Exception safety

Karel Richta a kol.

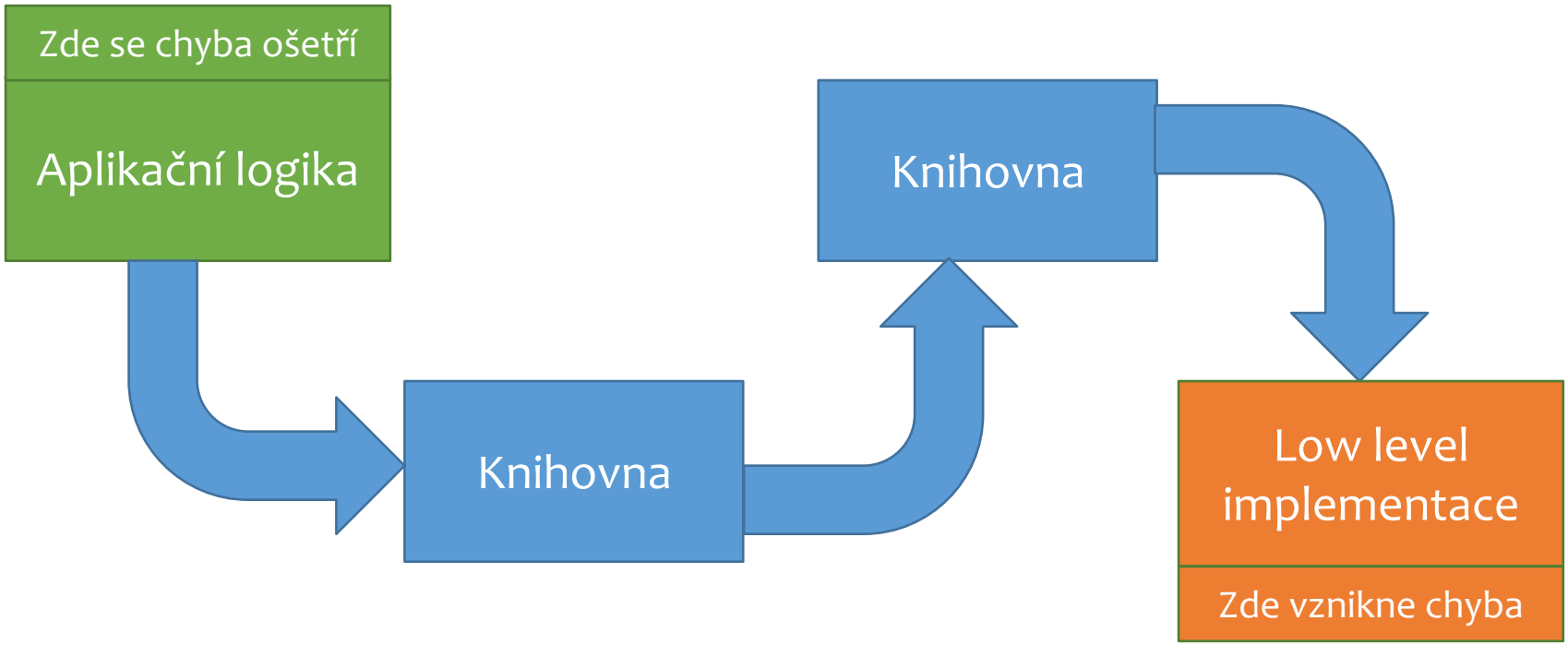
katedra počítačů FEL ČVUT v Praze

Programování v C++, B6B36PJC
1/2019, Lekce xx

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

Opakování

- Jak jsme si říkali, chyby se obvykle ošetřují jinde, než nastanou.



