

Programovanie 3.

2011 – 2012

12. Prednáška

Ing. František Gyárfáš, PhD.

Katedra aplikovanej informatiky

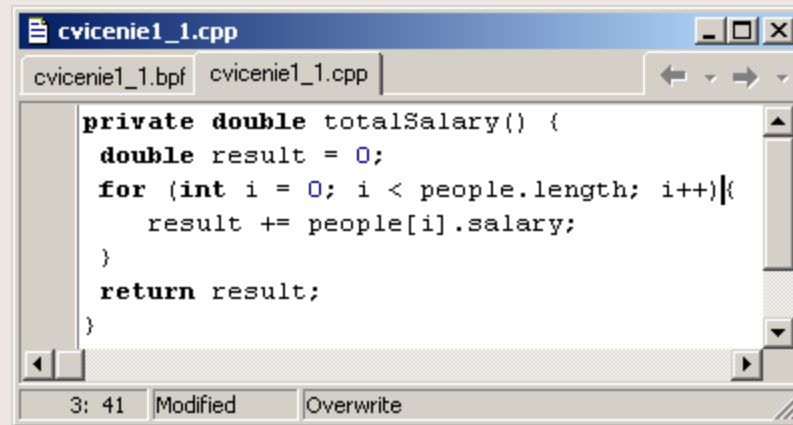
`gyarfas@ii.fmph.uniba.sk`

`http://www.ii.fmph.uniba.sk/~gyarfas/`

Čo je programovanie?

V ideálnom prípade:

Písanie programov.

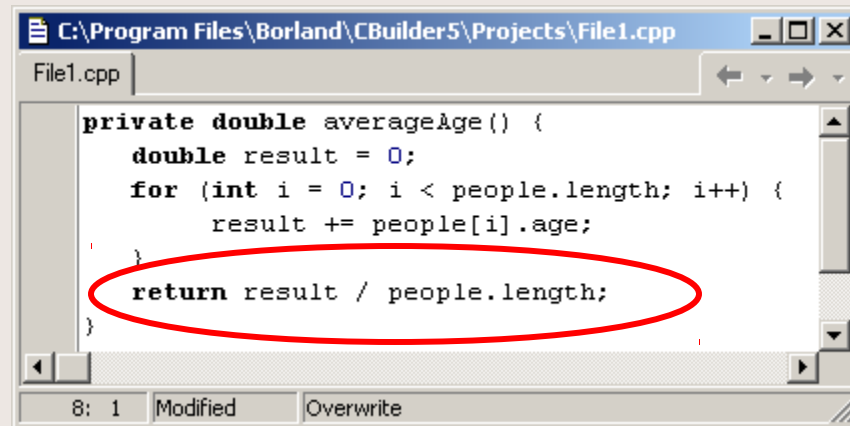


```
cvicenie1_1.cpp
cvicenie1_1.bpf cvicenie1_1.cpp
private double totalSalary() {
    double result = 0;
    for (int i = 0; i < people.length; i++){
        result += people[i].salary;
    }
    return result;
}
```

3: 41 Modified Overwrite

V realite:

Hľadanie chýb,
opravovanie,
testovanie,
refaktorovanie,
zmeny.



```
C:\Program Files\Borland\CBUILDER5\Projects\File1.cpp
File1.cpp
private double averageAge() {
    double result = 0;
    for (int i = 0; i < people.length; i++) {
        result += people[i].age;
    }
    return result / people.length;
}
```

8: 1 Modified Overwrite

Pravidlo o chybách v programe

Pravidlo o chybách v programe

Nepýtame sa, či sa dopustíme chýb pri programovaní. My to vieme.

Aké chyby nás čakajú?

- Chyby, ktoré sa objavia pri **kompilácii** alebo **linkovaní**.
- Chyby pôsobiace **pád programu**. Najdôležitejšie je dokázať **chybu zopakovať**.
- Program **nerobí**, čo od neho **očakávam**.
- Program robí, čo od neho očakávam, ale **v nepredvídateľných situáciách** urobí niečo **nečakané**.

Ako postupovať pri chybe

- Snažím sa **minimalizovať priestor**, kde sa chyba nachádza.
- Môžem **začať v bode**, kedy program už neide a vracať sa naspäť.
- Vyberiem **kontrolné body**, kde robím priebežné kontroly.

