Objektinis Programavimas

Klaidų valdymas. Išimtys

Iš čia kylama į žvaigždes

© dr. Remigijus Paulavičius, remigijus.paulavicius@mii.vu.lt

Turinys

Klaidų tvarkymas

Klaidų tvarkymas

— Klaidų paieška ir tvarkymas yra svarbus ir sudėtingas procesas, išeinantis už C++ kalbos ribų (pvz. debugeriai).

— Tačiau ir C++ suteikia keletą tam skirtų svarbių priemonių.

— Pirmiausia, C++ tipų sistema, vietoj integruotų tipų (char, int, double ir pan.) ir reiškinių (while, for ir pan.) leidžia saugiau (nes jų realizacijos automatiškai atlieka klaidų "gaudymą"), o dažniausiai ir efektyviau naudoti standartinėje bibliotekoje realizuotus tipus (std::vector, std::string ir t.t.) bei algoritmus (sort(), find() ir t.t.)

Klaidų tipai

Klaidas galima būtų suskirstyti į dvi esmines kategorijas:
- **sintaksės klaidos**;
- **semantinės klaidos**.

— **Sintaksės klaidas** beveik visada aptinka kompiliatoriai ir jas lengva ištaisyti. Todėl paprastai per daug nerimauti dėl jų nereikia. Pvz:

```
#include <iostream>
int main() {
    std::cour << "Ir kas gi čia blogai?" << std::endl;
}
```

```
In function 'int main()':
prog.cc:3:10: error: 'cour' is not a member of 'std'; did you mean 'cout'?
```

Semantinės klaidos (1)

— **Semantinės klaidos** atsiranda, kai sintaksės atžvilgiu reiškinys yra teisingas, tačiau daro ne visai tai, ką norėjo programuotojas:

```
#include <vector>
#include <iostream>
int main() {
    std::vector<int> vi{10,1};
    int suma = 0;
    for (uint i = 0; i <= vi.size(); ++i)
        suma += vi[i];
    std::cout << "suma = " << suma << std::endl;
}
```

— Kokią kintamojo `suma` reikšmę gausime?

Semantinės klaidos (2)

- Semantinių klaidų kompiliatorius neaptinka, todėl jos gali turėti įvairių "šalutinių poveikių", pvz.:
  - gali 99% proc. atvejų visvien viskas veikti gerai, tačiau tas 1% "varo iš proto" 😡
  - programa generuoja "neteisingą" išvestį;
  - sugadina programos duomenis;
  - užlaužia programą.

Semantinės klaidos (3)

— Semantinės klaidos atsiranda dėl įvairių priežasčių.

— Viena iš dažniausiai pasitaikančių semantinių klaidų yra **loginė klaida**.

— Loginė klaida atsiranda, kai neteisingai suprogramuojama reiškinio logiką.

Semantinės klaidos (4)

```
#include <vector>
#include <iostream>
int main() {
    // sukuriame vektorių iš 10 narių, visų reikšmės 1
    std::vector<int> vi{10,1};
    int suma = 0;
    // predikatas: sudėk pirmų 10 narių sumą
    for (uint i = 0; i <= 10; ++i)
        suma += vi[i];
    std::cout << "suma = " << suma << std::endl;
}
```

— Ar dabar lengviau pastebėti klaidas?

Semantinės klaidos (5)

— Kita tipinė semantinių klaidų priežastis yra pažeistos programos prielaidos:

```
void printElement(const std::vector<int>& vi, int i) {
    std::cout << vi[i] << std::endl;
}
```

— O kas nutiks jeigu i neigiamas arba už vi.size() ribų?

assert() **(1)**

— assert yra preprocesoriaus makro, kur joje apibrėžta sąlyginė išraiška įvertinama programos vykdymo (**runtime**) metu.

— Jei sąlyginė išraiška įvertinama kaip teisinga, assert išraiška nieko nedaro.

