällä olevat lauseet suoritetaan. Kun se on tehty ja ellei poikkeusta heitetä uudelleen catch-lauseessa, kontrolli välittyy takaisin ohjelmalle tässä vaiheessa. Jos esimerkiksi olisimme kirjoittaneet

```
catch( string exceptionMsg )
{
    cerr << "stats(): poikkeus tapahtunut: "
         << exceptionMsg << endl;
    pstats[0] = pstats[1] = pstats[2] = 0;
}
```

silloin catch-lauseen päättyessä ohjelman kontrolli välittyisi seuraavaan suoritettavaan lauseeseen catch-lauseessa olevan lausejoukon jälkeen. Meidän tapauksessamme lause

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

suoritetaan, minkä jälkeen suoritetaan do_something() ja pstats-muuttujan palautus. Funktio, joka käynnisti stats()-funktion, on tietämätön siitä, että poikkeusta koskaan edes käsiteltiin.

Seuraavassa on ehkä parempi poikkeuksen käsittely:

```
catch( string exceptionMsg )
{
        cerr  << "stats(): poikkeus tapahtui: "
              << exceptionMsg
              << " taulukkoa ei voi käsitellä "
              << endl;
```

```
        delete [] pstats;
        return 0;
}
```

Tässä tapauksessa catch-lause palaa käynnistyvään funktioon, jonka toivomme testaavan stats() -funktion paluuarvoa nolla-arvoon ennen sen indeksoinnin yritystä.

Ellei aktiivisessa try-lohkossa aiheutettua poikkeusta käsitellä siihen liittyvissä catch-lauseissa, funktio päättyy ja poikkeusmekanismi etsii käsittelijää funktiosta, joka käynnisti stats()-funktion.

Jos poikkeusmekanismi palaa takaisin funktioiden käynnistysketjua aina main()-funktioon saakka eikä käsittelijää löydy, käynnistetään vakiokirjaston terminate()-funktio. Oletusarvo on, että terminate() pysäyttää ohjelman.

Seuraavassa on erikoislaatuinen catch-lause, joka kykenee käsittelemään kaikentyyppisiä aiheutettuja poikkeuksia:

```
catch( ... )
{
    // käsittelee kaikki poikkeukset, vaikkakaan ei pysty
    // suoraan käsittelemään poikkeusoliota
}
```

Voimme ajatella sen olevat eräänlainen *ota kaikki kiinni* -lause!

Poikkeusten käsittely on kielitasoinen piirre yhdenmukaiselle ohjelman poikkeustilanteiden hoitamiselle. Tätä käsitellään yksityiskohtaisesti luvuissa 11 ja 19. Oheislukemistossamme, *Inside the C++ Object Model* ([LIPPMAN96a]), käsitellään sen toteutuskysymyksiä ja suorituskykyä kuten tehdään myös artikkelissa “Exception Handling: Behind the Scenes”, kirjoittanut Josée Lajoie julkaisussa [LIPPMAN96b]. Hyvä keskustelunavaus mahdollisista vaaroista poikkeusten käsittelyssä on “Exception Handling: A False Sense of Security”, jonka on kirjoittanut Tom Cargill ja joka on myös julkaisussa [LIPPMAN96b].

Harjoitus 2.18

Seuraavassa funktiossa ei ole tarkistuksia virheelliselle tiedolle tai operaation onnistumiselle. Yksilöi kaikki ne asiat, jotka voivat mennä pieleen tässä funktiossa (tässä harjoituksessa emme halua vielä huolestua mahdollisesti aiheutetuista poikkeuksista).

```
int *alloc_and_init( string file_name )
{
        ifstream infile( file_name );
        int elem_cnt;
        infile >> elem_cnt;
```

```
        int *pi = allocate_array( elem_cnt );

        int elem;
        int index = 0;
        while ( cin >> elem )
              pi[ index++ ] = elem;

        sort_array( pi, elem_cnt );
        register_data( pi );

        return pi;
}
```

Harjoitus 2.19

Seuraavat funktiot, jotka käynnistetään alloc_and_init()-funktiossa, aiheuttavat seuraavia poikkeustyyppejä epäonnistuessaan:

```
allocate_array() noMem
sort_array()     int
register_data()  string
```

Lisää yksi tai useampi try-lohko ja vastaavat catch-lauseet paikkaan, jossa poikkeukset on soveliainta käsitellä. Tulosta vain yksinkertaisesti virhetapahtuma catch-lauseessa.

