Obrada izuzetaka

Izuzetak

- U najvećem broju slučajeva će se program izvršavati u skladu sa implementiranim algoritmom
- U nekim izuzetnim situacijama program “odstupa” od “best case” scenarija izvršavanja tj. dešavaju se događaji koje nazivamo **izuzecima** (exceptions)
- Izuzetke treba obraditi na poseban način tj. izvan osnovnog toka programa
- **Primer:** U nekom izrazu se za neku kombinaciju vrednosti članova izraza dobija deljenje nulom što neizostavno dovodi do run-time greške i pada programa ili pokušaj čitanja podataka iz oštećene datoteke itd.
- Kako izbeći pojavu run-time grešaka i “pad” programa usled pojave izuzetaka?
- Šta ako program upravlja radom nuklearne elektrane???

Jezici koji ne podržavaju obradu izuzetaka (1)

Ako jezik ne podržava obradu izuzetaka, dolazi do sledećih problema:

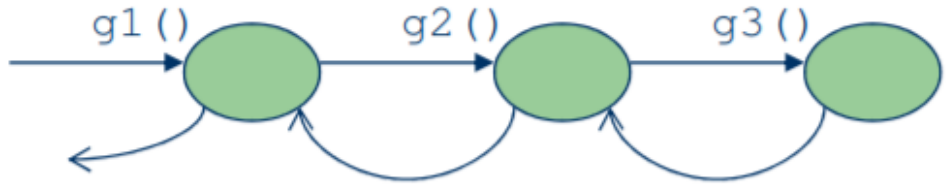
- posle izvršenja dela programa (ili poziva funkcije) u kojem može doći do greške vrši se testiranje statusa, te se obrada greške smešta u jednu granu a obrada uobičajene situacije u drugu granu if naredbe; - tkz. “**migracija u desno**”
- ako imamo više hijerarhijskih poziva funkcija, ukoliko grešku treba propagirati prema višem nivou, svaki nivo pozivanja treba da izvrši testiranje da li je došlo do greške u nižim nivoima i pomoću return vrati kod greške. – tkz. “**propagacija unazad**”

Jezici koji ne podržavaju obradu izuzetaka (2)

- Problem "migracije udesno"

```
if (f1()) {       // status==OK
                  // dalja obrada u slucaju pozitivnog ishoda f1()
  if ((f2()) {    // status==OK
                  // dalja obrada u slucaju pozitivnog ishoda f2()
    if ((f3()) {// status==OK
                  // dalja obrada u slucaju pozitivnog ishoda f3()
    } else {      // obrada greske koju je vratila f3()
    }
  } else {        // obrada greske koju je vratila f2()
  }
} else {          // obrada greske koju je vratila f1()
}
```

- Problem "propagacije unazad"

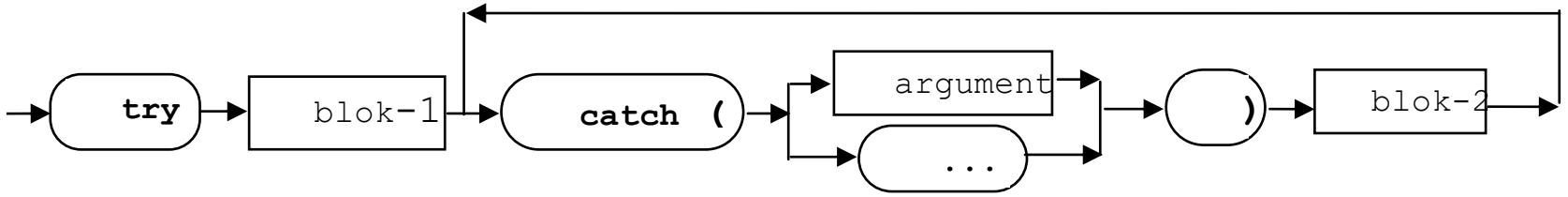


C++ - izuzeci

- C++ ima mehanizam za efikasnu obradu izuzetaka izvan osnovnog toka kontrole.
- U nekim situacijama se tokom izvršenja programa javljaju izuzeci:
  - izuzetak se kreira (“baca”) pomoću operatora **throw**;
  - Izuzetak može biti objekat klase ili nekog drugog tipa
- Pojavom izuzetka obrada se na tom mestu prekida i nastavlja u rutini (**hendleru -** rukovalac) za obradu izuzetka (catch);
  - izuzetak se smatra objektom klase (tipa) koji se dostavlja hendleru kao argument;
  - za svaki tip izuzetka se definiše poseban hendler.
- Nakon uspešne obrade detektovanog izuzetka obrada se nastavlja regularnim tokom
- Ako ne postoji odgovarajući handler izuzetak se automatski propagira unazad