— Jei sąlyginė išraiška įvertinama kaip klaidinga, parodomas klaidos pranešimas (kartu su failo vardu ir eilutės eil. nr), o programa nutraukiama (terminate'd).

assert() **(2)**

```
#include <iostream>
#include <vector>
#include <cassert> // assert() header'is

void printElement(const std::vector<int>& vi, uint i) {
    // assert'iname ar indeksas i yra tarp [0 ir vi.size())
    assert(i >= 0 && i < vi.size());
    std::cout << vi[i] << std::endl;
}

int main() {
    std::vector<int> vi(10,1);
    printElement(vi, 5);  // Ką gausime čia?
    printElement(vi, 10); // O čia ką gausime?
}
```

static_assert()

— Nuo C++11 atsirado static_assert, kuris skirtingai nuo assert (įvertina sąlyga programos vykdymo metu), įvertina sąlygą kompiliavimo metu:

```
static_assert(sizeof(int) == 4, "int tipas tūri būti 4 baitai");
```

standard error (std::cerr) **srautas (1)**

— std::cerr yra dar specialus **standart error** srautas, skirtas klaidų pranešimams spausdinti.

— std::cerr yra **output** srautas (kaip ir std::cout), kuris yra apibrėžtas <iostream> header'yje.

— Įprastai std::cerr rašo klaidos pranešimus ekrane (kaip ir std::cout), tačiau jį taip pat būtų galima nukreipti pvz. į failą - patogesnei klaidų analizei.

standard error (std::cerr) **srautas(2)**

```
// Išspausdinti failo turinį į std::cout
void printFileContent(const string& filename) {
    // atidaryti įvesties (input) failą
    std::ifstream file(filename);
    // ar atidarytas failas?
    if (!file) {
        // Į error srautą išspausdinti klaidos pranešimą
        std::cerr << "negali atidaryti įvesties failo - " << filename << std::endl;
        std::terminate() // nutraukti programos vykdymą
    }
    // nukopijuoti failo turinį (imant po simbolį) į `std::cout`
    char c;
    while (file.get(c)) {
        std::cout.put(c);
    }
} // uždaro failą automatiškai
```

standard error (std::cerr) **srautas(3)**

— Šis pavyzdys daugiau susijęs su darbu su failais.

— Vietoj to, kad nukopijuotumėte failo turinio simbolis po simbolio, efektyviau visą failo turinį nukopijuoti "vienu ūpu", t.y. operatoriui opertor<< dešiniuoju operandu perduodant rodyklę į failo srauto buferį rdbuf():

```
// Išspausdinti failo turinį į std::cout
void printFileContent(const string& filename) {
    // atidaryti įvesties (input) failą
    std::ifstream file(filename);
    // ar atidarytas failas?
    if (!file) {
        // Į error srautą išspausdinti klaidos pranešimą
        std::cerr << "negali atidaryti įvesties failo - " << filename << std::endl;
        std::terminate() // nutraukti programos vykdymą
    }
    // nukopijuoti failo turinį į `std::cout` vienu ūpu
    std::cout << file.rdbuf();
} // uždaro failą automatiškai
```

Klaidų kodų (error codes**) (1)**

— Vienas iš tradicinių programavimų būdų "klaidingas situacijas" valdyti panaudojant **return** reikšmes (kodus). Pvz.:

```
// Vector::operator[] realizacija
double& Vector::operator[](int i) {
    if (i < 0 || size() <= i)
        return -1; // grąžina -1 kaip klaidos ženklą
    return elem[i];
}
```

