KRITINIŲ SITUACIJŲ KONTROLĖ. IŠIMTYS (EXCEPTIONS).

Programos darbo metu gali atsirasti įvairių priežasčių, dėl kurių neįmanoma tęsti programos vykdymo, pvz., dalyba iš nulio, failų atidarymo (rašymo, skaitymo) klaida, OS klaidos ir kt. Tokios situacijos vadinamos *kritinėmis* (*išimtinėmis*).

Profesionaliai parengtose programose turi būti numatyta apsauga nuo tokių kritinių situacijų. Pvz., programa galėtų informuoti vartotoją apie išimtis ir siūlyti tam tikrus situacijos sprendimo variantus. Galimi tokie sprendimo variantai.

1. Jei pačioje programoje tokios apsaugos nėra, tai kritinė situacija apdorojama **standartinėmis priemonėmis** (pvz., *Delphi/Borland C++* sistemoje – *TApplication.HandleException* metodas). Jos išveda trumpą standartinį pranešimą (pvz., *Division by zero*) ir naikina išimties egzempliorių (programą).

Standartinio apdorojimo trūkumai:

- „resursų nutekėjimas“, pvz., lieka neištrinti laikinieji failai diske, lieka neišvalyti dinaminės atminties fragmentai ir pan.;
- programos vartotojui nesuteikiama jokia papildoma informacija apie kritinę situaciją, jis nežino, ką toliau daryti.

2. **Išimčių vengimas** – tai kiekvienos potencialiai pavojingos operacijos tikrinimas programoje, pvz., ***if*** *(b==0)* ... . Privalumas – didesnis programos patikimumas. Tačiau trūkumai taip pat akivaizdūs:

- auga kodo apimtis,
- sunku numatyti kritines situacijas bibliotekinėse funkcijose,
- objektinėse programose išimtys dažnai aptinkamos vienose programos vietose (duomenų apdorojimo objektuose), o galima situacijos korekcija turi būti atlikta kitoje vietoje (apdorojimą inicijavusiuose objektuose).

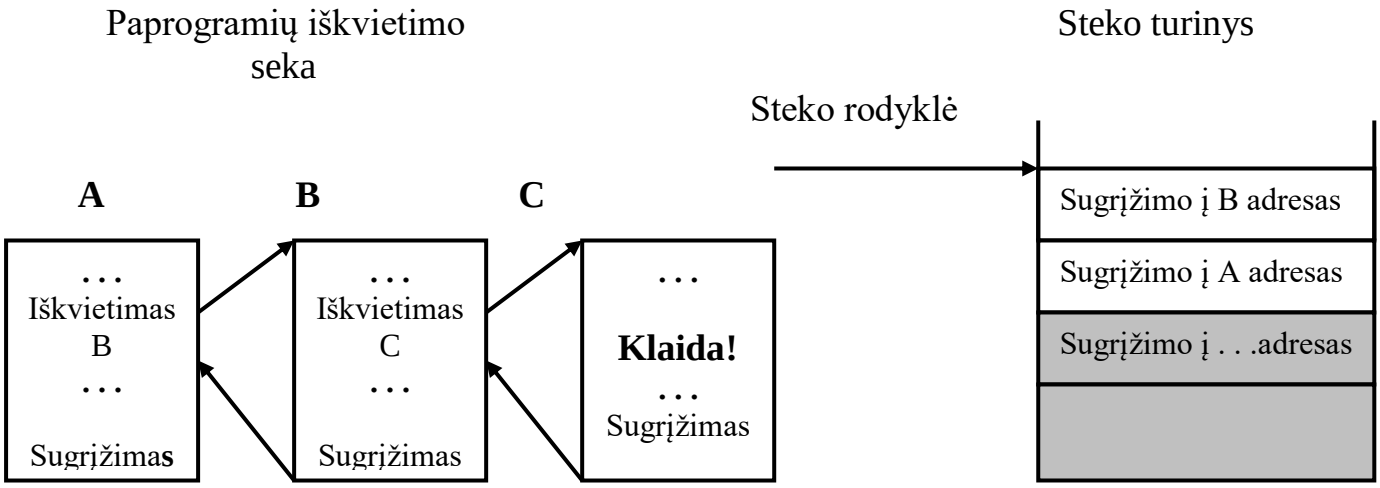
3. Kritinių situacijų apdorojimas programoje naudojant programinį **išimčių mechanizmą** (t. y., specializuotas valdymo konstrukcijas).

