11

Poikkeusten käsittely

Poikkeusten käsittely on mekanismi, joka mahdollistaa kahden erillään kehitetyn ohjelmankomponentin kommunikoida keskenään, kun ohjelman suorituksen aikana kohdataan epänormaali tilanne, jota kutsutaan *poikkeukseksi* (*exception*). Tässä luvussa katsomme ensin, kuinka poikkeus aiheutetaan tai heitetään paikassa, jossa ohjelman epänormaali tilanne havaitaan. Sitten katsomme, kuinka näihin liitetään käsittelijät eli catch-lauseet sekä try-lohkon ohjelmalauseet ja kuinka catch-lauseet käsittelevät poikkeukset. Sen jälkeen esittelemme poikkeusmääritykset: mekanismi, johon liittyy joukko poikkeuksia ja joka takaa, että funktio ei heitä minkään muun tyyppisiä poikkeuksia. Luku päättyy suunnittelun näkökohtien käsittelyyn ohjelmissa, jotka käyttävät poikkeuksia.

11.1 Poikkeuksen heittäminen

Poikkeukset ovat suorituksenaikaisia epänormaaleja tilanteita, joita ohjelmassa saattaa tapahtua, kuten nollalla jakaminen, taulukon käsittely yli sen rajojen tai keskusmuistin hupeneminen. Sellaiset poikkeukset ovat ohjelman normaalin toiminnan ulkopuolella ja vaativat ohjelmalta välitöntä käsittelyä. C++-kielessä on sisäinen kielen piirre, jolla voidaan aiheuttaa ja käsitellä poikkeuksia. Tämä kielen piirre aktivoi suorituksenaikaisen mekanismin, jota käytetään kahden toisiinsa liittymättömän (usein kehitetty erikseen) C++-ohjelmanosan poikkeusten väliseen kommunikointiin.

Kun C++-ohjelmassa havaitaan poikkeus, ohjelmanosa, joka sen havaitsee, voi ilmoittaa, että poikkeus on tapahtunut aiheuttamalla eli *heittämällä* poikkeus. Jotta näkisimme, kuinka poikkeuksia heitetään C++:ssa, toteuttakaamme kohdassa 4.15 esitelty iStack-luokka uudelleen niin, että se käyttää poikkeuksia epänormaalien tilanteiden ilmaisuun pinon käsittelyssä. iStack-luokan määrittely näyttää tältä:

```
#include <vector>
```

```
class iStack {
public:
    iStack( int capacity )
        : _stack( capacity ), _top( 0 ) { }

    bool pop( int &top_value );
    bool push( int value );

    bool full();
    bool empty();
    void display();

    int size();

private:
    int _top;
    vector< int > _stack;
};
```

Pino on toteutettu käyttämällä int-vektoria. Kun iStack-olio luodaan, iStack-muodostaja luo int-vektorin alkuarvon mukaisella koolla. Tämä koko on enintään elementtien lukumäärä, jonka iStack-olio voi sisältää. Esimerkiksi seuraavassa luodaan iStack-olio nimeltään myStack, joka voi sisältää 20 int-tyyppistä arvoa:

```
iStack myStack(20);
```

Mikä voi mennä pieleen, kun käsittelemme myStack-oliota? Tässä on kaksi epänormaalia tilannetta, jotka voidaan havaita iStack-luokassamme:

1. Pyydetään pop()-operaatiota ja pino on tyhjä.
2. Pyydetään push()-operaatiota ja pino on täysi.

Päätämme, että nämä erikoistilanteet tulisi välittää iStack-olioita käsitteleville funktioille poikkeuksia käyttämällä. Joten mistä aloitamme?

Ensiksi pitää määritellä poikkeukset, joita voidaan heittää. C++:ssa poikkeukset toteutetaan useimmin luokkia käyttämällä. Vaikka luokat esitellään täydellisesti vasta luvussa 13, määrittelemme tässä kaksi yksinkertaista luokkaa, joita voidaan käyttää poikkeuksina iStack-luokassamme ja sijoitamme näiden luokkien määrittelyt stackExcp.h-otsikkotiedostoon:

```
// stackExcp.h
class popOnEmpty { /* ... */ };
class pushOnFull { /* ... */ };
```

Luvussa 19 käsitellään luokkatyyppisiä poikkeuksia yksityiskohtaisemmin ja C++-vakiokirjaston poikkeusluokkahierarkiaa.

Sitten pitää muuttaa jäsenfunktioiden pop() ja push() määrittelyjä, jotta ne heittäisivät nämä juuri määritellyt poikkeukset. Poikkeus heitetään käyttämällä *throw-lauseketta*. Throw-lauseke näyttää paljon return-lauseelta. Throw-lauseke muodostuu throw-avainsanasta, jonka jälkeen tu-

lee lauseke, jonka tyyppi on heitettävän poikkeuksen tyyppi. Miltä throw-lauseke näyttää pop()-funktiossa? Kokeillaanpa tätä:

```
// hups, ei ihan oikein
throw popOnEmpty;
```

Valitettavasti tämä ei ole aivan oikein. Poikkeus on olio ja pop():in pitää heittää luokkatyyppinen olio. Throw-lausekkeen lauseke ei voi olla vain yksinkertaisesti tyyppi. Jotta voisimme luoda luokkatyyppisen olion, pitää kutsua luokan muodostajaa. Miltä muodostajan käynnistävä throw-lauseke näyttää? Tässä on throw-lauseke pop()-funktiossa:

```
// lauseke on muodostajakutsu
throw popOnEmpty();
```

Tämä throw-lauseke luo popOnEmpty-tyyppisen poikkeusolion.

Muista, että jäsenfunktiot pop() ja push() määriteltiin palauttamaan bool-tyyppinen arvo: paluuarvo true ilmaisee, että operaatio onnistui ja false ilmaisee, että se epäonnistui. Koska nyt käytetään poikkeuksia ilmaisemaan pop()- ja push()-operaatioiden onnistumista, ovat näiden funktioiden paluuarvot nyt tarpeettomia. Määrittelemme nyt nämä jäsenfunktiot void-paluutyypillä. Esimerkiksi:

```
class iStack {
public:
    // ...

    // ei enää paluuarvoa
    void pop( int &value );
    void push( int value );

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

Funktiot, jotka käyttävät iStack-luokkaamme, olettavat nyt, että kaikki on hyvin, ellei poikkeusta heitetä; niiden ei enää tarvitse testata jäsenfunktioiden pop() ja push() paluuarvoa nähdäkseen, onnistuiko operaatio. Tulemme näkemään seuraavissa kahdessa eri kohdassa, kuinka määritellään funktio, joka käsittelee poikkeukset.

Olemme nyt valmiita tekemään uudet toteutukset iStack-luokan pop()- ja push()-jäsenfunktioista:

```
#include "stackExcp.h"

void iStack::pop( int &top_value )
{
    if ( empty() )
        throw popOnEmpty();

    top_value = _stack[ --_top ];

    cout << "iStack::pop(): " << top_value << endl;
```

```
}

void iStack::push( int value )
{
    cout << "iStack::push( " << value << " )\n";

    if ( full() )
        throw pushOnFull();

    _stack[ _top++ ] = value;
}
```

Vaikka poikkeukset ovat useimmiten luokkatyyppisiä olioita, voi throw-lauseke heittää minkä tahansa tyyppisen olion. Vaikka seuraava onkin epätavallista, niin koodikatkelman mathFunc()-funktio heittää poikkeusolion, joka on luetellun joukon tyyppi. Tämä on kelvollista C++-koodia:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

int mathFunc( int i ) {
    if ( i == 0 )
        throw zeroOp; // luetellun joukon tyyppinen poikkeus

    // muussa tapauksessa jatkuu normaali käsittely
}
```

Harjoitus 11.1

Mitkä seuraavista throw-lausekkeista ovat virheellisiä, vai onko yksikään? Miksi? Ilmaise kelvollisten throw-lausekkeiden heittämä poikkeuksen tyyppi.

```
(a) class exceptionType { };
   throw exceptionType();
(b) int excpObj;
   throw excpObj;
(c) enum mathErr { overflow, underflow, zeroDivide };
   throw zeroDivide();
(d) int *pi = excpObj;
   throw pi;
```

Harjoitus 11.2

Kohdassa 2.3 määritellyssä IntArray-luokassa on jäsenoperaattorifunktio operator[](), joka käyttää assert()-funktiota ilmaistakseen, että indeksi on taulukon rajojen ulkopuolella. Muuta operator[]()-operaattorifunktion määrittelyä niin, että se heittää sen sijaan poikkeuksen tuossa tilanteessa. Määrittele poikkeusluokka, jota käytetään heitettävän poikkeuksen tyyppinä.

