Tampereen yliopisto
Tampere University

# Virhetilanteet ja poikkeusten hallinta

4.9.2019

Tampereen yliopisto
Tampere University

# Virhetilanteet

- Varautuminen ja reagoiminen usein vaikeaa:
  - Sekava koodi
  - Siivoustyön tarve
  - Toipuminen -> kesken jääneiden töiden peruutus

Tampereen yliopisto
Tampere University

# Virhe?

- Ulkoinen virhe: ohjelmaa pyydetään tekemään jotain, mitä se ei osaa tai mihin se ei pysty
- Sisäinen virhe: toteutus ajautuu tilanteeseen, jossa jotain menee pieleen
- Virhetilanteen havaitseminen helpoin vaihe

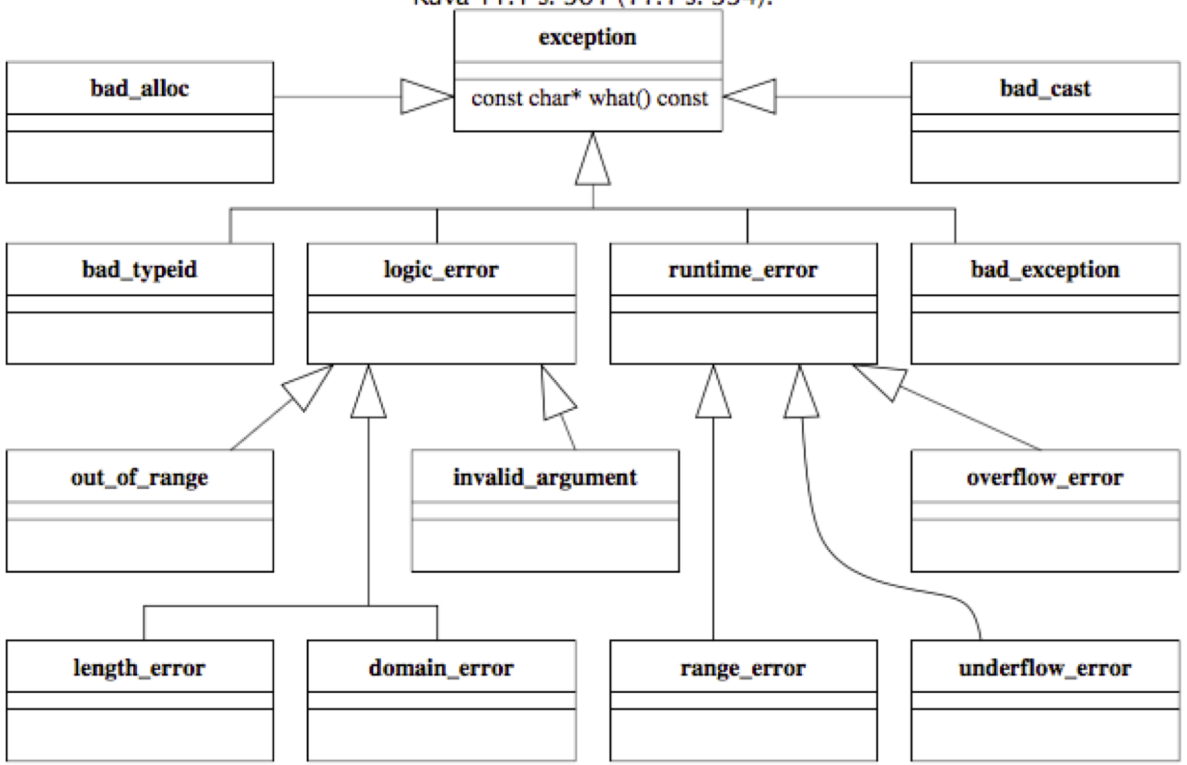
Tampereen yliopisto
Tampere University

# Varautuva ohjelmointi

- Varaudutaan siihen, että muut voivat toimia väärin
- Mitä voidaan tehdä, kun virhe havaitaan?
  - Keskeytys
  - Hallittu lopetus
  - Jatkaminen
  - Peruuttaminen
  - Toipuminen

# Virhehierarkiat

Kuva 11.1 s. 301 (11.1 s. 334):