Pagrindiniai kritinių situacijų apdorojimo etapai:

- informacijos apie kritine situacija generavimas (išimties generavimas),

- šios informacijos apdorojimas (išimties perėmimas).

Išimties perėmimas vyksta naudojant *iškvietimų steką*. Jame saugojami grįžimo į paprogrames adresai (žr. 11.1. pav.)



Pav. 11.1. Iškvietimų steko organizavimas.

Klaida aptikta paprogramėje C gali būti apdorojama vienoje iš paprogramių (A, B ar C) arba kitose paprogramėse, iš kurių buvo iškviesta A. Tam, vienoje iš jų turi būti realizuotas išimčių apdorojimo kodas. Jei jis nebus aptiktas, programa pasibaigs avariniu būdu.

<u>Išimčių mechanizmo programavimo konstrukcijos</u>.

- Išimčių apdorojimo kodo paieška ir vykdymas**.**

Delphi:

```
try
   {tikrinami sakiniai}
except
   {sakiniai realizuojantys reakcija į kritinę situaciją, jei ji įvyksta}
[else]
   {sakiniai atliekami, kai tikrinamos kritinės situacijos nėra}
end;
```

C++ **// SEH**

```
try ...
catch,
```

C – **_try ..._except.** **//EH**

- Išteklių (resursų) išlaisvinimas – užbaigimo konstrukcija.

Delphi:

```
try
   {tikrinami sakiniai}
finally
   {Sakiniai apdorojantys išimtis. Vykdomi bet kuriuo atveju, t.y. ir kai kritinė situacija įvyksta ir kai neįvyksta}
end;
```

C++ **: try ... finally**
C – **_try ..._finally**

- Išimčių generavimas (objekto ar kintamojo, kuriame saugoma informacija apie kritinę situaciją kūrimas).

C++ **– throw**
C – **RaiseException**

Delphi konstrukcija – **raise**

Pavyzdys 11.1. Kritinės situacijos apdorojimo kodo fragmentas *Delphi* programoje. Standartinių kritinių situacijų vardų panaudojimas trikampio ploto skaičiavimo programoje:

```
try
   a:= StrToFloat (Edit1.Text);
   b:= StrToFloat (Edit2.Text);
   c:= StrToFloat (Edit3.Text);
   p:= (a+b+c)/2.0;
   plotas:= sqrt(p*(p-a)*(p-b)*(p-c));
except
   on EConvertError do begin
                        ShowMessage("Įvedimo klaida");
                        Label6.Caption:='-'; Exit;
                        end;
   on EMathError do begin
                    ShowMessage("Negalima sudaryti trikampio");
                    Label6.Caption:='-'; Exit;
                       end;
end;
```

C++ KALBOS KLAIDŲ APDOROJIMO YPATUMAI.

Klaidų šaltiniai

Pirmas klaidų šaltinis – programavimo (algoritmavimo) klaidos. Čia reikėtų pabrėžti, kad dėl žemiau išvardintų naudingų ir tikslingų C++ kalbos savybių dauguma kompiliatorių vykdo gana griežtą programos kontrolę kompiliavimo metu:

- tipų kontrolė ir neleistinų operacijų aptikimas;
- visų vardų skelbimas prieš šių vardų naudojimą, funkcijų argumentų kontrolė;
- vardų kontekstai, matomumo apribojimai;
- objektinio programavimo mechanizmai leidžia derinti atskiras klases ir užtikrinti programos kūrimą iš jau patikrintų klasių bei daug kartų naudoti patikimus kodo fragmentus.

Nežiūrint į tai, programuotojui patartina (bent jau programos derinimo metu) vykdyti duomenų kontrolę ir pačiam tikrinti potencialiai pavojingus kodo fragmentus, pvz., tokius kaip rodyklių perdavimą parametrais (ar ne NULL), masyvo indeksų naudojimą (rėžių tikrinimą) ir kt.

Antras klaidų šaltinis – taip vadinamos „planinės klaidos“. Tai vartotojo veiksmų patikrinimas įvedant informaciją, apdorojamo teksto sintaksės klaidos, komunikacijų seansų klaidos, ryšių kanalų triukšmai ir pan. Nors tai ne programuotojo klaidos, bet jis (ji) turi planuoti galimą tokių klaidų atsiradimą. Gerai sudaryta programa turėtų tęsti darbą toliau ir esant planinėms klaidoms.