11.2 Try-lohko

Seuraava pieni ohjelma kokeilee iStack-luokkaamme sekä pop()- ja push()-jäsenfunktioita, jotka määriteltiin edellisessä kohdassa. main()-funktion for-silmukka tekee toiston 50 kertaa. Se työntää pinoon jokaisen arvon, joka on luvun 3 kerrannainen — 3, 6, 9 jne. Aina, kun arvo on luvun 4 kerrannainen kuten 4, 8, 12 jne., se näyttää pinon sisällön. Aina, kun arvo on luvun 10 kerrannainen kuten 10, 20, 30 jne., se vetää viimeisen arvon pinosta ja näyttää siten jälleen pinon sisällön. Kuinka muutamme main()-funktiota niin, että se käsittelee iStack-jäsenfunktioiden heittämät poikkeukset?

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

int main() {
    iStack stack( 32 );

    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        if ( ix % 3 == 0 )
            stack.push( ix );

        if ( ix % 4 == 0 )
            stack.display();

        if ( ix % 10  == 0) {
            int dummy;
            stack.pop( dummy );
            stack.display();
        }
    }
    return 0;
}
```

Try-lohko pitää sisältää lauseet, jotka voivat heittää poikkeuksia. Try-lohko alkaa try-avainsanalla, jonka jälkeen tulevat ohjelman lauseet aaltosulkujen sisään. Try-lohkon jälkeen on joukko käsittelijöitä, joita nimitetään *catch-lauseiksi*. Try-lohko ryhmittää lausejoukon ja liittää näihin lauseisiin käsittelijäjoukon, joka voi käsitellä poikkeukset, joita lauseet voivat heittää. Mihin sijoittaisimme try-lohkon tai -lohkot main()-funktiossa, jotta ne käsittelisivät poikkeukset popOnEmpty ja pushOnFull? Kokeillaanpa tätä:

```
for ( int ix = 1; ix < 51; ++ix ) {
    try { // try-lohko pushOnFull-poikkeuksille
        if ( ix % 3 == 0 )
            stack.push( ix );
    }
    catch  ( pushOnFull ) { ... }
```

```
    if ( ix % 4 == 0 )
        stack.display();

    try { // try-lohko popOnEmpty-poikkeuksille
        if ( ix % 10  == 0 ) {
            int dummy;
            stack.pop( dummy );
            stack.display();
        }
    }
    catch ( popOnEmpty ) { ... }
}
```

Näin toteuttamamme ohjelma toimii oikein. Sen normaalin käsittelyjärjestyksen kuitenkin sekoittaa poikkeusten käsittely, eikä se ole kovin hyvä ajatus. Loppujen lopuksi poikkeukset ovat harvinaisia tilanteita, jotka tapahtuvat vain poikkeustapauksissa. Haluamme erottaa koodin, joka käsittelee ohjelman epätavalliset tilanteet siitä koodista, joka toteuttaa pinon normaalia käsittelyä. Uskomme, että tämä strategia saa koodin helpommin luettavaksi ja ylläpidettäväksi. Tässä on parempana pitämämme ratkaisu:

```
try {
    for ( int ix = 1; ix < 51; ++ix )
    {
        if ( ix % 3 == 0 )
            stack.push( ix );

        if ( ix % 4 == 0 )
            stack.display();

        if ( ix % 10  == 0 ) {
            int dummy;
            stack.pop( dummy );
            stack.display();
        }
    }
}
catch ( pushOnFull ) { ... }
catch ( popOnEmpty ) { ... }
```

Try-lohkoon liittyy kaksi catch-lausetta, jotka kykenevät käsittelemään poikkeukset pushOnFull ja popOnEmpty. Ne voidaan heittää iStack-jäsenfunktioista `push()` ja `pop()`, kun niitä kutsutaan try-lohkossa. Kumpikin catch-lause määrittää sulkujen sisällä, minkä tyyppisen poikkeuksen se käsittelee. Koodi, joka käsittelee poikkeuksen, sijoitetaan catch-lauseen yhdistettyyn lauseeseen (aaltosulkujen sisään). Tutkimme catch-lauseita tarkemmin seuraavassa kohdassa.

Esimerkkimme ohjelman kulku on jokin seuraavista:

1. Ellei poikkeusta tapahdu, suoritetaan try-lohkon sisällä oleva koodi eikä try-lohkoon liittyviä käsittelijöitä huomioida. main()-funktio palauttaa arvon 0.
2. Jos push()-jäsenfunktio heittää poikkeuksen, jota kutsutaan for-silmukan ensimmäisestä if-lauseesta, jätetään for-silmukan toinen ja kolmas if-lause huomiotta, poistutaan for-silmukasta sekä try-lohkosta ja suoritetaan pushOnFull-poikkeustyypin käsittely.
3. Jos pop()-jäsenfunktio heittää poikkeuksen, jota kutsutaan for-silmukan kolmannesta if-lauseesta, jätetään display()-funktion kutsu huomiotta, poistutaan for-silmukasta sekä try-lohkosta ja suoritetaan popOnEmpty-poikkeustyypin käsittely.

Kun poikkeus on heitetty, heittävän lauseen jälkeiset lauseet jätetään huomiotta. Ohjelman suoritus jatkuu poikkeuksen käsittelevästä catch-lauseesta. Ellei yksikään catch-lause pysty käsittelemään poikkeusta, ohjelma jatkuu terminate()-funktiosta, joka on määritelty C++-vakiokirjastoon. Käsittelemme lisää terminate()-funktiota seuraavassa kohdassa.

Try-lohko voi sisältää mitä tahansa C++-lauseita — yhtä hyvin lausekkeita kuin lauseitakin. Try-lohko esittelee paikallisen käyttöalueen eikä sen ulkopuolelta voida viitata sen sisällä esiteltyihin muuttujiin, joihin kuuluvat myös catch-lauseet. Voisimme esimerkiksi kirjoittaa main()-funktiomme uudelleen niin, että stack-muuttujan esittely esiintyy try-lohkon sisällä. Tässä tapauksessa ei ole mahdollista viitata stack-muuttujaan catch-lauseissa:

```
int main() {
    try {
        iStack stack( 32 ); // ok: esittely try-lohkossa

        stack.display();
        for ( int ix = 1; ix < 51; ++ix )
        {
            // sama kuin aiemmin
        }
    }
    catch ( pushOnFull ) {
        // ei voida viitata stack-muuttujaan täällä
    }
    catch ( popOnEmpty ) {
        // ei voida viitata stack-muuttujaan täällä
    }

