Tampereen yliopisto
Tampere University

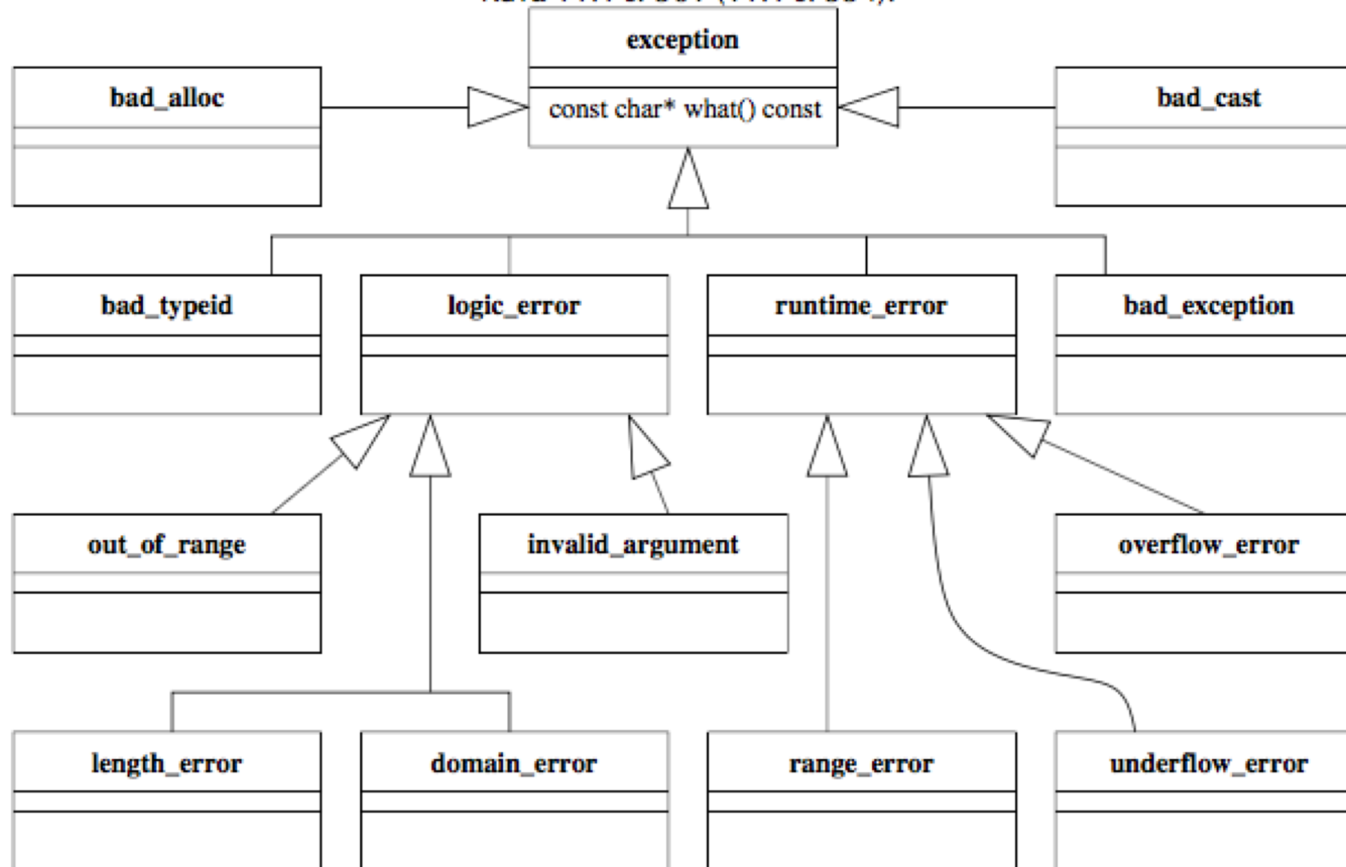
Poikkeusturvallisuus

2.10.2019

Tampereen yliopisto
Tampere University

Virhehierarkiat

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



Tampereen yliopisto
Tampere University

Poikkeukset ja rakentajat

- Olio rakennetaan vähitellen — milloin "syntyhetki"?
- Olennaista, jos rakentamisessa virheitä
- C++: olio syntynyt, kun kaikki rakentajat suoritettu onnistuneesti

