NPRG 041 – cvičení 3
Programování v C++

Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

mattermost:
https://ulita.ms.mff.cuni.cz/mattermost/ar2324zs/channels/nprg041-cpp-klepl

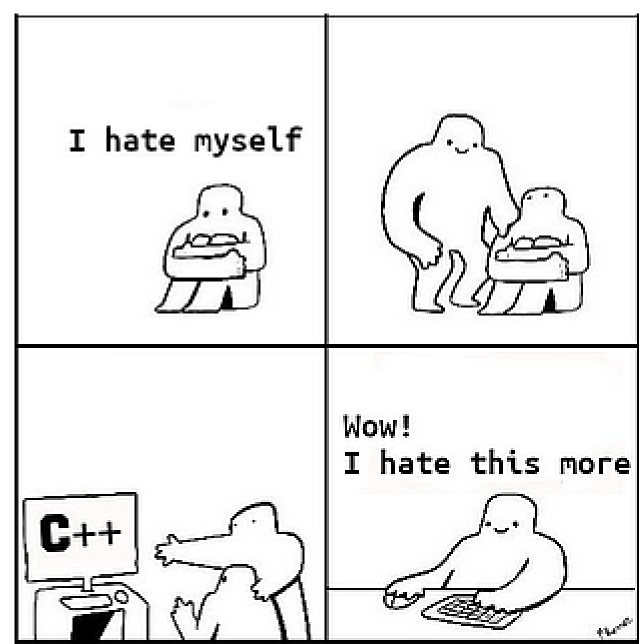
Feedback k úloze Násobilka

Viz nahrávky

Prostor na otázky z přednášky

Agenda

- **Druhá úloha: BST**
  - C-stringy, C++ stringy, std::string_view
  - Streamy (std::fstream a std::stringstream)
  - Memory management a pointery, reference
  - std::move, swap
  - std::tuple, std::pair, std::tie



Binární vyhledávací strom (BST)

- Cílem úkolu je naprogramovat primitivní ekvivalent set<string>
  - to je kontejner, který má v sobě uložený uspořádaný seznam stringů
  - A zachovává reference na hodnoty: když si vyrobíme pointer/iterátor na prvek, tak platí do jeho faktického smazání (pozor, toto např. u vectoru neplatí – dělá realokace)
- Vycházejte z kódu: https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/bst.cpp
- **Zadání**: Program čte stdin/soubor a odpovídá na následující příkazy:
  - `Přidání nového nodu: insert <string>`
  - `Odebrání nodu: remove <string>`
  - `Vytištění stromu: print`
  - `Vyčištění stromu: clear`
- Další detaily v kódu

Odevzdání BST

- Soubory úlohy dejte do složky „bst“ v adresáři „labs“ vašeho repozitáře
  - Necommitujte binární soubory ani žádné jiné vygenerované při sestavování
  - Úloha předpokládá 3 zdrojové soubory, ale součástí mohou být projektové soubory Visual Studia, CMakeLists.txt a nebo README.md
- Doma si nepovinně vypracujte EXTRA TODOs a dále experimentujte
  - Příště se na EXTRA TODO podíváme společně

Remove Node – the most difficult method

- First, **find the deleted node**
  - If it already doesn’t exist, we are done
- Otherwise, perform changes to the tree according to the following three cases
  1) The node has up to one child
  2) The node has two children, and its right child is also its successor
  3) The node has two children, and its right child is not its successor

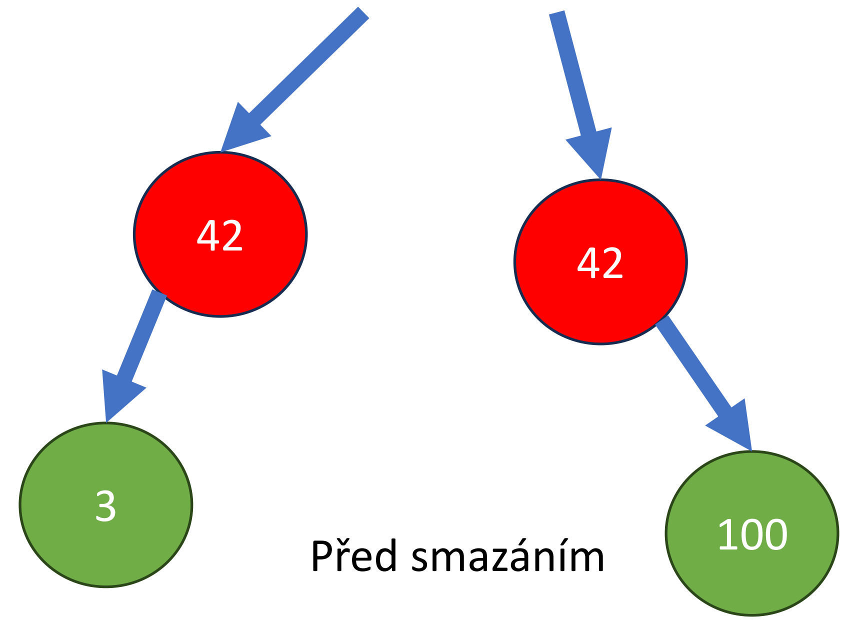
  All three cases are discussed in detail on the following three slides