C++ mehanizam za Obradu izuzetaka

- Sintaksa je sledeća:



- `blok-1` je programski blok unutar kojeg mogu da se jave izuzeci.
- `argument` se sastoji od oznake tipa i identifikatora argumenta.
- `...` označavaju univerzalni hendler – on se aktivira ako ne postoji hendler sa adekvatnim tipom izuzetka.
- `blok-2` je telo hendlera.
- Definicija hendlera liči na definiciju funkcije sa tačno jednim argumentom.
- Kaže se da je hendler tipa T ako je njegov `argument` tipa T, odnosno ako obrađuje izuzetke tipa T.
- `blok-2` obrađuje izuzetke koji su se javili neposredno u `blok-1` ili u nekoj funkciji koja je pozvana iz `blok-1`.
- Nakon izvršenja `blok-2` kontrola se ne vraća na mesto gde se pojavio izuzetak.
- Unutar `blok-1` ili u pozvanim funkcijama iz `blok-1` mogu da se pojave ugnježdene naredbe `try`.
- Ako se u `blok-1` ne javi izuzetak, preskaču se svi hendleri (kontrola se prenosi na kraj naredbe `try`).

Primer 1:

```
try {
   ...
   nekaFunkcija(); // i u funkciji može da se javi izuzetak
   ...
}

catch(const char *zn){// Obrada izuzetka tipa znakovnog niza}

catch(const int i){// Obrada izuzetka celobrojnog tipa}

catch(const double d){// Obrada izuzetka tipa double}

catch(Student s){// Obrada izuzetka tipa klase Student}

catch(...){//Obrada izuzetka proizvoljnog tipa koji nije
jedan od prethodnih}

// kod nakon poslednjeg hendlera
...
```

Izazivanje izuzetaka

- Izazivanje (prijavljivanje, bacanje) izuzetaka se vrši naredbom:

```
throw izraz
```

gde **izraz** svojim tipom određuje koji hendler će biti aktiviran;

- vrednost izraza se prenosi hendleru kao argument.

- Izuzetak se može izazvati iz try bloka ili iz bilo koje funkcije

direktno ili indirektno pozvane iz bloka naredbe **`try`**.

- Funkcije iz kojih se izaziva izuzetak mogu biti i:
  - članice klasa,
  - operatorske funkcije,
  - konstruktori,
  - destruktori.

Primer 2:

```
try {
  int godine = 15;
  if (godine >= 18) {
    cout << “Pristup dozvoljen – Imate dovoljno godina";
  } else {
    throw (godine);
  }
}
catch (int mojeGodine) {
  cout << “Pristup odbijen – Vi morate da imate najmanje 18
            godina\n";
  cout << “Vaše godine: " << mojeGodine;
}
```

Primer

```
int main()
{
  int x = -1;

  // neki kod
  cout << „Pre try \n";
  try {
    cout << „Unutar try \n";
    if (x < 0)
    {
      throw x;
      cout << „Nakon throw (Nikada se ne izvršava) \n";
    }
  }
  catch (int x ) {
    cout << „izuzetak obrađen \n";
  }

  cout << „Nakon catch (biće izvršen) \n";
  return 0;
}
```

```
Pre try
Unutar try
Izuzetak obrađen
Nakon catch (biće izvršen)
```

Primer

```
int main()
{
  try  {
    throw 'a';
  }
  catch (int x)  {
    cout << „Obrada int izuzetka" << x;
  }
  catch (...)  {
    cout << „Obrada default izuzetka\n";
  }
  return 0;
}
```

Izlaz: **Nije odrađena implicitna konverzija tipa (char u int)**

```
Obrada default izuzetka
```

Primer

```
int main()
{
  try  {
    throw 'a';
  }
  catch (int x)  {
     cout << “Obrađen izuzetak ";
  }
  return 0;
}
```

Izlaz:

Poziva se sistemska funkcija **terminate()** jer ne postoji handler za tip char i kreirani izuzetak nije obrađen

`throw` klauzula i funkcije

- U deklaraciji ili definiciji funkcija može da se navede spisak tipova izuzetaka koje funkcija izaziva (C++ 11 – ne preporučuje).
- Navođenje spiska tipova izuzetaka se postiže pomoću `throw(niz_identifikatora)` iza liste argumenata.
- Ako se stavi ovakva konstrukcija (**`dynamic exception specifications`**), a funkcija izazove izuzetak tipa koji nije u nizu identifikatora – to je greška tj. nepredvidivo ponašanje.
- Ako se ništa ne navede, funkcija sme da prijavi izuzetak proizvoljnog tipa.

```
void radi(...)throw(char *, int){
  if(...) throw "Izuzetak!";
  if(...) throw 100;
  if(...) throw Tacka(0,0);  // GRESKA: nije naveden tip izuzetka Tacka
}
```

```
void fun() throw (iz1,  iz2) {
…
}
```

ili

```
void f() {
      try {
            …
      }
      catch (iz1) {throw;}
      catch (iz2) {throw;}
      catch (…) {unexpected();}
}
```

Šta izaziva throw()?

Primer

```
// Deklaracija funkcije void fun(int *ptr, int x) je u redu za kompajler
// ali se preporučuje:

void fun(int *ptr, int x)
{
  if (ptr == NULL)
    throw ptr;
  if (x == 0)
    throw x;
  /* neka funkcionalnost */
}

int main()
{
  try {
    fun(NULL, 0);
  }
  catch(...) {
    cout << „Obrada izuzetka iz fun()";
  }
  return 0;
}
```

Izlaz: Obrada izuzetka iz fun()

Primer

```
// Deklaracija funkcije je u redu za kompajler ali se preporučuje:
// void fun(int *ptr, int x) throw (int *, int)

void fun(int *ptr, int x) throw (int*, int)
{
  if (ptr == NULL)
    throw ptr;
  if (x == 0)
    throw x;
  /* neka funkcionalnost */
}

int main()
{
  try {
    fun(NULL, 0);
  }
  catch(...) {
    cout << „Obrada izuzetka iz fun()";
  }
  return 0;
}
```

Izlaz: Obrada izuzetka iz fun()

noexcept (od verzije C++ 11)

- noexcept (kao alternativa za **throw()** u C++11, u C++17 izbačeno) i to kao:
  - **Specifikator** (`noexcept(izraz)`)
    - Navodi se iza liste parametara
    - Izraz – konstantan logički, izvršava se u toku prevođenja ako ima vrednost true – funkcija ne baca izuzetke
    - `noexcept` ili `noexcept(true)` ili `throw()`
  - **Operator** (`noexcept(izraz)`)
    - Rezultat je logičkog tipa (tačno - ako ne može da baci izuzetak)
    - Proverava se da li bi moglo doći do izuzetka u izrazu
    - **izraz** –
      - proizvoljnog tipa čak i void
      - Ne izračunava se, samo se proverava u fazi prevođenja

Primer: `noexcept` (specifikator i operator)

specifikator

```
void   f() noexcept {}                    // ne baca
int    g() noexcept(false){return 0;} // moze da baca
double h() {return 0.0;}                  // moze da baca
int    i() throw(){return 0;}             // ne baca
void   j() throw (int, double){}          // moze da baca
```

operator

```
//    int i double
bool p=noexcept(f());                     // true
bool q=noexcept(g());                     // false
bool r=noexcept(h());                     // false
bool s=noexcept(i());                     // true
bool t=noexcept(j());                     // false
bool v=noexcept(new int);                 // false
```

`noexcept` gde (ne)koristiti

- Koristi se za funkcije, metode klase, lambda funkcije i pointere na funkcije
- Od C++17 pointeri na funkcije sa `noexcept` ne mogu da ukazuju na funkcije koje potencijalno mogu da izazovu izuzetak
- Ne koristiti za virtuelne funkcije u osnovnoj klasi jer su time ograničene i u svim izvedenim klasama
- Virtuelna metoda u izvedenoj klasi ne sme da proširi listu izuzetaka u `throw()` specifikatoru ali sme da je suzi
- Ako se ipak pojavi izuzetak u funkciji koja je označena kao `noexcept` onda se pokreće `terminate` koja poziva `abort` tj. završava se program nasilno.