Trečias klaidų šaltinis – taip pat turi būti numatytas. Tai – išimtinės situacijos, kurios atsiranda ir kai programoje ir jos vartotojo veiksmuose nėra klaidų, pvz., disko ar duomenų bazės klaida, atminties nepakankamumas kuriant naujus objektus ir pan. Šiose situacijose reikia užtikrinti tarpinių rezultatų saugojimą ir tvarkingą programos užbaigimą.

Toliau panagrinėsime antrą ir trečią atvejus.

Klaidos požymių apdorojimas

Dažnai taikomas paprastas pranešimo apie įvykusią klaidą būdas – speciali funkcijos (metodo) gražinama reikšmė. Pvz., objekto saugojimo

duomenų bazėje funkcija gal gražinti loginę reikšmę: *true*, kai objektas sėkmingai išsaugotas ir *false* – priešingu atveju:

```
class Database
{
     public:
        bool SaveObject (const Object& ob);
};
```

Čia aprašyto metodo iškvietimas turėtų atrodyti taip:

```
if (Database.SaveObject(mano_objektas) == false)
{
   // klaidų apdorojimas
}
```

Kai funkcijos gražinamas rezultatas yra reikšmė, jos neleistinas dydis naudojamas kaip klaidos požymis, pvz., funkcija *abs(x)* gražina –1, arba gražinama nulinė rodyklė. Tačiau, pvz., konstruktoriai negražina jokių reikšmių. Tokiu atveju galima naudoti sukuriamų objektų būsenos vėliavėles (požymius). Jei minėtos *Database* klasės konstruktorius turi susijungti su DB serveriu, galima šį susijungimą patikrinti taip:

```
class Database
{
     public:
        Database(const char* serverName);
        . . .
        bool Ok (void) const { return  okFlag; };
     private:
        bool  okFlag;
};

Database :: Database (const char* serverName);
{
     if (connect(serverName) == true)
        okFlag = true;
     else
        okFlag = false;
};
void main ()
{
    Database  database("dbserver");
    if (!database.Ok()) {
        cerr  << "Ryšio su DB serveriu klaida"  <<  endl;
        return;
```

```
    }
}
```

Kitas galimas variantas – vietoje metodo *Ok* naudojimo perkrauti operaciją **!** :

```
class Database
{
    public:
      bool operator! () const { return  ! okFlag; };
};
```

Čia susijungimo su DB serveriu patikrinimas būtų toks:

```
   if (!database) {
      cerr  << "Ryšio su DB serveriu klaida"  <<  endl;
```

Šiaip, patartina nenaudoti konstruktorių, kuriuose gali atsirasti klaidų. Pateiktame pavyzdyje reikėtų išskirti iš konstruktoriaus susijungimo su DB serveriu metodą *Open*:

```
class Database
{
    public:
      Database();
      bool Open (const char* serverName);
};
```

Nors funkcijos (metodo) gražinamos reikšmės tikrinimas dažnai taikomas planinių klaidų apdorojimui, jis turi ir trūkumų:

- būtinybė perduoti klaidos požymį per funkcijų iškvietimų seką (žr. 11.1 pav.);
- yra sunkumų grąžinti klaidingą funkcijos reikšmę, kai funkcija jau grąžina reikiamą reikšmę, t. y. reikia modifikuoti sąsają ir pan.;
- papildomi ***if*** sakiniai kiek užgožia programos logiką.

Kritinių situacijų sužadinimas ir apdorojimas

Planinių klaidų ir išimtinių situacijų apdorojimui C++ siūlo naudoti specialų programinį **išimčių mechanizmą** (t. y., specializuotas valdymo konstrukcijas).

Kaip ja minėta, išimtinė situacija sužadinama (generuojama) vykdant sakinį ***throw***. Jo argumentais gali būti bet kokio standartinio tipo reikšmė arba programoje apibrėžtos klasės objektas. Kai atsiranda išimtinė situacija, esamos funkcijos vykdymas nutraukiamas, lokalūs

kintamieji naikinami ir valdymas perduodamas į ją iškvietusią funkciją, kur vėl generuojama išimtinė situacija ir t.t. iki funkcijos *main*, kol visa programa nebaigs savo darbo (žr. 11.1 pav.). Pavyzdžiui, iš pagrindinės funkcijos buvo iškviesta funkcija *access()*, kuri iškvietė metodą *Open()*, kuris savo ruožtu sugeneravo kritinę situaciją:

```
class Database
{
     public:
       void Open(const char* serverName);
};

void Database :: Open (const char* serverName);
{
     if (connect(serverName) == false)
       throw 2;
};

access()
{
    Database  database;
    Database.Open (“dbserver“);
    String y;
        . . .
}

main ()
{
        String x;
        access();
 }
```