Remove Node – first two easy cases

- The deleted node has at most 1 child

➔Just replace the node with the child

➔Or `nullptr` if it has none



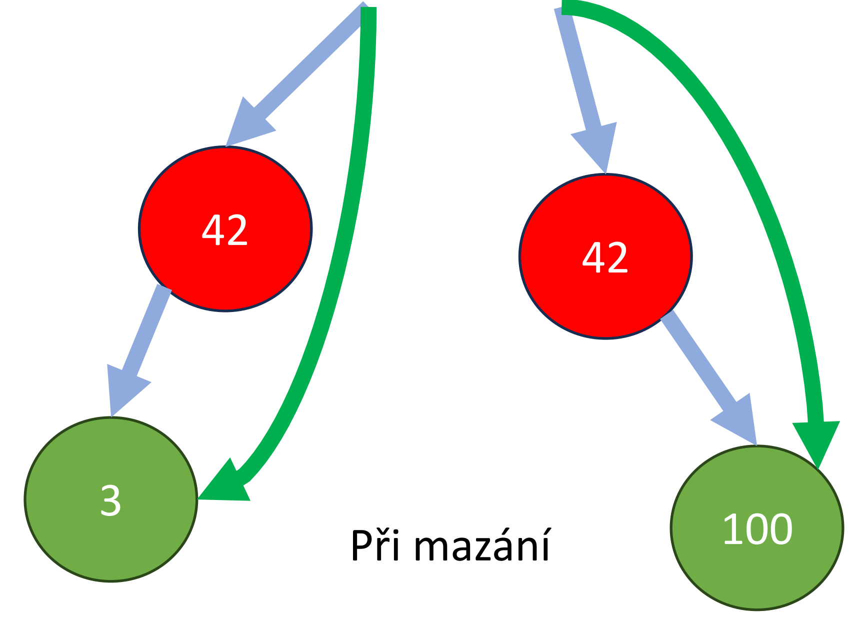
The deleted node
The replacement for the deleted node
Parent-child connection that stays
Parent-child connection that changes
New parent-child connection

Remove Node – first two easy cases

- The deleted node has at most 1 child

➔Just replace the node with the child

➔Or `nullptr` if it has none



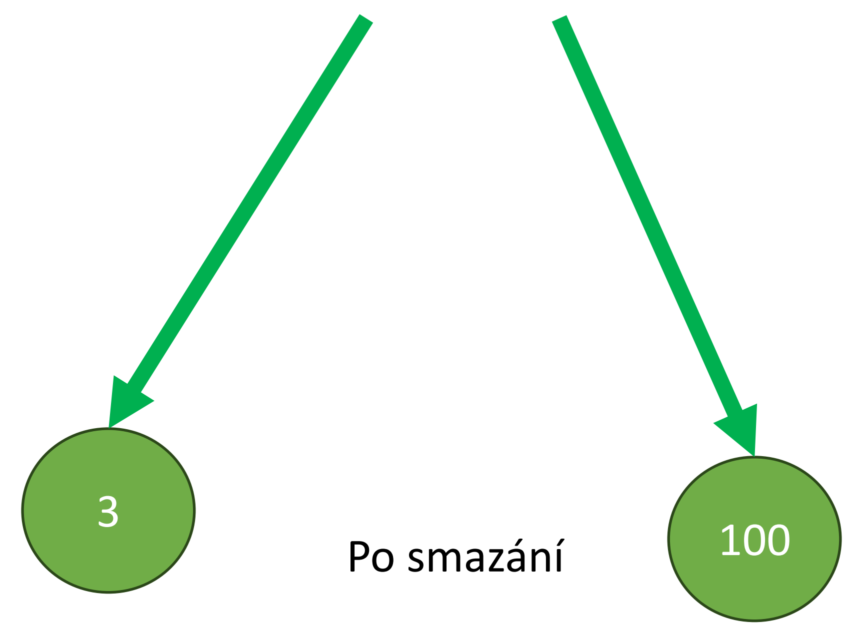
The deleted node
The replacement for the deleted node
Parent-child connection that stays
Parent-child connection that changes
New parent-child connection

Remove Node – first two easy cases

- The deleted node has at most 1 child

➔Just replace the node with the child

➔Or `nullptr` if it has none



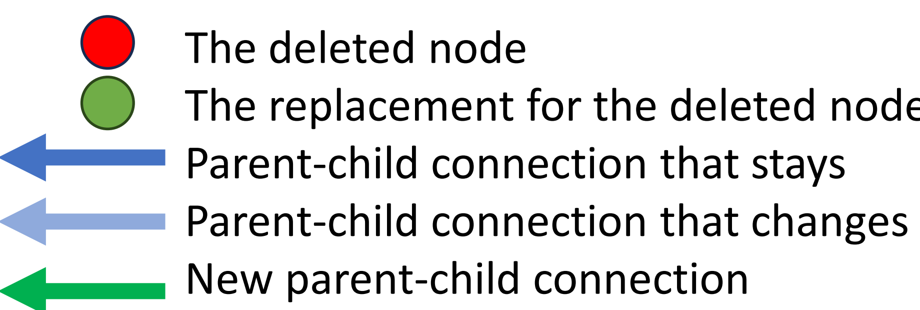
The deleted node
The replacement for the deleted node
Parent-child connection that stays
Parent-child connection that changes
New parent-child connection

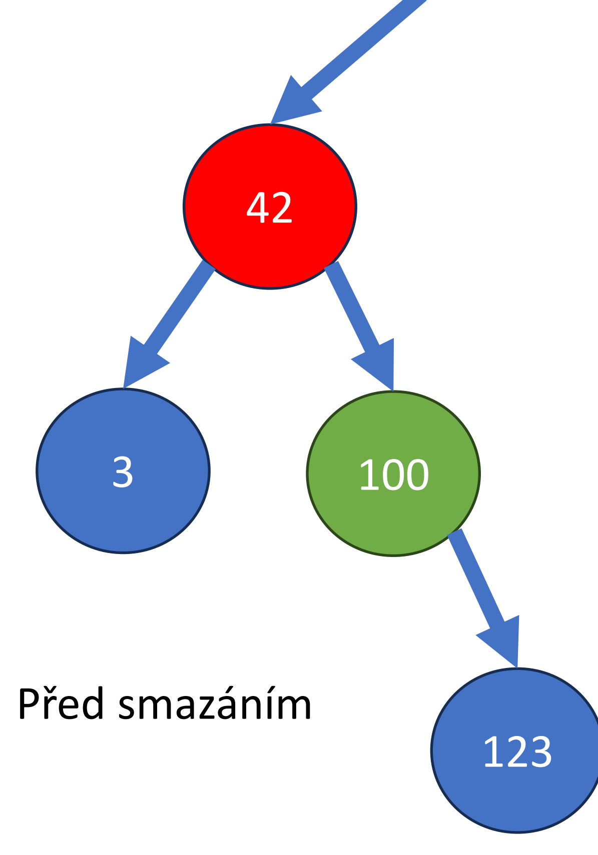
Remove Node – the second case

- The deleted node has two children
  - And the right child is the successor (has no left child)