363 *Olioiden ohjelmointi C++:lla*, Matti Rintala & Jyke Jokinen1941

Tampereen yliopisto
Tampere University

# Virhetyypit C++:n luokkina

```
// Nämä kaikki ovat std-nimiavaruudessa
class exception {
public:
    exception() throw(); // throw() selitetään myöhemmin
    exception(exception const& e) throw();
    exception& operator =(exception const& e) throw();
    virtual ~exception() throw();
    virtual char const* what() const throw();
    ...
};

class runtime_error : public exception {
public:
    runtime_error(std::string const& msg);
};
class overflow_error : public runtime_error {
public:
    overflow_error(std::string const& msg);
};
```

# Poikkeushierarkien käyttö

```
class LiianPieniArvo : public std::domain_error
{
public:
    LiianPieniArvo(std::string const& viesti,
                   int luku, int minimi);
    LiianPieniArvo(LiianPieniArvo const& virhe);
    virtual ~LiianPieniArvo() throw();
    int annaLuku(() const;
    int annaMinimi() const;
private:
    int luku_;
    int minimi_;
};
```

# Poikkeusten hallinta

- Virhealtis koodi kirjoitetaan **valvontalohkon** (*try*) sisään
- Jos havaitaan virhe, poikkeus **heitetään** (*throw*) “ilmaan”
- Poikkeuskäsittelijä **sieppaa** (*catch*) poikkeuksen ja käsittelee virheen
- Virhekäsittelyn päätyttyä:
  - Ei palata virhekohtaan, vaan jatketaan virhekäsittelyrakenteen jälkeisestä kohdasta

# Poikkeusten hallinta

- Poikkeuskäsittelijän etsiminen
  - Peruutetaan kutsuketjussa, kunnes sopiva käsittelijä löytyy
- Parametri määrää:
  - Minkä tyyppiset virheet siepataan
  - Viite virhekantaluokkaan -> mikä tahansa periytetty virhe
- Tai parametri voi puuttua

# Esimerkki

```
void lueLuvutTaulukkoon(vector<double>& taulu);

double summaaLuvut(vector<double> const& luvut) {
    double summa = 0.0;
    for (unsigned int i = 0; i < luvut.size(); ++i) {
        if (summa >= 0 && luvut[i] > numeric_limits<double>::max() - summa)
        {
            throw std::overflow_error("Summa liian suuri");
        }
        else if (summa < 0 && luvut[i] < -numeric_limits<double>::max() - summa)
        {
            throw std::overflow_error("Summa liian pieni");
        }
        summa += luvut[i];
    }
    return summa;
}
```

# Esimerkki (jatkuu)

```
double laskeKeskiarvo(vector<double> const& luvut) {
    unsigned int lukumaara = luvut.size();
    if (lukumaara == 0)
    {
        throw std::range_error("Lukumäärä keskiarvossa 0");
    }
    return summaaLuvut(luvut) / static_cast<double>(lukumaara);
}
```

# Esimerkki (jatkuu)

```
void keskiarvo1(vector<double>& lukutaulu) {
    try
    {
        lueLuvutTaulukkoon(lukutaulu);
        double keskiarvo = laskeKeskiarvo(lukutaulu);
        cout << "Keskiarvo: " << keskiarvo << endl;
    }
    catch (std::range_error const& virhe)
    {
         cerr << "Lukualuevirhe: " << virhe.what() << endl;
    }
    catch (std::overflow_error const& virhe)
    {
        cerr << "Ylivuoto: " << virhe.what() << endl;
    }
    cout << "Loppu" << endl;
}
```

# Virhe-kategoriat

```
void keskiarvo2(vector<double>& lukutaulu){
    try
    {
        lueLuvutTaulukkoon(lukutaulu);
        double keskiarvo = laskeKeskiarvo(lukutaulu);
        cout << "Keskiarvo: " << keskiarvo << endl;
    }
    catch (std::runtime_error const& virhe)
    {
        // Tänne tullaan minkä tahansa ajoaikaisen virheen seurauksena
        cerr << "Ajoaikainen virhe: " << virhe.what() << endl;
    }
    cout << "Loppu" << endl;
}
```

# Sieppaamattomat poikkeukset

- Esim. muistin loppuminen
  - Kutsutaan funktiota **terminate**
    - Lopettaa ohjelman suorituksen (ja antaa virheilmoituksen)
  - Ohjelmoija voi määritellä itse oman toteutuksen funktiolle