Ugnježđeni try blokovi

- Za (dinamički) ugnježđene naredbe **`try`**, hendler unutrašnje naredbe može da izazove izuzetak.
- Takav izuzetak se prosleđuje hendleru spoljašnje naredbe **`try`**.
- Unutar hendlera ugnježđene naredbe `try` izuzetak može da se izazove i pomoću naredbe **`throw`** bez izraza.
- Takav izuzetak ima tip hendlera u kojem je izazvan.

Primer: Ugnježdeni try-catch

```
void main()  {
        try {
                try {
                        throw 20;
                }
                catch (int n) {
                        cout << „Obrada unutrašnja ";
                        throw; // ponovno bacanje istog int izuzetka
                }
        }
        catch (int n) {
                cout << „Obrada spoljašnja ";
        }
}
```

Uništavanje lokalnih objekata

- Kada se kontrola predaje hendleru definitivno se napušta blok u kome je kreiran izuzetak i tada se uništavaju svi lokalni objekti kao i objekti u ugnježdenim blokovima
- Izuzetak kreiran u konstruktoru – uništavaju se prethodno kreirani atributi i nasleđeni podobjekti
- Nije dobro da rezultat izraza throw pokazuje na **lokalni objekat** jer će se taj objekat uništiti pre prihvatanja u catch bloku

try funkcijska naredba

- try blok obuhvata celo telo funkcije

```
tip funkcija(parametri) try { telo funkcije }
catch (param1)  {  hendler 1 }
catch (param2)  {  hendler 2 }
```

- Modifikatori metode (npr. `const`) i `throw` klauzula se pišu ispred `try`
- U hendlerima
  - Mogu da se koriste parametri funkcije
  - Ne mogu da se koriste lokalne promenljive
  - Ako funkcija nije tipa `void` mora da se izvrši `return` ako se ne kreira izuzetak

Primer: funkcijska naredba try

```
int f(int x) throw(double) try {
  int y=0;
  if (…) throw 1;      // baca se i obradjuje
  if (…) throw 2.0;    // baca se i propagira dalje
  …
  return x+y;    // regularan rezultat funkcije
} catch (int g) {
  int a=x;       // u redu
  int b=y;       // ! GRESKA
  return -1;     // rezultat u izuzetnoj situaciji
}
```

Funkcijski `try` u konstruktoru

- Omogućava hvatanje izuzetaka koji se bacaju iz
  - inicijalizatora atributa primitivnog tipa
  - konstruktora atributa klasnog tipa
  - konstruktora osnovnih klasa
- Lista inicijalizatora u definiciji konstruktora se piše iza `try`
- Ako se iz rukovaoca pristupa atributima ili nasleđenom podobjektu
  - posledice su nepredvidive
  - neki još nisu inicijalizovani, a one klasne koji su već bili konstruisani odgovarajući destruktor je uništio pre izvršenja rukovaoca
- Dolazak do kraja rukovaoca izaziva ponovno bacanje izuzetka
  - kao da je rukovalac završen naredbom `throw;` (bez operanda)
  - konstruktor može regularno da se završi samo ako uspešno stvori objekat
  - rukovalac može i da se završi pozivom funkcije `exit(int)`
  - u konstruktoru sa funkcijskim `try` nije dozvoljen `noexcept` ili `throw()`

Primer – funkcijski `try` u konstruktoru

```
class A {
public:
  A(int x) {… if (…) throw 'x';…}
  A(char x) {… if (…) throw 2;…}
}
class B {
  A a1=A(3);
  A a2;
public:
  B() try : a2('a'){      // izuzetak tipa int
      … if(…) throw 4.0; // izuzetak tipa double
  } catch (double g) {   // rukovalac za tip double
    A a3(a1);            // ! GRESKA - a1 unisten
  } catch (char g) {     // rukovalac za tip char
    …
  }                      // nije obradjen tip int
}
```

