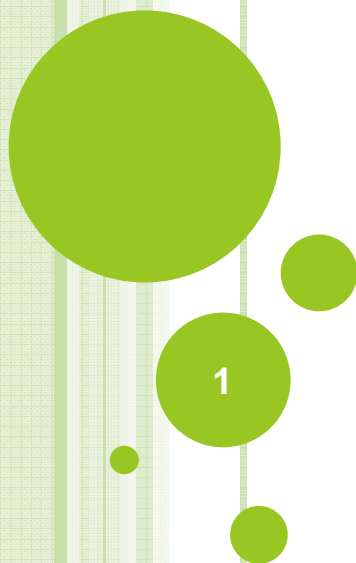


PB161 – PROGRAMOVÁNÍ V JAZYCE C++ OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

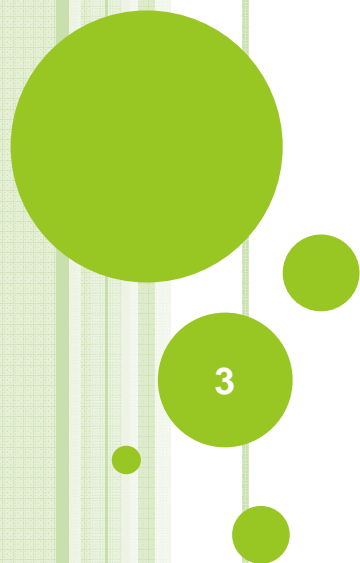


Functor, Výjimky

ORGANIZAČNÍ - ZVANÉ PŘEDNÁŠKY

- 2.12. Ondřej Krajíček (YSoft)
- 9.12. Jiří Weiser (Laboratoř Paradise)
 - C++11
 - + diskuze o předmětu

FUNKČNÍ OBJEKTY



FUNKČNÍ OBJEKT - MOTIVACE

- Z C/C++ jsou známy tzv. callback funkce

- pomocí funkčního ukazatele je možné předat jako parametr ukazatel na funkci, která je později zavolána
- používá se například u STL algoritmů

- `std::for_each(l.begin(), l.end(), print);`

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
{
    for ( ; first!=last; ++first ) f(*first);
    return f;
}
```

```
template < class InputIterator, class OutputIterator, class UnaryOperator >
OutputIterator transform ( InputIterator first1, InputIterator last1,
                           OutputIterator result, UnaryOperator op )
{
    while (first1 != last1)
        *result++ = op(*first1++); // or: *result++=binary_op(*first1++,*first2++);
    return result;
}
```

- Co když chceme přičítat pomocí `std::transform` číslo 10?

- vytvoříme funkci `int add10(int value){return value+10;}`
- `std::transform(l.begin(), l.end(), l.begin(), add10);`

- Co když chceme přičítat libovolné číslo?



může být zavolán

FUNKČNÍ OBJEKT – FUNCTOR

- Objekt, **který může zavolán**, jako by to byla běžná funkce
 - dosáhneme pomocí přetíženého operátoru `()`
 - `A prom; prom(5);`
- Operátor `()` musíme přetížit pro všechny potřebné typy **a** počty argumentů
 - tj. pro každou funkci, kterou chceme functorem nahradit
 - jeden functor může nahrazovat několik funkcí (jak?)
- Pokud se pokusíme použít objekt jako functor a neexistuje odpovídající přetížený operátor `()`
 - `error: no match for call to '(typ_objektu) (typ_argumentů)'`
 - *`stl_algo.h:4688:2: error: no match for call to '(A) (int&)'`*

FUNCTOR - UKÁZKA

```
#include <iostream>
#include <list>
#include <algorithm>

class CAddX {
    int m_whatAdd;
public:
    CAddX(int whatAdd) : m_whatAdd(whatAdd) {}
    void set(int whatAdd) { m_whatAdd = whatAdd; }
    int operator () (int value) {
        return value + m_whatAdd;
    }
};
```

