

# Izņēmumu apstrāde (Exception Handling)

---

"Izņēmums" ir situācija, kad objekta reprezentētā abstraktā koncepcija nav definēta.

- n Izņēmums ne vienmēr ir programmas kļūda!
- n Izņēmuma situācijas visbiežāk rodas, ja objektu lieto nekorekti, kā arī, ja objekta funkcijās nav paredzēta pietiekama datu pārbaude.
- n Piemēram:
  - § Dalīšanas funkcijā nav definēts rezultāts, ja dalītājs ir nulle;
  - § Datu struktūrai **stack** nav definēta situācija, ja izpilda operāciju **pop( )** tukšam stekam;
  - § Klases **Triangle** trīsstūra laukuma aprēķināšanas funkciju lieto negatīviem un/vai nepareiziem malu garumiem.

# Izņēmumu apstrāde (turpinājums)

n Kā apstrādāt izņēmuma situāciju?

- § Ignorēt to
- § Izvadīt kļūdas ziņojumu
- § Beigt programmu
- § Kā funkcijas vērtību atgriezt īpašu kļūdas kodu

n Klases vai funkcijas autors nevar noteikt, kādā veidā apstrādāt izņēmuma situāciju – to var tikai klases vai funkcijas lietotājs.

Autors (klases funkcija) <u>konstatē</u> izņēmumu	Lietotājs (izsaucošā funkcija) <u>apstrādā</u> izņēmumu
--	--

n Klases funkcijai un tās izsaucējam kaut kādā veidā ir jāsazinās.

# Izņēmumu apstrāde (turpinājums)

n Veidi, kā izsaucošajai funkcijai ziņot par izņēmumu:

§ Funkcijas vērtība ir pazīme par veiksmīgu vai neveiksmīgu izpildi:

```
bool myFunct1(int param1, char* param2, int& result);
```

```
if (myFunct1(5, "ABC", res) == false )
```

```
    // Funkcija nevarēja izpildīt savu uzdevumu ar tās  
    parametros norādītajām vērtībām ! Parametra result  
    vērtība šajā gadījumā nav noteikta.
```

§ Izmantot globālu kļūdas pazīmes mainīgo:

```
res = myFunct2(5, "ABC");
```

```
if (errno != 0)
```

```
    // notikusi kļūda ! Kļūdas aprakstu var iegūt, izsaucot  
    funkciju perror().
```

§ Izmantot izņēmumu apstrādes mehānismu, kas iebūvēts programmēšanas valodā.

# Izņēmumu apstrādes principi valodā C++

- n Valodā C++ tiek lietota izņēmumu apstrāde, izmantojot ***try, throw, catch***, mehānismu.
- n Funkcija, kas konstatē izņēmumu, izveido un “izsviež” (***throw***) īpašu izņēmuma objektu atpakaļ izsaucošajai funkcijai. Šajā gadījumā funkcija automātiski beidz darbu, neatgriežot nekādu vērtību (***void***).
- n Ja izņēmuma rašanās vieta vai funkcijas izsaukums, kas tieši vai netieši izraisīja izņēmumu, ir iekļauts mēģinājuma (***try***) blokā, tad izsaucošā funkcija var saņemt jeb “noķert” (***catch***) šo izņēmuma objektu, paredzot speciālu(s) ***catch*** bloku(s). Tiklīdz izņēmums tiek izraisīts, programmas vadība uzreiz pāriet pie pirmā ***catch*** bloka, kura tips sakrīt ar izņēmuma objekta tipu.
- n ***Catch*** bloka meklēšanas procesā automātiski tiek atbrīvota lokālo objektu atmiņa un izsaukti objektu destruktori.
- n Ja izņēmums netiek noķerts ar ***catch***, tad programma beidz darbu.

# Izņēmumu apstrāde – try, throw, catch

## (turpinājums)

---

### Klases funkcija:

```
void ClassName::funct() {  
    // ...  
    if (exception_is_detected) // konstatēts izņēmuma gadījums  
        throw new ExceptionObject; // izņēmuma objekta izveidošana un "sviešana"  
    // ...  
}
```

### Izsaucošā funkcija:

```
void caller() {  
    ClassName obj;           // rada objektu  
    // ...  
    try {                    // mēģinājuma bloka sākums  
        // ...  
        obj.funct();         // funkcijas izsaukums, kas var radīt izņēmumu  
        // ...               // ja izraisīts izņēmums, tad atlikušā try bloka daļa neizpildās  
    }                        // mēģinājuma bloka beigas  
  
    catch (ExceptionObject*) { //norāda konkrētā izņēmuma objekta tipu  
        // kods, kas apstrādā šo izņēmumu  
    }  
    catch (...) {  
        // visu pārējo izņēmumu noķeršana  
    }  
}
```

# Izņēmumu lietošana - klase Triangle

---

```
class Triangle {
private:
    int a, b, c;
public:
    Triangle() { a = b = c = 0; }
    Triangle(int, int, int);

    float area();
    int perimeter();
};

class SidesException{};

Triangle::Triangle(int x,int y,int z)
{
    if (x+y<=z || x+z<=y || y+z<=x)
        throw new SidesException;
    a = x;
    b = y;
    c = z;
}

float Triangle::area(void) {
    float p = perimeter() / 2.0;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

