Ošetření chyb a výjimky

Karel Richta a kol.

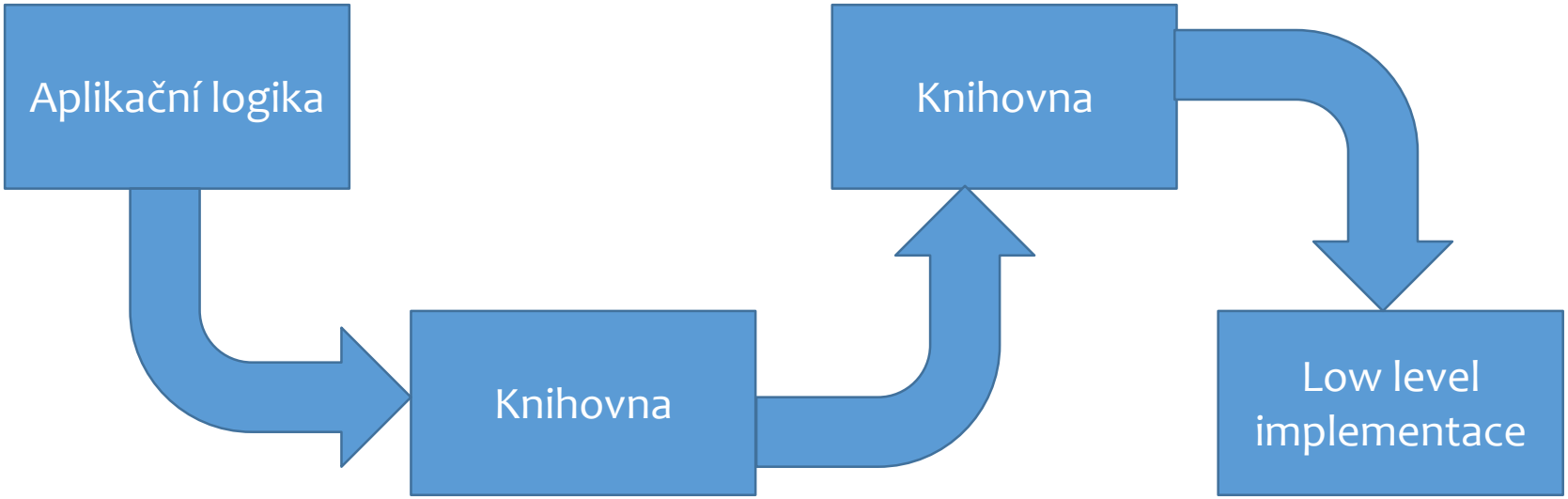
katedra počítačů FEL ČVUT v Praze

Programování v C++, B6B36PJC
11/2020, Lekce 9a

https://cw.fel.cvut.cz/wiki/courses/b6b36pjc/start

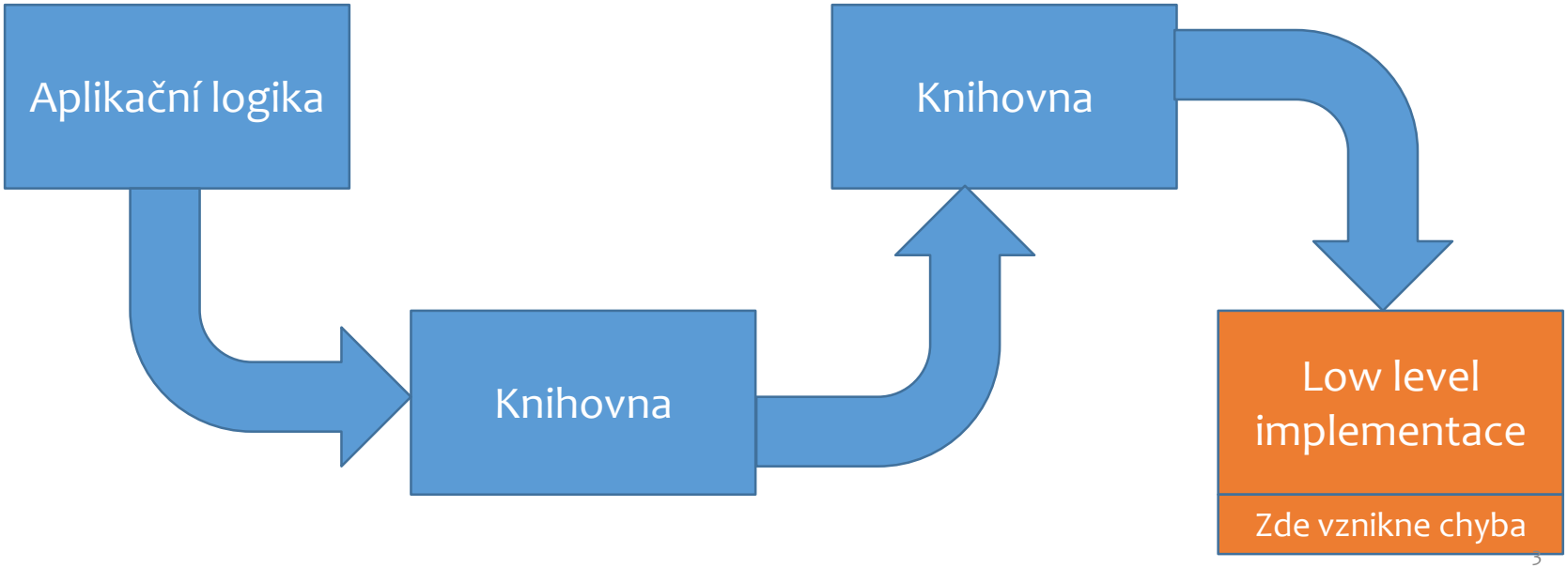
Problém

- Při běhu programu mohou vznikat různé chyby.
  - Například nemusíme mít práva ke čtení/zápisu souboru.
- Tyto chyby je potřeba nějak ošetřit.
- Místo, které si umí s chybou poradit, je obvykle jiné, než místo, kde chyba vznikla.