Tokiu atveju, valdymas perduodamas funkcijai *access()*, kur bus iškviestas objekto *database* destruktorius, toliau valdymas perduodamas funkcijai *main()*, kuri iškviečia objekto *x* destruktorių ir užbaigia programą. Čia išimtinė situacija suteikė galimybę užbaigti programą, išvalant jos kintamuosius, tačiau pati išimtinė situacija beveik nekaip neapdorojama.

Žymiai dažniau **išimtines situacijas reikia apdoroti**. Tam naudojama valdymo konstrukcija

```
try {
    . . .
} catch (išimtinės situacijos tipas) {
    . . .
}
```

Jei bloke ***try*** įvyko išimtinė situacija, ji perduodama į bloką ***catch***. Išimtinės situacijos tipas – tai ***throw*** argumento tipas. Jei jis sutampa su ***catch*** išimtinės situacijos tipu, šis blokas yra įvykdomas, jei ne – valdymas perduodamas aukščiau, kaip jau buvo aprašyta (žr. 11.1 pav.), ir kur gali būti apdorojamas situacijos tipą atitinkančiu bloku ***catch***. Jei tokio nebus aptikta, programa baigia darbą.

Blokas ***try*** dar vadinamas išimties inspekcijos bloku (angl. *exception inspection*), o blokas ***catch*** – išimties apdorojimo bloku (angl. *exception handler*).

Pavyzdžiui, dabar funkcijos *access()* blokas ***catch*** apdoroja išimtinę situaciją:

```
access()
{
    Database  database;
    int attemptCount = 0;

    again:
    try {
        Database.Open ("dbserver");
    } catch (int& ex) {
        cerr  << "Ryšio su DB serveriu klaidos numeris"<< x  <<  endl;
        if (++attemptCount) < 5)
        goto again;
        throw;
     }
    String y;
        . . .
}
```

Į bloką ***catch,*** kuris apdoroja išimtinę situaciją, perduodama nuoroda į ***throw*** argumentą. Apdorojimo metu galima arba vėl sužadinti tą pačią

išimtinę situaciją (naudojant ***throw*** be parametrų), arba kitą, arba nežadinti jokios. Pastaruoju atveju, manoma, kad išimtis apdorota ir programos vykdymas tęsiamas toliau po bloko ***catch.***

Su vienu bloku ***try*** gali būti surišti keli apdorojimo blokai ***catch***. Tokiu atveju išimtinės situacijos tipas bus palygintas su kiekvieno ***catch*** bloko argumento tipu, kol nebus rastas suderinamumas, tik suderintas apdorojimo blokas ir bus vykdomas. Taip pat yra specialaus pavidalo apdorojimo blokas ***catch(. . .)***, kuris suderintas su bet kuriu išimtinės situacijos tipu, tačiau į tokį bloką negalima perduoti argumentų.

Kritinių situacijų apdorojimas

Tikslingam minėtų konstrukcijų naudojimui programuotojai dažnai kuria išimtinių situacijų klasę. Įprasta tokios klasės paskirtis – klaidų kodų saugojimas. Pvz.:

```
class Exception
{
    public:
      enum ErrorCode {
            NO_MEMORY;
            DATABASE_ERROR;
            INTERNAL_ERROR;
            ILLEGAL_VALUE;
      };
      Exception (ErrorCode errorKind, const String& errMessage);
      ErrorCode GetErrorKind(void) const  { return kind; };
      const String& GetErrorMessage(void) const  { return msg; };
    private:
      ErrorCode kind;
      String msg;
};
```

Išimtinė situacija gali būti sužadinama taip:

```
    if (connect(serverName) == false)
      throw Exception (Exception::DATABASE_ERROR, serverName);
```

o jos tikrinimas:

```
  try {
      . . .
  } catch (Exception& e) {
      cerr  << "Įvyko klaida "<< e.GetErrorKind( ) <<  endl;
            << "Papildoma informacija "<< e. GetErrorMessage( );
```