```
void main()
{
    try{

        Triangle t(40, 50, 30);
        cout << "Area is " << t.area()
        << endl;
        cout << "Perimeter is "<<t.perimeter()
        << endl;

        Triangle f(4, 15, 3);
        cout << "Area is " << f.area()
        << endl;
        cout << "Perimeter is "<<f.perimeter()
        << endl;
    }

    catch (SidesException*) {
        cout << "Malu garumi nepareizi!"<<endl;
    }
    catch (...) {
        cout << "Unknown exception!"<<endl;
    }
}
```



```
c:\_oop\_Dev-Excepti
Area is 600
Perimeter is 120
Malu garumi nepareizi!
```

# Izņēmumu apstrāde – try, throw, catch

## (turpinājums)

---

n Izņēmuma objekta klase:

```
class NotInDatabase { };
```

n Datu bāzes klase:

```
class Database {
```

```
    // ...
```

```
public:
```

```
    Database(char* filename);
```

```
    char* search(int key);
```

```
    // ...
```

```
};
```

```
char* Database::search(int key) {
```

```
    char* value;
```

```
    bool found = false;
```

```
    // ...          meklē datubāzē pēc atslēgas key
```

```
    if (!found)
```

```
        throw NotInDatabase;
```

```
    // ...
```

```
    return value; // vērtība ir atrasta
```

```
}
```

# Izņēmumu apstrāde – try, throw, catch

## (turpinājums)

---

```
void main() {
    Database d("StudList.db");
    int k; char* v;
    // ...
    while (true) {
        // lietotājs ievada atslēgu k
        cout << "Ievadiet atslēgu ";
        cin >> k;
        try {
            v = d.search(k);
            cout << "Atrastais ieraksts: " << v << endl;
        }
        catch (NotInDatabase& e) {
            cout << "Šāds ieraksts datubāzē neeksistē!" << endl;
        }
        // ...
    }
    // ...
}
```



# Izņēmumi konstruktoru izpildes laikā

---

```
class DbFileErr { };

Database::Database(char* filename) {
    // ...
    if (cannot_open) throw DbFileErr;
    // ...
}

void main() {
    try {
        Database d("StudList.db");
        // ...
        v = d.search(k);
        // ...
    }
    catch (DbFileErr& e) {
        cout << "Nevar atvērt datu bāzi!" << endl;
    }
    catch (NotInDatabase& e) {
        cout << "Šāds ieraksts datubāzē neeksistē!" << endl;
    }
}
```

# Izņēmumu objekti

---

```
class DbException {
    int type;
public:
    DbException(int t) { type = t; }
    int getExceptionType() { return type; }
    char* getExceptionText();
};

char* DbException::getExceptionText() {
    switch (type) {
        case 1: return "Key not found!";
        case 2: return "Dublicate key!";
        case 3: return "The database is full!";
        case 4: return "Cannot open database!";
        default: return "Unknown error!";
    }
}
```

- n Šādu pieeju lieto reti. Biežāk veido izņēmumu klašu hierarhijas (DbNoKeyExcepti on, DbDubl i cateKeyExcepti on, DbI sFul l Excepti on, ...)

# Izņēmumu izmantošana algoritmu veidošanai

```
Database::Database(char* filename) {
    ...
    if (cannot_open)
        throw DbException(4);
    ...
}

char* Database::search(int key) {
    ...
    if (!found) throw DbException(1);
    ...
    return value;
}

void Database::add(int key, char* value) {
    if (size == 100000)
        throw DbException(3);
    try {
        char* dummy = search(key);
    }
    catch (DbException& e) {
        if (e.getExceptionType() == 1) {
            // tādas atslēgas nav
            // pievienojam jaunu rakstu ar atslēgu key
            ...
        }
    }
}
```

Izsaucošajā funkcijā  
var izmantot  
izņēmuma objektu.

# Neparedzētu izņēmumu apstrāde

---

n Kas notiek, ja izsaucošā funkcija neapstrādā izņēmuma objektu?

n Iemesli:

§ Funkcijas izsaukums nav iekļauts *try* blokā;

§ Funkcijas izsaukums ir *try* blokā, bet nav atbilstoša *catch* bloka.

n Sekas:

1. Funkcija uzreiz beidz darbu

2. Izņēmums tiek "sviests" tālāk nākošajai augstāka līmeņa izsaucošajai funkcijai, līdz par *main()*

3. Ja arī funkcijā *main()* nav atrasts atbilstošs *catch* bloks, tad programma beidz darbu

n Funkcijā *main()* var paredzēt visu neapstrādāto izņēmumu apstrādi:

```
void main() {  
    try {  
        // galvenās funkcijas operatori  
        ...  
    }  
    catch (...) {  
        // ziņojums par kļūdu un programmas beigšana  
    }  
}
```