Harjoitus 2.20

Käy läpi ne tilanteet, jotka yksilöitiin mahdollisina ohjelmavirheinä alloc_and_init()-funktiossa harjoituksessa 2.18. Yksilöi ne, jotka ovat niin vakavia, että niiden pitää heittää poikkeus. Uudista funktiota niin (joko harjoituksen 2.18 versiota tai harjoituksen 2.19 versiota, jos teit sen), että se heittää yksilöidyt poikkeukset (merkkijonoliteraalien heittäminen on tässä vaiheessa tarpeeksi hyvä ratkaisu).

2.7 Taulukko jollain muulla nimellä

Eräs vaikeus ohjelmakoodimme levittämisessä muuhun kuin omaan käyttöön on se, että emme tiedä itse asiassa, mitä nimiä globaalit nimemme voivat saada — jos mitään. Jos esimerkiksi joku Intelillä on kirjoittanut

```
class Array { ... };
```

silloin ei voisi käyttää samassa ohjelmassa tuota Array-luokkaa ja sitä, jonka me toteutimme. Nimien näkyvyys saa kaksi toteutusta toisensa poissulkeviksi.

Perinteinen tapa tämän ongelman ratkaisemiseksi ennen C++-standardia oli laittaa globaalisti näkyville nimille etuliite, joka oli leksikaalisesti yksilöllinen merkkijono. Voisimme esimerkiksi julkaista Array-luokan näin:

```
class Cplusplus_Primer_Third_Edition_Array { ... };
```

Vaikka tämä nimi on tietysti erittäin todennäköisesti yksilöllinen (emme voi taata sitä kuitenkaan), on siinä myös kirjoittamista. C++-standardin nimiavaruusmekanismi on kielitasoinen ratkaisu tähän ongelmaan.

Nimiavaruusmekanismilla voit kapseloida nimiä, jotka muutoin *pilaavat globaalin nimiavaruuden*. Yleensä käytämme nimiavaruuksia vain silloin, kun oletamme koodiamme käytettävän ulkopuolisissa ohjelmistoprojekteissa. Tässä on esimerkki siitä, kuinka voisimme kapseloida Array-luokkamme:

```
namespace Cplusplus_Primer_3E {
    template <class elemType>
      class Array { ... };
    // ...
}
```

Nimi namespace-avainsanan jälkeen ilmaisee nimiavaruutta, joka on erillään globaalista nimiavaruudesta, jonne voimme sijoittaa kokonaisuuksia, jotka haluamme esiteltävän funktion tai luokan ulkopuolella. Nimiavaruus ei muuta esittelyiden merkitystä sen sisällä, vaan muuttaa vain niiden näkyvyyttä. Ennen kuin jatkamme, laajentakaamme käytettävissämme olevien nimiavaruuksien joukkoa:

```
namespace IBM_Canada_Laboratory {
    template <class elemType>
      class Array { ... };
    class Matrix { ... };
    // ...
}

namespace Disney_Feature_Animation {
    class Point { ... };
    template <class elemType, int size>
      class Array { ... };

    // ...
}
```

Jos nimiavaruuden esittelyt eivät ole välittömästi näkyviä ohjelmalle, kuinka voimme käsitellä niitä? Me käytämme *määritetyn nimen merkitsemistapaa*, joka on muotoa

```
nimiavaruuden_tunniste::kokonaisuuden_nimi;
```

kuten seuraavassa:

```
Cplusplus_Primer_3E::Array<string> text;
IBM_Canada_Laboratory::Matrix mat;
Disney_Feature_Animation::Point origin( 5000, 5000 );
```