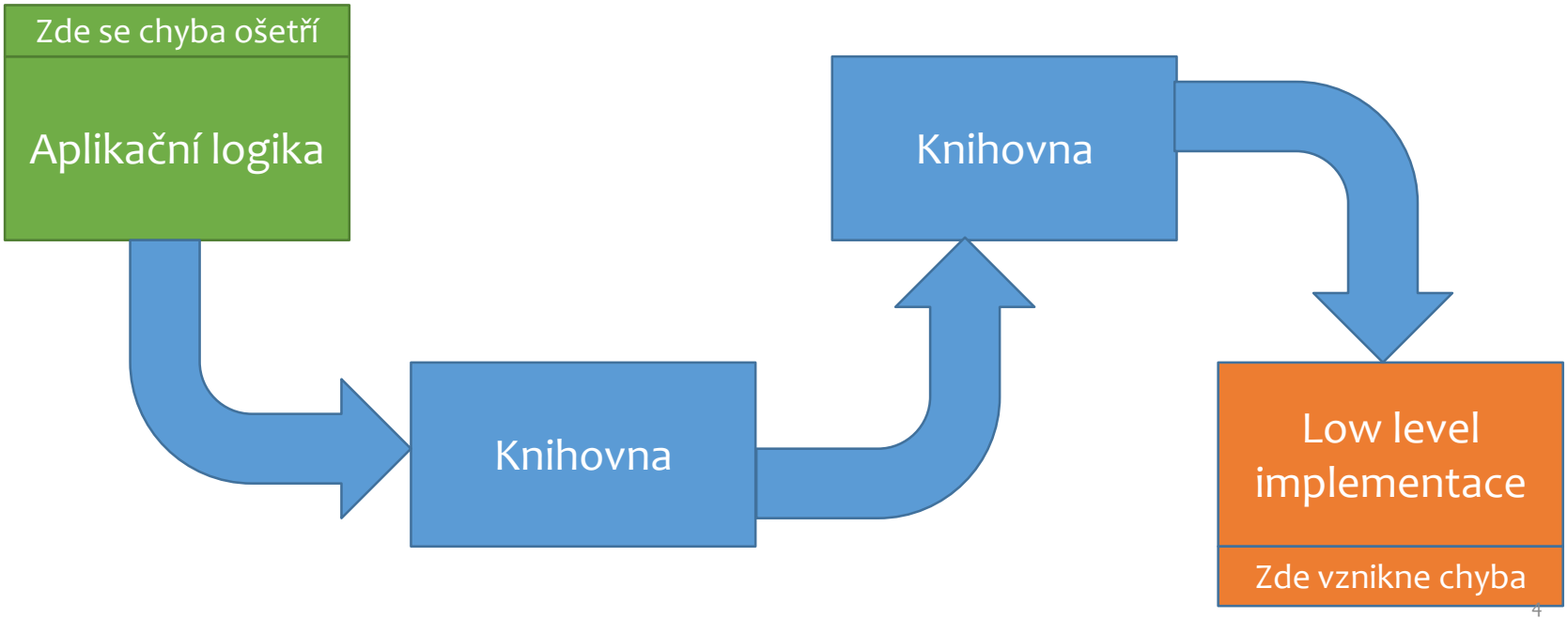
Problém

- Při běhu programu mohou vznikat různé chyby.
  - Například nemusíme mít práva ke čtení/zápisu souboru.
- Tyto chyby je potřeba nějak ošetřit.
- Místo, které si umí s chybou poradit, je obvykle jiné, než místo, kde chyba vznikla.



Problém

- Při běhu programu mohou vznikat různé chyby.
  - Například nemusíme mít práva ke čtení/zápisu souboru.
- Tyto chyby je potřeba nějak ošetřit.
- Místo, které si umí s chybou poradit, je obvykle jiné, než místo, kde chyba vznikla.



Důsledky

- K řešení chyb je tedy potřeba přenést informace o chybě z nižších vrstev do vyšších.
- Existují 3 různé způsoby:
  - Návratové hodnoty
  - Chybové příznaky
  - Výjimky
- Ve standardní knihovně C++ se vyskytují všechny tři.
  - Části převzaté z C obvykle používají návratové hodnoty.
  - Matematické funkce používají chybové příznaky.
  - STL používá výjimky.

Řešení pomocí chybových příznaků

- Základní schema:

```
int chyba = 0; // globální chybový příznak

void f(parametry) {
        // část kódu, kde může nastat chyba
        // pokud nastane:
        // – nastaví se chybový příznak na dohodnutou nenulovou hodnotu
        // - přeruší se normální zpracování a funkce se ukončí
        // pokud chyba nenastane, nic se neděje, příznak se nenastaví
};

int main() {
        f(skutečné parametry);
        if (chyba) { zpracování chyby }
        // normální zpracování
}
```

Chybové příznaky

- Funkce, která narazí na chybu, ji zapíše do globální proměnné.
- Funkce, která se chce o chybě dozvědět, zkontroluje zapsanou hodnotu.
- Chybové příznaky se používají v matematické knihovně a v POSIX standardu.