    // ei voida viitata stack-muuttujaan täällä
    return 0;
}
```

On mahdollista esitellä funktio niin, että funktion koko runko sisältyy try-lohkoon. Sellaisessa tapauksessa sen sijaan, että sijoittaisimme try-lohkon funktion määrittelyyn, voimme sijoittaa funktion rungon *try-lohkoon*. Tämä järjestely tukee selkeimmin niiden koodien erottelua, joissa tuetaan ohjelman normaalia käsittelyä ja niitä, jotka tukevat poikkeusten käsittelyä. Esimerkiksi:

```
int main()
try {
    iStack stack( 32 );

    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        // sama kuin aiemmin
    }

    return 0;
}
catch ( pushOnFull ) {
    // ei voi viitata stack-muuttujaan täällä
}
catch ( popOnEmpty ) {
    // ei voi viitata stack-muuttujaan täällä
}
```

Huomaa, että try-avainsana tulee ennen funktiorungon avaavaa aaltosulkua ja catch-lauseet on lueteltu funktiorungon sulkevan aaltosulun jälkeen. Tällä järjestelyllä koodi, joka tukee normaalia main()-funktion käsittelyä, on sijoitettu funktiorungon sisään selkeästi erilleen koodista, joka käsittelee poikkeukset catch-lauseissa. Kuitenkaan muuttujiin, jotka on esitelty main()-funktion rungon sisällä, ei voida viitata catch-lauseissa.

Funktion try-lohko liittää yhteen joukon catch-lauseita ja funktiorungon. Jos funktiorungon sisällä lause heittää poikkeuksen, otetaan ne poikkeusten käsittelijät huomioon, jotka tulevat funktiorungon jälkeen. Funktion try-lohkot ovat erityisen hyödyllisiä luokkien muodostajien yhteydessä. Tutkimme uudelleen try-lohkoja tässä yhteydessä luvussa 19.

Harjoitus 11.3

Kirjoita ohjelma, joka määrittelee IntArray-olion (jossa IntArray on luokkatyyppi, joka määriteltiin kohdassa 2.3) ja suorittaa seuraavat toimenpiteet. On olemassa kolme tiedostoa, jotka sisältävät kokonaislukuarvoja.

1. Lue ensimmäinen tiedosto ja sijoita ensimmäinen, kolmas, viides, ... , n:s luettu arvo (jossa n on pariton luku) IntArray-olioon; näytä sitten IntArray-olion sisältö.
2. Lue toinen tiedosto ja sijoita viides, kymmenes, ... , n:s luettu arvo (jossa n on 5:n kerrannainen) IntArray-olioon; näytä sitten IntArray-olion sisältö.

3. Lue kolmas tiedosto ja sijoita toinen, neljäs, kuudes, ... , n:s luettu arvo (jossa n on parillinen luku) IntArray-olioon; näytä sitten IntArray-olion sisältö.

Käytä harjoituksessa 11.2 määriteltyä IntArray-luokan operator[]()-operaattoria arvojen lukemiseen ja tallentamiseen IntArray-olioon. Koska operator[]() voi heittää poikkeuksen, käytä ohjelmassasi yhtä tai useampaa try-lohkoa ja catch-lausetta mahdollisten heitettyjen poikkeusten käsittelemiseksi. Perustele syyt, miksi sijoitit ohjelmasi try-lohkot valitsemaasi paikkaan.

11.3 Poikkeuksen sieppaaminen

C++:n poikkeuksen käsittelijä on *catch-lause*. Kun try-lohkon sisällä olevat lauseet heittävät poikkeuksen, etsitään try-lohkon jälkeisistä catch-lauseista sitä, joka voi käsitellä poikkeuksen.

Catch-lause muodostuu kolmesta osasta: catch-avainsanasta, yksittäisen tyypin tai olion esittelystä sulkujen sisällä (sanotaan *poikkeuksen esittelyksi)* ja lausejoukosta yhdistetyn lauseen sisällä. Jos catch-lause valitaan poikkeuksen käsittelyyn, suoritetaan yhdistetty lause. Tutkikaamme tarkemmin poikkeusten pushOnFull ja popOnEmpty catch-lauseita main()-funktiossa.

```
catch ( pushOnFull ) {
    cerr << "trying to push a value on a full stack\n";
    return errorCode88;
}
catch ( popOnEmpty ) {
    cerr << "trying to pop a value on an empty stack\n";
    return errorCode89;
}
```

Molemmilla catch-lauseilla on luokkatyyppinen poikkeuksen esittely; ensimmäinen on tyyppiä pushOnFull ja toinen popOnEmpty. Käsittelijä valitaan poikkeukselle, jos sen poikkeuksen esittely vastaa heitetyn poikkeuksen tyyppiä. (Tulemme näkemään luvussa 19, ettei tyyppien tarvitse olla täysin toisiaan vastaavia: kantaluokan käsittelijä voi käsitellä poikkeuksia, joiden luokkatyyppi on johdettu käsittelijän poikkeusesittelyn tyypistä.) Kun esimerkiksi iStack-luokan pop()-jäsenfunktio heittää popOnEmpty-poikkeuksen, siirrytään toiseen catch-lauseeseen. Sen jälkeen, kun virheilmoitus on annettu vakiovirheeseen cerr, main()-funktio palauttaa virhekoodin errorCode89.

Elleivät nämä catch-lauseet sisältäisi return-lausetta, mistä ohjelman suoritus jatkuisi? Sen jälkeen, kun catch-lause on saanut työnsä valmiiksi, ohjelman suoritus jatkuu lauseesta, joka on tuon catch-lauseen jälkeen. Esimerkissämme ohjelman suoritus jatkuu main()-funktion return-lauseesta ja sen jälkeen, kun popOnEmpty:n catch-lause generoi virheilmoituksen vakiovirheeseen cerr, main() palauttaa arvon 0.

```
int main() {
    iStack stack( 32 );
```

```
    try {
        stack.display();
        for ( int ix = 1; ix < 51; ++ix )
        {
            // sama kuin aiemmin
        }
    }
    catch ( pushOnFull ) {
        cerr << "trying to push a value on a full stack\n";
    }
    catch ( popOnEmpty ) {
        cerr << "trying to pop a value on an empty stack\n";
    }

    // ohjelman suoritus jatkuu täältä
    return 0;
}
```

C++:n poikkeusten käsittelyn mekanismin sanotaan olevan *takaisin palaamaton*; kun poikkeus on käsitelty, ohjelman suoritus ei jatku sieltä, missä poikkeus alun perin heitettiin. Kun esimerkissämme poikkeus on käsitelty, ohjelman suoritus ei jatku pop()-jäsenfunktiosta, josta poikkeus heitettiin.

11.3.1 Poikkeusoliot

Catch-lauseen poikkeuksen esittely voi olla joko tyyppiesittely tai olioesittely. Milloin catch-lauseen poikkeuksen esittelyn tulisi esitellä olio? Olio tulisi esitellä silloin, kun on tarpeellista saada arvo tai käsitellä throw-lausekkeen luomaa poikkeusoliota. Jos suunnittelemme poikkeusluokkamme niin, että ne voivat tallentaa tietoa poikkeusolioon, kun poikkeus heitetään, ja jos catch-lauseen poikkeuksen esittely esittelee olion, voidaan catch-lauseessa olevilla lauseilla käsitellä tuota oliota ja viitata throw-lausekkeen tallentamaan tietoon.

Muuttakaamme esimerkiksi pushOnFull-poikkeusluokan suunnitelmaa. Tallentakaamme poikkeusolioon arvo, jota ei voida työntää pinoon. Catch-lausetta muutetaan niin, että se näyttää tämän arvon, kun virheilmoitus generoidaan vakiovirheeseen cerr. Jotta voimme tehdä tämän, pitää ensiksi muuttaa pushOnFull-luokkatyypin määrittelyä. Tässä on uusi määrittely:

```
// uusi poikkeusluokka:
// tallentaa arvon, jota ei voida työntää pinoon
class pushOnFull {
public:
    pushOnFull( int i ) : _value( i ) { }
    int value() { return _value; }
private:
    int _value;
};
```

Uusi yksityinen tietojäsen, _value, sisältää arvon, jota ei voida työntää pinoon. Muodostaja saa int-tyyppisen arvon ja tallentaa tämän arvon _value-tietojäseneen. Seuraavasta nähdään, kuinka throw-lauseke voi käynnistää muodostajan ja tallentaa poikkeusolioon arvon, jota ei voida työntää pinoon:

```
void iStack::push( int value )
{
    if ( full() )
        // arvo tallennetaan poikkeusolioon
        throw pushOnFull( value );

    // ...
}
```

Luokalla pushOnFull on myös uusi jäsenfunktio, value(), jota voidaan käyttää catch-lauseessa poikkeusolioon tallennetun arvon näyttämiseen. Seuraavasta nähdään, kuinka sitä voidaan käyttää:

```
catch ( pushOnFull eObj ) {
    cerr << "trying to push the value " << eObj.value()
       << " on a full stack\n";
}
```

Huomaa, että catch-lauseen poikkeuksen esittelyssä on eObj-olio, jota käytetään luokan pushOnFull value()-jäsenfunktion käynnistämiseen.

Poikkeusolio luodaan aina throw-hetkellä, vaikka throw-lauseke ei olekaan muodostajan kutsu ja vaikka se ei näytä luovan poikkeusoliota. Esimerkiksi:

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;

int mathFunc( int i ) {
    if ( i == 0 ) {
        state = zeroOp;
        throw state; // poikkeusolio luotu
    }
    // muussa tapauksessa jatkuu normaali suoritus
}
```

Tässä esimerkissä ei state-oliota käytetä poikkeusoliona. Sen sijaan throw-lauseke luo EHstate-tyyppisen poikkeusolion ja alustaa sen globaalin state-olion arvolla. Kuinka ohjelma voi kertoa, että poikkeusolio on eri kuin globaali state-olio? Jotta tähän kysymykseen voitaisiin vastata, pitää ensiksi katsoa catch-lauseen poikkeuksen esittelyä tarkemmin.

Catch-lauseeen poikkeuksen esittely toimii melko samalla tavalla kuin parametrin esittely. Kun catch-lauseeseen saavutaan ja jos poikkeuksen esittelyssä on olio, alustetaan tämä olio poikkeusolion kopiolla. Esimerkiksi seuraava calculate()-funktio kutsuu aikaisemmin määriteltyä mathFunc()-funktiota. Kun calculate()-funktiossa saavutaan catch-lauseeseen, alustetaan eObj-olio poikkeusolion kopiolla, jonka throw-lauseke on luonut:

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate eObj ) {
        // eObj on kopio heitetystä poikkeusoliosta
    }
}
```

Tässä esimerkissä poikkeuksen esittely muistuttaa arvona välitettyä parametria. eObj-olio alustetaan poikkeusolion arvolla samalla tavalla kuin arvona välitetty funktion parametri alustetaan vastaavan argumentin arvolla (arvona välitettävät parametrit on käsitelty kohdassa 7.3).

Aivan kuten funktion parametrien yhteydessä, voidaan myös catch-lauseen poikkeuksen esittely muuttaa viittauksen esittelyksi. Silloin catch-lause viittaa suoraan poikkeusolioon, jonka throw-lauseke on luonut sen sijaan, että loisi paikallisen kopion siitä. Esimerkiksi:

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // eObj viittaa heitettyyn poikkeusolioon
    }
}
```

Samoista syistä, joista luokkatyyppiset parametrit tulisi esitellä viittauksina, jotta vältyttäisiin suurten luokkaolioiden kopioimiselta, pidetään myös parempana, että luokkatyyppiset poikkeusten esittelyt ovat viittauksia.

Kun poikkeuksen esittely on viittaustyyppinen, catch-lause pystyy muokkaamaan poikkeusoliota. Kuitenkin kaikki throw-lausekkeen määrittämät muuttujat säilyvät muuttumattomina. Kun esimerkiksi muokkaamme eObj-oliota catch-lauseessa, ei se vaikuta globaaliin state-muuttujaan, jonka throw-lauseke on määrittänyt:

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // korjaa poikkeustilanne
        eObj = noErr; // globaali state-muuttuja ei muutu
    }
}
```

Catch-lause asettaa eObj-olioon uuden noErr-arvon sen jälkeen, kun poikkeustilanne on korjattu. Koska eObj on viittaus, voimme olettaa, että tämä sijoitus muokkaa globaalia state-oliota. Sijoitus muokkaa kuitenkin vain poikkeusoliota, jonka throw-lauseke on luonut. Ja koska poikkeusolio on eri olio kuin globaali state-olio, säilyy state muuttumattomana, kun eObj-oliota muokataan catch-lauseessa.

11.3.2 Pinon aukikelaus

Catch-lauseen etsintä, joka käsittelee heitetyn poikkeuksen, etenee seuraavasti: jos throw-lauseke sijaitsee try-lohkon sisällä, tutkitaan, voiko joku tähän try-lohkoon liittyvistä catch-lauseista käsitellä poikkeuksen. Jos catch-lause löytyy, poikkeus käsitellään. Ellei catch-lausetta löydy, etsintä jatkuu kutsuvasta funktiosta. Jos funktion kutsu, joka päättyy heitettyyn poikkeukseen, sijaitsee try-lohkon sisällä, tutkitaan, voiko joku tuohon try-lohkoon liittyvistä catch-lauseista käsitellä poikkeuksen. Jos catch-lause löytyy, poikkeus käsitellään. Ellei catch-lausetta löydy, etsintä jatkuu kutsuvasta funktiosta. Tämä prosessi jatkuu sisäkkäisten funktioiden ketjua ylöspäin, kunnes poikkeukselle löytyy catch-lause. Niin pian kuin poikkeuksen käsittelevä catch-lause löytyy, siirrytään tuohon catch-lauseeseen ja ohjelman suoritus jatkuu tässä käsittelijässä.

Esimerkissämme ensimmäinen funktio, jota etsitään catch-lauseeksi, on iStack-luokan pop()-jäsenfunktio. Koska pop():in throw-lauseke ei sijaitse try-lohkon sisällä, pop() päättyy poikkeukseen. Seuraava tutkittava funktio on se, joka kutsuu pop()-jäsenfunktiota, joka esimerkissämme on main(). Jäsenfunktion pop() kutsu main()-funktiossa sijaitsee try-lohkon sisällä. Tähän try-lohkoon liittyvät catch-lauseet otetaan huomioon poikkeuksen käsittelemistä varten. Catch-lause löytyy popOnEmpty-tyyppisille poikkeuksille ja siihen siirrytään käsittelemään poikkeusta.

Prosessia, jossa yhdistetyt lauseet ja funkion määrittelyt päättyvät heitetyn poikkeuksen takia etsittäessä poikkeuksen käsittelevää catch-lausetta, sanotaan *pinon aukikelaamiseksi*. Kun pino on kelattu auki, esiteltyjen paikallisten olioiden elinaika päättyy yhdistetyissä lauseissa ja funktioiden määrittelyissä, joista on poistuttu. C++ takaa, että kun pino on kelattu auki, paikallisten luokkaolioiden tuhoajia kutsutaan, vaikka niiden elinaika päättyy heitetyn poikkeuksen takia. Katsomme tätä tarkemmin luvussa 19.

Mitä, jos ohjelmassa ei ole catch-lausetta heitetylle poikkeukselle? Poikkeus ei voi jäädä käsittelemättä. Poikkeus on liian vakava tilanne, jotta ohjelma voisi jatkaa normaalisti. Ellei käsittelijää löydy, ohjelma kutsuu terminate()-funktiota, joka on määritelty C++-vakiokirjastoon. terminate()-funktio käyttäytyy oletusarvoisesti niin, että se kutsuu abort()-funktiota ilmaistakseen ohjelman epänormaalin päättymisen. (Useimmissa tapauksissa abort()-funktion kutsuminen riittää. Kuitenkin joissakin tapauksissa on tarpeellista ohittaa terminate()-funktion toimenpiteet. Näet julkaisusta [STROUSTRUP97], kuinka se tehdään ja jossa asiaa käsitellään syvällisemmin.)

Nyt olet ehkä jo huomannut monia samankaltaisuuksia poikkeusten käsittelyn ja funktioiden kutsujen välillä. Throw-lauseke käyttäytyy jokseenkin samalla tavalla kuin funktion kutsu ja catch-lause kutakuinkin samalla tavalla kuin funktion määrittely. Pääero näiden kahden mekanismin välillä on, että kaikki tarvittava tieto funktion kutsun järjestämiseksi on saatavilla käännöksen aikana, kun taas poikkeusten käsittelymekanismiin se ei päde. C++:n poikkeusten käsittely vaatii suorituksenaikaista tukea. Kun on esimerkiksi kyse tavallisesta funktiosta, kääntäjä tietää kutsuhetkellä, mitä funktiota todellisuudessa kutsutaan, koska se on päätelty

funktion ylikuormituksen ratkaisun perusteella. Poikkeusten käsittelyn kohdalla kääntäjä ei tiedä tietystä throw-lausekkeesta, missä funktiossa catch-lause sijaitsee ja mistä suoritus jatkuu sen jälkeen, kun poikkeus on käsitelty. Nämä päättelyt tapahtuvat suorituksen aikana. Kääntäjä ei voi kertoa käyttäjille, milloin käsittelijää ei löydy poikkeukselle. Tästä syystä terminate()-funktio on olemassa: se on suorituksenaikainen mekanismi kertoa käyttäjille, että heitetylle poikkeukselle ei löydy sopivaa käsittelijää.

11.3.3 Uudelleenheitto

On mahdollista, että yksi catch-lause ei voi käsitellä poikkeusta loppuun. Joidenkin korjaavien toimenpiteiden jälkeen catch-lause voi päättää, että poikkeus pitää käsitellä ylempänä kutsuttujen funktioiden listassa. Catch-lause voi välittää poikkeuksen toiselle catch-lauseelle, joka sijaitsee ylemmällä funktiokutsujen listalla *heittämällä* poikkeuksen *uudelleen.* Uudelleenheitto-lausekkeen muoto on seuraava:

```
throw;
```

Uudelleenheitto-lauseke heittää poikkeusolion uudelleen. Uudelleenheitto voi esiintyä vain catch-lauseen yhdistetyssä lauseessa. Esimerkiksi:

```
catch ( exception eObj ) {
    if ( canHandle( eObj ) )
        // käsittele poikkeus
        return;
    else
        // heitä se uudelleen toiselle catch-lauseelle käsiteltäväksi
        throw;
}
```

Poikkeus, joka heitetään uudelleen, on *alkuperäinen* poikkeusolio. Tällä on joitakin seuraamuksia, jos catch-lause muokkaa poikkeusoliota ennen sen uudelleenheittoa. Seuraavassa ei muokata alkuperäistä poikkeusoliota. Huomaatko, miksi?

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

void calculate( int op ) {
    try {
        // poikkeuksella, jonka mathFunc() on heittänyt, on arvo zeroOp
        mathFunc( op );
    }
    catch ( EHstate eObj ) {
        // korjaa muutama asia

        // yritys muokata poikkeusoliota
        eObj = severeErr;

        // aikoo heittää poikkeuksen uudelleen arvolla severeErr
        throw;
    }
```

```
}
```

Koska eObj ei ole viittaus, catch-lause saa kopion poikkeusoliosta, jolloin kaikki käsittelijän eObj-olion muokkaukset kohdistuvat sen paikalliseen kopioon. Ne eivät vaikuta alkuperäiseen throw-lausekkeen heittämään poikkeusolioon. Tämä alkuperäinen poikkeusolio on se, jonka uudelleenheittolauseke heittää. Koska esimerkkimme catch-lause ei muokkaa alkuperäistä poikkeusta, on uudelleen heitetyn olion arvo yhä alkuperäinen zeroOp.

Jotta alkuperäistä poikkeusoliota voitaisiin muokata, pitää catch-lauseen poikkeuksen esittelyn olla viittaus. Esimerkiksi:

```
catch ( EHstate &eObj ) {
    // muokkaa poikkeusoliota
    eObj = severeErr;

    // uudelleen heitetyn poikkeuksen arvon on severeErr
    throw;
}
```

eObj viittaa throw-lausekkeen luomaan poikkeusolioon ja eObj:hin tehdyt muokkaukset catch-lauseessa vaikuttavat alkuperäiseen poikkeusolioon. Nämä muokkaukset ovat osa poikkeusoliota, joka heitetään uudelleen.

Tästä johtuen on toinenkin hyvä syy esitellä catch-lauseen poikkeusolio viittauksena: se varmistaa, että catch-lauseessa tehdyt muokkaukset poikkeusolioon vaikuttavat myös uudelleen heitettävään poikkeusolioon. Kohdassa 19.2 tulemme näkemään vielä yhden hyvän syyn siihen, miksi luokkatyyppisten poikkeusten esittelyiden tulisi olla viittauksia. Näemme siellä, kuinka catch-lauseet voivat käynnistää luokan virtuaalifunktioita.

11.3.4 Sieppaa kaikki -käsittelijä

Funktio voi halutessaan tehdä joitakin toimenpiteitä ennen kuin se päättyy poikkeuksen heittämiseen, vaikka se ei pystyisikään käsittelemään heitettyä poikkeusta. Funktio voi esimerkiksi vaatia joitakin resursseja kuten tiedoston avaamista tai muistin varaamista keosta ja se voi haluta vapauttaa nämä resurssit (sulkea tiedoston tai vapauttaa muistin) ennen kuin se päättyy heitettyyn poikkeukseen. Esimerkiksi:

```
void manip() {
    resource res;
    res.lock();    // lukitsee resurssin

    // käytä res-resurssia
    // joitakin toimenpiteitä, jotka aiheuttavat heitettävän poikkeuksen

    res.release(); // hypätään yli, jos poikkeus on heitetty
}
```