➔Give the deleted node's left child to the successor

➔Replace the node with the successor



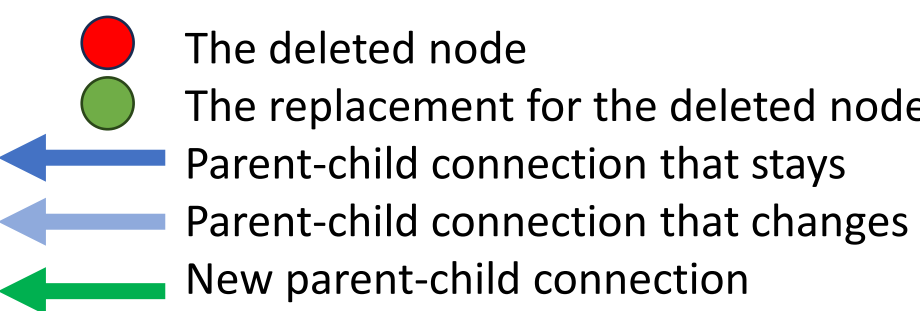


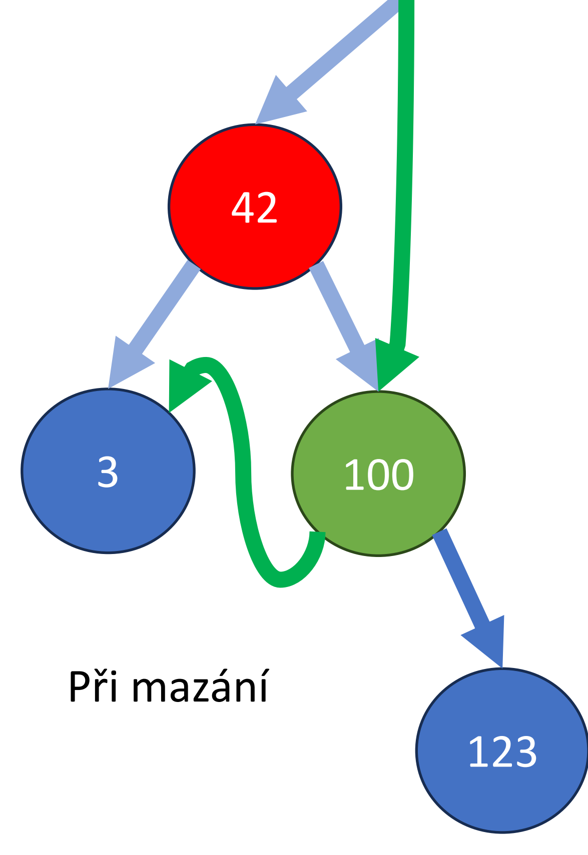
Remove Node – the second case

- The deleted node has two children
  - And the right child is the successor (has no left child)

➔Give the deleted node’s left child to the successor

➔Replace the node with the successor



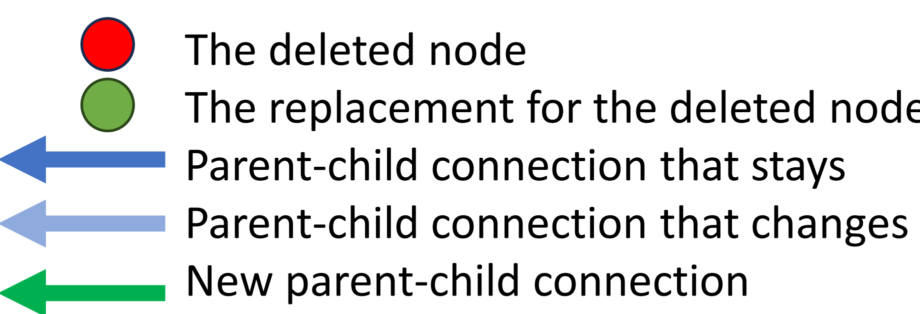


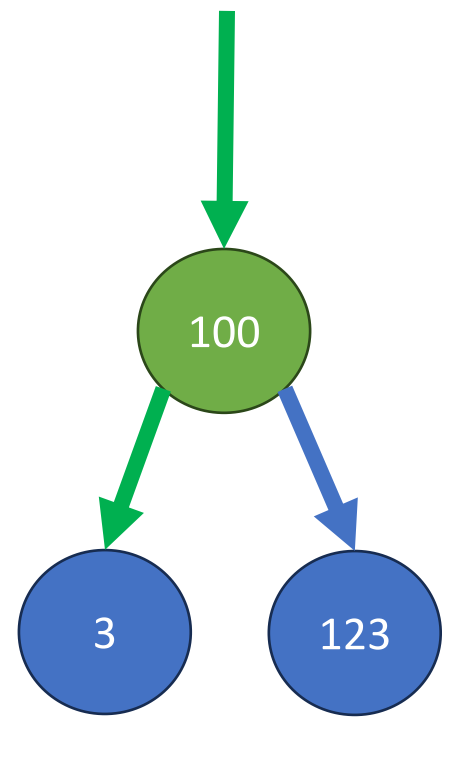
Remove Node – the second case

- The deleted node has two children
  - And the right child is the successor (has no left child)

➔Give the deleted node’s left child to the successor

➔Replace the node with the successor





Remove Node – the scary case

- The deleted node has two children
  - And the right child is not the successor

One solution is:

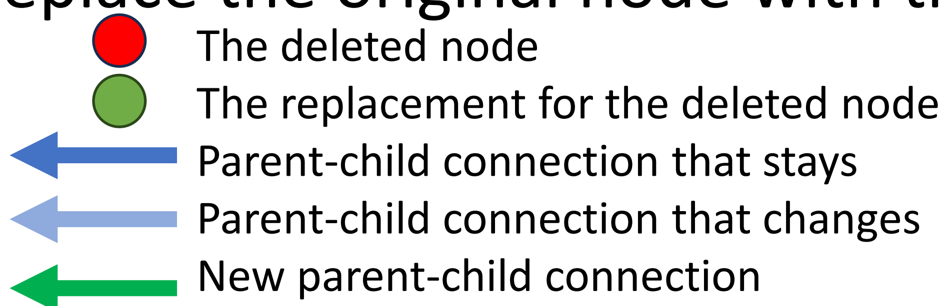
➔Store the successor in a new unique_ptr

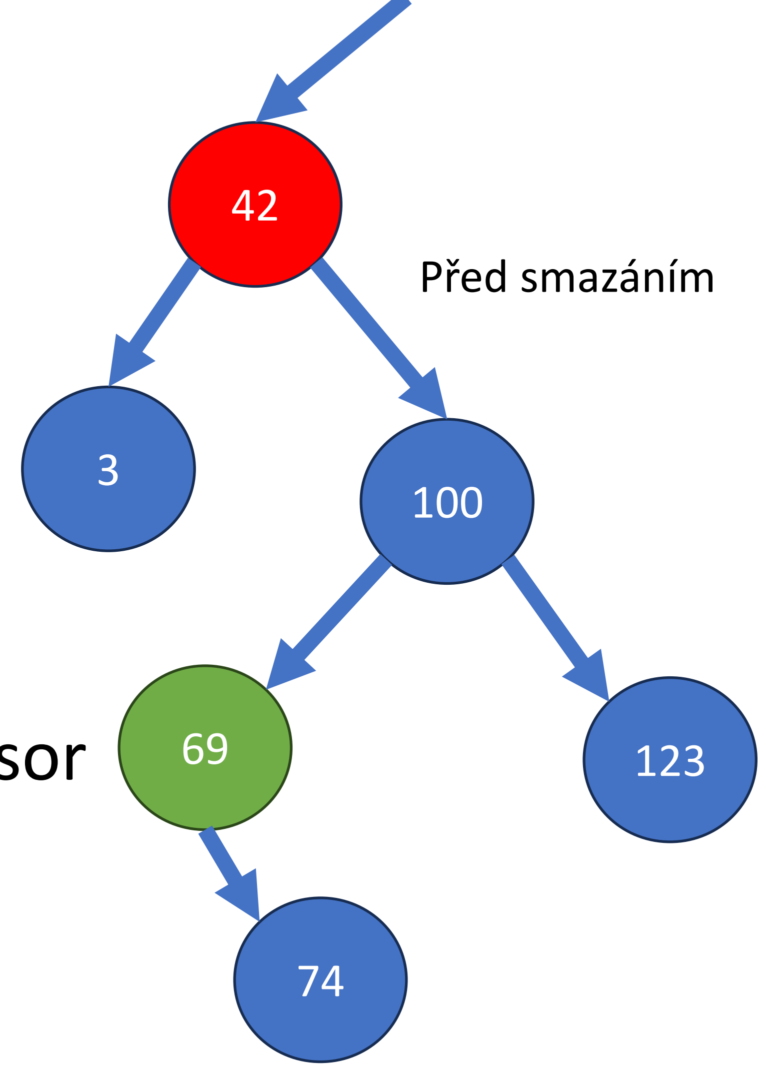
➔Give its right child to its parent as a replacement

➔Then, give it the deleted node's children

(it didn't have left child and we moved the right one)

➔Finally, replace the original node with the successor





Remove Node – the scary case

- The deleted node has two children
  - And the right child is not the successor

One solution is:

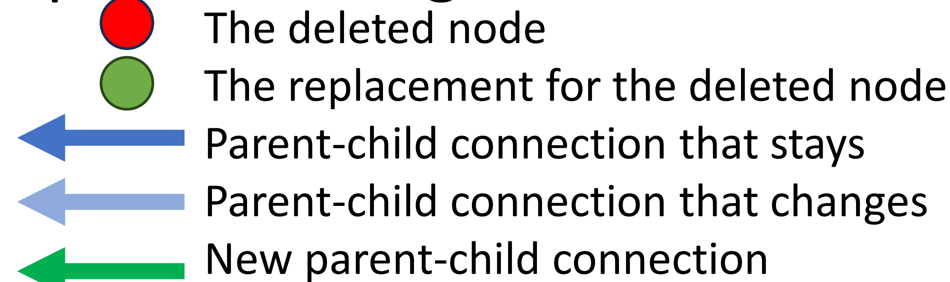
➔Store the successor in a new unique_ptr