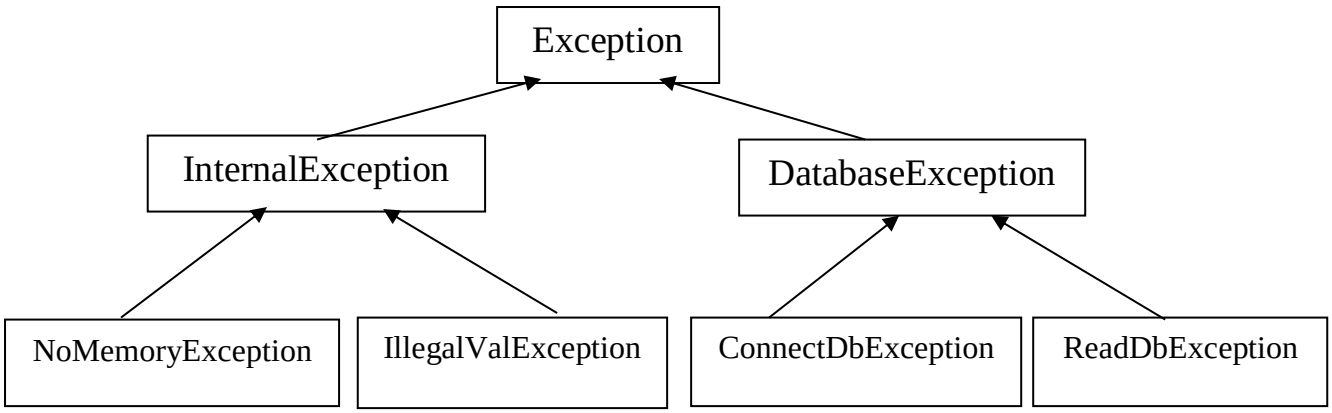
```
}
```

Lyginant su sveikojo tipo parametrų išimtinių situacijų klasė leidžia perduoti papildomą informaciją ir apdorojimo bloke reaguoti tik į tam tikro apibrėžto pavidalo klaidas, jei čia būtų sugeneruota kita išimtinė situacija, ***catch*** blokas būtų praleistas, jis reaguoja tik į *Exception* tipo situacijas. Tai ypač patogu jungiant programas ir bibliotekas, tokiu atveju kiekvienas klasių rinkinys atsako tik už savo klaidas.

Pateiktame pavyzdyje klaidos kodas saugomas klasės *Exception* objekte. Jei bloke ***catch*** laukiama kelių išimtinių situacijų, galima analizuoti klaidos kodą naudojant sąlyginius sakinius, pvz.:

```
try {
    . . .
} catch (Exception& e) {
    cerr  << "Įvyko klaida "<< e.GetErrorKind( ) <<  endl;
          << "Papildoma informacija "<< e. GetErrorMessage( );
  if (e.GetErrorKind( ) == Exception:: NO_MEMORY ||
    e.GetErrorKind( ) == Exception:: INTERNAL_ERROR)
    throw;
 else if    // ... ir t.t.
 }
```

Kitas būdas – sukurti klasių hierarchiją, kiekvienai išimtinei situacijai skiriant atskirą klasę:



Čia apdorojimas gali būti pateiktas taip:

```
try {
} catch (ConnectDbException& e) {
    // Ryšio su DB serveriu klaidos apdorojimas
```

```
    } catch (ReadDbException& e) {
        // Skaitymo iš DB klaidos apdorojimas
    } catch (DatabaseException& e) {
        // Kitų DB klaidų apdorojimas
    } catch (NoMemoryException& e) {
        // Atminties nepakankamumo klaidos apdorojimas
    } catch (. . .) {
        // Visų kitų išimtinių situacijų apdorojimas
{
```

Svarbu pabrėžti, kad blokų ***catch*** argumentų tikrinimas vyksta nuosekliai. Todėl, pvz., bloką klasei *DatabaseException* negalima rašyti prieš *ConnectDbException.* Pastaroji išimtinė situacija yra suderinta su klase *DatabaseException*, nes tai jos bazinė klasė. Todėl ji bus apdorojama bloku ***catch*** *(DatabaseException& e),* o blokas ***catch*** *(ConnectDbException& e)* nebus įvykdytas.