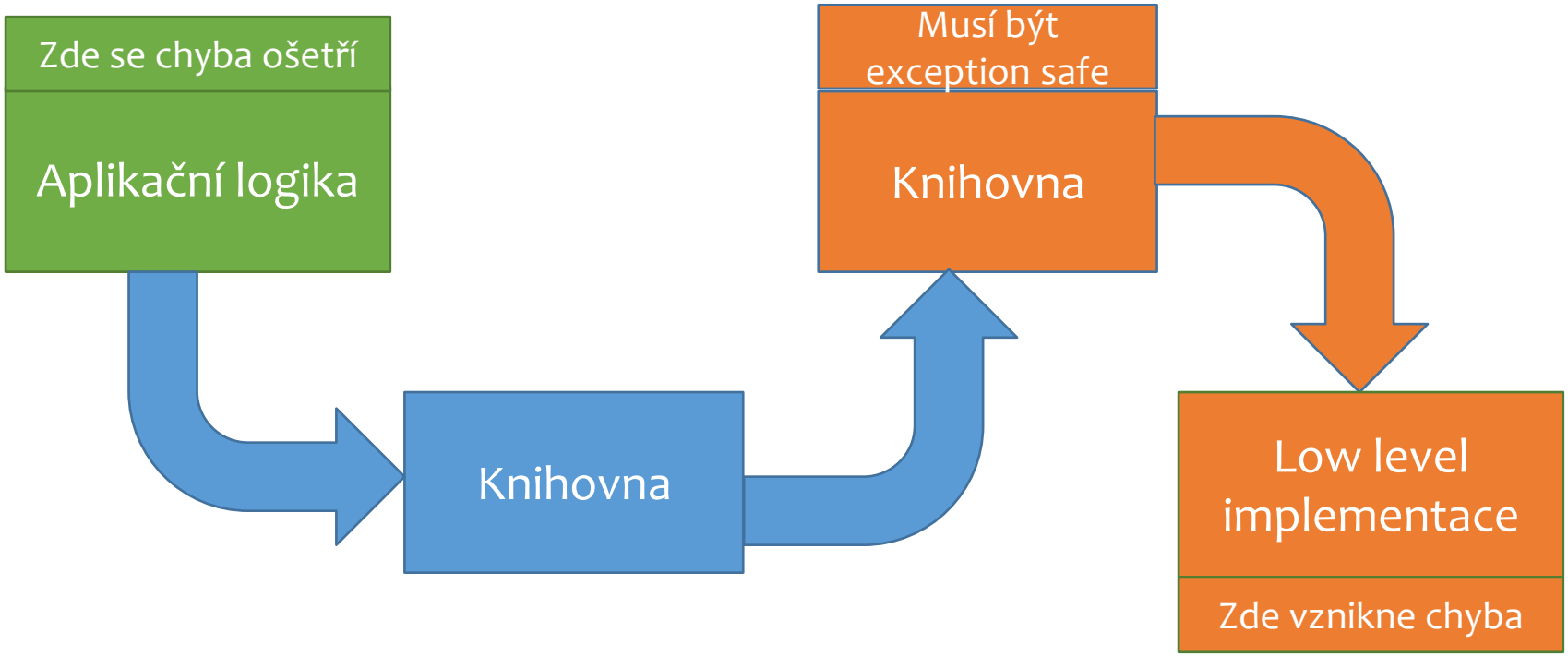
„Těžká část práce s výjimkami není explicitní zpracování, ale implicitní. Je to ta část, kde je potřeba psát všechen kód tak, aby arbitrární hozená výjimka mohla být chycena, aniž by její průchod kódem navodil další chybové stavy.“