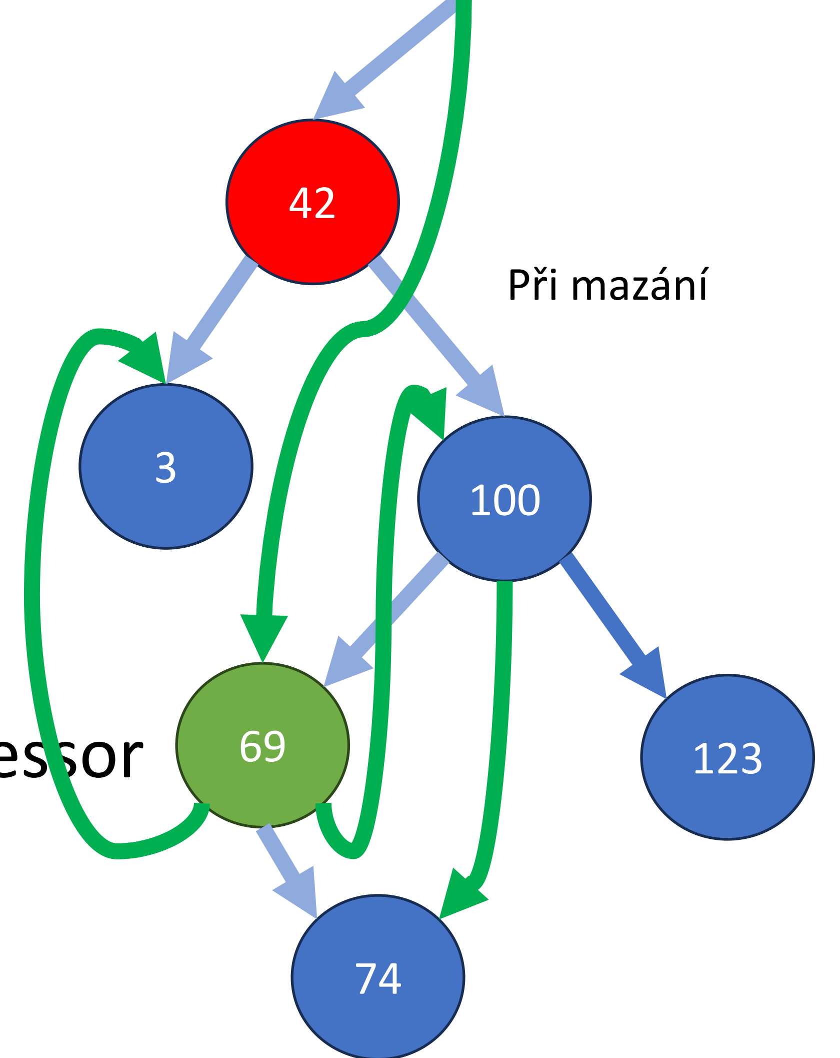
➔Give its right child to its parent as a replacement

➔Then, give it the deleted node's children

(it didn't have left child and we moved the right one)

➔Finally, replace the original node with the successor





Remove Node – the scary case

- The deleted node has two children
  - And the right child is not the successor

One solution is:

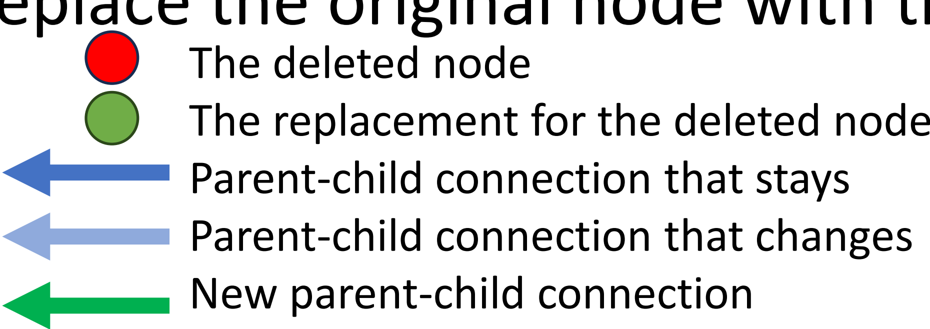
➔Store the successor in a new unique_ptr

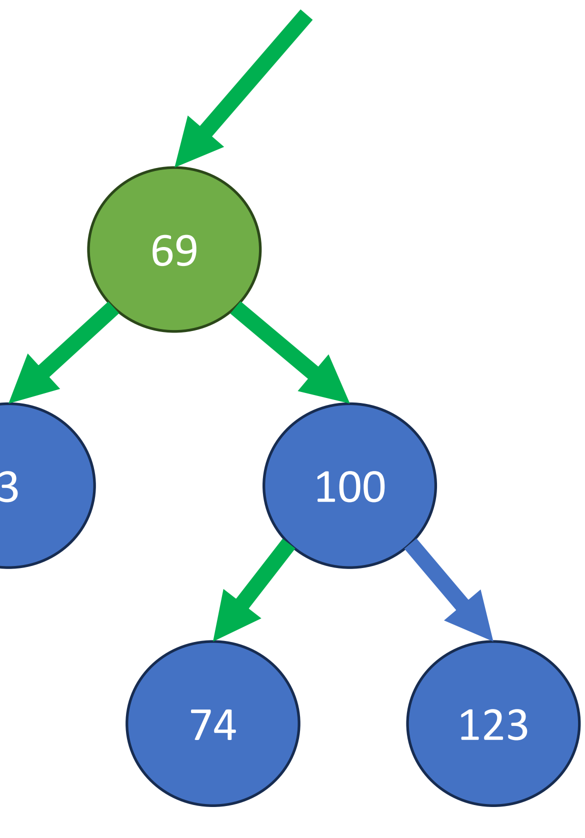
➔Give its right child to its parent as a replacement

➔Then, give it the deleted node's children

(it didn't have left child and we moved the right one)

➔Finally, replace the original node with the successor





Po smazání

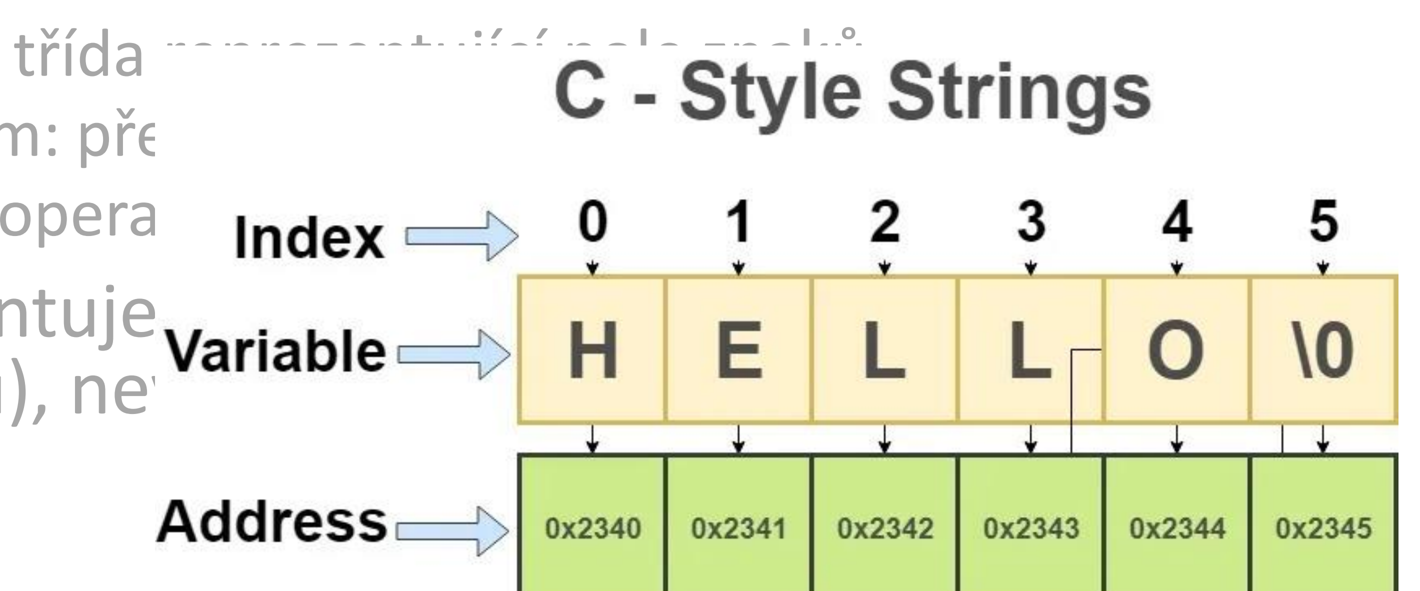
C-string vs std::string vs std::string_view

