

Preopterećeni operatori i pretvorbe

Objektno programiranje - 13. vježbe (2. dio)

Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

13. lipnja 2024. godine



Preopterećeni operatori

- preopterećeni operatori = funkcije čije je ime `"operator" + simbol operacije` (primjerice, `operator<<`)
- ne možemo promijeniti postojeće operatore za ugrađene tipove ni smisliti nove operatore:

Primjer.

```
int operator*(int, int);    X
int operator**(int, int);  X
```

Ako želimo napisati `operator?`:

- ako je `?` unaran/binaran, tada moramo imati jedan/dva argumenta (redoslijed za dva: prvi je lijevi, drugi je desni)
- ako je to funkcija članica klase, tada se implicitan `this` odnosi na prvi operand
- asocijativnost i prioritet operatora `?` ostaje nepromijenjena (npr. `x == y + z;` je ekvivalentno `x == (y + z);`)

Koje operatore (ne) možemo preopteretiti?

Operatori koje možemo preopteretiti

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	-
<<	>>	==	!=	&&	
+=	--	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operatori koje ne možemo preopteretiti

:: .* . ?:

- +, -, *, & mogu biti unarni ili binarni

Primjer: Klasa **Matrica**

- kod se može preuzeti na web stranici kolegija
- tri datoteke:
 - **Matrica.h**
 - **Matrica.cpp**
 - **main.cpp**
- sadržaj datoteke **Matrica.h** prikazan je na slici desno
- o matrici pamtimo: dimenziju (broj redaka i broj stupaca), elemente (tipa `double`)

```
1  #ifndef RACUN_H
2  #define RACUN_H
3
4  #include <iostream>
5  #include <utility>
6
7  class Matrica {
8  public:
9      Matrica();
10     Matrica(size_t);
11     Matrica(size_t, size_t);
12     ~Matrica();
13     Matrica(const Matrica &);
14     Matrica& operator=(const Matrica &);
15     Matrica(Matrica &&) noexcept;
16     Matrica& operator=(Matrica &&);
17 private:
18     void alociraj();
19     void dealociraj();
20     std::pair<size_t, size_t> dim;
21     double **elementi;
22 };
23
24 #endif
```

Datoteka Matrica.cpp - 1. dio (pomoćne funkcije)

```
1  #include <iostream>
2  #include <utility>
3  #include "Matrica.h"
4  using namespace std;
5
6  void Matrica::alociraj() {
7      elementi = new double*[dim.first];
8      for(size_t i = 0; i < dim.first; ++i) {
9          elementi[i] = new double[dim.second];
10     }
11 }
12
13 void Matrica::deallociraj() {
14     for(size_t i = 0; i < dim.first; ++i)
15         delete[] elementi[i];
16     delete[] elementi;
17     elementi = nullptr;
18 }
```

Datoteka `Matrica.cpp` - 2. dio (konstruktori i destruktor)

```
20 Matrica::Matrica(size_t a, size_t b) {
21     dim.first = a;
22     dim.second = b;
23     alociraj();
24     for(size_t i = 0; i < dim.first; ++i)
25         for(size_t j = 0; j < dim.second; ++j)
26             elementi[i][j] = 0;
27 }
28
29 Matrica::Matrica(size_t a) : Matrica(a,a) {}
30
31 Matrica::Matrica() : Matrica(0) {}
32
33 Matrica::~Matrica() {
34     dealociraj();
35 }
```



Datoteka Matrica.cpp - 3. dio (kopiranje)

```
37 Matrica::Matrica(const Matrica &m) {
38     dim = m.dim;
39     alociraj();
40     for(size_t i = 0; i < dim.first; ++i)
41         for(size_t j = 0; j < dim.second; ++j)
42             elementi[i][j] = m.elementi[i][j];
43 }
44
45 Matrica& Matrica::operator=(const Matrica &m) {
46     double** temp = new double*[m.dim.first];
47     for(size_t i = 0; i < m.dim.first; ++i) {
48         temp[i] = new double[m.dim.second];
49         for(size_t j = 0; j < m.dim.second; ++j)
50             temp[i][j] = m.elementi[i][j];
51     }
52     dealociraj();
53     dim = m.dim;
54     elementi = temp;
55     return *this;
56 }
```