Volný překlad, Tom Cargill

Exception Handling: False Sense of Security

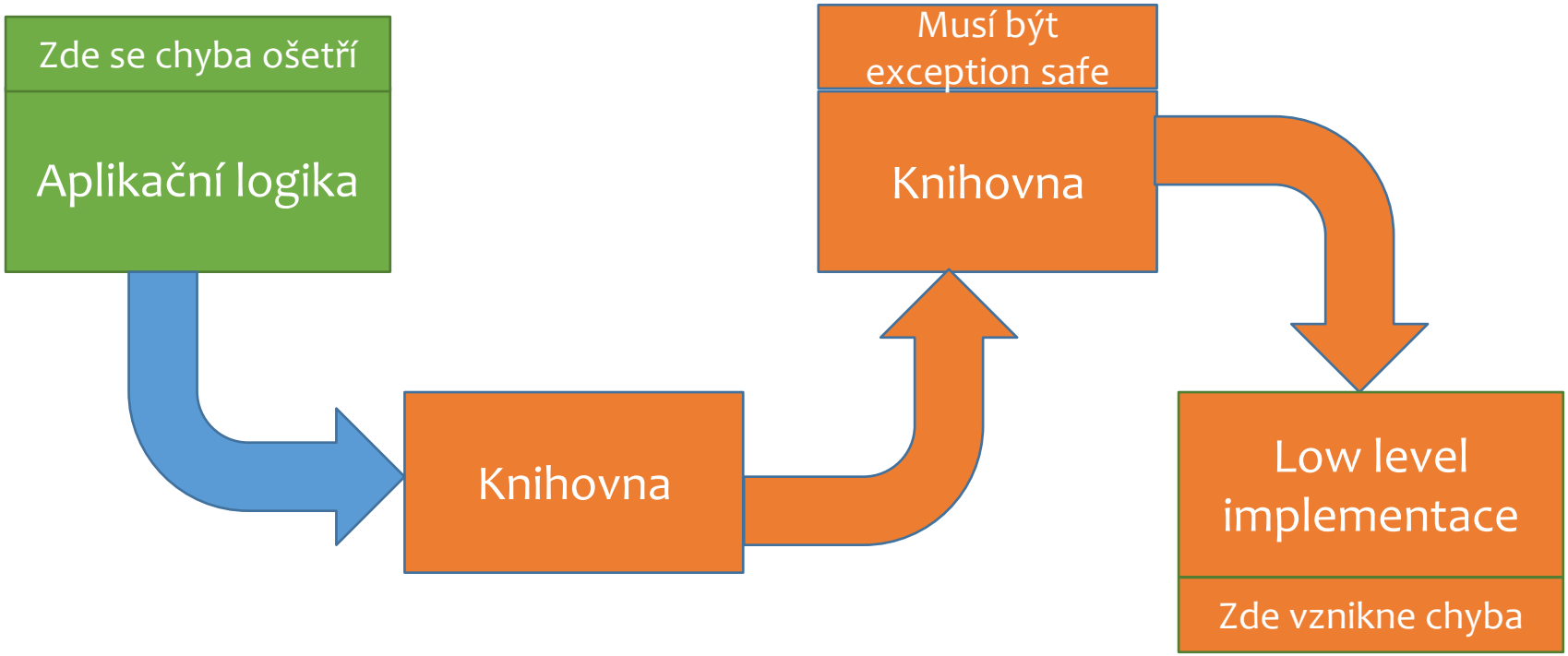
Důsledek výjimek

- Aby výjimky mohly fungovat, všechen kód, kterým projde výjimka musí být tzv. „Exception Safe“.