Vaikka Disney_Feature_Animation, IBM_Canada_Laboratory ja Cplusplus_Primer_3E yksilöivät jokaisen vastaavan nimiavaruuden, on niitä kömpelö käyttää ohjelmissamme. Nimiavaruustunnisteiden kuten P3E, DFA tai IBM_CL käyttö on kätevämpää, mutta ne ilmaisevat paljon vähemmän tietoa ja lisäävät nimitörmäyksen mahdollisuutta. Jotta saavutettaisiin sekä merkitsevät nimiavaruuden tunnukset että ohjelmoinnin mukavuus nimiavaruuksissa esiteltyihin kokonaisuuksiin, on tehty alias-piirre.

Nimiavaruuden alias mahdollistaa, että voidaan liittää vaihtoehtoinen, lyhyempi yleisnimi olemassaolevaan nimiavaruuteen. Esimerkiksi:

```
// anna yleisnimi aliakseksi
namespace LIB = IBM_Canada_Laboratory;

// anna yksikertaisesti lyhyempi nimi aliakseksi
namespace DFA = Disney_Feature_Animation;
```

Tätä voidaan käyttää sitten synonyyminä alkuperäiselle nimiavaruudelle. Esimerkiksi:

```
#include "IBM_Canada.h"
namespace LIB = IBM_Canada_Laboratory;

int main()
{
        LIB::Array<int> ia(1024);
        // ...
}
```

Alias-nimi voi toimia myös varsinaisesti käytetyn nimiavaruuden kapseloinnissa. Esimerkiksi, kun tässä skenaariossa vaihdamme aliakseen liitettyä nimiavaruutta, vaihdamme esittelyjoukkoa ilman, että meidän täytyy vaihtaa varsinaista koodia, jota aliaksen kautta käsitellään. Esimerkiksi:

```
namespace LIB = Cplusplus_Primer_3E;

int main()
{
        // tässä tapauksessa ei tätä esittelyä tarvitse vaihtaa
        LIB::Array<int> ia(1024);
        // ...
}
```

Jotta tämä tekniikka toimisi käytännössä, pitää kuitenkin kahdessa nimiavaruudessa olevilla esittelyillä olla täsmälleen samanlaiset rajapinnat. Esimerkiksi seuraava ei toimi, koska Disney Array -luokka haluaa sekä tyyppi- että kokoparametrin Array-luokalleen:

```
namespace LIB = Disney_Feature_Animation;

int main()
{
        // ei enää kelvollinen esittely
        LIB::Array<int> ia(1024);
```

```
        // ...
}
```

Yhä useammin ohjelmoijat haluaisivat määrittämättömän pääsyn nimiavaruudessa esiteltyihin nimiin. Vaikka on mahdollisuus käyttää lyhyempää alias-nimeä nimiavaruuden tunnukselle, voi jokaisen nimiavaruudessa esitellyn nimen käsittely määrittämällä osoittautua usein kömpelöksi. *Using-direktiivi* saa nimiavaruuden esittelyt näkyviksi niin, että niihin voidaan viitata ilman määritystä. Esimerkiksi:

```
#include "IBM_Canada_Laboratory.h"

// saa kaikki nimet näkyviksi
using namespace IBM_Canada_Laboratory;

int main()
{
        // ok: IBM_Canada_Laboratory::Matrix
        Matrix mat( 4,4 );

        // ok: IBM_Canada_Laboratory::Array
        Array<int> ia( 1024 );
        // ...
}
```

Sekä `using` että `namespace` ovat avainsanoja. Nimiavaruus, johon viitataan, pitää olla esiteltynä tai tapahtuu käännösvirhe.