Prihvatanje izuzetaka

- Hendler tipa B može da prihvati izuzetak tipa D ako:
  - B i D su istih tipova
  - B je javna osnovna klasa za izvedenu klasu D
  - B i D su pokazivački tipovi i D može da se standardnom konverzijom konvertuje u tip B.
- Na mestu izazivanja izuzetka formira se privremeni objekat sa vrednošću izraza.
- Privremeni objekat se prosleđuje najbližem (prvom na koji se naiđe) hendleru.
- Ako su `try` naredbe ugnježdene, izuzetak se obrađuje u prvom odgovarajućem hendleru tekuće naredbe `try`.
- Ako se ne pronađe odgovarajući hendler - izuzetak se prosleđuje hendleru sledećeg (višeg) nivoa naredbe `try`.
- Prilikom navođenja hendlera treba se držati sledećih pravila:
  - hendlere tipa izvedenog iz neke osnovne klase treba stavljati ispred hendlera tipa te osnovne klase
  - univerzalni hendler treba stavljati na poslednje mesto.

- Predaja kontrole hendleru podrazumeva definitivno napuštanje bloka u kojem se dogodio izuzetak.
- Napuštanje bloka podrazumeva uništavanje svih lokalnih objekata u tom bloku i ugnježdenim blokovima.

- Objekat koji se pojavi kao operand naredbe `throw` se uništava prvi, ali se zato prethodno kopira u privremeni.
- Privremeni objekat će se uništiti tek po napuštanju tela hendlera.
- Ako se izuzetak iz hendlera H1 prosleđuje hendleru višeg nivoa H2, privremeni objekat živi do kraja hendlera H2.
- Nije dobro da tip izuzetka bude pokazivački tip, jer će se pokazivani objekat uništiti pre dohvatanja iz hendlera.

- Kada se baci izuzetak svi objekti kreirani u try bloku se uništavaju pre nego što se kontrola preda catch bloku

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

class Test {
public:
  Test() { cout << „Konstruktor za Test " << endl; }
  ~Test() { cout << "Destruktor za Test " << endl; }
};

int main()
{
  try {
    Test t1;
    throw 10;
    ... // neke naredbe
  }
  catch (int i) {
    cout << „Obrada izuzetka" << i << endl;
  }
}
```

```
Konstruktor za Test
Destruktor za Test
Obrada izuzetka 10
```

Neprihvaćeni izuzeci

Javljaju se:

- Ako se za neki izuzetak ne pronađe hendler koji može da ga prihvati
- kada se detektuje poremećen stek poziva
- kada se u destruktoru, u toku odmotavanja steka, postavi izuzetak

Izvršava se sistemska funkcija:

```
void terminate();
```

- Podrazumeva se da ova funkcija poziva funkciju **`abort()`** koja kontrolu vraća operativnom sistemu.
- Ovo se može promeniti pomoću funcije **`set_terminate`**.
- Njoj se dostavlja pokazivač na funkciju koju treba da pozove funkcija `terminate` umesto funkcije `abort`.
- Pokazivana funkcija mora biti bez argumenata i bez rezultata (`void`).
- Vrednost funkcije `set_terminate` je pokazivač na staru funkciju koja je bila pozivana iz `terminate`.
- Iz korisničke funkcije **`(*pf)`** treba pozvati **`exit()`** za povratak u operativni sistem.
- Pokušaj povratka sa `return` iz korisničke funkcije `(*pf)` dovešće do nasilnog prekida programa sa `abort()`.

```
typedef void (*PVF)(); // tip korisničke funkcije koja menja abort
PF set_terminate(PVF pf); // prototip funkcije set_terminate
```

Primer: set_terminate

```
void term_func() {
    cout << "term_func se poziva umesto terminate." ;
    exit( -1 );
}

int main() {
    try {
        set_terminate( term_func );
        throw "Nema više memorije!"; // Ne postoji handler
    }
    catch( int ) { cout << "Bacen izuzetak tipa int." <<
endl; }
    return 0;
}
```

IZLAZ:

term_func se poziva umesto terminate.

Neočekivani izuzeci

• Ako se u nekoj funkciji izazove izuzetak koji nije na spisku naznačenih izuzetaka, izvršava se funkcija:

```
void unexpected();
```

•Poziva se unexpected_handler()

•Podrazumeva se da ova funkcija poziva funkciju **terminate()**.

• Ovo se može promeniti pomoću funcije **set_unexpected**.

• Njoj se dostavlja pokazivač na funkciju koju treba da pozove funkcija `unexpected` umesto `terminate`.

• Pokazivana funkcija mora biti bez argumenata i bez rezultata (`void`).

• Vrednost funkcije `set_unexpected` je pokazivač na staru funkciju koja je bila pozivana iz `unexpected`.

• Pokušaj povratka sa `return` iz korisničke funkcije `(*pf)` dovešće do nasilnog prekida programa sa `abort()`.