Datoteka Matrica.cpp - 4. dio (premjestanje)

```
58 Matrica::Matrica(Matrica && m) noexcept {
59     dim = m.dim;
60     elementi = m.elementi;
61     m.dim = make_pair(0,0);
62     m.elementi = nullptr;
63 }
64
65 Matrica& Matrica::operator=(Matrica && m) {
66     if(this != &m) {
67         deallociraj();
68         dim = m.dim;
69         elementi = m.elementi;
70         m.dim = make_pair(0,0);
71         m.elementi = nullptr;
72     }
73     return *this;
74 }
```



Datoteka main.cpp - primjer upotrebe

```
1  #include <iostream>
2  #include "Matrica.h"
3  using namespace std;
4
5  Matrica f(Matrica a){
6      return a;
7  }
8
9  int main() {
10     Matrica a, b(3), c(3,4), d(c);
11     b = c;
12     Matrica e = std::move(c); //dalje ne koristiti c!
13     b = f(d); //dalje ne koristiti d!
14     return 0;
15 }
```

operator<< za ispis matrice

```
ostream &operator<<(ostream &os, const Matrica &m) {
    for(size_t i = 0; i < m.dim.first; ++i) {
        for(size_t j = 0; j < m.dim.second; ++j) {
            os << m.elementi[i][j] << " ";
        }
        os << endl;
    }
    return os;
}
```

Uočite poredak argumenata - primjer upotrebe:

```
cout << mat;
1. arg   2. arg
```

⇒ to ne može biti funkcija članica naše klase (1. argument!)

- prvi parametar nije `const` jer pisanje/čitanje mijenja *stream*!



operator>> za unos matrice

```
istream& operator>>(istream &is, Matrica &m) {  
    for(size_t i = 0; i < m.dim.first; ++i)  
        for(size_t j = 0; j < m.dim.second; ++j)  
            is >> m.elementi[i][j];  
    return is;  
}
```

Pitanja.

- Zašto je prvi argument referenca?
- Zašto drugi argument nije tipa `const Matrica&?`
- Zašto je povratni tip `istream&?`

Uočite poredak argumenata - primjer upotrebe:

```
cin >> mat;  
1. arg 2. arg
```

Funkcije friendovi naše klase

- zbog pristupa dijelovima klase `Matrica` koji nisu `public`
- dodamo u datoteku `Matrica.h`:

```
class Matrica {  
    friend std::istream &operator>>(std::istream&,  
                                    Matrica&);  
    friend std::ostream &operator<<(std::ostream&,  
                                    const Matrica&);  
    ...  
};
```

Primjer: ne mijenjamo matricu za neuspjeli unos

```
istream &operator>>(istream &is, Matrica &m) {  
    double **temp = new double*[m.dim.first];  
    for(size_t i = 0; i < m.dim.first; ++i) {  
        temp[i] = new double[m.dim.second];  
        for(size_t j = 0; j < m.dim.second; ++j)  
            is >> temp[i][j];  
    }  
    if(is) { //ako sve uspješno učitali, to spremimo  
        m.dealociraj();  
        m.elementi = temp;  
    } else { //inače ostavimo prethodnu vrijednost  
        for(size_t i = 0; i < m.dim.first; ++i)  
            delete[] temp[i];  
        delete[] temp;  
    }  
    return is;  
}
```

Aritmetički i relacijski operatori

```
Matrica operator+(const Matrica &lm,
                  const Matrica &dm) {
    Matrica zbroj(lm);
    for(size_t i = 0; i < zbroj.dim.first; ++i)
        for(size_t j = 0; j < zbroj.dim.second; ++j)
            zbroj.elementi[i][j] += dm.elementi[i][j];
    return zbroj;
}
```