Using-esittely on valikoivampi mekanismi nimen näkyvyydelle. Se mahdollistaa yksittäisen esittelyn näkyville saamisen nimiavaruudesta. Esimerkiksi:

```
#include "IBM_Canada_Laboratory.h"

// saa vain Matrixin näkyville
using IBM_Canada_Laboratory::Matrix;

int main()
{
        // ok: IBM_Canada_Laboratory::Matrix
        Matrix mat(4,4);

        // virhe: IBM_Canada_Laboratory::Array ei ole näkyvillä
        Array<int> ia( 1024 );
        // ...
}
```

Jotta estettäisiin se, että C++-standardikirjaston komponentit pilaisivat käyttäjän ohjelmien nimiavaruutta, pitää C++-standardikirjaston komponenttien olla esiteltynä nimiavaruudessa nimellä `std`. Kuten mainitsimme luvussa 1, että vaikka ottaisimme mukaan C++-kirjaston otsikkotiedoston ohjelman tekstitiedostoon, eivät otsikkotiedostossa olevat komponentit ole automaattisesti näkyviä tekstitiedostossa. Esimerkiksi seuraava C++-standardin koodiesimerkki

ei käänny kunnolla:

```
#include <string>
// virhe: string ei ole näkyvillä
string current_chapter = "Matka C++-kieleen";
```

Kaikki esittelyt <string>-otsikkotiedostossa ovat suljettuina std-nimiavaruuteen. Kuten mainitsimme luvussa 1, voimme käyttää using-direktiiviä esikääntäjän #include-direktiivin jälkeen, jotta saamme <string>-otsikkotiedoston std-nimiavaruudessa esitellyt komponentit näkyville tekstitiedostossa:

```
#include <string>
using namespace std;

// ok: string on näkyvillä
string current_chapter = "Matka C++-kieleen";
```

Using-direktiivi nähdään usein kehnona ratkaisuna std-nimiavaruudessa esiteltyjen nimien näkyville saamiseksi ohjelmissamme. Esimerkissä using-direktiivi saa std-nimiavaruuden kaikki <string>-otsikkotiedostossa esitellyt komponentit näkyville ohjelman tekstitiedostossa. Tämä tuo takaisin globaalin nimiavaruuden pilaantumisongelman, jota nimiavaruus yrittää ensi sijassa välttää, ja lisää mahdollisuutta, että C++-vakiokirjaston komponenttien nimet törmäävät ohjelmissamme esiteltyjen globaalien nimien kanssa.

Nyt, kun olemme nähneet hieman enemmän nimiavaruuden mekanismista, tiedämme, että on olemassa kaksi muuta mekanismia, joita voidaan käyttää using-direktiivin sijasta, kun haluamme viitata std-nimiavaruuteen piilotettuun string -nimeen. Voimme käyttää määritettyä nimeä kuten seuraavassa:

```
#include <string>
// ok: käytä määritettyä nimeä
std::string current_chapter = "Matka C++-kieleen";
```

Tai voimme käyttää using-esittelyä kuten seuraavassa:

```
#include <string>
using std::string;

// ok: using-esittely saa stringin näkyville
string current_chapter = "Matka C++-kieleen";
```

Suosittelemme std-nimiavaruudessa esitelyjen nimien käyttöä varten valikoivampaa using-esittelyn käyttöä using-direktiivien sijasta. Tämä on toinen syy siihen, että using-direktiivejä ei käytetä tämän kirjan koodiesimerkeissä. Ihannetapauksessa pitäisi koodiesimerkissä olla using-esittely jokaiselle käytetylle kirjastokomponentille. Koska haluamme rajoittaa esimerkkien kokoa ja koska monet tämän kirjan esimerkit käännetään toteutuksissa, jotka eivät tue nimiavaruuksia, using-esittelyitä ei käytetä. Kohdassa 8.6 käsitellään lisää sitä, kuinka using-esittelyjä käytetään C++-vakiokirjaston komponenteille.