```
typedef void (*PF) (); // tip korisničke funkcije
PF set_unexpected(PF pf); // prototip funkcije set_unexpected
```

Nije ista situacija u svim standardima C++ !!!!

Standardni izuzeci

- U biblioteci `<exception>` je klasa `exception`
- `exception` je u korenu hijerarhije svih standardnih izuzetaka
- Metode standardnih klasa i neki operatori mogu da prijave izuzetke klasa izvedenih iz klase `exception`
- Iz ovih klasa se po pravilu izvode korisničke klase za izuzetke

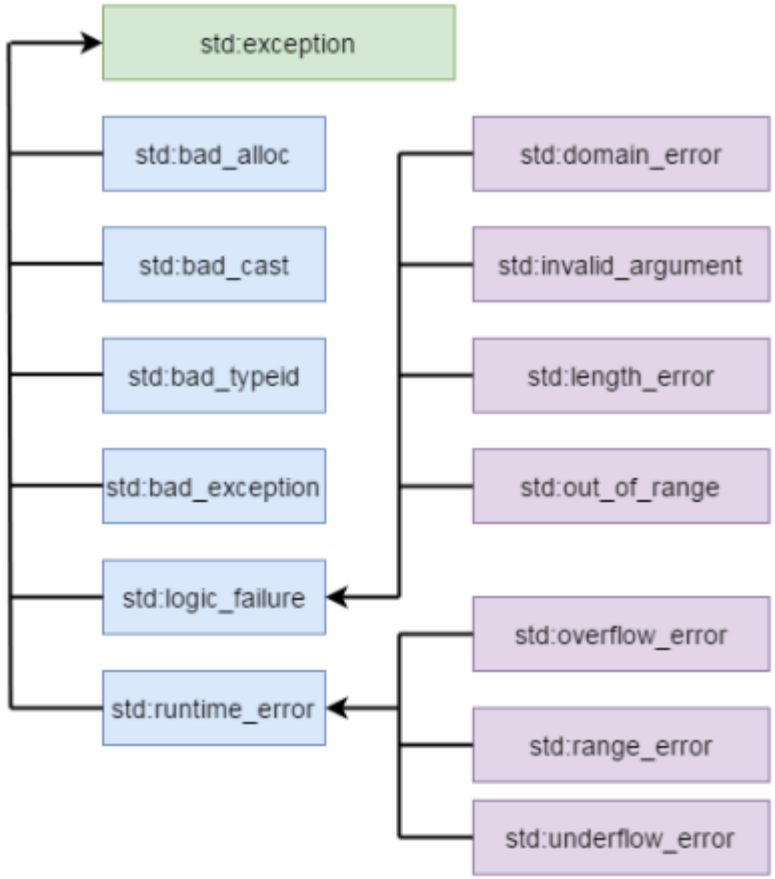
Klasa `exception`

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

- `what()` vraća pokazivač na tekstualni opis izuzetka (std. ne propisuje tekst poruka)
- Ni jedna metoda ne sme da prijavi ni jedan izuzetak:
  - obezbeđuje se sa `noexcept`
  - to automatski važi i za date metode u izvedenim klasama (lista izuzetaka se ne sme proširiti)

Hijerarhija standardnih izuzetaka

Sr.No	Exception & Description
1	**std::exception** An exception and parent class of all the standard C++ exceptions.
2	**std::bad_alloc** This can be thrown by **new**.
3	**std::bad_cast** This can be thrown by **dynamic_cast**.
4	**std::bad_exception** This is useful device to handle unexpected exceptions in a C++ program.
5	**std::bad_typeid** This can be thrown by **typeid**.
6	**std::logic_error** An exception that theoretically can be detected by reading the code.
7	**std::domain_error** This is an exception thrown when a mathematically invalid domain is used.
8	**std::invalid_argument** This is thrown due to invalid arguments.
9	**std::length_error** This is thrown when a too big std::string is created.
10	**std::out_of_range** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
11	**std::runtime_error** An exception that theoretically cannot be detected by reading the code.
12	**std::overflow_error** This is thrown if a mathematical overflow occurs.
13	**std::range_error** This is occurred when you try to store a value which is out of range.
14	**std::underflow_error** This is thrown if a mathematical underflow occurs.



Kreiranje korisničke klase za izuzetke