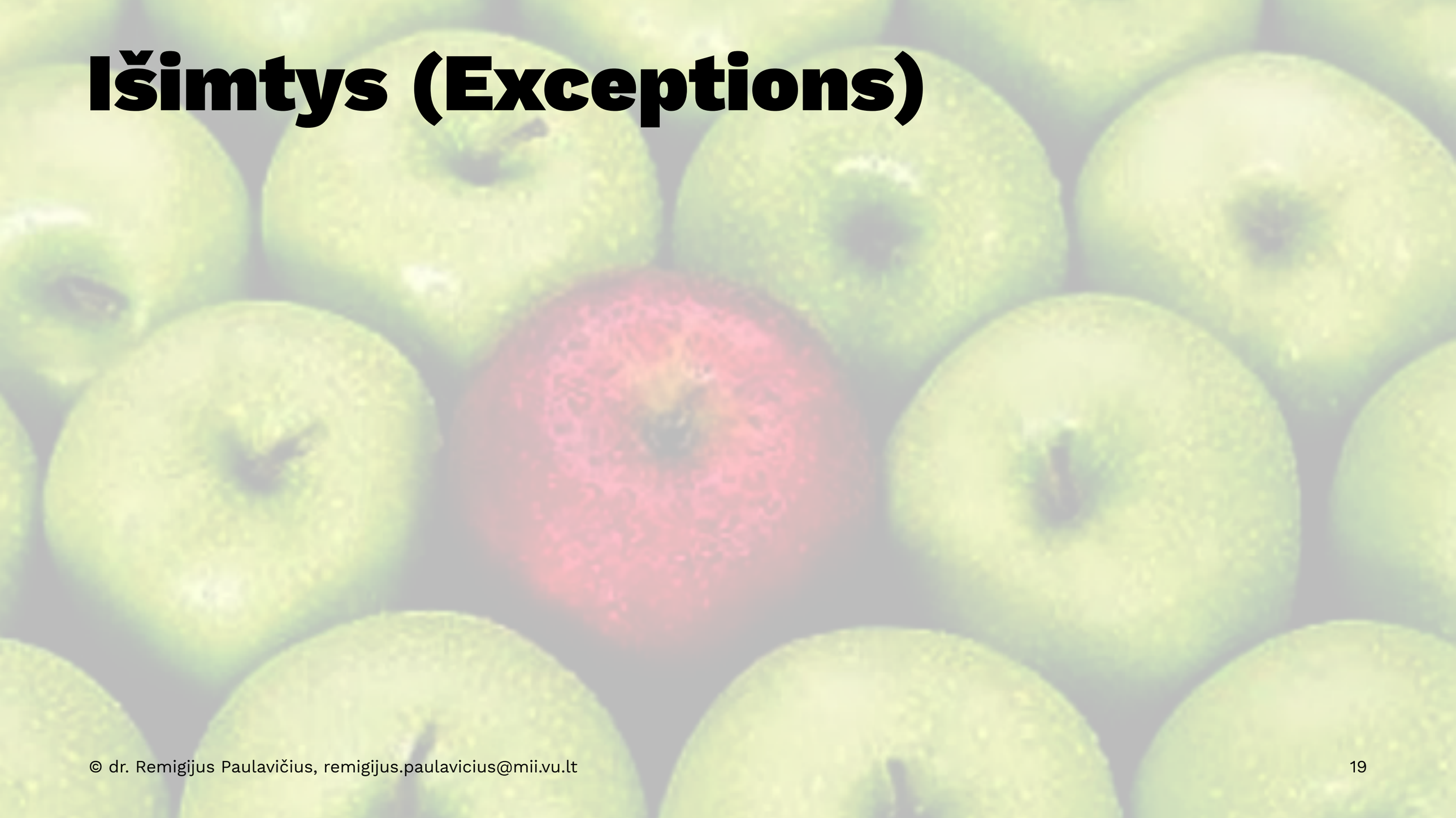
Klaidų kodų (error codes**) (2)**

— Pagrindinis šio klaidų valdymo būdo privalumas yra tai, kad jis yra labai paprastas.

— Tačiau, valdant klaidingas situacijas panaudojant **return** kodus, yra keletas labai svarbių trūkumų:

1. Iš grąžinimo reikšmės ne visada aišku, ar ji reiškia klaidos kodą ar iš tiesų yra teisinga grąžinta reikšmė?
2. Funkcijos gali grąžinti tik vieną vertę, taigi, kas atsitinka, kai reikia grąžinti funkcijų rezultatą ir klaidos kodą?