Seuraavissa neljässä luvussa käymme läpi neljän lisäluokan suunnittelun. Luvussa 3 suun-

nitellaan ja toteutetaan String-luokka, luvussa 4 suunnitellaan kokonaislukujen Stack-luokka, luvussa 5 List-luokka, ja luvussa 6 suunnitellaan luvun 4 Stack-luokka uudelleen. Nimiavaruusmekanismi mahdollistaa jokaisen luokan sijoittamisen omaan otsikkotiedostoonsa, mutta kapseloi silti niiden nimet yksittäiseen Cplusplus_Primer_3E-nimiavaruuteen. Katsomme tätä ja muuta tekniikkaa nimiavaruuksista luvussa 8.

Harjoitus 2.21

Kun on olemassa seuraava nimiavaruuden määrittely:

```
namespace Exercise {
    template <class elemType>
      class Array { ... };

    template <class Etype>
      void print( Array< Etype > );

    class String { ... };
    template <class listType>
      class List { ... };
}
```

ja seuraava ohjelma:

```
int main() {
    const int size = 1024;
    Array< String > as( size );
    List< int > il( size );

    // ...

    Array< String > *pas = new Array<String>(as);
    List <int> *pil = new List<int>(il);

    print( *pas );
}
```

ohjelman nykyisen toteutuksen käännös epäonnistuu, koska tyyppien nimet on kapseloitu nimiavaruuteen. Muokkaa ohjelmaa, ja

a. Käytä määritetyn nimen merkintätapaa, jotta voit käsitellä Exercise-nimiavaruuden tyyppimäärittelyjä.
b. Käytä using-esittelyä, jotta voit käsitellä tyyppimäärittelyjä.
c. Käytä nimiavaruuden alias-mekanismia.
d. Käytä using-direktiiviä.

2.8 Standardinmukainen taulukko on vektori

Vaikka sisäinen taulukko tukee säiliön mekanismeja kuten olemme nähneet, se ei tue säiliö-abstraktion semantiikkaa. Ohjelmoidaksemme tuolla tasolla, pitää C++-standardin mukaan joko vaatia tai toteuttaa sellainen luokka itse. C++-standardissa taulukkoluokka on nyt osa vakiokirjastoa. Sitä ei kutsuta taulukoksi vaan vektoriksi.

Vektori on luonnollisesti luokkamalli. Täten kirjoitamme

```
vector<int>   ivec( 10 );
vector<string> svec( 10 );
```

kun haluamme määritellä vastaavasti kymmenen kokonaislukuolion vektorin ja kymmenen merkkijono-olion vektorin.

Array-luokkamme ja vektoriluokkamallien toteutuksessa on kaksi pääeroa. Ensimmäinen ero on, että vektoriluokkamalli tukee sekä olemassaolevan taulukon elementin sijoituksen että lisäelementin lisäyksen merkintätapaa — tämä tarkoittaa, että vektorin taulukko kasvaa dynaamisesti suorituksen aikana, jos ohjelmoija haluaa käyttää tuota piirrettä. Toinen ero on laajakantoisempi ja edustaa merkittävää suunnitteluparadigman muutosta. Sen sijaan, että vektoriluokassa olisi laaja joukko jäsenoperaatioita, joita vektoriin voitaisiin käyttää, kuten sort(), min(), max(), find() jne., on niitä vain minimaalinen joukko: operaatiot kuten yhtäsuuruus- ja pienempi kuin -operaattorit, size() ja empty(). Yleiset operaatiot kuten sort(), min(), max(), find() jne. on sen sijaan toteutettu itsenäisinä *geneerisinä algoritmeina*.