Kontrolné body

Na zvolené miesta vkladám **pomocné výpisy** so stavom premenných.

Kam výpisy posielat'? Na **štandardný výstup**, do **súboru**, do **databázy**?
(Pri nekonečnom cykle prichádza nekonečné množstvo kontrolných správ.)

Kontrolné výpisy **vypisovať podmiennečne**: iba vtedy, ak hľadám chybu.

```
#define DEBUG
```

```
int main() {
```

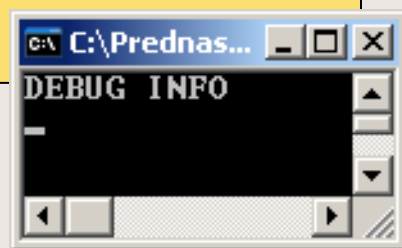
```
    . . .
```

```
#ifdef DEBUG
```

```
    cout << "DEBUG INFO";
```

```
#endif
```

```
    . . .
```



```
//#define DEBUG
```

```
int main() {
```

```
    . . .
```

```
#ifdef DEBUG
```

```
    cout << "DEBUG INFO";
```

```
#endif
```

```
    . . .
```



Použitie debuggera

Ak máme debugger, môžeme program vytvoriť **v ladiacom režime**.

Ladiaci režim nie je identický s normálnym behom programu. Premenné môžu mať inak inicializované hodnoty, využitie pamäti je iné a pod.

V debuggeri môžeme:

- **krokovat' program** (Step over, Trace into)
- vytvárať **body zastavenia** (Breakpoints)
- sledovať **stav programu** (Call stack, Variables, ...)

Bezpečné programovanie

Pravidlo bezpečného programovania

Bezpečné programovanie je krásny sen.

Pravidlo programátorskej schizofrénie

Programovanie je trvalý konflikt medzi programátorom – **byrokratom** a programátorom – **hazardérom**. Obaja sa ukrývajú v každom z nás.

Ako programovať čo najbezpečnejšie

Kontrolujte si **všetky varovania** kompilátora. Snažte sa ich odstrániť.

Minimalizujte počet **globálnych premenných**.

Ochraňujte premenné používaním **const**.

Zatíkat', zatíkat', zatíkat'. Ukrývanie informácií znižuje počet ohrození.

Kontrolujte **kritické operácie**, ako sú **alokácia** pamäti, či **otváranie** súborov.

Opatrne s **indexmi**. Vždy kontrolujte, či nie sú mimo priestoru polí.

Jedna entita by mala mať práve **jednu presne určenú úlohu**.
(premenná, trieda, funkcia, menný priestor, modul, knižnica)

Nulová tolerancia k chybe

V prípade, že nejaká **chyba nesmie nastať**, doporučuje sa radikálne riešenie.

```
assert(index < POCET);
```

Ak podmienka nie je splnená, **assert ukončí program** a vypíše zdrojový kód podmienky, ktorá nebola splnená.

`assert` je **najnekompromisnejší nástroj** pri boji s chybami.

Príklad pre assert

```
#include <stdio>
#include <assert>

const int POCET = 5;
int cisla[POCET] = {1,2,3,4,5};

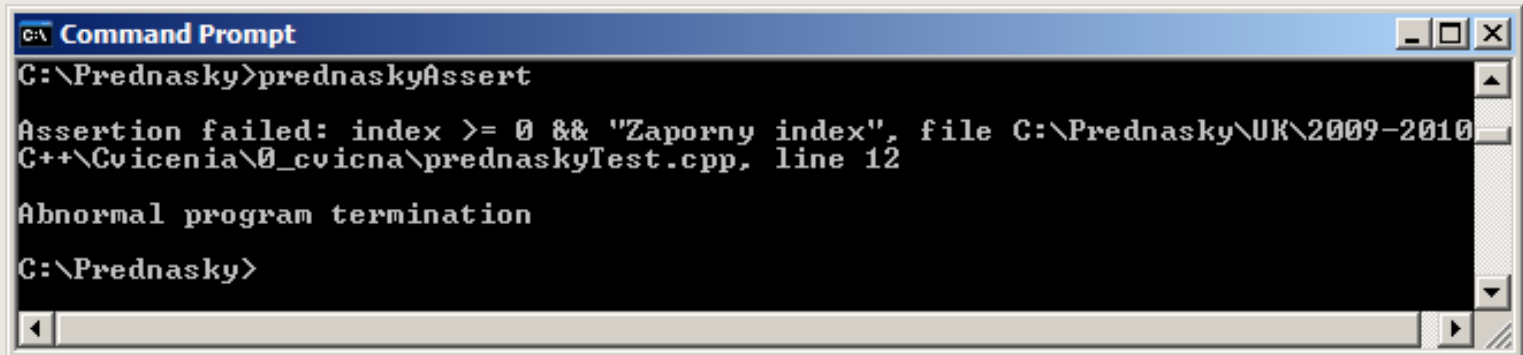
int main(){
    int index = -1;

    assert(index < POCET && "Prilis velky index");

    assert(index >= 0 && "Zaporny index");
    getchar();
}
```