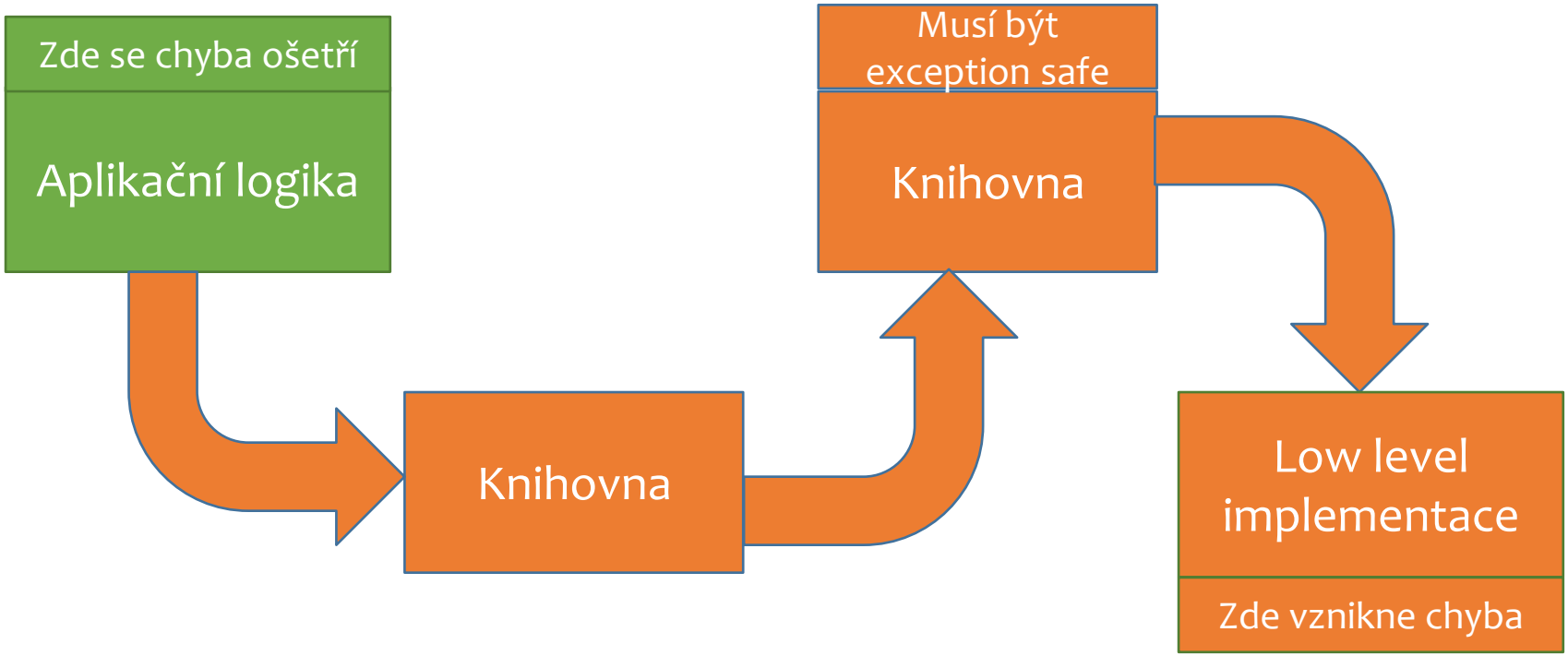
Důsledek výjimek

- Aby výjimky mohly fungovat, všechen kód, kterým projde výjimka musí být tzv. „Exception Safe“.



Důsledek výjimek

- Aby výjimky mohly fungovat, všechen kód, kterým projde výjimka musí být tzv. „Exception Safe“.



Řešení chyb

- Řešení chyb je těžké vždy
  - Jediné snadné řešení je napodobit pštrosa.

- Vrstvy mezi aplikační logikou a low level kódem musí být uzpůsobeny ošetřování chyb.
  - Pokud správně nepoužívají chybové přiznaky, zavlečou spoustu dalších chyb.
  - Pokud správně nepoužívají a nevracejí návratové hodnoty, je to ještě horší.
  - Pokud správně neřeší výjimky, dojde ke ztrátě zdrojů. (resources)

Řešení chyb (2)

- Každý způsob řešení chyb vyžaduje uzpůsobení rozhraní funkcí.
  - Chybové příznaky zavádějí globální stav a neumožňují paralelizaci
  - Návratové hodnoty vedou k out parameterům
  - Výjimky mění rozhraní funkcí méně nápadným způsobem

Terminologie (Abrahams)

Existují 3 různé úrovně záruky bezpečnosti.

- **Základní (Basic Guarantee)**
  - Pokud dojde k výjimce, nedojde ke ztrátě zdrojů nebo k porušení invariant.
- **Silná (Strong Guarantee)**
  - Pokud dojde k výjimce, nedojde ke změně interního stavu.
- **Neházející (No-throw Guarantee)**
  - K výjimce nemůže dojít.

Žádná vs Základní záruka

```
int* read_numbers(int n, std::istream&
in) {
    int* numbers = new int[n];
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        in >> numbers[i];
        sum += numbers[i];
    }

    return numbers;
}
```

```
std::unique_ptr<int[]>
read_numbers(int n, std::istream& in) {
    auto numbers = std::make_unique<int[]>(n);
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        in >> numbers[i];
        sum += numbers[i];
    }

    return numbers;
}
```

Práce s výjimkami – předpoklady

- Každá funkce poskytuje alespoň základní záruku vůči výjimkám.
- Pokud funkce neposkytuje alespoň základní záruku, nelze pokračovat.
  - Takováto funkce může přesunout program do špatného vstupu bez ohledu na naše akce.
- Každá funkce poskytuje POUZE základní záruku vůči výjimkám.
  - Pokud nedeklaruje jinak.

Práce s výjimkami – předpoklady

- Každá funkce může vyhodit výjimku.
  - Pokud nedeklaruje jinak.
- Destruktor, swap a přesunující akce jsou no-throw.
  - Házející swap a přesunující akce jsou ospravedlnitelné, destruktory nejsou.
- Operace nad primitivními typy výjimky nehází
  - Sčítání, odčítání, přiřazování...

Jak psát bezpečný kód?

