

# Výjimky (Exceptions)

# Obsluha chyby – bez výjimek

```
int foo(std::string name) {  
    int status = 0;  
    Person person(name);  
    if (status == 0) {  
        if (person.isValidName()) {  
            cout << "Valid person " << person.getName() << endl;  
        }  
        else status = -1; // signalize problem with name  
    }  
    if (status == 0) { // Still no problem?  
        std::string message;  
        message = "My favourite person is ";  
        message += person.getName();  
        cout << message << endl;  
    }  
    if (status == 0) { // Still no problem?  
        // ... do something else  
    }  
    if (status != 0) { // Some problem?  
        // ... report problem, roll back info....  
    }  
    return status;  
}
```

Proměnná s  
hodnotou chyby  
(0 == OK)

Test, zda se  
nevyskytla chyba

V případě chyby  
nastav status

Každý další logický  
blok kódu obalen  
testem na OK

Udělej něco v reakci  
na problém

Vrať informaci o  
průběhu

# Výjimky – motivace

- Výskyt situace, kterou je potřeba okamžitě řešit
  - nedostatek paměti
  - nepovolený přístup do paměti
  - dělení nulou
- Může nastat kdekoli v kódu a může být řešitelné
  - uvolníme nepotřebnou paměť a zkusíme znovu
  - vypíšeme varování uživateli a nulou dál nedělíme
- Zároveň ale není praktické mít reagující kód na v každé metodě
  - nedostatečná paměť může nastat kdekoli

# Výjimky - syntaxe

- Označení oblasti kde zachytávat výjimku

- klíčové slovo **try**

- Vyvolání výjimky

- klíčové slovo **throw**

- Zachycení výjimky

- klíčové slovo **catch**

- jeden nebo více bloků

- specifikace objektu výjimky typicky jako reference

```
try {  
    // ... some code  
  
    throw std::logic_error("Problem");  
  
    // ... some code (not executed)  
}  
catch (std::logic_error& ex) {  
    // ... handle somehow exception  
    cout << ex.what() << endl;  
}
```

# Výjimky (Exception) - princip

1. V případě výskytu problému se vyvolá výjimka
  - objekt obsahující informaci o problému
  - můžeme vyvolat i vývojář programově (throw)
2. Výjimka postupně “stoupá” volajícími funkcemi
  - (callstack), dokud ji někdo nezachytí
3. Výjimka je zachycena obslužným blokem
  - specifikuje, v jaké části kódu a jaké výjimky zachytávat
  - try/catch blok
4. Na základě výjimky proběhne programová reakce
  - ukončení programu
  - výpis varování uživateli
  - exception handling

# Ukázka

```
class Person {  
    std::string m_name;  
public:  
    Person(const std::string name) : m_name(name) {  
        if (m_name.size() == 0)  
            throw WrongNameExceptionExt("Invalid empty name");  
    }  
    void print() const { cout << m_name << endl; }  
};
```

1.

Výjimka vyvolána

2.

Výjimka zachycena,  
obsloužena a zaniká

3.

Program pokračuje

```
int main() {  
    cout << "Code before exception handling" << endl;  
    try {  
        Person p1("Pepa Novak"); // No problem  
        Person p2("");  
  
        p1.print();  
        p2.print();  
    }  
  
    catch (WrongNameExceptionExt& ex) {  
        cout << "WrongNameExceptionEx: " << ex.what();  
    }  
  
    cout << "Continuing after block with exception handling";  
  
    return 0;  
}
```

Kód nebude vykonán

# Datový typ výjimky

## ● Primitivní datové typy

- např. int, char...
- `throw 1; catch (int ex) { }`
- `throw 'e'; catch (char ex) { }`

## ● Struktury, třídy

- struktury nebo třídy definované programátorem
- `MyStructException ex; throw ex;`
- `catch (MyStructException& ex) { }`

## ● Potomci std::exception

- nejčastěji potomci std::logic\_error nebo std::runtime\_error
- `class MyException : public std::logic_error {`
- `catch (std::logic_error& ex) { }`

```

#include <iostream>
#include <stdexcept>
#include <string>
using std::cout;
using std::endl;

struct MyStructException {
    std::string reason;
    int     someValue;
};

class MyException : public std::invalid_argument {
public:
    MyException(const std::string& reason = "") :
        std::invalid_argument(reason) {}
};

void foo(int what) {
    if (what == 1) throw 1;
    if (what == 2) throw 'e';
    if (what == 3) {
        MyStructException ex;
        ex.reason = "Just testing";
        ex.someValue = -1;
        throw ex;
    }
    if (what == 4) throw MyException("Just testing");
}