```
1
2   // Primer za kreiranje korisniček klase za obradu
3   // izuzetka koji nastaje pri pokušaju deljenja nulom
4   #include <iostream>
5
6   using std::cout;
7   using std::cin;
8   using std::endl;
9
10  #include <exception>
11
12  using std::exception;
13
14  // DeljenjeNulomIzuzetak objekat korisniček klase se baca u funkciji
15  // ako se detektuje pokušaj deljenja nulom u nekom izrazu
16  class DeljenjeNulomIzuzetak : public exception {
17
18  public:
19
20  // konstruktor specificira default poruku o grešci
21  DeljenjeNulomIzuzetak :: DeljenjeNulomIzuzetak()
22        : exception( “Pokušaj deljenja nulom" ) {}
23
24  };  // kraj kalse DeljenjeNulomIzuzetak
```

```
26  // izvršava deljenje i baca objekat klase DeljenjeNulomIzuzetak
27  // ako se pojavi deljenje nulom
28  double kolicnik( int deljenik, int delilac )
29  {
30     // baca objekat  DeljenjeNulomIzuzetak ako se proba deljenje nulm
31     if ( delilac == 0 )
32        throw DeljenjeNulomIzuzetak(); // završava funkciju
33
34     // vraća rezultat deljenja
35     return static_cast< double >( deljenik ) / delilac;
36
37  }  // kraj funkcije kolicnik
38
39  int main()
40  {
41     int broj1;
42     int broj2;
43     double rezultat;
44
45     cout << “Unesi dva cela broja (end-of-file to end): ";
46
```

```
47      // korisnik unosi dva cela broja za deljenje
48      while ( cin >> broj1 >> broj2 ) {
49
50         // try block sadrži kod koji može da izazove izuzetak
51         // i tada ne treba da se izvrši ako se pojavi izuzetak
52         try {
53            rezultat = kolicnik( broj1, broj2 );
54            cout << “Kolicnik je: " << rezultat << endl;
55
56         } // end try
57
58         // hendelr za izuzetak deljenje nulom
59         catch ( DeljenjeNulomIzuzetak &deljenjeNulom ) {
60            cout << “Pojavio se izuzetak: "
61                 << deljenjeNulom.what() << endl;
62
63         } // end catch
64
65         cout << "\nUnesi dva cela broja (end-of-file to end): ";
66
67      }  // end while
69      cout << endl;
71      return 0;  // normalan zavrsetak
72
73   }  // end main
```

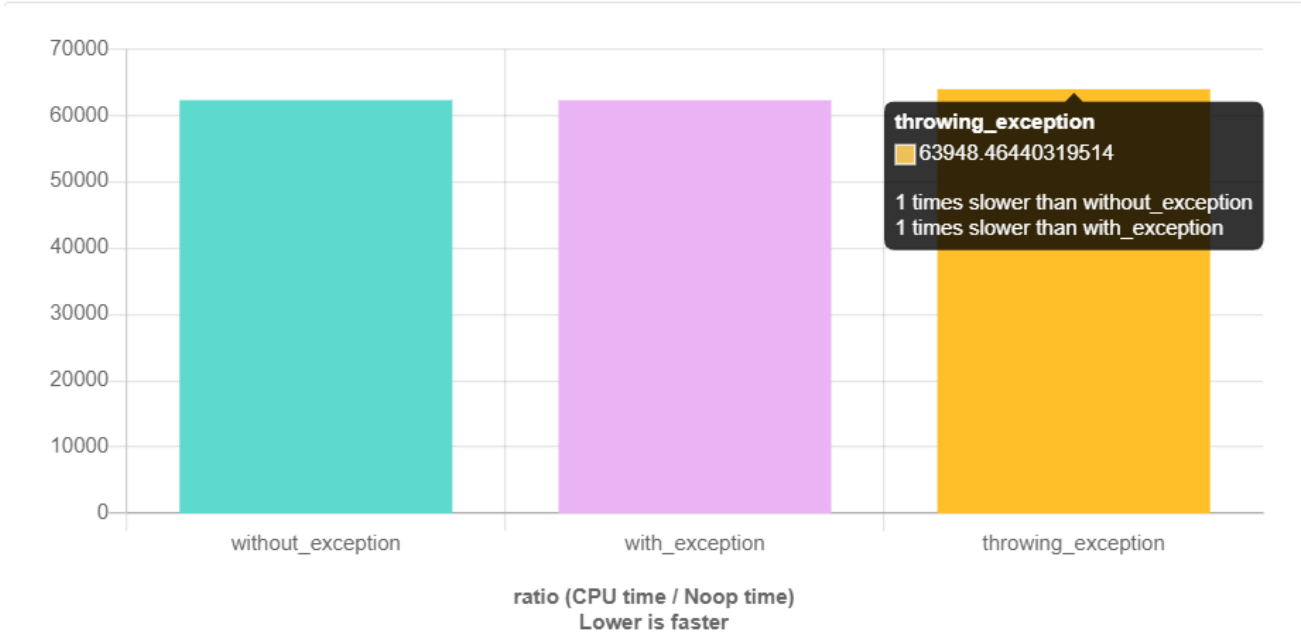
Unesi dva cela broja (end-of-file to end): 100 7
Kolicnik is: 14.2857

Unesi dva cela broja (end-of-file to end): 100 0
Pojavio se izuzetak: Pokusaj deljenja nulom

Unesi dva cela broja (end-of-file to end): ^Z

Neke greške/zablude:

- ne korišćenje izuzetaka u konstruktoru - kada se objekat ne kreira uspešno i ne baci se izuzetak on ostaje u “zombi” stanju te je ispravno koristiti izuzetak da ne bi morali da uvodimo neku promenljivu čije bi stanje trebali da proveravamo nakon svakog kreiranja objekta (da li je bilo uspešno kreiranje ili ne)
- Koja je cena postojanja mehanizma za obradu izuzetaka? Minimalna! Koristite ga!



Kod experimenta za cenu