assert na príliš veľký index.

assert na záporný index



The screenshot shows a Windows Command Prompt window with the title "C:\ Command Prompt". The command prompt is at "C:\Prednasky>prednaskyAssert". The output shows an assertion failure: "Assertion failed: index >= 0 && 'Zaporny index', file C:\Prednasky\UK\2009-2010\C++\Cvicenia\0_cvicna\prednaskyTest.cpp, line 12". This is followed by "Abnormal program termination" and the prompt "C:\Prednasky>".

```
C:\ Command Prompt
C:\Prednasky>prednaskyAssert

Assertion failed: index >= 0 && "Zaporny index", file C:\Prednasky\UK\2009-2010\C++\Cvicenia\0_cvicna\prednaskyTest.cpp, line 12

Abnormal program termination

C:\Prednasky>
```

Katastrofa

4. júna 1996 bol spustený počítačový program. Po 30 sekundách jeden **assert** zlyhal a program sa zastavil.

Program bežal na rakete Ariadne 5 a jeho úlohou bolo udržiavať raketu v správnom smere. Vypnutie programu znamenalo pre Ariadne 5 jej zničenie.

Pravidlo cvičiska a bojiska

Po ukončení testovania je rozumné radikálne riešenia konfliktných situácií ako **assert** vypnúť.

Vypnutie assert

```
#include <stdio>

#define NDEBUG ← .....

#include <assert>

const int POCET = 5;
int cisla[POCET] = {1,2,3,4,5};

int main(){
    int index = -1;

    assert(index < POCET);
    . . .
}
```

Po ukončení testovania vložíme makro
`#define NDEBUG`
pred volanie
`#include <assert>`
Tým je **assert** vypnutý.

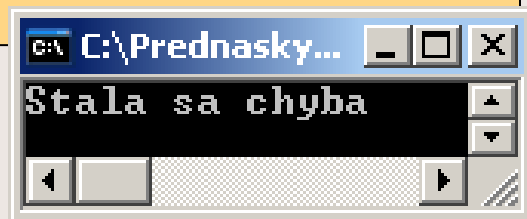
Chytanie chyby pomocou goto

Predstavme si, že odhalíme nežiaducu situáciu a chceme ju riešiť tak, že program nebude nepokračovať, ale začne riešiť krízový stav.

Ako by sme to riešili s **goto**?

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

int main(){
    if (true){
        string nieco;
        if (nieco == "CHYBA"){
            goto chyba;
        }
        return 0;
    }
    chyba: cerr << "Stala sa chyba";
    getchar();
}
```



Pred skokom **goto** sa zrušia všetky premenné bloku.

Potom program skáče na návestie **chyba**.

Návestie **chyba** sa musí nachádzať v tej istej funkcii ako príkaz **goto**.

Chytanie chyby pomocou throw

Veľmi podobne sa správa chytanie chyby s pomocou príkazu throw.

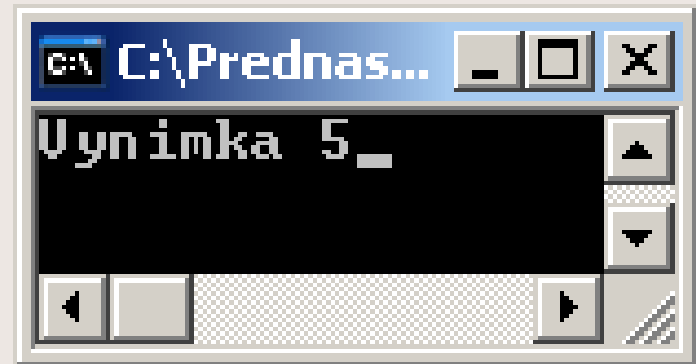
```
#include <iostream>
using namespace std;