```

Vyhazuje  
různé typy  
výjimek podle  
what

```

int main() {
    cout << "Code before exception handling";

    try {
        foo(4); // throw exception

        cout << "Will not be printed";
    }
    catch (int ex) { foo(1)
        cout << "Integer exception: " << ex;
    }
    catch (char ex) { foo(2)
        cout << "Char exception: " << ex;
    }
    catch (MyStructException& ex) { foo(3)
        cout << "Struct exception : " << ex.reason;
    }
    catch (MyException& ex) { foo(4)
        cout << "class MyException : " << ex.what();
    }

    cout << "Continuing after block with EH";

    return 0;
}

```



# Standardní výjimky

- Výjimky jsou vyvolávané
  - jazykovými konstrukcemi (např. `new`)
  - funkcemi ze standardní knihovny (např. STL algoritmy)
  - uživatelským kódem (výraz `throw`)
- Základní třída `std::exception`
  - `#include <stdexcept>`
  - metoda `exception::what()` pro získání specifikace důvodu
- Dvě základní skupiny standardních výjimek
  - `std::logic_error` – chyby v logice programu
    - např. chybný argument, čtení za koncem pole...
  - `std::runtime_error` – chyby způsobené okolním prostředím
    - např. nedostatek paměti (`bad_alloc`)
  - v konstruktoru můžeme specifikovat důvod (`std::string`)
  - <http://www.cplusplus.com/reference/std/stdexcept/>

# Vlastní výjimky - třídy

- `std::exception` nemá konstruktor s možností specifikace důvodu (řetězec)
- Dědíme typicky z `logic_error` a `runtime_error`
  - mají konstruktor s parametrem důvodu (`std::string`)
- Nebo jejich specifitějších potomků
  - např. `std::invalid_argument` pro chybné argumenty

<http://www.cplusplus.com/reference/std/stdexcept/>

## Logic errors:

<code>logic_error</code>	Logic error exception (class)
<code>domain_error</code>	Domain error exception (class)
<code>invalid_argument</code>	Invalid argument exception (class)
<code>length_error</code>	Length error exception (class)
<code>out_of_range</code>	Out-of-range exception (class)

## Runtime errors:

<code>runtime_error</code>	Runtime error exception (class)
<code>range_error</code>	Range error exception (class)
<code>overflow_error</code>	Overflow error exception (class)
<code>underflow_error</code>	Underflow error exception (class)

# Vlastní výjimky - ukázka

Jméno výjimky

Výjimka může nést  
dodatečné informace dle  
našich potřeb

rodič výjimky  
zvolte co  
nejspecifičtější

```
class WrongNameExceptionExt : public std::invalid_argument {  
    std::string m_wrongName;  
public:  
    WrongNameExceptionExt(const std::string& reason = "", const std::string name = "") :  
        std::invalid_argument(reason), m_wrongName(name) {}  
    const std::string getName() const { return m_wrongName; }  
    ~WrongNameExceptionExt() throw () {}  
};
```

Můžeme přidat vlastní  
dodatečné metody

Konstruktor pro specifikace  
informací o výjimce.  
Inicializuje i předka.

Destruktor – nutné pokud  
máme atributy, které mají  
také destruktory


# Zachycení výjimek – využití dědičnosti

- Výjimky mohou tvořit objektovou hierachii
  - typicky nějakí potomci `std::exception`
- Při zachytávání můžeme zachytávat rodičovský typ
  - nemusíme chytat výjimky podle nejspecifičtějšího typu
  - obslužný kód může reagovat na celou třídu výjimek
  - např. zachytává výjimku typu `std::runtime_error` a všechny potomky

Vyvoláme `MyException`,  
chytáme  
`invalid_argument`

```
class MyException : public std::invalid_argument;

int main() {
    try {
        throw MyException("Test");
    }
    catch (std::invalid_argument& ex) {
        cout << "invalid argument : " << ex.what();
    }
    return 0;
}
```



# Pořadí vyhodnocování catch klauzulí

- Dle pořadí v kódu
  - pokud je více klauzulí, postupně se hledá klauzule s odpovídajícím datovým typem
  - klauzule “výše” budou vyhodnoceny dříve
- Vhodné řadit od nejspecifičtější po nejobecnější
  - nejprve potomci, potom předci
  - jinak se pozdější specifičtější nikdy neuplatní
- Kompilátor nás upozorní warningem
  - *warning: exception of type 'WrongNameException' will be caught by earlier handler for 'std::logic\_error'*

# Pořadí zachycení – ukázka problému

```
class WrongNameExceptionExt : public std::invalid_argument;

class Person;

int main() {
    cout << "Code before exception handling" << endl;
    try {
        Person p2(""); // Exception WrongNameExceptionExt thrown
    }
    catch (std::invalid_argument& ex) {
        cout << "Exception from group std::logic_error : " << ex.what();
    }
    catch (WrongNameExceptionExt& ex) {
        cout << "WrongNameExceptionExt: " << ex.what() << " " << ex.getName();
    }

    cout << "Continuing after block with exception handling" << endl;

    return 0;
}
```

**Není nikdy provedeno – všechny výjimky WrongNameExceptionExt jsou zachyceny jako std::invalid\_argument**