- ne zaboraviti dodati tu funkciju kao frenda klasi `Matrica`
- ne mijenja ni lijevi ni desni operand (`const!`), a vraća rezultat kao novu vrijednost (kopiju lokalne matrice)
- to je primjer simetrične operacije - one su obično ne-članice klase kako bi moglo doći do konverzije bilo kojeg operanda:

```
string s = "abc";
string t = s + "!";
string u = "hi" + s;
```

X ako + član od `string`

Primjer upotrebe (funkcija main)

```
int main() {  
    Matrica a(2,3), b(2,3);  
    cin >> a;  
    b = a;  
    cout << a + b;  
    return 0;  
}
```

Primjeri. (dva primjera unosa i ispisa)

za unos:

```
1 2 3  
4 5 6
```

dobivamo ispis:

```
2 4 6  
8 10 12
```

za unos:

```
1 2 3  
a
```

dobivamo ispis:

```
0 0 0  
0 0 0
```

Operatori za provjeru jednakosti

```
bool operator==(const Matrica &lm,
                const Matrica &dm) {
    if(lm.dim != dm.dim)
        return false;
    for(size_t i = 0; i < lm.dim.first; ++i)
        for(size_t j = 0; j < lm.dim.second; ++j)
            if(lm.elementi[i][j] != dm.elementi[i][j])
                return false;
    return true;
}
```

```
bool operator!=(const Matrica &lm,
                const Matrica &dm) {
    return !(lm == dm);    //lako uz upotrebu ==
}
```

- ne zaboraviti dodati kao friendove klasi `Matrica`
- ako imamo `==`, korisnik često očekuje i `!=` (i obratno)

Primjer upotrebe (funkcija `main`)

```
int main() {  
    Matrica a(2,3), b(2,3);  
    cin >> a;  
    cout << "Matrice " << (a != b ? "ni" : "")  
        << "su jednake." << endl;  
    b = a;  
    cout << "Matrice " << (a != b ? "ni" : "")  
        << "su jednake." << endl;  
    return 0;  
}
```

- ako korisnik kao matricu `a` unese neku različitu od nul-matrice, ispis je:

Matrice nisu jednake.

Matrice su jednake.

Za vježbu možete probati napisati:

- operator $-$ za oduzimanje dvije matrice
- operator $*$ za množenje matrica
- operator $/$ nam nema neku logičku smislenu definiciju za matrice pa ga ne treba implementirati (osim ako baš želite)
- slično, možete definirati (iako mi to nećemo napraviti jer ponovo nedostaje neka logična smisljena definicija) operator $<$
- ako ste napravili operator $<$, a već imamo $i ==$, razmislite kako pomoću toga lako dobiti operatore $>$, $<=$ i $>=$

Operatori pridruživanja

- poput *copy* i *move* pridruživanja, članice klase (ali sad desna strana ne mora biti matrica)
- npr. možemo navesti (bar početne) elemente matrice

Primjer. (što želimo)

$$\begin{array}{l} \text{Matrica } a(2, 3); \quad \rightarrow \quad \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ a = \{1, 2, 3, 4\}; \quad \rightarrow \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \end{pmatrix} \end{array}$$

- no, prvo treba reći nešto o varijabilnom broju argumenata funkcije

Funkcije s promjenjivim parametrima

- ako ne znamo unaprijed koliko će argumenata dobiti naša funkcija
- dva glavna načina:
 - (1.) ako su svi argumenti istog tipa - upotreba posebnog tipa `initializer_list`
 - (2.) inače: posebna vrsta funkcije (tzv. *variadic* predložak)
- promatramo način (1.) - potrebno zaglavlje:

```
#include<initializer_list>
```
- to je **predložak** za tip
- primjeri:
 - `initializer_list<int>`
 - `initializer_list<double>`
 - `initializer_list<string>`
 - ...
- kao kod vektora imamo `begin()`, `end()` i `size()`
- no, za razliku od vektora svi elementi su `const` vrijednosti