```
int main() {
    std::list<int> myList;

    myList.push_back(1);  myList.push_back(2);
    myList.push_back(3);  myList.push_back(4);

    CAddX variableAdder(3);
    // variableAdder is functor object, now adding number 3
    transform(myList.begin(), myList.end(), myList.begin(), variableAdder);
    // change added value
    variableAdder.set(101);
    transform(myList.begin(), myList.end(), myList.begin(), variableAdder);

    return 0;
}
```

Nyní přičítač trojky

Nyní přičítač stojedničky

Adder: 'e'
je prekryte
bielou
šestkou

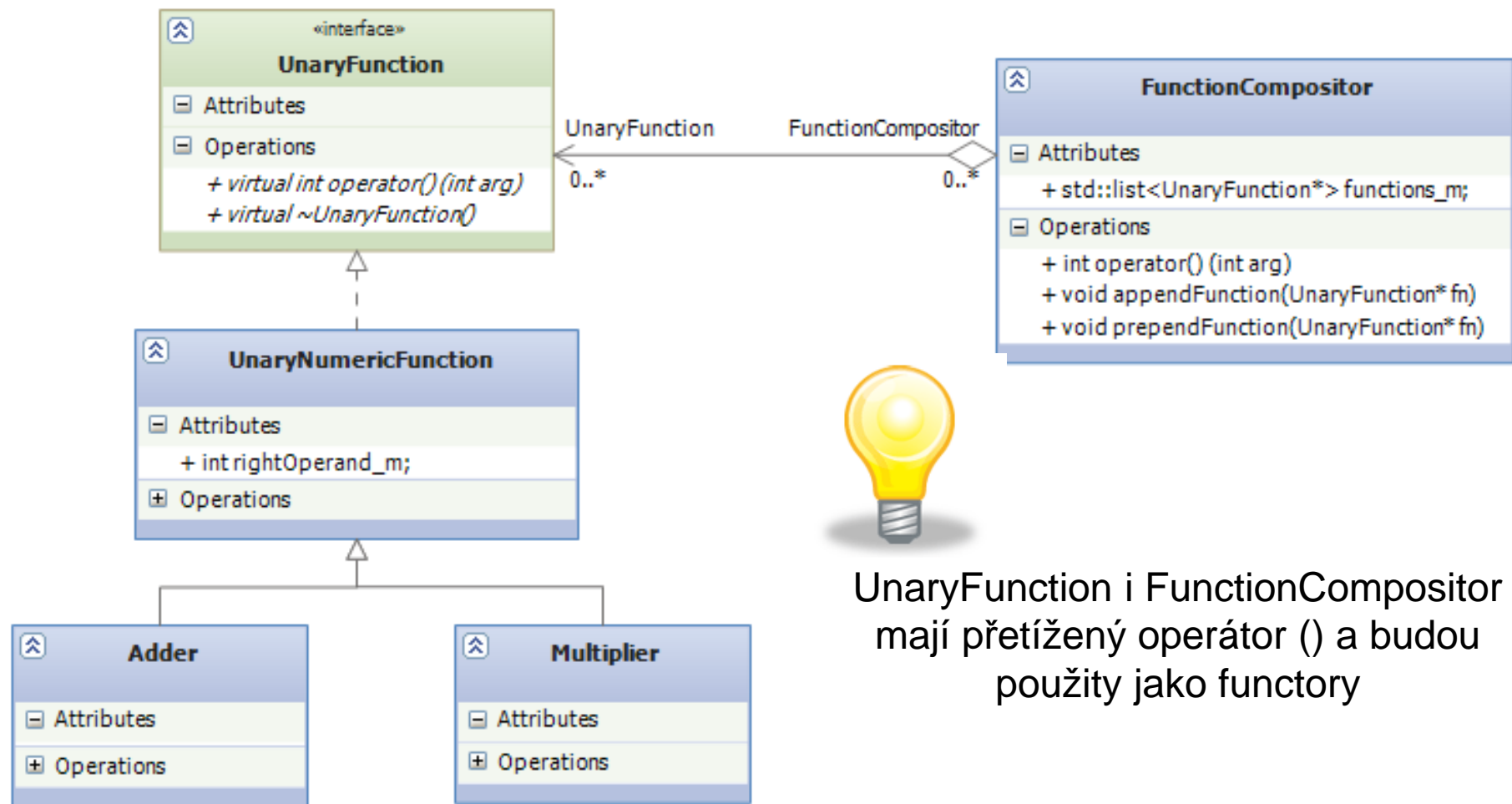
FUNCTORY – VHODNOST POUŽITÍ

- Výhodou je možnost uchovávat (a měnit) stav functoru
 - u funkcí bychom museli použít globální proměnné
- Functor může být výrazně rychlejší než funkce
 - překladač může vložit kód functoru namísto volání
- u funkčních ukazatelů (callback) není možné
- Využívá myšlenek z funkcionálním programování
 - současné imperativní jazyky mají podporu pro některé funkční prvky
 - functor v C++, delegates a lambda výrazy v C#...

FUNCTOR A TESTOVACÍ ZÁPOČTOVÝ PŘÍKLAD

- Testovací zápočtový příklad
 - http://cecko.eu/public/pb161_cviceni#prednaskacviceni_tyden_21-27112011
- Aplikace série funkcí na zadaný argument
- Definice vhodného rozhraní a využití dědičnosti
- (autor původního kódu Petr Pilař)

HIERARCHIE



UnaryFunction i FunctionCompositor mají přetížený operátor `()` a budou použity jako functory