class TypVynimky{
    int chyba;
public:
    TypVynimky(int i):chyba(i){};
    void printChyba(){
        cerr << "Vynimka " << chyba;
    };
};

int main() {
    try {
        string nieco;
        if (nieco == "CHYBA"){
            throw TypVynimky(5);
        }
        return 0;
    }catch (TypVynimky &id){
        id.printChyba();
    }
}
```

Ak program narazí na **throw**, opustí blok, v ktorom je a skáče na najbližší **catch** a začne tam skúmať, čo má urobiť.

Pred týmto skokom riadne zrušia premenné



Blok strážiaci výskyt výnimky

```
. . .  
try { ← .....  
    strážený blok  
    . . .  
    if (problem) {  
        throw TypVynimky;  
    }  
} ← .....  
catch (TypVynimky &id) {  
    spracovanie výnimky  
}  
pokračovanie programu  
}
```

Kľúčové slovo **try** označuje začiatok bloku, v ktorom strážime, či nedôjde k výnimke.

Koniec bloku, v ktorom strážime, či nedôjde k výnimke.

Postupnosť spracovanie výnimky

```
class TypVynimky{  
    TypVynimky() ;  
};  
  
int test(){  
    . . .  
    try {  
        A a;  
        B b;  
        if (problem){  
            throw TypVynimky;  
        }  
    }  
    catch (TypVynimky &id) {  
        spracovanie výnimky  
    }  
    pokračovanie programu  
}
```

(1)

(2)

(3)

(4)

Najprv sa opustia všetky vnorené funkcie a bloky (**stack-unwinding**), pričom sa na všetky lokálne premenné (a, b) volajú ich deštruktory.

Skočí sa na výnimku **TypVynimky** a privolá sa jej konštruktor .

Prevedie sa kód v príslušnom **catch** bloku. Hodnota **throw TypVynimky** sa odloží pre prípad ďalšieho posielania.

Po ukončení **catch** bloku program pokračuje ďalšími inštrukciami.

Označenie výnimky mimo bloku **try**

```
class TypVynimky1 {};  
class TypVynimky2 {};  
  
f() {  
    if (problem1) {  
        throw TypVynimky1;  
    }  
    if (problem2) {  
        throw TypVynimky2;  
    }  
}  
  
g() {  
    try {  
        . . .  
        f();  
    } catch (TypVynimky1 &id) {  
        spracovanie výnimky  
    } catch (TypVynimky2 &id) {  
        spracovanie výnimky  
    }  
    pokračovanie programu  
}
```

Volania výnimiek sú umiestnené vo funkcii **f()**, ktorá nemá strážený blok.

Volanie funkcie **f()** z vnútra stráženého bloku vo funkcii **g()**. Odkiaľ je volaná funkcia **f()** sa určí až počas behu programu.

Typy výnimiek

Typy výnimiek môžu byť:

- Jednoduché typy – int, float, double, char, char *

```
. . .  
try {  
    if (a) throw 333;  
    . . .  
    if (b) throw "Chyba X";  
    . . .  
}  
catch (int e) {  
    cout << "int " << e;  
}  
catch (char *c) {  
    cout << "char " << c;  
}  
. . .
```

Typ výnimky: int

Typ výnimky: char *c

Typy výnimiek - triedy

- triedy, smerníky na triedy

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

int main() {
    string s = "JA";
    try {
        if (false) {
            throw s;
        }
        else{
            throw &s;
        }
    }
    catch (string &s){
        cout << "string - " << s;
    }
    catch (string *s){
        cout << "smernik - " << s->c_str();
    }
}
```