Išimtys (Exceptions)



Motyvacija

1. Klaidų (išimčių) tvarkymo kodo atskyrimas nuo įprastinio programos kodo.
2. Laisvė Funkcijoms/metodams pasirinkti į kurias (kokio tipo) išimtis jie nori atsižvelgti, o kurias palikti tvarkyti "kitiems".
3. Klaidų tipų grupavimas (susisteminimas).

vector::operator[]

```
double& Vector::operator[](int i) {
    if (i < 0 || size() <= i)
        throw std::out_of_range { "Vector::operator[]" };
    return elem[i];
}
```

— Pasvarstykime, kas turėtų nutikti jei bandytume pasiekti elementą, neegzistuojantį (pvz. **out of range**) konteineryje (pvz. Vector'iuje)?
  — Vektorių kūrėjai nežino, ką vartotojas norėtų daryti šiuo atveju.
  — Kūrėjai nežino, kokiuose kontekstuose vektoriai bus naudojami.
  — Vektorių vartotojai negali aptikti kas šią problemą lemia.
— Sprendimas būtų Vektorių kūrėjams aptikti (**out of range**) bandymą ir apie tai informuoti vartotojui. Tada vartotojas gali imtis atitinkamų veiksmų.

vector::operator[] vs. vector::at

— vector::operator[]

  Returns a reference to the element at specified location pos. No bounds checking is performed.[1]

— vector::at

  Returns a reference to the element at specified location pos, with bounds checking. If pos is not within the range of the container, an **exception** of type `std::out_of_range` is **thrown**.[2]

[1] http://en.cppreference.com/w/cpp/container/vector/operator_at

[2] http://en.cppreference.com/w/cpp/container/vector/at

vector::operator[] pavyzdys

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v {1, 2, 3, 4, 5};
    int i = v[5];
    std::cout << "i = " << i << std::endl;
}