Klasių hierarchijos sukūrimas yra imlus darbui procesas, tačiau manoma, kad tokios programos yra lankstesnės. Apdorojant išimtines situacijas patartina atsižvelgti į tokias ypatybes:

- tam, kad būtų palengvintas klaidų apdorojimas programoje, galima papildyti funkcijų ir metodų aprašus papildoma informacija apie tai, kokio tipo išimtines situacijas jos gali generuoti, pvz.:

```
class Database
{
    public:
void Open(const char* serverName)  throw ConnectDbException;
};
```

- kai tam tikroje vietoje generuojama išimtine situacija, likęs funkcijos kodas nėra vykdomas, todėl prieš ***throw*** naudojimą reikia išlaisvinti funkcijos naudojamus resursus (t. y. atitinkamam ***new*** panaudoti ***delete***);
- reikia vengti išimtinių situacijų naudojimo destruktoriuose, nes destruktorius jau prieš tai gali būti iškviestas apdorojant jau įvykusią išimtį (išlaisvinant iškvietimų steką, kaip parodyta 11.1 pav.);
- kai išimtine situacija atsiranda konstruktoriuje, manoma, kad atitinkamas objektas nėra pilnai sukurtas ir destruktorius jam nebus kviečiamas.

Kritinių situacijų klasės standartinėje C++ bibliotekoje.

Šios klasės aptarnauja pačią biblioteką ir suteikia programuotojui galimybę jų pagrindu kurti savo išvestines klaidų apdorojimo klases.

Standartinėje C++ bibliotekoje yra pagrindinė klasė, specialiai sukurta deklaruoti išimčių objektus. Ji vadinama *std::exception* ir yra apibrėžta antraštiniame faile <exception>, kuris turėtų būti įtrauktas į programą.

Bibliotekoje visos išimtinės situacijos, kurias sukelia jos komponentai, išvedamos iš bazinės klasės *exception* :

```
class exception
{
    public:
      exception (); throw();
      exception (const exception&); throw();
      exception& operator= (const exception&); throw();
      virtual  ~exception (); throw();
      virtual const char* what(); const  throw();
    private:
      . . .
};
```

Šioje klasėje apibrėžtas virtualus metodas *what()*, kuris grąžina ASCIIZ pranešimą (t. y. null-terminuotą teksto eilutę). Šį metodą galima perrašyti išvestinėse klasėse, kad būtų galima suformuoti tam tikrą informacija apie įvykusią išimtį.

Pavyzdys 11.2. Standartinės *exception* klasės naudojimas su *what()* metodu.

```
#include <iostream>
#include <exception>
using namespace std;

class thisexception: public exception
{
        virtual const char* what() const throw() {
                return "This exception occurred.";
        }
} this_ex;

int main () {
        try  {
                throw this_ex;
        }
        catch (exception& e)  {
```

```
            cout << e.what() << '\n';
        }
        return 0;
}
```

Čia apdorojimo blokas ***catch*** gauna išimties objektą pagal nuorodą, todėl jis gali perimti ir klases, kurios yra išvestas iš *exception* klasės. Šioje programoje tai yra *this_ex* objektas, kuris yra klasės *thisexception*, išvestinės iš *exception* klasės, objektas. Tad, šios programos rezultatas yra:

This exception occurred.

Yra 3 grupės išimtinių situacijų: loginės klaidos (*logic_error*), vykdomo klaidos (*runtime_error*) ir kitos klaidos. Galimos išimtinių situacijų priežastys pateiktos lentelėje:

Išimtinės situacijos vardas	Galimos priežastys
bad_alloc	Atminties skirstymo klaida, sugeneruota *new* operatoriumi
bad_cast	Operacijos *dynamic_cast* klaida
bad_typeid	Neteisingas tipas operacijoje *typeid*
bad_exception	Neegzistuojančios išimtinės situacijos nurodymas operatoriuje *throw*
bad_function_call	Išmeta tuščias objektas-funkcija
bad_weak_ptr	Išmetama *shared_ptr* , kai perduodama bloga rodyklė *weak_ptr*
domain_error	Argumentas arba reikšmė neatitinka tipo reikšmių intervalo
invalid_argument	Neteisingas argumentas
ios_base_failure	Įvedimo-išvedimo operacijos klaida
length_error	Neteisingas ilgis
out_of_range	Neleistina indekso reikšmė
overflow_error	Aritmetikos klaida, perpildymas

range_error	Neleistinas intervalas
underflow_error	Aritmetikos klaida, reikšminių skaitmenų išnykimas

Iš viso apibrėžta 12 klasių, kurių pagrindu programuotojas gali kurti savo išvestines klaidų apdorojimo klases.