Kun vektori määritellään, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include < vector >
```

Kaikki seuraavat ovat sallittuja vektoriolioiden määrittelyjä:

```
#include < vector >

// tapoja luoda vektoriolio
vector<int> vec0;    // tyhjä vektori

const int size  = 8;
const int value = 1024;

// vektorin koko on 8
// ja jokainen elementti alustetaan arvolla 0
vector<int> vec1( size );

// vektorin koko on 8
// ja jokainen elementti alustetaan arvolla 1024
vector<int> vec2( size, value );

// vec3:n koko on 4
// alustetaan ia:n neljällä arvolla
int ia[4] = { 0, 1, 1, 2 };
vector<int> vec3( ia, ia+4 );
```

```
// vec4 on kopio vektorista vec2
vector<int> vec4( vec2 );
```

Nyt, kun olemme määritelleet vektorimme, sen elementit pitää käydä läpi. Kuten Array-luokkamallimme, niin myös standardin mukainen vektoriluokkamalli tukee indeksioperaattorin käyttöä. Esimerkiksi:

```
#include <vector>
extern int getSize();

void mumble()
{
        int size = getSize();
        vector< int > vec( size );

        for ( int ix = 0; ix < size; ++ix )
                vec[ ix ] = ix;

        // ...
}
```

Vaihtoehtoinen läpikäyntitapa on käyttää *iteraattoriparia* (*iterator pair*), jolla merkitään vektorin alku ja loppu. Iteraattori on luokkaolio, joka tukee osoitintyypin abstraktiota. Vektoriluokkamallissa on `begin()`- ja `end()`-operaatiopari, joka palauttaa iteraattorin vastaavasti vektorin alkuun ja yhden yli vektorin lopun. Yhdessä iteraattoripari merkitsee elementtialuetta, jota voidaan käydä läpi. Esimerkiksi tässä on yhtäpitävä toteutus edellisestä koodikatkelmasta:

```
#include < vector >
extern int getSize();

void mumble()
{
        int size = getSize();
        vector< int > vec( size );

        vector< int >::iterator iter = vec.begin();

        for ( int ix = 0; iter != vec.end(); ++iter, ++ix )
                *iter = ix;

        // ...
}
```

Tässä `iter`-määrittely

```
vector< int >::iterator iter = vec.begin();
```

alustaa itsensä vec-vektorin ensimmäisen elementin osoitteella. iterator on typedef-nimi ja määritelty vektoriluokkamallin sisään, joka sisältää int-tyyppisiä elementtejä. Seuraavassa siirretään iteraattori vektorin seuraavaan elementtiin:

```
++iter
```

Seuraavassa käytetään käänteisviittausta, jotta päästäisiin käsittelemään todellista elementtiä:

```
*iter
```

Operaatiota, joita voimme tehdä vektorilla, on yllättävän monenlaisia. Niitä ei kuitenkaan ole tehty vektoriluokkamallin jäsenfunktioiksi, vaan itsenäisiksi geneerisiksi algoritmeiksi vakiokirjastoon. Seuraavassa on kokoelma käytettävissä olevista geneerisistä algoritmeista:

- *Etsintä*algoritmeja: find(), find_if(), search(), binary_search(), count() ja count_if().
- *Lajittelu-* ja yleisiä *järjestely*algoritmeja: sort(), partial_sort(), merge(), partition(), rotate(), reverse() ja random_shuffle().
- *Poisto*algoritmeja: unique() ja remove().
- *Numeerisia* algoritmeja: accumulate(), partial_sum(), inner_product() ja adj- acent_difference().
- *Generointi-* ja *muunnos*algoritmeja: generate(), fill(), transform(), copy() ja for_each().
- *Vertailu*algoritmeja: equal(), min() ja max().

Geneeriset algoritmit saavat iteraattoriparin, joka ilmaisee läpikäytävän elementtialueen. Jos esimerkiksi lajittelemme sort()-algoritmilla ivec-vektorin, jonka elementit ovat tietyn tyyppisiä ja kokoisia, tarvitsee meidän kirjoittaa yksinkertaisesti seuraavasti:

```
sort( ivec.begin(), ivec.end() );
```

Kun haluamme lajitella vain ensimmäisen puoliskon, kirjoitamme

```
sort( ivec.begin(), ivec.begin()+ivec.size()/2 );
```

Geneeriset algoritmit hyväksyvät myös osoitinparin sisäiseen taulukkoon. Jos esimerkiksi on annettu taulukko