// Output (Clang 3.8/GCC 7.1):
i = 0
```

vector::at pavyzdys (1)

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v {1, 2, 3, 4, 5};
    int i = v.at(5);
    std::cout << "i = " << i << std::endl;
}

// Output:
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

vector::at pavyzdys (2)

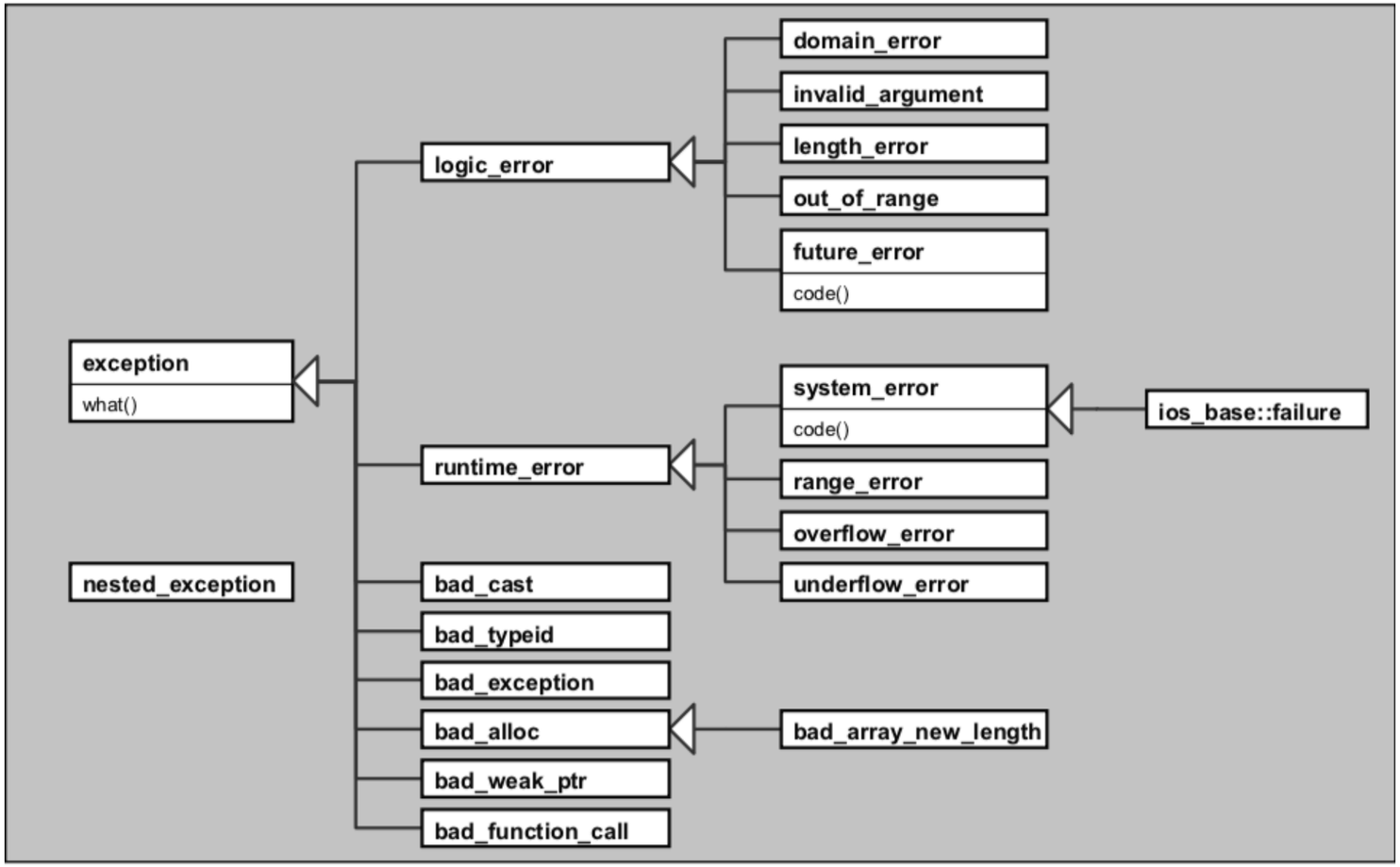
— Šiuo atveju `vector::at` aptinka bandymą pasiekti neegzistuojantį (**out-of-range**) elementą, `throw`'ina **out_of_range** išimtį (`<stdexcept>) ir nutraukia (**terminate**) programos vykdymą.

— `throw` perduoda **out_of_range** tipo išskirčių valdymą funkcijai, kuri tiesiogiai (ar netiesiogiai) naudojo `vector::at`.

vector::at pavyzdys (3)

```
#include <iostream>
#include <vector>
#include <stdexcept>
int main() {
    std::vector<int> v {1, 2, 3, 4, 5};
    try {
        int i = v.at(5);
        std::cout << "i = " << i << std::endl;
    }
    catch (const std::out_of_range& msg) {
        std::cout << "Pagavau `std::out_of_range išimtį!\n";
    }
    // std::cout << "Ar čia daro?\n";
}
```

Standartinių išimčių hierarchija



Header failai išimties klasėms

— Išimčių klasės yra apibrėžtos daugelyje skirtingų header failų:

```
#include <exception>     // for classes exception and bad_exception
#include <stdexcept>     // for most logic and runtime error classes
#include <system_error>  // for system errors (since C++11)
#include <new>           // for out-of-memory exceptions
#include <ios>           // for I/O exceptions
#include <future>        // for errors with async() and futures (since C++11)
#include <typeinfo>      // for bad_cast and bad_typeid
```

Bendros funkcijos išimčių apdorojimui (1)

Šis kodas parodo, kaip naudoti bendrą funkciją, skirtą apdoroti skirtingas išimtis:

```
#include <iostream>
#include <vector>
#include <exception>
#include <stdexcept>
#include <system_error>
#include <new>
#include <ios>
#include <future>
#include <typeinfo>
```

Bendros funkcijos išimčių apdorojimui (2)