  - Pokud `log` narazí na chybu, třeba negativní vstup, zapíše do `errno` hodnotu `EDOM`.
  - Funkce volající log má následně možnost `errno` přečíst.
- Nemění signaturu funkce.

```
int main() {
    double foo = std::log(-1.23);
    if (errno == EDOM) {
        std::cout << "log() has failed because: " << std::strerror(errno) << '\n';
    }
}
```

```
>>> log() has failed because: Numerical argument out of domain
```

Chybové příznaky – knihovna errno.h

- Definuje makro **`errno`** – lze si představit jako deklaraci proměnné **`int errno = 0;`**
- Obsahuje kód poslední zjištěné chyby (0 znamená OK).
- Nejméně tři konstanty - kódy: EDOM (domain error), ERANGE (range error) a EILSEQ (illegal sequence).
- Pokud některá funkce narazí na chybu, např. log na negativní vstup, zapíše do errno kód chyby, zde např. hodnotu EDOM.
- Funkce volající log má následně možnost errno přečíst.

```
enum class errc;
```

http://en.cppreference.com/w/cpp/error/errc

Chybové příznaky – Problémy

- Chyby mohou být ignorovány.
  - Programátor musí vynaložit úsilí, aby ignorovány nebyly.
- Není jisté, kdy a kde k chybě došlo.
  - Pokud po volání funkce nezkontrolujeme `errno`, už se nedá určit, zda k chybě došlo.
  - Ani není možné zjistit, jestli se chyb nevyskytlo více.
- Je to spousta kódu, která se špatně čte a špatně píše.
  - Volání funkce následuje mnoho řádků, které čtou `errno` a reagují na chybu.
  - Funkce se dají skládat, ale pouze za cenu možného přehlédnutí chyby.

```
double sumlogs(const std::vector<double>& numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    return result;
}
```