```
class UnaryFunction {
public:
    virtual ~UnaryFunction() {}
    virtual int operator() (int arg) const = 0;
};
```

Přetížíme operátor () pro aplikaci unární funkce

```
class UnaryNumericFunction : public UnaryFunction {
public:
    UnaryNumericFunction(int rightOperand) : rightOperand_m(rightOperand) {}

    virtual int rightOperand() const { return rightOperand_m; }
    virtual void setRightOperand(int r) { rightOperand_m = r; }
```

```
private:
    int rightOperand_m;
};
```

Numerická unární funkce je functor s parametrizací v rightOperand_m

```
class Adder : public UnaryNumericFunction {
public:
    Adder(int r) : UnaryNumericFunction(r) {}
    int operator() (int arg) const { return arg + rightOperand(); }
};
```

Parametrizace functoru

```
class Multiplier : public UnaryNumericFunction {
public:
    Multiplier(int r) : UnaryNumericFunction(r) {}
    int operator() (int arg) const { return arg * rightOperand(); }
};
```

V přetíženém operátoru () přičítáme

```
// Function for deallocation
void delete_ptr(UnaryFunction* ptr) { delete ptr; }

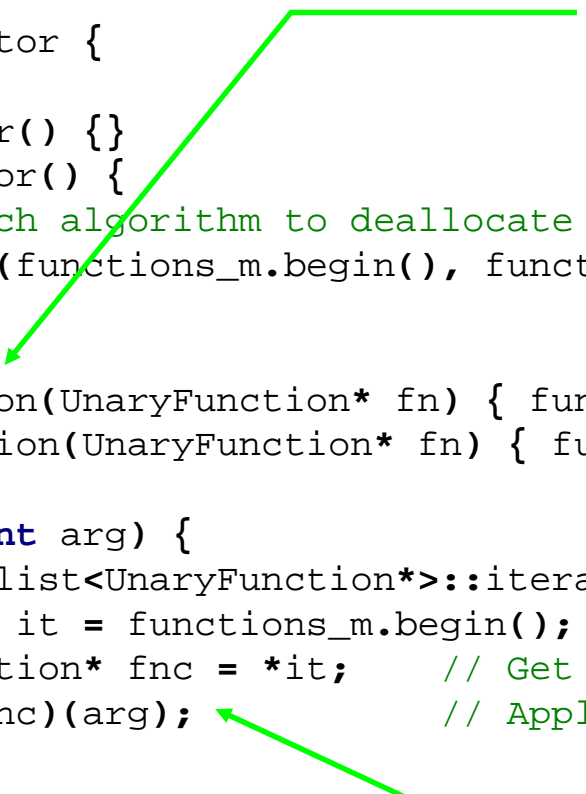
class FunctionCompositor {
public:
    FunctionCompositor() {}
    ~FunctionCompositor() {
        // Use for_each algorithm to deallocate separate functions
        std::for_each(functions_m.begin(), functions_m.end(), delete_ptr);
    }