Typ výnimky: trieda

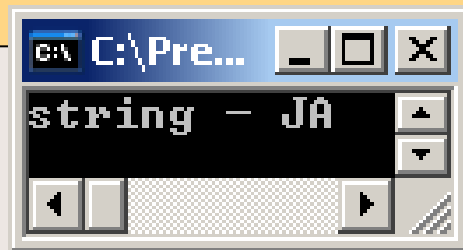
Typ výnimky: smerník na triedu



Stratené JA

V príklade presunieme definíciu string `s` do vnútra bloku `try`.

```
int main(){
    try{
        string s = "JA";
        if (true){
            throw s;
        }
        else{
            throw &s;
        }
    }
    catch (string &s){
        cout << "string - " << s;
    }
    catch (string *s){
        cout << "smernik - "
            << s->c_str();
    }
}
```



```
int main(){
    try{
        string s = "JA";
        if (false){
            throw s;
        }
        else{
            throw &s;
        }
    }
    catch (string &s){
        cout << "string - " << s;
    }
    catch (string *s){
        cout << "smernik - "
            << s->c_str();
    }
}
```



Definícia výnimiek

```
class TypVynimky1 {};
```

Najjednoduchší typ výnimky

```
class TypVynimky2 {  
    string text;  
    public:  
        TypVynimky2(char *chyba);  
        print();  
};
```

Normálna trieda s premennou, konštruktorom a metódou.

```
class TypVynimky4 {};  
  
class TypVynimky3 : public TypVynimky4 {  
    string text;  
    public:  
        TypVynimky3(char *chyba);  
        print();  
};
```

Hierarchia výnimiek

Výber v hierarchii výnimiek

```
class TypVynimky4 {};  
  
class TypVynimky3 : public TypVynimky4 {  
    . . .  
};  
  
int main(){  
    try {  
        if (problem3)  
            throw TypVynimky3;  
    }  
    catch (TypVynimky4 &id){  
        spracovanie výnimky  
    }  
    catch (TypVynimky3 &id){  
        spracovanie výnimky  
    }  
    pokračovanie programu  
}
```

Ak sú výnimky definované v hierarchii, vyberie prvú, ktorá sa nachádza v príslušnej hierarchii aj v prípade, že to je iba rodičovská výnimka.

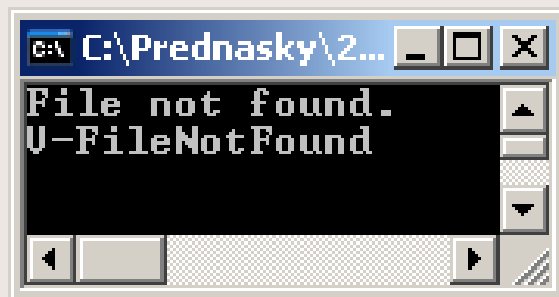
Pravidlo subordinácie výnimiek

Chytanie výnimiek vždy usporadujte od najdetailnejšej po najvšeobecnejšiu.

Príklad usporiadania výnimiek

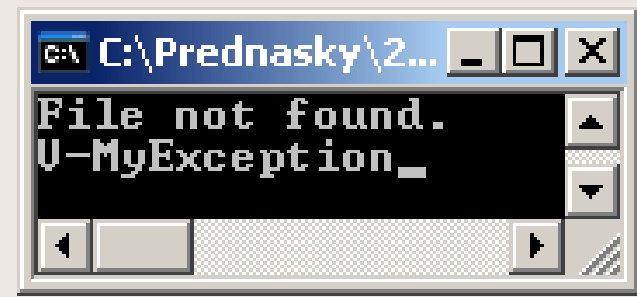
```
class MyException {...};  
class FileNotFound : public MyException {...};
```

```
int main(){  
    try{  
        if (true){  
            throw FileNotFound();  
        }  
  
    } catch (FileNotFound &e) {  
        cout << "V-FileNotFound";  
  
    } catch (MyException &e) {  
        cout << "V-MyException";  
  
    }  
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Prednasky\2...'. The window contains two lines of text: 'File not found.' on the first line and 'V-FileNotFound' on the second line. The window has standard Windows controls (minimize, maximize, close) and a scroll bar on the right.