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

int zbroji(initializer_list<int> a) {
    int suma = 0;
    for(auto br : a)
        suma += br;
    return suma;
}

int main() {
    cout << zbroji({1,2,3,4,5}) //vitičaste zagrade!
        << endl;
    return 0;
}
```

Upotreba na našem primjeru matrica

- svuda dodati `#include <initializer_list>`
- u datoteku `Matrica.h` dodati:

```
class Matrica {  
    ...  
public:  
    ...  
    Matrica& operator=(std::initializer_list<double>);  
    ...  
}
```

- u datoteci **Matrica.cpp**:

```
Matrica& Matrica::operator=
    (initializer_list<double> il) {
    auto it = il.begin();
    for(size_t i = 0; i < dim.first; ++i)
        for(size_t j = 0; j < dim.second; ++j)
            elementi[i][j] = (it == il.end()) ? 0 : *it++;
    return *this;
}
```

- **Pitanje.** Zašto se neće kompajlirati ako `*it++` zamijenimo s `(*it)++`?

Primjer upotrebe (funkcija main)

```
#include <iostream>
#include <initializer_list>
#include "Matrica.h"
using namespace std;
```

```
int main() {
    Matrica a(2,3);
    a = {4,2,3,5,1,2};
    cout << a << endl;
    a = {2,3};
    cout << a << endl;
    return 0;
}
```

Ispis:

```
4 2 3
5 1 2

2 3 0
0 0 0
```


Složena pridruživanja

- nije nužno, ali često **funkcije članice klase**
- uočite koji je povratni tip (Zašto je takav?)

```
class Matrica {  
    ...  
public:  
    Matrica& operator+=(const Matrica&);  
    ...  
};
```

```
Matrica& Matrica::operator+=(const Matrica &dm) {  
    for(size_t i = 0; i < dim.first; ++i)  
        for(size_t j = 0; j < dim.second; ++j)  
            elementi[i][j] += dm.elementi[i][j];  
    return *this;  
}
```

Primjer upotrebe (main funkcija)

```
int main() {  
    Matrica a(2,3), b(2,3);  
    cin >> a >> b;  
    a += b;  
    cout << a;  
    return 0;  
}
```

Primjer. Za unos:

```
1 2 3  
4 5 6  
2 4 1  
3 4 5
```

dobivamo ispis:

```
3 6 4  
7 9 11
```

Zadatak. Probati napisati i operatore `--`, `*=` (eventualno i `/=` i `%=` ako to ima smisla).

Operator indeksiranja

- funkcija članica
- u skladu s uobičajenim značenjem, vraća **referencu** na element
- također, trebao bi biti preopterećen po `const`
- za matricu nije jednostavno dobiti [][] za dva indeksa - umjesto toga koristit ćemo operator () ([više o \[\]\[\] na linku](#))

Primjer. Operator indeksiranja na klasi `Vekt` - klasa:

```
class Vekt {
public:
    Vekt(std::initializer_list<int>);
    ~Vekt();
    double& operator[] (size_t n);
    const double& operator[] (size_t n) const;
private:
    size_t velicina;
    double *elementi;
};
```

Nastavak primjera s klasom Vekt (konstr/destr/op[])

```
Vekt::Vekt(initializer_list<int> il)
    : velicina(il.size()){
    elementi = new double[il.size()];
    size_t i = 0;
    for(auto it = il.begin(); it != il.end(); ++it)
        elementi[i++] = *it;
}
```

```
Vekt::~Vekt() {
    delete[] elementi;
}
```

```
double& Vekt::operator[] (size_t n) {
    return elementi[n];
}
```

```
const double& Vekt::operator[] (size_t n) const {
    return elementi[n];
}
```



Nastavak primjera s klasom `Vekt` (primjer upotrebe)

```
const Vekt a = {1,2,3,4,5};  
Vekt b = {2,3};  
b[0] = 5;  
cout << b[0] << endl;    //ispis: 5  
cout << a[3] << endl;    //ispis: 4
```

Operatori inkrementiranja i dekrementiranja

- mijenjanju stanje objekta pa stavimo kao funkcije članice (iako to nije nužno!)
- imaju prefiks i postfiks verziju

Prefiks inkrementiranje i dekrementiranje

Primjer. Klasa Razlomak (s operatorom <<):