    void appendFunction(UnaryFunction* fn) { functions_m.push_back(fn); }
    void prependFunction(UnaryFunction* fn) { functions_m.push_front(fn); }

    int operator() (int arg) {
        typedef std::list<UnaryFunction*>::iterator iterator;
        for (iterator it = functions_m.begin(); it != functions_m.end(); ++it) {
            UnaryFunction* fnc = *it;    // Get particular function
            arg = (*fnc)(arg);           // Apply it on argument
        }

        return arg;
    }

private:
    std::list<UnaryFunction*> functions_m;
};
```



**Parametrizace functoru
probíhá přidáváním
unárních funkcí**

**Aplikací functoru pomocí
operátoru () postupně
zavoláme všechny
obsažené funkce**

POUŽITÍ FUNCTORŮ

```
#include <iostream>
#include <algorithm>
#include <list>
```

```
int main() {
    FunctionCompositor comp;
    comp.prependFunction(new Adder(10));
    comp.prependFunction(new Multiplier(2));
    comp.prependFunction(new Adder(-5));
    comp.prependFunction(new Adder(2));

    int number = 0;
    std::cin >> number;
    std::cout << comp(number) << std::endl;
}
```

Vytváříme functor comp

Vytváříme functor typu
Adder (a další) a
vkládáme do comp

Aplikujeme functor comp

VÝJIMKY (EXCEPTIONS)

13

```

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

```
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/>



```
class A {  
    int m_value;  
public:  
    A(int value) : m_value(value) {}  
    void print() const {cout << m_value << endl; }  
};
```

```
// Allocate A object dynamically  
// If fail, std::bad_alloc exception is thrown  
A*    pTest2 = 0;  
try {  
    pTest2 = new A(20);  
    pTest2->print(); // no need to test pTest2 != NULL  
    delete pTest2;  
}  
catch (std::bad_alloc& ex) {  
    cout << "Fail to allocate memory";  
}
```

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 specifičtějších 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

ZACHYCENÍ A ZNOVUPOSLÁNÍ VÝJIMKY

- Někdy zjistíme že ji nedokážeme obsloužit
 - nebo chceme ještě provést obsluhu jinde
- Výjimka typicky zaniká na konci bloku `catch`, který ji zachytí
 - pokud neurčíme jinak
- Výjimku můžeme poslat dál
 - do bloku `catch` umístíme `throw` bez argumentů
 - pošle dál aktuální výjimku

ZNOVUPOSILÁNÍ VÝJIMKY - UKÁZKA

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

void foo() {
    try {
        Person p1("Pepa Novak"); // No problem
        Person p2(""); // Exception thrown
    }
    catch (WrongNameExceptionExt& ex) {
        cout << "WrongNameExceptionExt: " << ex.what() << endl;
        throw; // Let it propagate further
    }
}
```

Prázdné jméno způsobí výjimku

Výjimka odchycena, ale poslána dál

```
int main() {
    try {
        foo();
    }
    catch (std::logic_error& ex) {
        cout << "Exception std::logic_error : " << ex.what();
    }
    return 0;
}
```

Výjimka znovu zachycena a finálně zpracována.