# Yleispoikkeuskäsittelijä

- Sieppaa minkä tahansa poikkeuksen
  - Lupaa käsitellä kaikki virheet! ->älä käytä turhaan!
  - Kätevä siivoukseen, jos virhe heitetään lopussa uudelleen

```
catch (...) // Todellakin ... eli kolme pistettä
{
      // Tämä poikkeuskäsittelijä sieppaa kaikki poikkeukset
}
```

Tampereen yliopisto
Tampere University

# Sisäkkäiset valvontalohkot

```
int main() {
    vector<double> taulu;
    try {
        keskiarvo2(taulu); // Lue luvut ja laske keskiarvo
    }
    catch (std::bad_alloc&) {
        cerr << "Muisti loppui!" << endl;
        return EXIT_FAILURE;
    }
    catch (std::exception const& virhe) {
        cerr << "Virhe pääohjelmassa: " << virhe.what() << endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Tampereen yliopisto
Tampere University

# Poikkeukset & olioiden tuhoaminen

- Poikkeuskäsittelijän etsiminen -> Poistutaan koodilohkoista -> Olioiden elinkaari saattaa päättyä
- -> Nämä oliot tuhotaan automaattisesti
- -> Poikkeuksista ei ongelmia, jos olion elinkaari staattinen

# Poikkeukset & purkajat

- Yhdessä lohkossa vain yksi poikkeus kerrallaan
  - Purkaja ei saa vuotaa poikkeusta ulos!
  - Jos näin käy -> terminate
  - Ei yleensä ongelma (siivoaminen onnistuu aina)

Tampereen yliopisto
Tampere University

# Poikkeukset & dynaamisesti luodut oliot

- Dynaamisesti luoduilla olioilla ei minkään ohjelmarakenteen määräämää elinkaarta
- -> Poikkeusmekanismi ei automaattisesti tuhoa **new**'llä luotuja olioita
- Paikalliset osoitinmuuttujat tuhotaan -> roikkuvia olioita

Tampereen yliopisto
Tampere University

# Poikkeukset & dynaamisesti luodut oliot

- Ratkaisu 1:
  - Otetaan kiinni kaikki virheet, tuhotaan **delete**llä oliot, heitetään poikkeus uudelleen
  - Kieli keskellä suuta, jos useita olioita (virhe ennen kuin kaikki luotu)
- Ratkaisu 2: fiksut osoittimet

# Ratkaisu 1

```
void siivousfunktio()
{
    vector<double>* taulup = new vector<double>();
    try
    {   // Jos täällä sattuu virhe, vektori pitää tuhota
        keskiarvo2(*taulup);
    }
    catch (…)
    {   // Otetaan kiinni kaikki virheet ja tuhotaan vektori
        delete taulup; taulup = 0;
        throw;  // Heitetään poikkeus edelleen käsiteltäväksi
    }
    delete taulup; taulup = 0;  // Tänne päästään, jos virheitä
                                // ei satu
}
```

Tampereen yliopisto
Tampere University

# Poikkeusturvallisuus

- Kapselointi piilottaa tietoa toteutuksesta — myös mahdollisista poikkeustilanteista
- Periytyminen & polymorfismi — toteutus yhä kauempana ja vaihteleva
- **-> Rajapinnan dokumentointi** äärimmäisen tärkeää

Tampereen yliopisto
Tampere University

# Poikkeusturvallisuus

- Periytetyt luokat eivät saa rikkoa kantaluokan lupauksia
- Kantaluokka ei saa luvata liikoja virheistä tai niiden puuttumisesta
- Dokumentointi helpompaa, jos yleisiä termejä eri tilanteille: **poikkeustakuut**

Tampereen yliopisto
Tampere University

# Poikkeustakuut

- **Minimitakuu** (minimal guarantee) **–** Ei hukata resursseja
  - olio tuhottavissa/resetoitavissa mutta ei muuten käyttökelpoinen
  - luokkainvariantti ei välttämättä voimassa

# Poikkeustakuut

- **Perustakuu** (basic guarantee)
  - olion tila ei-ennakoitava, mutta järjellinen
  - luokkainvariantti edelleen voimassa
  - olio sinänsä edelleen käyttökelpoinen

# Poikkeustakuut

- **Vahva takuu** (strong guarantee)
  - Kaikki tai ei mitään (commit or rollback)
- **Nothrow-takuu** (nothrow guarantee)
  - Ohjelmoijan taivas
- Lisäksi näistä erillään **poikkeusneutraalius** (exception neutrality)