Poikkeukset ja rakentajat

- Yksikin käsittelemätön virhe rakentajissa → oliota ei ole
- Jo rakennettujen jäsenmuuttujien ja kantaluokkien purkajat suoritetaan
- Oltava tarkkana, jos rakentajassa dynaamisesti luotuja olioita

Tampereen yliopisto
Tampere University

Poikkeukset ja rakentajat

```
class Henkilo {
public:
    Henkilo(int p, int k, int v,
            std::string const& nimi,
            std::string const& hetu);
    ~Henkilo();
private:
    Paivays syntymapvm_;
    std::string nimi_;
    std::string* hetup_;
};
```

Tampereen yliopisto
Tampere University

Poikkeukset ja rakentajat

```
Henkilo::Henkilo(int p, int k, int v, std::string const& nimi,
                          std::string const& hetu)
      : syntymapvm_(p, k, v), nimi_(nimi), hetup_(0) {
    try {
        hetup_ = new std::string(hetu);
    }
    catch (...) {
        // Tänne päästään, jos hetun luominen epäonnistuu
        // Siivotaan tarvittaessa, olion luominen epäonnistui
        throw; // Heitetään virhe edelleen käsiteltäväksi
    }
}
```

Luomisvirheisiin reagoiminen

- Jäsenmuuttujan tai kantaluokan rakentaja epäonnistuu → luominen ei voi onnistua
- Dynaamisesti luodun luomista voi yrittää uudelleen, jos järkevää
- Jäsenmuuttujien ja kantaluokkien rakentajien virheet kiinni **funktion valvontalohkossa** (*function try block*)

Luomisvirheisiin reagoiminen

- Vain virheen muuttaminen toiseksi, toipuminen ei mahdollista
- Jäsenmuuttujat ja kantaluokkaosat jo tuhottu → ei pääsyä

Tampereen yliopisto
Tampere University

Funktion valvontalohko rakentajassa

```
Henkilo::Henkilo(int p, int k, int v, std::string const& nimi,
                               std::string const& hetu)
try // Huomaa try-sanan paikka!
  : syntymapvm_(p, k, v), nimi_(nimi), hetup_(0) {
    …
}
catch (...) { // Tänne päästään, jos jäsenmuuttujan (tai
kantaluokan)
    // luominen epäonnistuu. Tehdään tarvittaessa toimenpiteitä.
    throw; // … tai heitetään jokin toinen poikkeus
}
```

Poikkeukset ja purkajat

- Purkajista **ei** saisi vuotaa poikkeuksia!
- Purkajan pitää itse käsitellä aiheuttamansa poikkeukset pois päiväjärjestyksestä
- Jos tämä ei mahdollista, erillinen siivousjäsenfunktio parempi
- Funktion valvontalohko periaatteessa mahdollinen, mutta lähes hyödytön purkajissa
- `uncaught_exception` — myös lähes hyödytön

Tampereen yliopisto
Tampere University

Poikkeusturvallisuus

- Kapselointi piilottaa tietoa toteutuksesta — myös virhevaaroista
- Periytyminen & polymorfismi — toteutus yhä kauempana ja vaihteleva

→ **Rajapinnan dokumentointi** äärimmäisen tärkeää

Poikkeustakuut

- 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)

Tampereen yliopisto
Tampere University

Esimerkki: Poikkeusturvallinen sijoitus

- Tutun luokan Kirja sijoitusoperaattorin analysointi ja parantaminen
- **Vaihe 1**: analysoidaan olemassaoleva poikkeustakuu
- **Vaihe 2**: parannetaan, jos mahdollista ja järkevää

Yksinkertainen luokka, jolla on sijoitusoperaattori

```
class Kirja {
public:
    …
    Kirja& operator =(Kirja
                 const& kirja);
private:
    std::string nimi_;
    std::string tekija_;
};
```

```
Kirja& Kirja::operator
=(Kirja const& kirja)
{
    if (this != &kirja) {
        nimi_ = kirja.nimi_;
        tekija_ =
          kirja.tekija_;
    }
    return *this;
}
```

Tavoitteena vahva takuu

```
Kirja& Kirja::operator =(Kirja const& kirja) {
    if (this != &kirja) {
        std::string vanhanimi(nimi_);
        std::string vanhatekija(tekija_);
        try {
            nimi_ = kirja.nimi_;
            tekija_ = kirja.tekija_;
        }
        catch (...) {
            nimi_ = vanhanimi;
            tekija_ = vanhatekija;
            throw;
        }
    }
    return *this;
}
```