Resurssin res vapautus ohitetaan, jos poikkeus heitetään. Jotta voisimme varmistua tuon resurssin vapauttamisesta ja koska emme tiedä kaikkia mahdollisia heitettäviä poikkeuksia, voimme käyttää *sieppaa kaikki* -catch-lausetta sen sijaan, että laittaisimme tietyt catch-lauseet jokaista poikkeusta varten. Tämän catch-lauseen poikkeuksen esittely on muotoa (...), jossa kolmea pistettä sanotaan *ellipsiksi*. Tähän catch-lauseeseen tullaan kaikilla poikkeustyypeillä. Esimerkiksi:

```
// tullaan millä tahansa heitetyllä poikkeuksella
catch ( ... ) {
    // sijoita koodimme tähän
}
```

catch(...)-lausetta käytetään yhdessä uudelleenheittolausekkeen kanssa. Resurssi, joka on lukittu, vapautetaan catch-lauseen yhdistetyssä lauseessa ennen kuin suorituksen annetaan jatkua ylempänä funktiokutsujen ketjussa uudelleenheittolausekkeen jälkeen:

```
void manip() {
    resource res;
    res.lock();
    try {
        // käyt res-resurssia
        // joitakin toimenpiteitä, jotka aiheuttavat heitettävän poikkeuksen
    }
    catch (...) {
        res.release();
        throw;
    }
    res.release(); // hypätään yli, jos poikkeus heitetään
}
```

Jotta voisimme varmistua, että resurssi on kunnolla vapautettu poikkeuksen heiton yhteydessä manip():in päättyessä poikkeukseen, catch(...)-lausetta käytetään resurssin vapauttamiseen ennen kuin poikkeus välitetään ylemmäksi funktiokutsujen listalla. Voimme myös järjestää resurssin hankkimisen ja vapauttamisen kapseloimalla sen luokkaan ja antaa muodostajan hankkia resurssi ja tuhoajan vapauttaa se automaattisesti. Katsomme luvussa 19, kuinka se tehdään.

catch(...)-lausetta voidaan käyttää itsenäisesti tai yhdessä muiden catch-lauseiden kanssa. Jos sitä käytetään yhdessä muiden catch-lauseiden kanssa, pitää olla hieman huolellinen, kun catch-lauseita järjestetään niihin liittyvään try-lohkoon.

Catch-lauseet tutkitaan vuorollaan siinä järjestyksessä, missä ne esiintyvät try-lohkon jälkeen. Kun kelvollinen löytyy, ei jäljessä olevia lauseita tutkita. Tämä tarkoittaa, että jos catch(...)-lausetta käytetään yhdessä muiden catch-lauseiden kanssa, se pitää sijoittaa poikkeustenkäsittelijäjoukon viimeiseksi; muussa tapauksessa saadaan aikaan käännöksenaikainen virhe. Esimerkiksi:

```
try {
    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        // sama kuin aikaisemmin
    }
}
catch ( pushOnFull ) { }
catch ( popOnEmpty ) { }
catch (...) { } // pitää olla viimeinen catch-lause
```

Harjoitus 11.4

Selitä, miksi sanomme, että C++:n poikkeusten käsittely on takaisin palaamaton.

Harjoitus 11.5

Olkoot seuraavat poikkeusten esittelyt. Tee throw-lauseke, joka luo poikkeusolion, joka voidaan siepata seuraavilla catch-lauseilla:

```
(a) class exceptionType { };
   catch( exceptionType *pet ) { }
(b) catch(...) { }
(c) enum mathErr { overflow, underflow, zeroDivide };
   catch( mathErr &ref ) { }
(d) typedef int EXCPTYPE;
   catch( EXCPTYPE ) { }
```

Harjoitus 11.6

Selitä, mitä tapahtuu pinon aukikelaamisen aikana.

Harjoitus 11.7

Anna kaksi syytä, miksi catch-lauseen poikkeuksen esittelyn tulisi olla viittaus.

Harjoitus 11.8

Käytä koodia, jonka kehitit harjoituksessa 11.3 ja muokkaa luomaasi poikkeusluokkaa niin, että kun operator[]()-operaattorin yhteydessä käytetään kelvotonta indeksiä, se tallennetaan poikkeusolioon poikkeuksen heiton yhteydessä ja näytetään myöhemmin catch-lauseessa. Muokkaa ohjelmaasi niin, että operator[]() heittää poikkeuksen ohjelman suorituksen aikana.

11.4 Poikkeusmääritykset

Kun katsoo iStack-luokan pop()- ja push()-jäsenfunktioiden esittelyitä, ei ole mahdollista päätellä, että nämä funktiot saattaisivat heittää poikkeuksia. Eräs mahdollinen ratkaisu on lisätä kyseinen kommentti jokaisen funktion esittelyyn. Tällä tavalla otsikkotiedostossa näkyvä luokan rajapinta myös dokumentoi poikkeukset, joita jäsenfunktiot saattavat heittää:

```
class iStack {
public:
    // ...

    void pop( int &value ); // heittää popOnEmpty
    void push( int value ); // heittää pushOnFull

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

Tämä ei kuitenkaan ole kovin hyvä ratkaisu. Ei ole takuita siitä, että tämä dokumentointi säilyy ajan tasalla iStack-luokkamme myöhemmissä versioissa. Eikä se myöskään anna tietoja kääntäjälle takuiksi, ettei muunlaisia poikkeuksia heitettäisi. *Poikkeusmääritys* on ratkaisu, jota voidaan käyttää poikkeuksien luettelemiseksi funktion esittelyssä, joita funktio voi heittää. Se takaa, että funktio ei heitä minkään muun tyyppisiä poikkeuksia.

Poikkeusmääritys tulee funktion parametriluettelon jälkeen. Se alkaa avainsanalla throw, jonka jälkeen tulevat poikkeustyypit sulkujen sisällä. Esimerkiksi iStack-luokan jäsenfunktiot voidaan muokata seuraavasti, kun niihin halutaan lisätä vastaavat poikkeusmääritykset:

```
class iStack {
public:
    // ...

    void pop( int &value ) throw(popOnEmpty);
    void push( int value ) throw(pushOnFull);

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

pop():in kutsu takaa, että se ei heitä minkään muun tyyppistä poikkeusta kuin popOnEmpty. Samalla tavalla push()-kutsu takaa, että se ei heitä minkään muun tyyppistä poikkeusta kuin pushOnFull.

Poikkeuksen esittely on osa funktion rajapintaa ja se pitää olla funktioiden esittelyiden joukossa otsikkotiedostoissa. Poikkeusmääritys on sopimus funktion ja muun ohjelman välillä. Se on takuu, että funktio ei heitä sellaista poikkeusta, jota ei ole lueteltu poikkeusmäärityksessä.

Jos funktion esittelyssä on poikkeusmääritys, pitää saman funktion uudelleenesittelyssä määrittää myös poikkeusmääritys samoilla tyypeillä. Saman funktion eri esittelyiden poikkeusmääritykset eivät ole kumulatiivisia. Esimerkiksi:

```
// kaksi saman funktion esittelyä
extern int foo( int = 0 ) throw(string);

// virhe: poikkeusmääritys jätetty pois
extern int foo( int parm ) { }
```

Mitä tapahtuu, jos funktio heittää poikkeuksen, jota ei ole lueteltu poikkeusmäärityksessä? Poikkeuksia heitetään vain, jos ohjelmassa kohdataan epätavallisuuksia eikä käännöksen aikana ole mahdollista tietää, kohtaako ohjelma näitä poikkeustilanteita suorituksen aikana. Tästä syystä funktion poikkeusmäärityksen rikkomukset voidaan havaita vain suorituksen aikana. Jos funktio heittää poikkeuksen, jota ei ole lueteltu poikkeusmäärityksessä, käynnistetään unexpected()-funktio, joka on määritelty C++-vakiokirjastoon. Funktio unexpected() käyttäytyy oletusarvoisesti niin, että se kutsuu terminate()-funktiota. (Joissain tilanteissa voi olla tarpeen ohittaa unexpected()-funktion suorittamat toimenpiteet. C++-vakiokirjastossa on mekanismi, jolla voidaan korvata unexpected()-funktion oletuskäyttäytyminen. Julkaisussa [STROUSTRUP97] käsitellään tätä aihetta tarkemmin.)