ZACHYCENÍ VŠECH VÝJIMEK `CATCH (...)`

- Klauzule `catch(...) { }` zachytí většinu běžných výjimek
 - závisí na překladači, někdy i nezotavitelné jako SIGSEGV (VS6)
 - není přístup k objektu výjimky
- V běžném kódu zachytávejte pouze výjimky, které umíte zpracovat
 - pokud opravdu nevíte že potřebujete zachytit všechny
 - okolní kód může být schopen na výjimku reagovat lépe

ZACHYCENÍ VŠECH VÝJIMEK `CATCH (...)`

- V některých případech je naopak velmi vhodné zachytávat všechny výjimky
 - výjimky by neměly být propagovány mimo váš modul
 - není dána přesná realizace objektu výjimky, záleží na překladači
 - kód přeložený jiným může spadnout při ošetření
- Obalení těla funkce `main()`
 - zabránění neřízenému pádu programu
- Obalení těla destruktoru
 - destruktory by neměl nikdy vyvolat výjimku (kdo by ošetřil?)
- Obalení těla callback funkce
 - výjimka z naší callback funkce jde do kontextu uživatele callbacku

CATCH (...) PRO LOGOVÁNÍ

- Využívá se např. v kombinaci se znovuvyvoláním pro uložení logu o problému

```
int main() {  
    try {  
        foo();  
    }  
    catch (...) {  
        cerr << "error in " << __FILE__ << __LINE__;  
        throw;  
    }  
    return 0;  
}
```


NEZACHYCENÁ VÝJIMKA

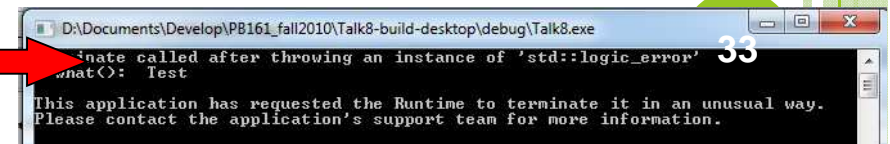
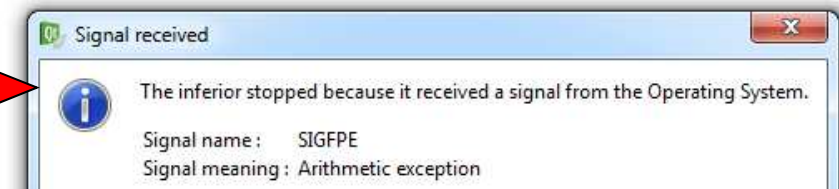
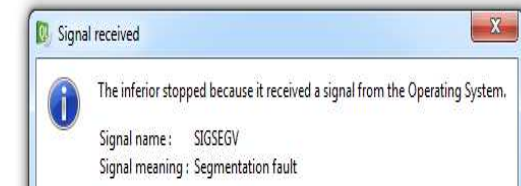
- Může nastat když
 - neobsluhujeme výjimky vůbec
 - nastane mimo blok s obsluhou výjimky
- Následuje ukončení programu

```
#include <stdexcept>

int main() {
    // 1. Segmentation fault (SIGSEGV)
    int array[10];
    array[100] = 1;

    // 2. Division by zero (SIGFPE)
    float result = 10 / 0;

    // 3. Own uncoght exception
    throw std::logic_error("Test");
    return 0;
}
```