```
static void without_exception(benchmark::State &state){
    for (auto _ : state){
        std::vector<uint32_t> v(10000);
        for (uint32_t i = 0; i < 10000; i++) v.at(i) = i;
    }
}
BENCHMARK(without_exception);//---------------------------------------

static void with_exception(benchmark::State &state){
    for (auto _ : state){
        std::vector<uint32_t> v(10000);
        for (uint32_t i = 0; i < 10000; i++){
            try{
                v.at(i) = i;
            }
            catch (const std::out_of_range &oor){}
        }
    }
}
BENCHMARK(with_exception);//------------------------------------------

static void throwing_exception(benchmark::State &state){
    for (auto _ : state){
        std::vector<uint32_t> v(10000);
        for (uint32_t i = 1; i < 10001; i++){
            try{
                v.at(i) = i;
            }
            catch (const std::out_of_range &oor){}
        }
    }
}
BENCHMARK(throwing_exception);//---------------------------------------
```

Neke preporuke - kako izbeći neke greške u korišćenju izuzetaka

- https://www.acodersjourney.com/top-15-c-exception-handling-mistakes-avoid/
- **Mistake # 1: Dismissing Exception Handling as expensive in favor of using error codes**
- **Mistake # 2: Not understanding the stack unwinding process**
- **Mistake # 3: Using exceptions for normal code-flow**
- **Mistake # 4: Not using exceptions in constructors when object creation fails**
- **Mistake # 5: Throwing exceptions in destructors or in overloaded delete or delete[] operator**
- **Mistake # 6: Not Throwing an exception by value**
- **Mistake # 7: Not catching an exception by reference or consts reference**
- **Mistake # 8: Using Exception specifications in code**
- **Mistake # 9: Not realizing the implications of "noexcept" specification**
- **Mistake # 10: Mixing Exceptions and Error codes**
- **Mistake # 11: Not Deriving Custom Exception classes from from a common base class, std::exception or one of it's subclasses**
- **Mistake # 12: Throwing exception in an exception class constructor**
- **Mistake # 13: Not understanding the difference between throw and throw e from a catch block**
- **Mistake # 14: Using setjmp and longjmp in c++ code for exception handling**
- **Mistake # 15: Swallowing Exceptions**