```
int main(){  
    try{  
        if (true){  
            throw FileNotFound();  
        }  
  
    } catch (MyException &e) {  
        cout << "V-MyException";  
  
    } catch (FileNotFound &e) {  
        cout << "V-FileNotFound";  
  
    }  
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Prednasky\2...'. The window contains two lines of text: 'File not found.' on the first line and 'V-MyException_' on the second line. The window has standard Windows controls (minimize, maximize, close) and a scroll bar on the right.

Všeobecná výnimka

```
. . .  
try {  
    if (problem3)  
        throw TypVynimky3;  
}  
catch (TypVynimky1 &id){  
    spracovanie výnimky  
}  
catch (...){  
    spracovanie všetkých  
    výnimiek  
}  
  
pokračovanie programu  
}
```

Všeobecná výnimka **catch (...)** zachytáva všetky výnimky.

Táto výnimka sa musí uvádzať vždy ako posledná v zozname výnimiek.

Všeobecná výnimka by sa mala používať čo najmenej. Môže sa totiž stať, že cez ňu sa schová aj výnimka, ktorá by nemala ostať neodhalená.

Implicitná výnimka

```
. . .  
try {  
    if (problem3)  
        throw TypVynimky3;  
}  
catch (TypVynimky1 &id){  
    spracovanie výnimky  
}  
catch (TypVynimky2 &id){  
    spracovanie výnimky  
}  
  
catch (...){  
    throw;  
}  
  
pokračovanie programu  
}
```

V prípade, že sa objaví výnimka, ktorá nie je spracovaná ani jedným z **catch** príslušného bloku, privolá sa implicitný všeobecný **catch**, ktorý odošle tú istú výnimku `TypVynimky3` ďalej.

Špecifikácia povolených výnimiek

```
void f() {... throw(...); ...}
```

Funkcia **f()** povoľuje všetky výnimky

```
void f() throw() {...throw(...);}
```

Funkcia **f()** nepovoľuje žiadne výnimky a ukončí beh programu

```
void f() throw(MyException) {...}
```

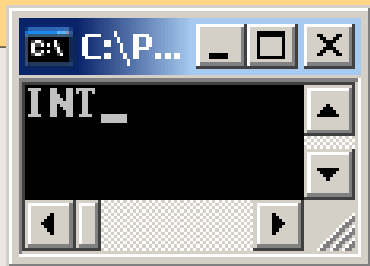
Funkcia **f()** povoľuje iba výnimku **MyException**, inak ukončí beh programu. Špecifikácia povoľuje celú hierarchiu dedenia triedy **MyException**.

```
void f() throw(Ex1, Ex2) {...}
```

Funkcia **f()** povoľuje zoznam výnimiek **Ex1, Ex2**.

Dobrá a zlá špecifikácia výnimiek

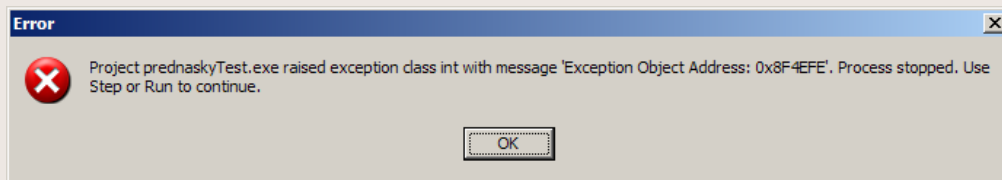
```
void f() throw (int){  
    throw 7;  
}  
void g() throw (int){  
    f();  
}  
main()  
{  
    try {  
        g();  
    }  
    catch (int) {  
        cout << "INT";  
    }  
    catch (float) {  
        cout << "FLOAT";  
    }  
}
```



```
void f() throw (int){  
    throw 7;  
}  
void g() throw (float){  
    f();  
}  
main()  
{  
    try {  
        g();  
    }  
    catch (int) {  
        cout << "INT";  
    }  
    catch (float) {  
        cout << "FLOAT";  
    }  
}
```

Zlá špecifikácia
výnimiek.

Program padne bez ošetrenia výnimky.