Chybové příznaky – Problémy

- Chyby mohou být ignorovány.
  - Programátor musí vynaložit úsilí, aby ignorovány nebyly.

```
double sumlogs(const std::vector<double>& numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    return result;
}

double foo(const std::vector<double>& numbers) {
    return std::exp(sumlogs(numbers));
}
```

- Když zavoláme `std::log` s neplatným parametrem, nastaví se `errno`. To samé platí o `std::exp`.
- Došlo k chybě?
  - Nevíme.

Chybové příznaky – Problémy

- Není jisté, kdy a kde k chybě došlo.
  - Pokud po volání funkce nezkontrolujeme `errno`, už se nedá určit, zda k chybě došlo.
  - Ani není možné zjistit, jestli se chyb nevyskytlo více.

```
// Toto už zkontroluje chybu, ale neví kde nastala
double sumlogs(const std::vector<double>& numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    if (errno == EDOM) {
        // handle error
    }
    return result;
}
```

- Chybu kontrolujeme, ale nevíme, kdy k ní došlo.
  - V tomto případě to nevadí, ale jindy může.

Chybové příznaky – Problémy

- Je to spousta kódu, která se špatně čte a špatně píše.
  - Volání funkce následuje mnoho řádků, které čtou `errno` a reagují na chybu.
  - Funkce se dají skládat, ale pouze za cenu možného přehlédnutí chyby.

```
double sumlogs(const std::vector<double>& numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    if (errno == EDOM) {
        std::cerr << "Doslo k chybe v sumlogs: EDOM\n";
        return NAN;
    }
    return result;
}

double foo(const std::vector<double>& numbers) {
    auto temp = sumlogs(numbers);
    if (errno == EDOM || isnan(temp)) {
        std::cerr << "foo chyba: suma logaritmu skoncila chybou\n";
        return NAN;
    }
    auto res = std::exp(temp);
    if (errno == ERANGE) {
        std::cerr << "foo chyba: vysledek je prilis velky.\n";
        return NAN;
    }
    return res;
}
```

Zvětšeno na příštím snímku

Chybové příznaky – Problémy

```
double sumlogs(const std::vector<double>& numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    if (errno == EDOM) {
        std::cerr << "Doslo k chybe v sumlogs: EDOM\n";
        return NAN;
    }
    return result;
}

double foo(const std::vector<double>& numbers) {
    auto temp = sumlogs(numbers);
    if (errno == EDOM || isnan(temp)) {
        std::cerr << "foo chyba: suma logaritmu skoncila chybou\n";
        return NAN;
    }
    auto res = std::exp(temp);
    if (errno == ERANGE) {
        std::cerr << "foo chyba: vysledek je prilis velky.\n";
        return NAN;
    }
    return res;
}
```

Chybové příznaky – Problémy

```
double sumlogs(const std::vector<double>& numbers) {
    double result = 0;
    for (auto n : numbers) {
        result += std::log(n);
    }
    if (errno == EDOM) {
        std::cerr << "Doslo k chybe v sumlogs: EDOM\n";
        return NAN;
    }
    return result;
}

double foo(const std::vector<double>& numbers) {
    auto temp = sumlogs(numbers);
    if (errno == EDOM || isnan(temp)) {
        std::cerr << "foo chyba: suma logaritmu skoncila chybou\n";
        return NAN;
    }
    auto res = std::exp(temp);
    if (errno == ERANGE) {
        std::cerr << "foo chyba: vysledek je prilis velky.\n";
        return NAN;
    }
    return res;
}
```