```
class Razlomak {
    friend std::ostream& operator<<(std::ostream&,
                                   const Razlomak&);

public:
    Razlomak(double a, double b) : br(a), naz(b) {}
    Razlomak& operator++();
    Razlomak& operator--();

private:
    double br, naz;
};
```

Prefiks inkrementiranje i dekrementiranje (Razlomak)

```
ostream& operator<<(ostream &os, const Razlomak &r)
{
    os << r.br << "/" << r.naz;
    return os;
}
```

```
Razlomak& Razlomak::operator++() {
    br += naz;
    return *this;
}
```

```
Razlomak& Razlomak::operator--() {
    br -= naz;
    return *this;
}
```

- prefiksni operatori \Rightarrow vraćamo referencu na promijenjeni objekt

Postfiksno inkrementiranje i dekrementiranje

Problem.

Isto ime i tip argumenata kao prefiksni - kako ih onda preopteretiti?

Rješenje.

Dodatan (nekorišten!) parametar tipa `int`
(kompajler tom argumentu pridružuje 0).

Primjer. (Ilustracija postfiks operacija na klasi `Razlomak`)

```
class Razlomak {  
    ...  
public:  
    ...  
    Razlomak& operator++();  
    Razlomak& operator--();  
    Razlomak operator++(int);    //ne vraća ref.  
    Razlomak operator--(int);    //ne vraća ref.  
    ...  
};
```


Postfiks (nastavak primjera s klasom Razlomak)

```
Razlomak Razlomak::operator++(int) {  
    Razlomak r(br, naz);  
    ++*this;  
    return r;  
}
```

```
Razlomak Razlomak::operator--(int) {  
    Razlomak r(br, naz);  
    --*this;  
    return r;  
}
```

- `int` parametar ne koristimo pa mu ne dajemo ime
- postfiks → spremimo stanje objekta prije promjene radi vraćanja te vrijednosti

Primjer (post/prefiks operacije za Razlomak)

```
Razlomak a(1,2);  
cout << a++ << endl;    //ispis: 1/2  
cout << ++a << endl;    //ispis: 5/2  
cout << --a << endl;    //ispis: 3/2
```

Operator poziva funkcije

- funkcije članice
- objekt klase kao funkcija (tzv. **funkcijski objekti**)
- npr. za klasu `Matrica` i objekt `a` te klase, želimo da `a(1, 2)` daje element u 1. retku i 2. stupcu (standardno brojimo od 0)

```
class Matrica {  
    ...  
public:  
    double& operator() (size_t i, size_t j) {  
        return elementi[i][j];  
    }  
    double operator() (size_t i, size_t j) const {  
        return elementi[i][j];  
    }  
    ...  
};
```

Primjer upotrebe

```
Matrica a(2,3);  
const Matrica b(3,4);  
a(1,2) = 5;  
cout << a;  
cout << b(1,2) << endl;
```

Ispis.

```
0 0 0  
0 0 5  
0
```

Operatori konverzije (pretvaranje `u` tip klase)

- konvertirajući konstruktor (ne-`explicit` konstruktor s jednim argumentom) omogućava implicitnu konverziju u tip klase

Primjer. Sljedeći kod se kompajlira (Zašto i što se ispiše?):

```
Matrica a(5, 5);  
cout << a + 5;
```

Isto dobivamo i ovako:

```
Matrica a(5, 5);  
cout << 5 + a;
```

Sljedeće se kompajlira, ali pri pokretanju daje *Segmentation fault*:

```
Matrica a(5, 5);  
cout << a + 4;
```

Eksplicitni konstruktor

- ako deklariramo konstruktor kao **explicit** on se neće koristiti u implicitnim konverzijama
- *explicit* navodimo samo u deklaraciji (ne i u definiciji!)
- dakle, ako stavimo:

```
class Matrica {  
    ...  
    explicit Matrica(size_t);  
    ...  
};
```

tada se sljedeći niti jedan od sljedećih kodova ne kompajlira:

```
Matrica a(5,5);  
cout << a + 5;  X
```

*error: no match for 'operator+'
(operand types are 'Matrica' and 'int')*

```
Matrica a = 5;  X
```

*error: conversion from 'int' to non-
scalar type 'Matrica' requested*

- no, možemo eksplicitno pretvoriti: `cout << a + Matrica(5);`

- sljedeći kod se ne kompajlira:

```
Matrica a(2,3);
```

```
if(a)
```

```
    cout << "Nije nul-matrica!" << endl;
```

error: could not convert 'a' from 'Matrica' to 'bool'

- potrebno je definirati operator konverzije
- korisnički definirane konverzije = konstruktori konverzije + operatori konverzije

Primjer: operator pretvorbe ($\text{Matrica} \rightarrow \text{bool}$)

- funkcija članica, ne mijenja objekt, nema parametara i ne navodimo povratni tip

```
class Matrica {  
    ...  
public:  
    ...  
    operator bool() const;  
    ...  
};
```

```
Matrica::operator bool() const {  
    for(size_t i = 0; i < dim.first; ++i)  
        for(size_t j = 0; j < dim.second; ++j)  
            if(elementi[i][j] != 0)  
                return true;  
    return false;  
}
```


Primjer.

- pretpostavimo da nismo za matrice implementirali `operator<<`
- no, ipak se sljedeći kod kompajlira i ispiše 0 (Objasnite!)

```
Matrica a(2,3);  
cout << a << endl;
```

- rješenje: stavimo operator pretvorbe u `bool` kao eksplicitan:

```
class Matrica {  
    ...  
    explicit operator bool() const;  
    ...  
};
```

- sad gornji primjer pri kompajliranju daje poruku o grešci:
*error: no match for 'operator<<' (operand types are 'std::ostream
aka std::basic_ostream<char>' and 'Matrica')*

- iako sad prethodni primjer ne prolazi, ipak je sljedeći kod sasvim u redu:

```
Matrica a(2,3);  
if(a)  
    cout << "Nije nul-matrica!" << endl;
```

Objašnjenje. Kompajler će primijeniti eksplicitnu konverziju na izraz koji koristimo kao uvjet za:

- `if`
- `while`, `do...while`, `for`
- operand logičkih operatora `ne (!)`, ili `(||)`, i `(&&)`
- uvjetni operator `?:`

Malo o iznimkama

- iznimka - anomalija **tijekom izvršavanja**

Primjer. U datotekama "1.txt", "2.txt", "3.txt", ... zapisane su matrice na sljedeći način:

- prvi red datoteke sadrži dva broja (dimenzije matrice),
- ostali redovi u datoteci sadržaju redove matrice.

Primjerice, datoteka "2023.txt" može imati sadržaj:

	2	3	
	1	2	3
	4	5	6

Sadržaj datoteke "2023.txt" odgovara sljedećoj matrici:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Želimo ispisati zbroj svih matrica iz datoteka počevši redom od datoteke "1.txt" (ili 0 ako ne postoji datoteka "1.txt").

Nastavak primjera (funkcija main)

```
ifstream dat;           //treba #include <fstream>
vector<Matrica> matrice; //treba #include <vector>
size_t dim1, dim2;
int br = 1;
while(1) { //učitavanje matrica iz datoteka
    dat.open(to_string(br++) + ".txt");
    if(!dat)
        break;
    dat >> dim1 >> dim2;
    matrice.push_back(Matrica(dim1, dim2));
    dat >> matrice.back();
    dat.close();
}
cout << "Ucitane matrice:" << endl; //ispis matrica
for(auto m : matrice)
    cout << m << "-----" << endl;
zbroji(matrice); //funkcija je na idućem slajdu
```

Funkcija zbroji za zbrajanje elemenata