Zachytenie všetkých výnimiek

catch (. . .) by mal byť na nasledujúcich miestach:

- **main**: zachytí všetky nezachytené výnimky
- **hranice rozhrania modulu**: public metódy triedy.
- **vo vnútri deštruktorov**: deštruktory nesmú vypustiť výnimky.
- **vo vnútri konštruktorov globálnych premenných**: takéto konštruktory nesmú vypustiť výnimky. Nemali by sa kde zachytiť.

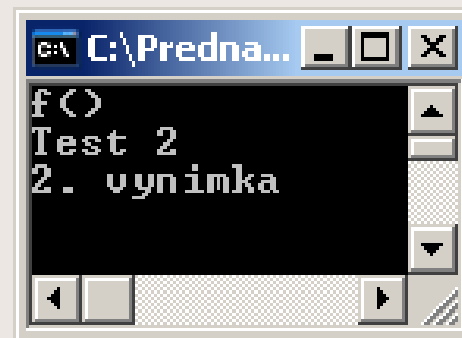
Výnimka v **throw** časti

Nová výnimka nahradí pôvodnú výnimku.

```
#include <iostream>
using namespace std;
class test {
public:
    test(int i){cout << "Test\n";};
};
class test2 {
public:
    test2(int i){cout << "Test 2\n";};
};
int f() {
    cout << "f()" << endl;
    throw test2(1);
    return 1;
}
int main(int argc, char* argv[]) {
    try{
        throw test(f());
    } catch (test &e){
        cout << "1. vynimka ";
    } catch (test2 &e){
        cout << "2. vynimka";
    };
    getchar();
}
```

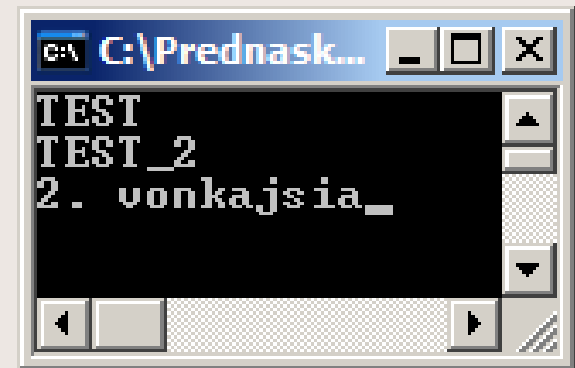
throw test(f()) privolá funkciu **f()**.

Vo funkcii **f()** je ďalší **throw test2(1)**. Ten sa vyhodnotí a nahradí **throw test(f())**, ktorý sa už nevyhodnotí.



Výnimka v catch časti

```
#include <iostream>
using namespace std;
class TEST {
public:
    TEST(int i) {cout << "TEST" << endl;};
};
class TEST_2 {
public:
    TEST_2(int i) {cout << "TEST_2" << endl;};
};
int main() {
    try {
        try {
            throw TEST(1);
        } catch (TEST &e){
            throw TEST_2(2);
            cout << "1. vynimka";
        } catch (TEST_2 &e){
            throw TEST_2(1);
            cout << "2. vnutorna";
        }
    } catch (TEST_2 &e){
        cout << "2. vonkajsia";
    };
}
```



Programovanie: posledný krát



Cvičenia

```
for (unsigned int i = 0; i < slovo.length(); i++) {  
    if ((slovo[i] >= 'a' && slovo[i] <= 'z')  
        || (slovo[i] >= 'A' && slovo[i] <= 'Z')) {  
        return true;  
    }  
}
```

```
for (unsigned int i=0; i+1<=pocet; i++){  
    vysledok+=student1[i][0].getName();  
    if (i+1!=pocet) {  
        vysledok+=", ";  
    }  
}
```

```
map<string, PESNICKA>::iterator it;  
for (it = spevnik.begin(); it != spevnik.end(); it++) {  
    if (it->first == nazovPiesne) {  
        spevnik.erase(it);  
        break;  
    }  
}
```

Remeslo

Programovanie je **remeslo**.

Máme **majstrov**, **tovarišov** a **učňov**.

Trvá **roky**, kým sa naučíme robiť to **dobre**.

Veľmi pomáha, keď nás to **baví**.

Tak nech Vás to baví čo najdlhšie.