SEZNAM VYVOLÁVANÝCH VÝJIMEK

- Můžeme specifikovat seznam výjimek, které daná metoda může vyvolat (*exception-specification*)
 - informace pro uživatele metody (co hlídat a zkoušet odchyťovat)
 - informace pro překladač (možné optimalizace, **ale viz. dále**)
- Pokud metoda vyvolá jinou výjimku než deklarovanou
 - je zavolána funkce `std::unexpected()`
 - defaultní implementace přeruší program
 - můžeme upravit `set_unexpected(naše_obslužná_fce)`
- Pozor na odlišnost od Javy!
 - u Javy kontroluje překladač (nedovolí vyvolávat jinou výjimku)
 - u C++ kontroluje až běhové prostředí (pomocí `unexpected()`)
- Problematické, spíše nepoužívejte
 - viz. diskuze <http://www.gotw.ca/publications/mill22.htm>
 - vývojáře nekontroluje (až za běhu), překladači téměř nepomáhá

SEZNAM VYVOLÁVANÝCH VÝJIMEK - UKÁZKA

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

void foo(int what) throw(int, MyException) {
    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");
}

// Note: will compile even when foo()
// has different exception-specification
void foo2(int what) throw(int) {
    foo(what);
}

void handleUnexpected () {
    std::cerr << "unexpected exception thrown";
    // throws exception with int type
    //(allowed in exception-specification)
    throw -1;
    // ... or e.g., terminate
    //std::terminate();
}
```

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

int main() {
    std::set_unexpected(handleUnexpected);

    cout << "Code before exception handling" << endl;

    try {
        foo2(3); // throw MyStructException

        cout << "Will not be printed";
    }
    catch (int ex) {
        cout << "Integer exception: " << ex << endl;
    }
    catch (MyStructException& ex) {
        cout << "Struct exception : " << ex.reason;
    }

    cout << "Continuing after block with EH";

    // We may reset unexpected handler to default
    std::set_unexpected(std::terminate);

    return 0;
}
```

VHODNOST POUŽITÍ – OPAKOVANÁ OBSLUHA CHYB

- Výjimky mohou výrazně omezit podmíněný kód pro obsluhu chyby
 - kód s podmínkami obsahuje nejčastěji chybu
 - výjimky omezují množství kódu s podmínkami
 - musí však být používány správně (viz. zneužívání)
- Při použití výjimek nemusíme mít kód pro vyhodnocení a předání chyby v každé funkci

VHODNOST POUŽITÍ VÝJIMEK - KONSTRUKTORY

- Chybu v konstrukturu nelze předat návratovou hodnotou
 - žádná není 😊
 - typicky chybné argumenty
- Lze řešit nějakou validační metodou “isValid()”
 - může být nepohodlné
 - musíme testovat každý vytvářený objekt
- Při chybě v konstrukturu lze vyhodit výjimku
 - viz. prázdné jméno u třídy Person
 - typicky instance `std::invalid_argument` nebo její potomek
 - **throw** `std::invalid_argument("Invalid empty name");`

VHODNOST POUŽITÍ VÝJIMEK - OPERÁTORY

- Chyby při použití operátorů
 - operátory typicky nevrací chybový status
 - návratová hodnota operátoru nemusí být použitelná
 - např. operátor sčítání může způsobit přetečení výsledku
- Některé operátory nastavují “fail” bit
 - např. >> a << pro iostream
 - může být nutné testovat po každé operaci - nepraktické
- Při chybě v operátoru lze vyhodit výjimku
 - např. při detekci přetečení u sčítání
 - **throw** `std::overflow_error("+ result overflow");`

VÝJIMKY PRO I/O OPERÁTORY

```
int x;  
cin >> x;  
if (cin.fail()) { }
```

```
#include <iostream>  
  
int main() {  
    std::cin.exceptions( std::istream::failbit);  
    try {  
        int i;  
        std::cin >> i;  
        std::cout << "Succesfully read: " << i << std::endl;  
    }  
    catch (std::istream::failure & ex) {  
        std::cout << "Exception: " << ex.what() << std::endl;  
    }  
  
    return 0;  
}
```