- C-stringy jsou jen pole bajtů (psáno **char*** nebo **char []**)
  - Jsou zakončeny \0 (většina funkcí brající C-stringy vypisuje dokud nepřečte \0)
  - String literály: např. když v kódu napíšeme jen tak "ahoj", pointery do read-only paměti (vždy const char*)
  - Chovají se jako pointery (operátor == ověří, jestli to je totožný objekt)
  - Při předávání není jasné vlastnictví (kdo/jestli vůbec má to pole uklidit)

```
// string literal (pointer
na pole bajtů, neviditelné \0)
const char *c_string = "hello";

// c-string na stacku (opět končí
\0)
char c_string2[] = "hello";
```

třída
m: pře
opera
ntuje
u), ne



C-string vs std::string vs std::string_view

- C-stringy jsou jen pole bajtů (char* nebo char[], const char*)
  - Jsou zakončeny \0 (většina funkcí brající C-stringy vypisuje dokud nepřečte \0)
  - String literály: např. když v kódu napíšeme jen tak "ahoj", pointery do read-only paměti (vždy const char*)
  - Chovají se jako pointery (operátor == ověří, jestli to je totožný objekt)
  - Při předávání není jasné vlastnictví (kdo má to pole uklidit)
- C++ stringy (std::string) jsou třída reprezentující pole znaků
  - Vlastnictví dáno objektem = když zahodíme string, sám se uklidí
  - Podporují appendování, operator + (ten udělá kopii!), atd.

- std::string_view: třída reprezentující "pohled" na část stringu (ať už C-stringu nebo C++ stringu), nevlastní ho

#include <string_view>

Jak získat ze jména a příjmení celé jméno C++ string vs C string

```
string cele_jmeno( const string& jm, const string& prijm)
{
  return jm + " " + prijm;
}
```

```
int cele_jmeno( char * buf, size_t bufsize,
                    const char * jm,
                    const char * prijm)
{
  size_t lj = strlen( jm);
  size_t lp = strlen( prijm);
  if ( lj + lp + 2 > bufsize )
  { /* error */ return -1; }
  memcpy( buf, jm, lj);
  buf[ lj] = ' ';
  memcpy( buf + lj + 1, prijm, lp);
  buf[ lj + lp + 1] = 0;
  return lj + lp + 1;
}
```

Pitfally při práci se stringy

- Definice funkce: void foo(const std::string &text) { … }
  Volání: foo("Hello")
  - Tohle vypadá nevinně a bude to fungovat (compiler si nebude stěžovat)
  - **SKRYTÁ KOPIE**: z literálu "Hello" vyrobí std::string a ten až předá
  - **Řešení: používat foo(std::string_view text)**
    - Tohle je důvod, proč se string_view zavedly – předá se jen pohled, **žádná kopie**
- Porovnávání C-stringů
  - Už jsme viděli dříve, porovnáme je na **úplnou (referenční) totožnost**
    - Rovnost pointerů
  - Proto je **lepší se C-stringům vyhýbat**

Užitečné funkce

https://en.cppreference.com/w/cpp/string/basic_string/stol

- std::isdigit(c) – rozpozná, zda je znak číslo
  - Dále: isalpha, isalnum, islower, isupper
  - Pro převody: tolower, toupper
- Nikdy nepředpokládejte uspořádání znaků: if (c >= 'a' && c <= 'z’)
  - Naproti tomu, čísla uspořádaná jsou:
    - If (c >= '0' && c <= '9’) je povoleno (ale std::isdigit je lepší)
    - int cifra = c - '0' je povoleno
- Nezapomeňte na starý známý int number = std::stoi(std::string)
  - Lze experimentovat s: std::size_t first_nondigit; std::stoi(s, &first_nondigit)
  - Další varianty: stol, stoul, stoll, stof, stod (podle typu parsovaného čísla)

Streamy

- Už známe základní streamy: std::cin, std::cout, std::cerr
- Další často používané jsou:

#include <fstream>

  - **File streamy**: std::ifstream, std::ofstream, std::fstream
    - Konstruktorem nebo metodou open se otevře soubor (umí různé flagy na nastavení módu, práv)
    - Zavření metodou close nebo zahozením objektu (destruktor)

#include <sstream>

  - **String streamy**: std::istringstream, std::ostringstream, std::stringstream
    - Umí číst ze stringu, zapisovat do stringu, nebo obojí <- podle typu
    - Stringy předávejte přes std::move, abyste předešli kopírování