Řešení pomocí návratových hodnot

- Základní schema:

```
int f(parametry) {
        // část kódu, kde může nastat chyba
        // pokud nastane:
        // - přeruší se normální zpracování a funkce se ukončí
        // - návratová hodnota indikuje chybu
        // pokud chyba nenastane, funkce vrátí hodnotu 0
};

int main() {
        if (f(skutečné parametry)) { zpracování chyby }
        // normální zpracování
}
```

Řešení pomocí návratových hodnot

- Funkce, které vrací numerickou hodnotu (tradičně `int`)
  - Návratová hodnota je pak určuje chybu, ke které došlo při průběhu funkce.
  - Hodnoty značící chybu (nebo úspěch) jsou předem definovány.
- Funkce často přijímají i tzv. výstupní parametr.
- Navrácené hodnoty z funkce se pak vrací výš.

```
int main() {
    printf("Kolik cisel bude nasledovat?\n");
    int n;
    int res = scanf("%d", &n);
    if (res != 1) {
        printf("To neni cislo\n");
        exit(1);
    }
    ...
}
```

Návratové hodnoty – Problémy

- Chyby mohou být ignorovány.
  - Programátor musí vynaložit úsilí, aby ignorovány nebyly.
- Pokud někdy ignorujete chybu, její hlášení je navždy ztraceno.
- Je to spousta kódu, která se špatně čte a špatně píše.
  - Vrácená hodnota z každé funkce se musí uložit, zkontrolovat, vyřešit a případně vrátit z funkce.
  - Funkce se nedají skládat.

```
int main() {
    printf("Kolik cisel bude nasledovat?\n");
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) {
        ...
    }
}
```

Návratové hodnoty – Problémy

- Chyby mohou být ignorovány.
  - Pokud někdy ignorujete chybu, její hlášení je navždy ztraceno.

```
int nacti_cislo() {
    int temp;
    scanf("%d", &temp);
    return temp;
}

int main() {
    printf("Kolik cisel bude nasledovat?\n");
    int n = nacti_cislo();
    ...
}
```

- Pokud se nepovedlo načíst číslo, v n je podivná hodnota.
  - Zbytek programu o tom neví.

Návratové hodnoty – Problémy

- Chyby mohou být ignorovány.
  - Programátor vynaložit úsilí, aby ignorovány nebyly.

```
int nacti_kladne_cislo() {
    int temp;
    int res = scanf("%d", &temp);
    if (res != 1) {
        return -1;
    }
    if (temp < 0) {
        return -2;
    }
    return temp;
}
```

- Už nejsme schopni vrátit všechny možné hodnoty datového typu `int`.
  - Musíme vybrat, které hodnoty obětujeme.

Návratové hodnoty – Problémy

- Je to spousta kódu, který se špatně čte i píše.
  - Vrácená hodnota z každé funkce se musí uložit a zkontrolovat; v případě chyby musíme správně zareagovat.
  - Funkce se nedají skládat.

```
int nacti_kladne_cislo() {
    int temp;
    int res = scanf("%d", &temp);
    if (res != 1) {
        return -1;
    }
    if (temp < 0) {
        return -2;
    }
    return temp;
}

int main() {
    printf("Kolik cisel bude nasledovat?\n");
    int cislo = nacti_kladne_cislo();
    if (cislo < 0) {
        if (cislo == -2) {
            printf("Potrebuju KLADNE cislo.\n");
        }
        if (cislo == -1) {
            printf("Potrebuju cislo.\n");
        }
        exit(1);
    }
    ...
}
```

Zvětšeno na příštím snímku

Návratové hodnoty – Problémy

```
int nacti_kladne_cislo() {
    int temp;
    int res = scanf("%d", &temp);
    if (res != 1) {
        return -1;
    }
    if (temp < 0) {
        return -2;
    }
    return temp;
}

int main() {
    printf("Kolik cisel bude nasledovat?\n");
    int cislo = nacti_kladne_cislo();
    if (cislo < 0) {
        if (cislo == -2) {
            printf("Potrebuju KLADNE cislo.\n");
        }
        if (cislo == -1) {
            printf("Potrebuju cislo.\n");
        }
        exit(1);
    }
    ...
}
```