```
template <typename T>
void processCodeException (const T& e) {
  using namespace std;
  auto c = e.code();
  cerr << "- category: " << c.category().name() << endl;
  cerr << "- value:    " << c.value() << endl;
  cerr << "- msg:      " << c.message() << endl;
  cerr << "- def category: "
       << c.default_error_condition().category().name() << endl;
  cerr << "- def value: "
       << c.default_error_condition().value() << endl;
  cerr << "- def msg: "
       << c.default_error_condition().message() << endl;
}
```

Bendros funkcijos išimčių apdorojimui (3)

```
void processException() {
  using namespace std;
  try {
    throw; // rethrow išimtį, kad ją apdoroti čia
  }
  catch (const ios_base::failure &e) {
    cerr << "I/O EXCEPTION: " << e.what() << endl;
    processCodeException(e);
  }
  catch (const system_error &e) {
    cerr << "SYSTEM EXCEPTION: " << e.what() << endl;
    processCodeException(e);
  }
  catch (const future_error &e) {
    cerr << "FUTURE EXCEPTION: " << e.what() << endl;
    processCodeException(e);
  }
  catch (const bad_alloc &e) {
    cerr << "BAD ALLOC EXCEPTION: " << e.what() << endl;
  }
  catch (const exception &e) {
    cerr << "EXCEPTION: " << e.what() << endl;
  }
  catch (...) {
    cerr << "EXCEPTION (unknown)" << endl;
  }
}
```

Bendros funkcijos išimčių apdorojimui (4)

```
int main() {
    std::vector<int> v{0, 1, 2, 3, 4};
    // tuomet galime naudoti štai tokią struktūrą
    try {
      // pavyzdžiai, kurie "iššaukia" išimtis
      std::cout << v.at(5) << std::endl;
      //int* array = new int[50000000000000];
    }
    catch (...) {
      processException();
    }
    return 0;
}
```

Kuomet nereiktų vartoti išimčių?[isocpp]

C++ exceptions are designed to support **error handling**!

— Use throw **only to signal an error** (which means specifically that the function couldn't do what it advertised, and establish its postconditions).

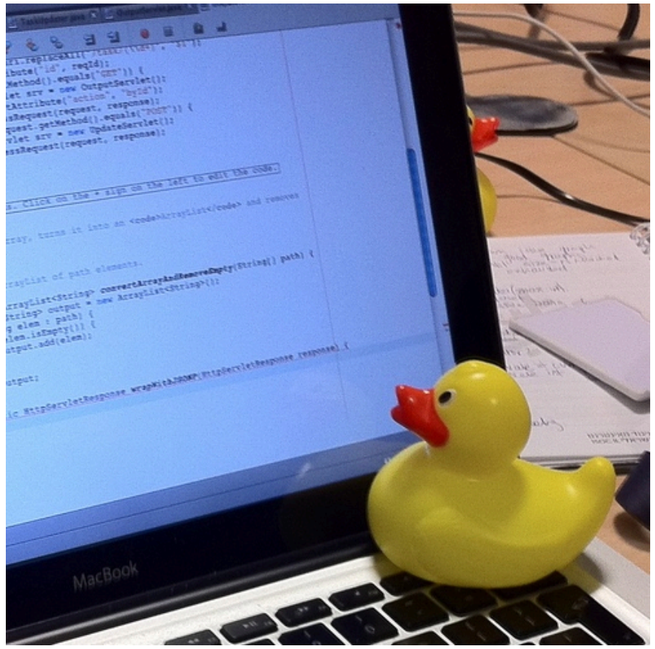
— Use catch only to specify error handling actions **when you know you can handle an error** (possibly by translating it to another type and rethrowing an exception of that type, such as catching a `bad_alloc` and rethrowing a `no_space_for_file_buffers`).

— Do not use throw to indicate a coding error in usage of a function. Use assert or other mechanism to either send the process into a debugger or to crash the process and collect the crash dump for the developer to debug.

— Do not use throw if you discover unexpected violation of an invariant of your component, use assert or other mechanism to terminate the program. **Throwing an exception will not cure memory corruption and may lead to further corruption of important user data.**

[isocpp] https://isocpp.org/wiki/faq/exceptions

Klausimai?![rdd]



[rdd] https://en.wikipedia.org/wiki/Rubberduckdebugging