Počítání Oveček

počítání znaků, řádek, slov a vět v textu; součet

- Program čte stdin (std::cin), pokud nedostane cmd argumenty; jinak pro každý atgument otevře soubor s danou cestou
- Nejspíš využijeme čtení znak po znaku:

std::cin, std::ifstream,...

```
void func(std::istream& in) {
    for (;;) {
        char c = in.get();
        // could be EOF or error:
        if (in.fail()) break;
        process(c);
    }
}
```

Pokus o načtení znaku (nemuselo načíst nic)

Detekce chyb (včetně EOF)

Teď už víme, že c je platný znak

#include <fstream>

```
std::vector<std::string> args(
    argv + 1, argv + argc);
func(std::cin);
for (auto&& arg : args) {
    std::ifstream in(arg);
    if (!in.good()) ...
    func(in);
}
```

Otevření souboru

Detekce chyb

Zde se soubor sám zavře

Požadované kvality kódu

- **Enkapsulace stavu programu v objektech, rozdělení na .cpp moduly**
  - **Přehledné rozhraní (API) v .hpp**,
  - Žádné globální proměnné! – konstanty můžete, ale jen s využitím **constexpr**
- Rozdělení tak, **aby každá funkce (metoda) řešila jeden jasný úkon**
    - Např.: process_char(c), print_results(), read_input(in)
- Pokud má **třída nejasnou zodpovědnost** -> **rozdělení na více tříd**
- **Přehledná implementace**, neopakování výpočtů (nebo kusů kódu)
- **Ladění**: dejte si pozor na okrajové případy
- **Program se zkompiluje bez warningů**

Na co si dát pozor

- **Složité funkce** (nejsou one-linery) implementované **v .cpp souboru**
  - Jednoduché mohou být přímo ve třídě v .hpp hlavičce
- Každý header (nejspíš bude jen jeden) musí mít **header-guard**
- **V headeru nepoužívejte „using namespace std;“** (ani jiný podobný)
  - V .cpp souborech můžete podle svého uvážení
  - Můžete použít i přímo v těle funkce (pak platí jen v ní) – ale to doporučuju jen u „**using namespace std::string_literals**“ – pak je **"hello"s** typu **std::string**
- **Inicializace data fieldů ve třídách pomocí initializer-listů** (ten následuje dvojtečku v konstruktoru; před samotným tělem konstruktoru)

Počítání oveček – upřesnění (okopírováno)

- počet **znaků**, **řádek**, **slov**, **vět**, **počet** a **součet** čísel
- **znaky**: vše včetně mezer, konců řádek apod.
- **slovo:** nejdelší posloupnost alfanumerických znaků nezačínající číslicí
  - neuvažujte diakritiku, resp. všechny speciální (ne-isalnum()) znaky považujte za nepísmena
- **číslo**: posloupnost číslic následující za nealfanumerickým znakem
  - `'.12ab.'` je jedno číslo a žádné slovo
- **řádky:** započítat jen ty, kde je alespoň jedno slovo nebo číslo
  - poslední řádka nemusí být ukončená `'\n'`
- **věta:** neprázdná posloupnost **slov** ukončená oddělovačem
  - oddělovače vět jsou `'.'`, `'!'`, `'?'`
  - `'...'` ani `'31.12.2049'` není několik vět
- spočítat z `cin` nebo ze **všech** souborů uvedených na příkazové řádce
  - žádné číslo/slovo/věta/řádek nejde přes hranici souboru
- dekompozice, objektovost, modularita, efektivita
  - elegantní a efektivní rozhraní třídy pro vstup (data) a výstup (výsledky)
  - separace výpočtu a I/O
- DÚ - **Recodex**
  - opět do půlnoci před následujícím cvičením
  - důkladně otestujte vč. okrajových případů
  - bez warningů

```
znaku: 999
slov: 999
vet: 999
radku: 999
cisel: 999
soucet: 999
```

multiplatformnost

Objekty *reprezentující* referenční sémantiku: Pointery a chytré reference

- Mají value sémantiku jako ostatní věci (bez & v argumentu je kopírujeme)
  - Ale dovolují přístup do hodnot uložených jinde (přes indexace, dereference, atd.)
- Chytré reference na pole
  - Minule jsme viděli std::span, dnes std::string_view
  - Mají API blízké tomu, co sledují: span jako array, string_view jako string
- Pointery a iterátory (nejčastější objekty reprezentující reference)
  - Lze je dereferencovat (operátor *, u metod ->), inkrementovat, odečítat, …
    - Některé iterátory podporují jen část těchto věcí (nebo s různými časovými složitostmi)
- Když před ně napíšeme const, tak ovlivníme jen tu referenci, ne sledovaný objekt (pozor, u pointerů opticky naopak: const T* vs T* const).

  Typicky v C++, pro objekty *nereprezentující* reference, se const propaguje dál:
  mějme: `struct Obj { int a, b; }; const Obj x{42, 7};`
  pak: x.a i x.b jsou také const. (stejně by to bylo u vectoru a jeho prvků)

Pointery (raw vs smart)

- Obecně je dělíme na raw pointery (observer pointery) a smart pointery
  - Raw pointer = např int*, přejato z C
    - Neměly by reprezentovat vlastnictví (používat jen na pozorování)
      - Pokud reprezentují vlastnictví, je potřeba ručně uklidit (free), lze zapomentou/udělat dvakrát
        - Proto je nepoužívejte
    - Příklad definice: int* pointer = &ukazovaná_hodnota
    - Nejsou třída -> nemají metody, nemají destruktor, co by je uklidil
  - Smart pointer = např. unique_ptr<int>
    - Reprezentují vlastnictví = zahození pointeru způsobí úklid
      - Automatický úklid předchází chybám a usnadňuje programování
    - Oproti raw pointerům nepřidávají významný overhead

Raw pointery a jejich použití pro observování

- Definice: int* pointer = &number; (operátor & veme adresu čísla)
- Použití: *pointer = 5; (zapíšeme do ukazovaného čísla)
  - Operátoru * se zde říká dereference, pro metody se používá ->
    - Příklad s metodou: object_pointer->do_cool_stuff()
- **Pointerová aritmetika:**