```
void zbroji(const vector<Matrica> &m) {
    cout << "Zbroj:" << endl;
    if(m.empty())
        cout << 0 << endl;
    else {
        Matrica zbroj = m[0];
        for(size_t i = 1; i < m.size(); ++i)
            zbroj += m[i];
        cout << zbroj;
    }
}
```

Problem. Za sljedeće datoteke pokretanje daje *Segmentation fault*:

"1.txt"	"2.txt"	"3.txt"	"4.txt"
2 3	2 3	1 1	2 3
1 2 3	0 2 4	3	4 1 2
4 5 6	3 5 7		2 3 4

Problem: pribrajanje matrica različite dimenzije

```
Matrica.cpp
```

```
#include <stdexcept>
```

```
...
```

```
Matrica& Matrica::operator+=(const Matrica &dm) {  
    if(dim != dm.dim)  
        throw runtime_error("+= nisu iste dim!");  
    for(size_t i = 0; i < dim.first; ++i)  
        for(size_t j = 0; j < dim.second; ++j)  
            elementi[i][j] += dm.elementi[i][j];  
    return *this;  
}
```

- dio programa koji je otkrio problem, prijavljuje to (ne mora znati kome!) i prekida svoje izvršavanje
- ovdje je iznimka objekt tipa `runtime_error` (zaglavljuje `stdexcept`)
 - inicijaliziran C-ovskim stringom = poruka o problemu koji je nastao

Obrada iznimke

- funkcija `zbroji` sad može detektirati i preskočiti „problematične“ datoteke - izmijenjen sadržaj for petlje:

```
for(size_t i = 1; i < m.size(); ++i) {  
    try {  
        zbroj += m[i];  
    } catch (runtime_error greska) {  
        cout << "Preskacem matricu iz datoteke "  
            << (i+1) << ".txt:  " << endl  
            << '\t' << greska.what() << endl;  
    }  
}
```

- za isti `try` blok, moguće staviti više `catch` dijelova - mi stavili za obradu iznimke tipa `runtime_error`
- klasa `runtime_error` ima metodu `what()` - vraća C-ovski string = kopija stringa kojim inicijalizirali objekt (na prethodnom slajdu)

Dobiveni ispis (za ranije prikazane datoteke)

```
sebastijan@DESKTOP:~/Matrice$ ./prog
```

```
Ucitane matrice:
```

```
1 2 3
```

```
4 5 6
```

```
-----
```

```
0 2 4
```

```
3 5 7
```

```
-----
```

```
3
```

```
-----
```

```
4 1 2
```

```
2 3 4
```

```
-----
```

```
Zbroj:
```

```
Preskacem matricu iz datoteke 3.txt:
```

```
+= nisu iste dim!
```

```
5 5 9
```

```
9 13 17
```


Standardne klase iznimaka iz zaglavlja `stdexcept`

<u>exception</u>	općeniti problem (treba zaglavlje <code>exception</code>)
<u>runtime_error</u>	problem koji se može detektirati tek prilikom izvršavanja programa
<u>range_error</u>	prilikom izvršavanja dobiven rezultat izvan raspona vrijednosti kojeg razmatramo
<u>overflow_error</u>	prilikom izvršavanja rač. daje <i>overflow</i>
<u>overflow_error</u>	prilikom izvrš. računanje daje <i>underflow</i>
<u>logic_error</u>	greška u logici programa
<u>domain_error</u>	logička greška: argument za koji ne postoji rezultat
<u>invalid_argument</u>	logička greška: neprikladan argument
<u>length_error</u>	logička greška: pokušaj stvaranja objekta koji je veći od maksimalne veličine tog tipa
<u>out_of_range</u>	logička greška: koristi se vrijednost koja je izvan valjanog raspona

underflow/overflow - vrijednost manja/veća od min/max koju objekt tog tipa može imati

Primjer. Izvršavanjem sljedećeg koda:

```
Matrica a(3, 4);  
cout << a;
```

dobivamo ispis:

```
0 0 0 0  
0 0 0 0  
0 0 0 0
```

- umjesto konstruktora koji stvara nul-matricu tipa $a \times b$, želimo stvoriti matricu sa slučajnim brojevima
- zaglavlje **random** - dvije vrste tipova:
 - **engine** - stvaraju niz slučajnih nenegativnih cijelih brojeva
 - **distribution** - koriste *engine* za vraćanje brojeva prema određenoj vjerojatnosnoj distribuciji
- **generator slučajnih brojeva** = *engine* + *distribution*

- promjene u datoteci `Matrica.cpp`:

```
#include <random>
```

```
...
```

```
Matrica::Matrica(size_t a, size_t b) {  
    dim.first = a;  
    dim.second = b;  
    alociraj();  
    default_random_engine e;  
    for(size_t i = 0; i < dim.first; ++i)  
        for(size_t j = 0; j < dim.second; ++j)  
            elementi[i][j] = e();  
}
```

- `e` je funkcijski objekt \rightsquigarrow `e()` daje sljedeći slučajan broj

Transformacija sirovih u upotrebljive slučajne brojeve

- sad kod:

```
Matrica a(3,4);  
cout << a;
```

daje ispis poput:

```
16807 2.82475e+08 1.62265e+09 9.84944e+08  
1.14411e+09 4.70211e+08 1.01028e+08 1.45785e+09  
1.45878e+09 2.00724e+09 8.23564e+08 1.11544e+09
```

- to možda nije raspon koji smo željeli - koristimo distribucijski objekt - primjerice, uniformna distribucija od 0 do 9 (uključivo):

```
Matrica::Matrica(size_t a, size_t b) {  
    ...  
    uniform_int_distribution<unsigned> u(0,9);  
    default_random_engine e;  
    ... //for petlje  
        elementi[i][j] = u(e);  
}
```

Više matrica - ali sve su iste!

primjerice, izvršavanjem koda

```
Matrica a(3,4), b(3,4), c(3,4);  
cout << a << b << c;
```

možemo dobiti ispis:

```
sebastijan@DESKTOP:~/Matrice$ ./prog
```

```
0 1 7 4
```

```
5 2 0 6
```

```
6 9 3 5
```

```
0 1 7 4
```

```
5 2 0 6
```

```
6 9 3 5
```

```
0 1 7 4
```

```
5 2 0 6
```

```
6 9 3 5
```

- ako objekte stavimo kao `static`, oni će zadržati svoje stanje kroz funkcijske pozive

```
Matrica::Matrica(size_t a, size_t b) {  
    ...  
    static uniform_int_distribution<unsigned> u(0, 9);  
    static default_random_engine e;  
    ...  
}
```

- sad bi se kodom s prethodnog slajda mogle dobiti ovakve matrice:

```
0 1 7 4  
5 2 0 6  
6 9 3 5
```

```
8 0 0 5  
6 0 3 0  
4 6 5 9
```

```
8 5 0 6  
4 7 9 7  
2 0 7 3
```

Daljnji problem

- no, svako pokretanje istog programa daje iste vrijednosti
- **Rješenje.** dajemo sjeme (*seed*) - vrijednost koju *engine* koristi kako bi počeo generirati brojeve od nove vrijednosti
- umjesto fiksne vrijednosti (npr. 32540), koristimo `time(0)`

```
Matrica::Matrica(size_t a, size_t b) {  
    ...  
    static uniform_int_distribution<unsigned> u(0, 9);  
    static default_random_engine e(time(0));  
    ...  
}
```

Druge razdiobe (distribucije)

- dobivali `unsigned` brojeve, svaki s jednakom vjerojatnošću
- za `double` brojeve između 0 i 1:

```
Matrica::Matrica(size_t a, size_t b) {  
    ...  
    static uniform_real_distribution<double> u(0,1);  
    static default_random_engine e(time(0));  
    ...  
}
```

Osim uniformne distribucije, mogu se koristiti i [ostale](#):

- normalna, Bernoullijeva, Poissonova, Cauchyjeva, Fisherova, ...