(NE)VHODNOST POUŽITÍ VÝJIMEK – DEALOKACE

- Výjimka může způsobit *memory leak*
 - kód s dealokací díky výjimce neproběhne
- Lze řešit kombinací zachycení a znovuvypuštění
 - `catch (...) { delete[] data; throw; }`
- Lze řešit použitím `auto_ptr`, `unique_ptr`
 - vhodnější, můžeme deklarovat i alokovat v podblocích

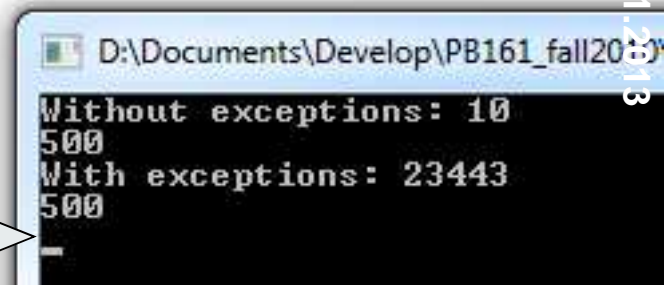
ZNEUŽÍVÁNÍ A NADUŽÍVÁNÍ VÝJIMEK

- Zachytávání, ale neošetřování výjimek
 - `catch (...) { /* do nothing */ }`
 - zachytíme vše a nereagujeme
- Nadužívání výjimek
 - např. nadměrné testování přístupu k poli pomocí `at()`

bez výjimky

s výjimkou

```
std::vector<int> myVect(100, 5);  
float result = 0;  
  
for (unsigned int i = 0; i < 10000000; i++) {  
    if (i < myVect.size()) result += myVect[i];  
}  
for (unsigned int i = 0; i < 10000000; i++) {  
    try {  
        result += myVect.at(i);  
    }  
    catch (std::out_of_range& ex) {  
        // do nothing  
    }  
}
```



D:\Documents\Develop\PB161_fall2013
Without exceptions: 10
500
With exceptions: 23443
500
-

PROBLÉM S DEFAULTNÍM DESTRUKTOREM

- Může nastat při vytváření vlastní výjimky jako potomek exception
 - *error: looser throw specifier for ...*
 - pozor, ne všechny překladače hlásí
- Nastává, pokud u třídy výjimky definujeme atributy s vlastním destruktorem
 - např. `std::string` pro uložení dalších informací
 - atribut může způsobit vyvolání výjimky, kterou předek v `throw` deklaraci nespecifikuje
- Více viz.
<http://www.agapow.net/programming/cpp/looser-throw-specifier>

DALŠÍ INFORMACE K VÝJIMKÁM

- Motivace a pravidla pro vhodnost použití výjimek
 - <http://www.parashift.com/c++-faq-lite/exceptions.html>
- Princip výjimek a vhodnost (např. RAII)
 - <http://www.gamedev.net/reference/articles/article953.asp>

STANDARDNÍ VÝJIMKY

- `bad_alloc` – při dynamické alokaci (typicky nedostatek paměti)
- `bad_cast` – chybné přetypování
- `bad_exception` – vyvolaná výjimka není v seznamu deklarovaných výjimek
- `bad_typeid` – výjimka vyvolaná funkcí `typeid`
- `ios_base::failure` – problém

VÝJIMKY - UKÁZKY

- `exceptionTypeDemo.cpp`
 - zachycení výjimek různých typů
- `exceptionDemo.cpp`
 - zachycení obecnější výjimky před specifitější (chyba)
- `exceptionReThrowDemo.cpp`
 - ukázka znovuvyvolání výjimky
- `unhandledException.cpp`
 - ukázka neošetřené výjimky
- `exceptionSpeedDemo.cpp`
 - ukázka nadužívání výjimek
- `exceptionSpecifDemo.cpp`
 - ukázka použití specifikace vyvolávaných výjimek

SHRNUTÍ

- Výjimky jsou nástroj pro usnadnění ošetření chyb
 - lze využít dědičnosti
 - pozor na nadužívání výjimek
- Functor
 - objekt využitelný namísto funkčního ukazatele