- Vždy kontrolujte návratové hodnoty.
- Zjistěte, které funkce mohou hodit výjimku, a rozmyslete si, jak na ně reagovat.
- Používejte specifikace výjimek.
- Používejte `try{} catch(){}` bloky.

Jak psát bezpečný kód?

- ~~Vždy kontrolujte návratové hodnoty.~~
- ~~Zjistěte, které funkce mohou hodit výjimku, a rozmyslete si, jak na ně reagovat.~~
- ~~Používejte specifikace výjimek.~~
- ~~Používejte `try{} catch(){}` bloky.~~
- Vždy zachovejte invarianty.
  - I v případě chyby.
- Rozdělte kód na část, která může selhat a která selhat nemůže.
  - V části která selhat může se připravují změny.
  - V části která selhat nemůže se změny ukl

Transakce?

Jak psát bezpečný kód?

- Rozdělte kód na část, která může selhat a která selhat nemůže.

V části, která selhat může, se změny připravují.

```
list& operator=(const list& rhs) {
    list temp(rhs);
    temp.swap(*this);
    return *this;
}
```

V části, která selhat nemůže, se změny ukládají.

Všeobecná pravidla

- VŽDY poskytujte základní záruku.
- Silnou záruku poskytujte pouze tam, kde je to „přirozené".
  - Pouze tam, kde se dá implementovat bez zbytečné pesimizace výkonu.
- Zdokumentujte všechny funkce se silnou zárukou.
- Implementujte neházející destruktory!
- Implementujt
  - A označte je j

```
class list {
public:
    list(list&& rhs) noexcept;
    list& operator=(list&& rhs) noexcept;
    void swap(list& rhs) noexcept;
    ...
};
```

Základní záruka

- Základní záruku nabízejí všechny RAII třídy.
- Zároveň platí, že pokud všechny prostředky spravujete pomocí RAII, máte základní záruku ve funkci zdarma.
- Pozor, pokud ji funkce neposkytuje, nelze ji doimplementovat!

Silná záruka

- Mnoho funkcí umožňuje implementovat silnou záruku „zdarma“.
  - Zdarma z hlediska běhu programu, ne bez úsilí programátora.

- Zpravidla se jedná o případ, kdy se funkce dotkne pouze jednoho zdroje.
  - Například: `vector::push_back`

Silná záruka (2)

- Neměla by se ale implementovat za každou cenu
  - `vector::insert` se dá implementovat se silnou zárukou, ale pouze za cenu zpomalení běhu programu
- Uživatel si ji téměř vždy může doimplementovat sám.
- Funkce se silnou zárukou připomínají ACID transakce.

Příklad – push_back

```
void push_back(const T& t) {
    if (size < capacity) {
        // Pokud konstruktor hodí výjimku, nic se nestane
        new (&buffer[size]) T(t); // Může hodit výjimku
        size++;                   // Nemůže hodit výjimku
        // Výsledek: silná záruka zdarma
    } else {
        // Viz dále
        ...
    }
}
```

Příklad – push_back

```
void push_back(const T& t) {
    if (size < capacity) {
        // Již jsme viděli
        ...
    } else {
        // Připravíme si větší kapacitu
        auto temp = std::make_unique<T[]>(capacity * 2);

        // Nakopírujeme staré prvky
        std::copy(buffer.get(),
                  buffer.get() + size,
                  temp.get());

        new (&temp[size]) T(t); // Přidáme nový prvek
        swap(temp, buffer);     // Přebereme nový buffer
        size++;                 // Nemůže hodit
        capacity *= 2;          // Nemůže hodit
    }
}
```

Cargillův zásobník

- V roce 1994 Tom Cargill prošel implementaci zásobníku a identifikoval v něm množství chyb.
- Implementace byla převzata z časopisu *C++ Report*, kde byla publikována jako modelová implementace.
- Na závěr vyzval širší komunitu, aby přišla s implementací zásobníku, která je bezpečná vůči výjimkám.
- Projdeme si nyní tuto implementaci společně.
  - Schválně kolik najdete chyb.
    - Kód je starý.
      - Velmi, velmi starý.