Návratové hodnoty – Problémy

```
int nacti_kladne_cislo() {
    int temp;
    int res = scanf("%d", &temp);
    if (res != 1) {
        return -1;
    }
    if (temp < 0) {
        return -2;
    }
    return temp;
}

int main() {
    printf("Kolik cisel bude nasledovat?\n");
    int cislo = nacti_kladne_cislo();
    if (cislo < 0) {
        if (cislo == -2) {
            printf("Potrebuju KLADNE cislo.\n");
        }
        if (cislo == -1) {
            printf("Potrebuju cislo.\n");
        }
        exit(1);
    }
    ...
}
```

Řešení pomocí výjimek

- Základní schema:

```
int main() {
    try {
            // část kódu, kde může nastat chyba
            // pokud nastane – způsobí změnu ve zpracování, tzv. „hodí výjimku“
            // - přeruší se normální zpracování, hledá se ovladač této výjimky
            // - výjimka je identifikována pomocí typu
            // pokud chyba nenastane, nic se neděje
    } catch (… typ výjimky…) {
        // zpracování výjimky daného typu
    } catch (… jiný typ výjimky…) {
        // zpracování výjimky jiného typu
    }
    // další případné ovladače

}
```

Výjimky

- Chyba nemůže být ignorována.
  - Ignorování chyby ukončí proces.
- Funkce, která hlásí chybu, hodí výjimku.
- Funkce, která umí chybu zpracovat, ji chytá.
- Funkce se dají skládat.

```
int main() {
    try {
        while (true) {
            new int[100000000ul];
        }
    } catch (const std::bad_alloc& e) {
        std::cout << "Allocation failed: " << e.what() << '\n';
    }
}
```

Výjimky – použití

- Při hození výjimky dojde k přerušení normálního běhu programu.
- Příkazy následující po hození výjimky nejsou provedeny.
  - Destruktory objektů na zásobníku ale zavolány jsou.

```
void throws() {
    std::vector<int> vec;
    while (true) {
        std::string str;
        throw 1;
        baz(); // Nezavolá se
    } // str je destruováno
}     // vec je destruováno

int main() {
    throws();
    doesnt(); // Nezavolá se
}
```

```
void throws() {
    std::vector<int> vec;
    while (true) {
        std::string str;
        throw 1;
        baz(); // Nezavolá se
    } // str je destruováno
}     // vec je destruováno

int main() {
    try {
        throws();
        doesnt(); // Nezavolá se
    } catch (int) {
        std::cout << "Vyjimka chycena.\n";
    }
    doesnt(); // Zavolá se
}
```

Výjimky – použití

- Při hození výjimky dojde k přerušení normálního běhu programu.
- Příkazy následující po hození výjimky nejsou provedeny.
  - Destruktory objektů na zásobníku ale zavolány jsou.

```
void throws() {
    std::vector<int> vec;
    while (true) {
        std::string str;
        throw 1;
        baz(); // Nezavolá se
    } // str je destruováno
}     // vec je destruováno

int main() {
    throws();
    doesnt(); // Nezavolá se
}
```

```
void throws() {
    std::vector<int> vec;
    while (true) {
        std::string str;
        throw 1;
        baz(); // Nezavolá se
    } // str je destruováno
}     // vec je destruováno

int main() {
    try {
        throws();
        doesnt(); // Nezavolá se
    } catch (int) {
        std::cout << "Vyjimka chycena.\n";
    }
    doesnt(); // Zavolá se
}
```

Výjimky – házení

- Výjimku hodíme pomocí klíčového slova `throw`.
- Hodit se dá libovolný typ/objekt.
- Obvykle se ale hází nějaký potomek `std::exception`.
- Hozená výjimka přeruší normální běh programu a dojde k tzv. „stack unwinding“. Je to proces, při kterém se postupně ruší objekty na zásobníku a hledá se odpovídající catch blok.
- Pokud není nalezen vhodný catch blok, dojde k ukončení programu.

Nedoporučujeme	Doporučujeme

`throw 1;`
`throw "chyba";`
`std::vector<int> vec;`
`throw &vec;`
`throw std::vector<int>{};`
	`throw std::runtime_error("ooops\n");`
	`throw std::system_error(std::error_code{}, "argh\n");`
	`throw std::future_error(std::future_errc::broken_promise);`

Výjimky – chytání

- Výjimka se chytá pomocí klíčového slova `catch()`.
- Chytat se dá libovolný typ.
- Chytání je polymorfní – pokud chytáme třídu, chytáme i její potomky.
- Kód v catch bloku se provede pouze, pokud je dotyčná výjimka zachycena.