Meidän tulisi olla tietoisia, että unexpected()-funktiota ei kutsuta ainoastaan silloin, kun funktio heittää poikkeuksen, jota ei ole lueteltu poikkeusmäärityksessä. Jos funktio käsittelee poikkeuksen itse ennen kuin “pakenee” funktion ulkopuolelle, silloin kaikki on ok. Esimerkiksi:

```
void recoup( int op1, int op2 ) throw(ExceptionType)
{
    try {
        // ...
        throw string("we’re in control");
    }
    // käsittelee heitetyn poikkeuksen
    catch ( string ) {
        // tee, mitä tarvitaan
    }
} // OK, unexpected()-funktiota ei kutsuta
```

Vaikka recoup()-funktiosta heitetään string-tyyppinen poikkeus ja vaikka recoup()-funktio takaa, että ei heitä muun tyyppisiä poikkeuksia kuin ExceptionType, koska poikkeus käsitellään ennen kuin recoup()-funktiosta poistutaan, ei unexpected()-funktiota kutsuta tämän poikkeuksen heittämisen tuloksena.

Funktion poikkeusmääritysten rikkomukset havaitaan vain suorituksen aikana. Kääntäjä ei generoi käännöksenaikaisia virheitä, jos lauseke voi heittää poikkeustyypin, jota poikkeusmäärityksessä ei sallita. Ellei tätä lauseketta koskaan suoriteta tai ekkeu se koskaan heitä poikkeusta, joka rikkoo poikkeusmääritystä, ohjelman kulku on odotettua, eikä poikkeusmääritystä koskaan rikota. Esimerkiksi:

```
extern void doit( int, int ) throw(string, exceptionType);

void action (int op1, int op2 ) throw(string) {
    doit( op1, op2 ); // ei ole käännöksenaikainen virhe
    // ...
}
```

Funktio doit() voi heittää poikkeustyypin exceptionType, jota action()-funktion poikkeusmäärityys ei salli. Vaikka action()-funktio ei salli tämän tyyppistä poikkeusta, funktio kääntyy onnistuneesti. Sen sijaan kääntäjä generoi koodin varmistaakseen, että jos heitetään poikkeus, joka rikkoo poikkeusmääritystä, kutsutaan suorituksenaikaisesta kirjastosta (run-time) unexpected()-funktiota.

Tyhjä poikkeusmääritys takaa, että funktio ei heitä yhtään poikkeusta. Esimerkiksi no_problem()-funktio takaa, että ei heitä poikkeusta:

```
extern void no_problem() throw();
```

Ellei funktion esittelyssä määritetä poikkeuksia, voi funktio heittää minkä tahansa tyyppisen poikkeuksen.

Tyyppikonversioita ei sallita heitetyn poikkeuksen tyypin ja poikkeusmäärityksen tyypin välillä. Esimerkiksi:

```
int convert( int parm ) throw(string)
{
    // ...
    if ( somethingRather )
        // ohjelmavirhe:
        // convert() ei salli poikkeustyyppiä const char*
        throw "help!";
}
```

convert()-funktion throw-lauseke heittää C-tyylisen merkkijonon. Poikkeusolio, joka tästä luodaan, on tyyppiä const char*. Usein const char* -tyyppinen lauseke voidaan konvertoida string-tyyppiseksi. Poikkeusmääritys ei kuitenkaan salli tyyppikonversioita heitetyn poikkeuksen tyypistä poikkeusmäärityksen tyypiksi. Jos convert() heittää tämän poikkeuksen, silloin kut-

sutaan unexpected()-funktiota. Jotta tämä tilanne voitaisiin korjata, voidaan throw-lauseketta muokata niin, että se konvertoi eksplisiittisesti lausekkeen arvon string-tyyppiseksi, kuten tässä:

```
throw string( "help!" );
```

11.4.1 Poikkeusmääritykset ja osoittimet funktioihin

Poikkeusmääritys voidaan laittaa myös osoitin funktioon -esittelyyn. Esimerkiksi:

```
void (*pf)( int ) throw(string);
```

Tämä esittely ilmaisee, että pf osoittaa funktioon, joka voi heittää string-tyyppisiä poikkeuksia. Aivan kuten funktion esittelyissä, eivät eri osoitin funktioon -esittelyiden poikkeusmääritykset ole kumulatiivisia, jolloin kaikkiin pf-osoittimen esittelyihin pitää laittaa samat poikkeusmääritykset. Esimerkiksi:

```
extern void (*pf)( int ) throw(string);

// virhe: poikkeusmääritys puuttuu
void (*pf)( int );
```

Kun osoitin sellaiseen funktioon alustetaan (tai siihen sijoitetaan), jossa on poikkeusmääritys, on olemassa rajoituksia osoittimen tyypille, jota alustamiseen käytetään (tai käytetään *rvaluena* sijoituksen oikealla puolella). Molempien osoittimien poikkeusmäärityksen ei tarvitse olla samanlaisia. Kuitenkin sen osoittimen poikkeusmäärityksen, jota käytetään alustamiseen tai rvaluena, pitää olla joko yhtä rajoittava tai rajoittavampi kuin sen osoittimen poikkeusmääritys, jota alustetaan tai johon sijoitetaan. Esimerkiksi:

```
void recoup( int, int ) throw(exceptionType);
void no_problem() throw();
void doit( int, int ) throw(string, exceptionType);

// ok: recoup() on yhtä rajoittava kuin pf1
void (*pf1)( int, int ) throw(exceptionType) = &recoup;

// ok: no_problem() on rajoittavampi kuin pf2
void (*pf2)() throw(string) = &no_problem;

// virhe: doit() ei ole niin rajoittava kuin pf3
void (*pf3)( int, int ) throw(string) = &doit;
```

Kolmannessa alustuksessa ei ole järkeä. Osoittimen esittely takaa, että pf3 osoittaa funktioon, joka ei heitä muun tyyppisiä poikkeuksia kuin string. Kuitenkin doit()-funktio saattaa heittää exceptionType-tyyppisen poikkeuksen. Koska doit()-funktio ei tyydytä pf3:n poikkeusmääritystä, se ei ole kelvollinen pf3:n alustamiseen ja saa aikaan käännösvirheen.

Harjoitus 11.9

Käytä koodia, jonka kehitit harjoituksessa 11.8 ja muuta IntArray-luokan operator[]()-operaattorin esittelyä niin, että lisäät siihen sopivan poikkeusmäärityksen kuvaamaan poikkeusta, jonka tämä operaattori voi heittää. Muokkaa ohjelmaasi niin, että operator[]() heittää poikkeuksen, jota ei ole lueteltu sen poikkeusmäärityksessä. Mitä silloin tapahtuu?

Harjoitus 11.10

Mitä poikkeuksia funktio heittää, jos sen poikkeusmääritys on muotoa throw()? Entä jos sillä ei ole poikkeusmääritystä?