Cargillův zásobník – kód

Deklarace

```
template <class T>
class Stack {
    unsigned nelems;
    int top;
    T* v;

public:
    unsigned count();
    void push(T);
    T pop();

    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
};
```

- Základní implementace, kterou známe.
- Pointer v na pole prvků, `nelems` kapacita pole a `top`, index aktuálního prvku.
- Funkce umožňující sebrat prvek z vrcholu zásobníku, vložit prvek a zjistit kolik je v zásobníku prvků.
- Sada základních operací typu: kopírující operace, destruktor, konstruktor.

Cargillův zásobník – kód (2)

Implementace

```
template <class T>
Stack<T>::Stack() {
    top = -1;
    v = new T[nelems = 10];
    if (v == 0)
        throw "out of memory";
}

template <class T>
Stack<T>::~Stack() {
    delete[] v;
}

template <class T>
unsigned Stack<T>::count() {
    return top + 1;
}
```

- V dnešním C++ operátor new při nepodařené alokaci hodí výjimku, takže vlastní házení není potřeba.
- Jinak zde není žádný problém.

Cargillův zásobník – kód (3)

```
template <class T>
void Stack<T>::push(T element) {
    top++;
    if (top == nelems - 1) {
        T* new_buffer = new T[nelems += 10];
        if (new_buffer == 0)
            throw "out of memory";
        for (int i = 0; i < top; i++)
            new_buffer[i] = v[i];
        delete[] v;
        v = new_buffer;
    }
    v[top] = element;
}
```

```
template <class T>
T Stack<T>::pop() {
    if (top < 0)
        throw "pop on empty stack";
    return v[top--];
}
```

Cargillův zásobník – řešení

- Zásobník nelze opravit bez změny rozhraní

Stará deklarace

```
template <class T>
class Stack {
    unsigned nelems;
    int top;
    T* v;

public:
    unsigned count();
    void push(T);
    T pop();
    ...
};
```

Nová deklarace

```
template <class T>
class Stack {
    unsigned nelems;
    int top;
    T* v;

public:
    unsigned count();
    void push(T);
    void pop();
    T& top();
    ...
};
```

Cargillův zásobník – kód (4)

```
template <typename T>
Stack& Stack::operator=(const Stack& s) {
    delete[] v;
    v = new T[nelems = s.nelems];
    if (v == 0)
        throw "out of memory";
    if (s.top > -1) {
        for (top = 0; top <= s.top; top++)
            v[top] = s.v[top];
        top--;
    }
    return *this;
}
```

Cargillův zásobník – kód (4)

```
template <typename T>
Stack::Stack(const Stack& s) {
    v = new T[nelems = s.nelems];
    if (v == 0)
        throw "out of memory";
    if (s.top > -1) {
        for (top = 0; top <= s.top; top++)
            v[top] = s.v[top];
        top--;
    }
}
```

Rekapitulace – Základy

- Házejte výjimky hodnotou, chytejte referencí.
- Používejte RAII.
- Správné pořadí operací je užitečnější než try – catch.
- Rozmyslete si jaké záruky chcete/potřebujete
- Používejte noexcept, tam kde to dává smysl.
- Nepište házející destruktory.
- Umožněte dostat objekt do výchozího stavu.

Rekapitulace – Pokročilé

- Úklid (cleanup) by vždy měl být prováděn destruktorem.
- Implementujte no-throw swap.
- Pokud chcete silnou záruku, rozdělte kód na dvě části.
  - Neimplementujte silnou záruku tam kde je to drahé
- Preferujte výjimky vůči ostatním způsobům řešení chyb.
- Pozor při tvorbě šablon.
- Uživatel musí být schopen dát dohromady silnou záruku za použití vašeho rozhraní.

Co udělá výjimka v konstruktoru?

- Destruktor proběhne pokud alespoň jeden konstruktor úspěšně skončil.

```
~list() {
    while (head != nullptr) {
        auto* temp = head;
        head = head->next;
        delete temp;
    }
}
```

```
list(const list& rhs):list() {
    for (const auto& el : rhs) {
        push_back(el);
    }
}
```

```
 list(): head{ nullptr },
         tail{ nullptr },
         num_elements{ 0 }{}
```

```
list(const list& rhs) {
    head = nullptr;
    tail = nullptr;
    num_elements = 0;

    for (const auto& el : rhs) {
        push_back(el);
    }
}
```

Co když push_back vyhodí výjimku?

Co udělá výjimka v konstruktoru?