Nedoporučujeme	Doporučujeme

`catch (...) {}  // Chytne cokoliv`
`catch (int) {}  // Chytne throw 1;`
`catch (char) {} // Chytne throw 'a';`
`catch (int*) {} // Chytne throw &<int>`
	`catch (const std::exception& ex) {}  // Chytne exception a potomky`
	`catch (const std::bad_alloc& ex) {}  // Chytne bad_alloc a potomky`
	`catch (const std::logic_error& ex) {} // Chytne logic_error a potomky`

Výjimky – Pravidla použití

- **VŽDY** házejte hodnotou.
- **VŽDY** chytejte referencí (pokud můžete, `const&`)
- Používejte své výjimky, které dědí z nějaké standardní výjimky.
- Nepoužívejte `catch(...)`.

Výjimky – házení, chytání a polymorfismus

- Jak jsme si řekli, chytání výjimek je polymorfní.
- Házení ale není.
- Co to znamená?

```
class B {};
class D : public B {};

void f(B& throwable) {
    throw throwable;
}

int main() {
    D status;
    try {
        f(status);
    } catch (D& e) {
        std::cout << "Chytil jsem D!\n";
    } catch (...) {
        std::cout << "To je divny...\n";
    }
}
```

```
>>> To je divny...
```

```
class B {
public: virtual void raise() {throw *this;}
};
class D : public B {
public: virtual void raise() {throw *this;}
};
void f(B& throwable) {
    throwable.raise();
}

int main() {
    D status;
    try {
        f(status);
    } catch (D& e) {
        std::cout << "Chytil jsem D!\n";
    } catch (...) {
        std::cout << "To je divny...\n";
    }
}
```

```
>>> Chytil jsem D!
```

Výjimky – Problémy

- Cesta kódem není jasná
  - U žádného řádku si nemůžeme být jisti, kam bude pokračovat exekuce.
- Zpočátku jsou neintuitivní.
  - Popravdě, často i později.
- Dlouho byly používány špatně.
  - Stále často jsou.

```
int* nacti_cisla(int n) {
    int* pole = new int[n];
    for (int i = 0; i < n; ++i) {
        pole[i] = nacti_cislo(); // Tahle funkce rozhodně nehází výjimky. Doufám.
    }
    return pole;
}
```

Ještě k výjimkám - noexcept

- K deklaraci funkce můžeme přidat specifikaci týkající se výjimek **`throw`** případně **`noexcept`** (C++17).
- Tato specifikace není součástí typu funkce a může se použít jen na hlavní úrovni, nebo v lambda funkci (bude probráno později). Nelze ji použít ve specifikaci **`typedef`**.

```
void f() throw(int); // funkce f může hodit výjimku
void f() noexcept; // funkce f() nehází výjimky
void (*fp)() noexcept(false); /* fp ukazuje na
funkci, která může hodit výjimku */
void g(void pfa() noexcept);  /* g má jako parametr
ukazatel na funkci, která nehází výjimky */
// typedef int (*pf)() noexcept; // chyba - nelze
```

Ještě k výjimkám (pokr.)

Samotná třída **`exceptions`** toho využívá:

```
class exception {
public:
  exception () noexcept;
  exception (const exception&) noexcept;
  exception& operator= (const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
}
```

Nevýhodou výjimek je, že spotřebovávají značné množství systémových prostředků.
Proto je nutné užívat jich obezřetně. Rozhodně nenahrazují ošetřování kódu pomocí podmínek!

Příklad:

```
#include <iostream>        // std::cerr
#include <typeinfo>        // operator typeid
#include <exception>       // std::exception

class Polymorphic { virtual void member() {} };

int main() {
    try
    {
        Polymorphic* pb = 0;
        typeid(*pb);  // throws a bad_typeid exception
    }
    catch (std::exception& e)
    {
        std::cerr << "exception caught: " << e.what() << '\n';
    }
    return 0;
}
```

Děkuji za pozornost.