Tampereen yliopisto
Tampere University

Lisätään epäsuoruutta

```
class Kirja {
public:
    Kirja(std::string const& nimi, std::string const& tekija);
    // Tarvitaan myös oma kopiorakentaja (dynaaminen muistinhallinta)!
    ~Kirja();
    // ...
    Kirja& operator =(Kirja const& kirja);
private:
    std::string* nimip_;
    std::string* tekijap_;
};
```

Vahva takuu epäsuoruudella

```
Kirja::Kirja(std::string const& nimi, std::string const& tekija) :
             nimip_(0), tekijap_(0) {
    try {
        nimip_ = new std::string(nimi);
        tekijap_ = new std::string(tekija);
    }
    catch (...) {
        delete nimip_; nimip_ = 0;
        delete tekijap_; tekijap_ = 0;
        throw;
    }
}
```

Vastaavasti:

```
Kirja::~Kirja() {
    delete nimip_; nimip_ = 0;
    delete tekijap_; tekijap_ = 0;
}
```

... jatkoa

```
Kirja& Kirja::operator =(Kirja const& kirja) {
    if (this != &kirja) /* Periaatteessa tarpeeton! */ {
        std::string* uusinimip = 0;
        std::string* uusitekijap = 0;
        try {
            uusinimip = new std::string(*kirja.nimip_);
            uusitekijap = new std::string(*kirja.tekijap_);
            // Jos päästiin tänne, ei virheitä tullut
            delete nimip_; nimip_ = uusinimip; // Onnistuvat aina
            delete tekijap_; tekijap_ = uusitekijap; // Samoin nämä
        }
        catch (...) {
            delete uusinimip; uusinimip = 0;
            delete uusitekijap; uusitekijap = 0;
            throw;
        }
    }
    return *this;
}
```

Tampereen yliopisto
Tampere University

Tilan eriyttäminen (*pimpl*)

```
class Kirja {
public:
    Kirja(std::string const& nimi, std::string const& tekija);
    // Tarvitaan myös oma kopiorakentaja!
    ~Kirja();
    // ...
    Kirja& operator =(Kirja const& kirja);
private:
    struct Tila;
    std::unique_ptr<Tila> tilap_;
};
```

Tampereen yliopisto
Tampere University

Tilan eriyttäminen (*pimpl*)

```
struct Kirja::Tila {
    std::string nimi_;
    std::string tekija_;
    Tila(std::string const& nimi, std::string const& tekija) :
         nimi_(nimi), tekija_(tekija) {}
};

Kirja::Kirja(std::string const& nimi, std::string const& tekija) :
             tilap_(new Tila(nimi, tekija))
{
}

Kirja::~Kirja()
{ // Uniikkiosoitin tuhoaa tilan automaattisesti
}
```

Tampereen yliopisto
Tampere University

Tilan eriyttäminen (*pimpl*)

```
Kirja& Kirja::operator =(Kirja const& kirja)
{
    std::unique_ptr<Tila> uusitilap =
                                std::make_unique<Tila>(*kirja.tilap_);
    tilap_ = std::move(uusitilap);  // Ei voi epäonnistua ja
                                    // tuhoaa vanhan tilan
    return *this;
}
```

Tampereen yliopisto
Tampere University

Tilan vaihtaminen päikseen (nothrow)

```
class Kirja
{
public:
    // ...
    Kirja& operator =(Kirja const& kirja);
    void vaihda(Kirja& kirja);
private:
    std::string nimi_;
    std::string tekija_;
};
```

Tilan vaihtaminen päikseen (nothrow)

```
void Kirja::vaihda(Kirja& kirja) {
    nimi_.swap(kirja.nimi_); // Ei voi epäonnistua
    tekija_.swap(kirja.tekija_); // Eikä tämäkään
}

Kirja& Kirja::operator =(Kirja const& kirja) {
    Kirja kirjakopio(kirja); // Kopio sijoitettavasta
    vaihda(kirjakopio); // Vaihdetaan itsemme siihen,
                        // ei epäonnistu
    return *this; // Vanha tila tuhoutuu kirjakopion myötä
}
```

Tampereen yliopisto
Tampere University

C++: erityismääreet

- `override` (virtuaalifunktiolle) → aliluokka tarjoaa oman toteutuksen
- `final` (virtuaalifunktiolle) → aliluokka ei voi tarjota omaa toteutusta
- **`noexcept`** → funktio ei heitä poikkeusta
- `= 0` → puhdas virtuaalifunktio