- Destruktor proběhne pokud alespoň jeden konstruktor úspěšně skončil.

Doběhl základní konstruktor

```
list(const list& rhs):list() {
    for (const auto& el : rhs) {
        push_back(el);
    }
}
```

```
list(): head{ nullptr },
        tail{ nullptr },
        num_elements{ 0 }{}
```

Žádný konstruktor nedoběhl

```
list(const list& rhs) {
    head = nullptr;
    tail = nullptr;
    num_elements = 0;

    for (const auto& el : rhs) {
        push_back(el);
    }
}
```

Co když push_back vyhodí výjimku?

Problémy s evaluací argumentů

- Jak jsme si říkali, pořadí evaluace argumentů funkce není dáno.
  - Jediné co je dáno, je pořadí závislostí.

- To znamená, že na první pohled nevinné rozdíly mohou vést k zásadním chybám.

```
void foo(std::unique_ptr<A> a, std::unique_ptr<B> b);
```

Volání

```
// Pozor! Neposkytuje základní záruku
foo(std::unique_ptr<A>(new A),
    std::unique_ptr<B>(new B));
// V pořádku.
foo(std::make_unique<A>(), std::make_unique<B>());
```

Možné pořadí evaluace argumentů

```
foo(std::unique_ptr<A>(new A), std::unique_ptr<B>(new B));
```

```
new A
std::unique_ptr<A>()
new B
std::unique_ptr<B>()
foo
```

```
new A
new B
std::unique_ptr<A>()
std::unique_ptr<B>()
foo
```

```
new B
new A
std::unique_ptr<A>()
std::unique_ptr<B>()
foo
```

```
new B
new A
std::unique_ptr<B>()
std::unique_ptr<A>()
foo
```

```
foo(std::make_unique<A>(), std::make_unique<B>());
```

```
std::make_unique<A>()
std::make_unique<B>()
foo
```

```
std::make_unique<B>()
std::make_unique<A>()
foo
```

Cargillův widget

- Jedná se o další Cargillův "rébus"
- Lze implementovat kopírující přiřazení pro `Widget` se silnou zárukou, pokud libovolná operace na `T1` a `T2` může hodit výjimku?

Interface

```
template <typename T1, typename T2>
class Widget {
public:
// Poskytuje silnou záruku
Widget& operator=(const Widget& rhs);

private:
    T1 t1_;
    T2 t2_;
};
```

Cargillův widget

- Pokud může opravdu každá operace hodit výjimku, a nemůžeme změnit deklaraci `Widgetu`, přiřazení nelze implementovat.
- Pokud je `swap` neházející, přiřazení implementovat lze. Jak jsme již říkali, `swap` by *měl* být neházející.

```
Widget& Widget::operator=(const Widget& rhs) {
    T1 t1_temp = rhs.t1_;
    T2 t2_temp = rhs.t2_;
    using std::swap;
    swap(t2_temp, t2_);
    swap(t1_temp, t1_);
    return *this;
}
```

Cargillův widget

- Řekněmě, že `swap` `T1` a `T2` může hodit výjimku. Co teď?
  - Nejdřív seřvat programátora `T1`, `T2`!
  - Co dál? Spokojit se se základní zárukou?

- Použijeme tzv. „PIMPL“ idiom
  - *pointer to implementation*
  - Vytvoříme novou privátní třídu, do které uložíme T1 a T2.
  - Widget nyní neobsahuje T1 a T2, ale ukazatel ke své soukromé třídě.

Cargillův widget

```
class Widget {
public:
    ...
    Widget& operator=(const Widget& rhs);
private:
    class WidgetImpl;
    std::unique_ptr<WidgetImpl> pimpl_;
};

class Widget::WidgetImpl {
public:
    // ...
    T1 t1_;
    T2 t2_;
};
```

Cargillův widget

- Kopírující přiřazení se nyní dá rozdělit na vytvoření kopie T1 a T2 a následné prohození ukazatelů.
- Protože prohození ukazatelů nemůže hodit vyjímku, máme kopírující přiřazení se silnou zárukou.

```
Widget& Widget::operator=(const Widget& rhs) {
    auto temp = std::make_unique<WidgetImpl>(pimpl_);
    using std::swap;
    swap(temp, pimpl_);
    return *this;
}
```

Děkuji za pozornost.