Harjoitus 11.11

Mitkä seuraavista ovat virheellisiä osoittimen sijoituksia, vai onko yksikään? Miksi?

```
void example() throw(string);
(a) void (*pf1)() = example;

(b) void (*pf2)() throw() = example;
```

11.5 Poikkeukset ja suunnittelunäkökohdat

Poikkeusten käsittelyyn C++-ohjelmissa liittyy joitakin suunnittelunäkökohtia. Vaikka poikkeusten käsittelyn tuki on rakennettuna kieleen, ei jokaisen C++-ohjelman tulisi käyttää poikkeusten käsittelyä. Poikkeusten käsittelyä tulisi käyttää ohjelman epätavallisten tilanteiden välittämiseen ohjelmanosien välillä, jotka on kehitetty toisistaan riippumatta, koska poikkeuksen heittäminen ei ole yhtä nopea kuin normaalin funktion kutsu. Esimerkiksi kirjaston toteuttaja voi päättää välittää ohjelman poikkeustilanteet kirjaston käyttäjille poikkeusten avulla. Jos kirjaston funktio havaitsee epätavallisen tilanteen, jota se ei voi käsitellä paikallisesti, voi se heittää poikkeuksen tiedoksi kirjastoa käyttävälle ohjelmalle.

Esimerkkimme kirjastossa on määritelty iStack-luokka ja sen jäsenfunktiot. main()-funktio käyttää kirjastoa ja meidän tulisi olettaa, että main()-funktion kirjoittaja ei ole kirjaston toteuttaja. iStack-luokan jäsenfunktiot kykenevät havaitsemaan, että pop()-operaatiota on pyydetty tyhjältä pinolta tai että push()-operaatiota on pyydetty täydelle pinolle, mutta kirjaston toteuttaja ei tiedä ohjelman tilaa, joka sai aikaan pyydetyt pop()- tai push()-operaatiot alun perin, eikä voi ottaa yhteyttä pop()- ja push()-funktioihin tässä tilanteessa. Koska näitä virheitä ei voida käsitellä jäsenfunktioissa, päätimme heittää poikkeukset tiedoksi kirjastoa käyttävälle ohjelmalle.

Vaikka C++ tukee poikkeusten käsittelyä, tulisi C++-ohjelmien käyttää muita virheenkäsittelytekniikoita (kuten virhekoodin palauttaminen) aina, kun se on mahdollista. Ei ole olemassa selvää vastausta kysymykseen “Milloin virheestä pitäisi tulla poikkeus?”. On todella kirjaston toteuttajan päätettävissä, millainen “poikkeuksellinen tilanne” on. Poikkeukset ovat osa kirjaston rajapintaa ja päätös siitä, mitkä poikkeukset kirjasto heittää, on tärkeä vaihe kirjaston suun-

nittelussa. Jos kirjasto on tarkoitettu käytettäväksi ohjelmien kanssa, joilla ei ole varaa romahtaa (kaatua), silloin sen pitää käsitellä ongelma itse. Jollei se pysty, sen pitää välittää ohjelman poikkeustilanteet tiedoksi sellaiseen ohjelmanosaan, joka käyttää kirjastoa, ja antaa kutsujalle mahdollisuus valita, mikä toimenpide tulisi tehdä, kun kirjaston omasta koodista ei sellaista löydy. Päätös siitä, mitä tulisi käsitellä poikkeuksena, on vaikea osa kirjaston suunnittelua.

iStack-esimerkissämme voidaan keskustella siitä, pitäisikö push()-jäsenfunktion heittää poikkeus, kun pino on täynnä. Jotkut voisivat sanoa, että parempi push()-toteutus olisi käsitellä tilanne paikallisesti ja kasvattaa pinoa, kun se on täynnä. Loppujen lopuksi ainoa todellinen rajoitus on ohjelmallemme käytettävissä oleva muistin määrä. Päätöksemme heittää poikkeus, kun ohjelma yrittää työntää arvon täyteen pinoon, saattaa olla harkitsematon. Voimme toteuttaa uudelleen push()-jäsenfunktion niin, että se kasvattaa pinoa, jos sitä pyydetään työntämään arvo täyteen pinoon:

```
void iStack::push( int value )
{
    // jos täynnä, kasvata taustalla olevaa vektoria
    if ( full() )
        _stack.resize( 2 * _stack.size() );

    _stack[ _top++ ] = value;
}
```

Samalla tavalla voitaisiin kysyä, tulisiko pop()-jäsenfunktion heittää poikkeus, kun sitä pyydetään vetämään arvo tyhjästä pinosta? Eräs mielenkiintoinen piirre on, että C++-vakiokirjaston pinoluokka (esitelty luvussa 6) ei heitä poikkeusta, jos veto-operaatiota pyydetään pinon ollessa tyhjä. Sen sijaan operaation käyttäytyminen on tuntematon: ei tiedetä, kuinka ohjelma käyttäytyy sellaisen operaatiokutsun jälkeen. Kun C++-vakiokirjasto suunniteltiin, päätettiin, että poikkeusta ei tulisi heittää tässä tapauksessa. Ohjelman suorituksen salliminen, vaikkakin kelpaamattomassa tilassa, katsottiin sopivammaksi tässä tilanteessa. Kuten mainitsimme, eri kirjastoilla on erilaiset poikkeukset. Ei ole olemassa oikeaa vastausta kysymykseen, mistä poikkeus muodostuu.

Ei kaikkien ohjelmien tulisi olla huolissaan kirjastojen heittämistä poikkeuksista. Vaikka on totta, että jotkut järjestelmät eivät siedä *alhaallaoloa* (suoritus keskeytynyt), ja ne tulisi rakentaa niin, että ne käsittelevät poikkeukselliset tilanteet, ei kaikilla ohjelmilla ole sellaisia vaatimuksia. Poikkeuksen käsittely on pääasiassa apukeino vikasietoisen järjestelmän toteuttamiseen. Jälleen päätös siitä, käsittelevätkö ohjelmamme kirjastojen heittämät poikkeukset vai tulisiko meidän antaa ohjelmiemme päättyä, on vaikea osa suunnitteluprosessia.

Vielä eräs näkökohta ohjelmansuunnitteluun kannalta on, että poikkeusten käsittely on usein kerrostettu ohjelmassa. Ohjelma koostuu usein komponenteista ja jokaisen komponentin pitää päättää, mitkä poikkeukset se käsittelee paikallisesti ja mitkä se välittää ohjelman ylemmille tasoille. Mitä me tarkoitamme komponentilla? Esimerkiksi luvussa 6 esitelty tekstinkyselyjärjestelmä voidaan jakaa kolmeen komponenttiin eli kerrokseen. Ensimmäinen kerros on

C++-vakiokirjasto, josta saadaan tuki merkkijonojen, karttojen ym. perusoperaatioille. Toinen kerros on itse tekstinkyselyjärjestelmä, joka määrittelee funktiot kuten string_caps() ja suffix_text(), joilla käsitellään prosessoitavaa tekstiä ja joka käyttää C++-vakiokirjastoa alikomponenttina. Kolmas kerros on ohjelma, joka käyttää tekstinkyselyjärjestelmäämme. Jokainen komponentti eli kerros on rakennettu itsenäisesti ja jokaisen niistä pitää päättää, mitkä poikkeukselliset tilanteet ne käsittelevät ja mitkä ne välittävät ohjelman ylemmille tasoille.

Ei jokaisen komponentin funktion tai kerroksen pitäisi pystyä käsittelemään poikkeuksia. Usein try-lohkoja ja niihin liittyviä catch-lauseita käyttävät funktiot, jotka ovat sisääntulopisteitä ohjelmakomponenttiin. Catch-lauseet on suunniteltu käsittelemään poikkeuksia, joita komponentti ei halua levittää ohjelman ylemmille tasoille. Poikkeusmäärityksiä (käsitelty kohdassa 11.4) käytetään myös funktioissa, jotka ovat komponenttien sisääntulopisteissä vartioimassa ei-toivottujen poikkeusten livahdusta ohjelman korkeammille tasoille.

Katsomme luvussa 19 muita suunnittelunäkökohtia poikkeusten kannalta sen jälkeen, kun luokat ja luokkahierarkiat on esitelty.