```
int ia[7] = { 10, 7, 9, 5, 3, 7, 1 };
```

voimme lajitella koko taulukon seuraavasti:

```
sort( ia, ia+7 );
```

Lajittelemme vain neljä ensimmäistä elementtiä seuraavasti:

```
sort( ia, ia+4 );
```

Jotta algoritmeja voitaisiin käyttää, pitää ottaa mukaan niihin liittyvä otsikkotiedosto:

```
#include <algorithm>
```

Seuraavassa on esimerkkejä, kuinka voisimme käyttää monia eri algoritmeja vektoriluokan olioon:

```
#include <vector>
#include <algorithm>
#include <iostream>
```

```
int ia[ 10 ] = {
        51, 23, 7, 88, 41, 98, 12, 103, 37, 6 };

int main()
{
        vector< int > vec( ia, ia+10 );

        // lajittele taulukko
        sort( vec.begin(), vec.end() );

        // pyydä etsittävä arvo
        int search_value;
        cin >> search_value;

        // etsi elementti
        vector<int>::iterator found;
        found = find( vec.begin(), vec.end(), search_value );
        if ( found != vec.end() )
      cout << "arvo search_value löytyi!\n";
    else cout << "arvoa search_value ei löytynyt!\n";

        // laita taulukko päinvastaiseen järjestykseen
        reverse( vec.begin(), vec.end() );

        // ...
}
```

Vakiokirjastossa on myös tuki assosiatiiviselle taulukolle (*map*) — tarkoittaa taulukon elementtejä, joita voidaan indeksoida jollain muulla kuin kokonaislukuarvoilla. Esimerkiksi puhelinluetteloa voidaan tukea puhelinnumerotaulukolla, jota voidaan indeksoida henkilön nimellä, jolle puhelinnumero kuuluu:

```
#include <map>
#include <string>
#include "TelephoneNumber.h"

map< string, telephoneNum > telephone_directory;
```

Katsomme luvussa 6 vektoreita, assosiatiivisia taulukoita ja muita C++-vakiokirjaston tukemia säiliötyyppejä, kun käymme läpi tekstinkyselyjärjestelmän toteutuksen, joka tuo esille näiden tyyppien käyttöä. Luvussa 12 katsomme geneerisiä algoritmeja ja tämän kirjan liitteessä on aakkosjärjestyksessä kuvaus, kuinka algoritmeja tulisi käyttää.

Tässä luvussa on käyty läpi, myönnettäköön, reippaaseen tahtiin C++:n pääasiallinen tuki tietoabstraktiolle (oliopohjainen ohjelmointi), oliokeskeiselle ohjelmoinnille ja geneeriselle ohjelmoinnille (mallit, säiliötyypit ja geneeriset algoritmit) sekä ohjelmointi suurelta osin (poikkeusten käsittely ja nimiavaruudet). Tästä eteenpäin käsittelemme C++:n perus- ja lisäpiirteitä yksityiskohtaisemmin ja hitaampaan tahtiin.

Harjoitus 2.22

Selitä jokaisen vektorin määrittelyn tulokset:

```
string pals[] = {
    "pooh", "tigger", "piglet", "eeyore", "kanga" };

(a) vector<string> svec1( pals, pals+5 );
(b) vector<int>    ivec1( 10 );
(c) vector<int>    ivec2( 10, 10 );
(d) vector<string> svec2( svec1 );
(e) vector<double> dvec;
```

Harjoitus 2.23

Kun on annettu seuraava funktioesittely, toteuta min()-funktion runko, joka löytää ja palauttaa vec-vektorin pienimmän elementin käyttäen for-silmukkaa vec-vektorin indeksointiin ja sitten for-silmukkaa, joka käyttää iteraattoria vec-vektorin läpikäyntiin:

```
template <class elemType>
elemType
  min( const vector<elemType> &vec );
```