  - std::vector<int> numbers = {1, 2, 3, 4, 5};
    int *pointer1 = &numbers[0], *pointer2 = &numbers[4];
    (pointer2 - pointer1) bude mít hodnotu 4, jde používat ++, sčítat s čísly, atd.
  - Snažte se jí vyhnout, jak to jen jde; ale je dobré o ní vědět
    - Pointerová aritmetika je totiž inspirací iterátorů (to jsou pseudoukazatele do kontejnerů, např. do vectoru) - o iterátorech, a různých kontejnerech, se budeme bavit jindy

Raw pointery v archaické správě paměti (a jejich probémy)

- Vyrobení a úklid objektu na haldě:
  - Object* object = new Object(parametry);
  - delete object; (volání destruktoru objektu = uklizení podobjektů atd; a pak naváže samotná dealokace)
- Vyrobení a úklid pole objektů na haldě:
  - Object* objects = new Object[10]; nebo new Object[10](parametry);
  - delete[] objects; (to samé, ale naznačujeme, že je těch objektů víc)
- **Problémy:**
  - je možné zapomenout delete (nebo ho zavolat víckrát)
  - jde si splést delete (pro jednotlivé objekty) a delete[] (pro pole objektů)
  - Není jasné, jestli pointer reprezentuje vlastnictví (ani jestli jednoho, nebo více objektů)

Správa paměti dnes: RAII (zjednodušeně) https://en.cppreference.com/w/cpp/language/raii

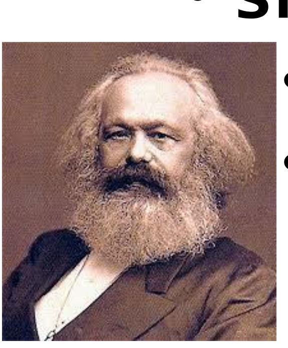
- Spousta *technoblábolu*, ale princip je jasný:
  každý resource reprezentován objektem,s jeho zánikem se resource automaticky uvolní
- Objekty se vytváří konstruktorem a zanikají po zavolání destruktoru
  - **Kdy?** Na konci scopu, který je definuje:
    - Zanořené scopy ve funkcích
    - Data member (subobject) umírá s rodičem
      1. Rodič uklidí: zavolá se tělo ~Rodič()
      2. Zavolají se destruktory jeho dětí (pozpátku)
         - Jako objekty v top scopu mainu v ukázce:
      3. Zavolají se destruktory tříd, od kterých dědí (opět pozpátku)
         - class A: B, C {} zavolá ~C() a pak ~B()
- Použití (pár typických příkladů):
  - Kontejnery (známe aspoň vector, array)
    - Zahodíme kontejner -> uklidí prvky
  - Smart pointery (řekneme si dnes)
  - std::fstream: resourcem je otevřený soubor

```cpp
struct Object {
    std::string name_;
    Object(string name) : name_(name) {
        cout << "Object " << name_ << " created" << endl;
    }
    ~Object() {
        cout << "Object " << name_ << " destroyed" << endl;
    }
};

int main() {
    Object a("a");
    Object b("b");
    {
        Object c("c");
    } // c is destroyed here
    Object d("d");
}
// destroyed in reverse order:
//   d, b, a
```

Správa paměti dnes: smart pointery

- **unique_ptr<Object>**
  - Vyrobení objektu na haldě: **pointer = std::make_unique<Object>(parametry);**
  - Podporuje dereferenci jako raw pointer, i pro metody (operátory * a ->)
  - Když **vlastníme tento pointer**, tak víme, že **vlastníme i Object**
    - **Compiler nám zakáže ten pointer omylem okopírovat, musíme: std::move(pointer)**
  - Úklid Objectu vyvoláme zahozením pointeru (nebo přiřazením **nullptr**)
- **shared_ptr<Object>**



  - Vyrobení objektu na haldě: **pointer = std::make_shared<Object>(parametry)**
  - Jako unique_ptr, ale reprezentuje **sdílené vlastnictví** = ten pointer může mít víc vlastníků a Object **se uklidí až, když ho všichni zahodí** (jde volně kopírovat)
    - **Hlídá si počet referencí** při každém předání/zahození – když je 0, tak spustí úklid

Správa paměti dnes: smart pointery

- Výhody smart pointerů:
  - **Nehrozí double-free, nejde zapomenout na zavolání delete**
  - **Je jasné vlastnictví**
- Kdy použít `unique_ptr` a kdy `shared_ptr`
  - Použijte unique_ptr vždy, pokud to jde a víte, že ten object musí být na haldě
    - U malých objectů a u těch, co samy odkládají věci na haldu (třeba vector) dávejte věci na stack (**prostě Object object**, bez pointerů) – nebo prostě tam, kde je definujete
    - **shared_ptr je malinko dražší a méně bezpečný**
      - **V čem je nebezpečný?** – Můžeme omylem převést vlastnictví na někoho jiného a pak mylně očekávat úklid… (viz další bod, ale zde i obecněji)
- Chytré pointery nejsou stříbrná kulka, stále může nastat leak
  - Link na Compiler Explorer, kde „omylem“ vznikne cyklus unique pointerů
  - **A hrozba je o to větší se shared pointery**, např. pokud se s nimi pokusíme implementovat obecný graf (proto třeba Java a C# mají GC)

std::move(objekt)

#include <utility>

- Tato funkce nic nedělá, jen objekt castne na **T&&** referenci
  - Oficiálně: T&& je rvalue reference (T& je lvalue reference)
    - *Vzniklo z left&right, protože v lvalue=rvalue nedává smysl, aby hodnota nalevo byla temporary*; zároveň: rvalue jakože *returned value* (často vracíme temporary, např u a+b)
- Značí, že objekt má být chápán jako *temporary*
  - Funkce, co jej přijme, ho může volně „vykrást“ – pokud to je vektor, tak přebrat jeho data, u unique_ptr přebrat vlastnictví cílového objektu, atd.
- Díky std::move můžeme rozlišovat overloady funkcí brající T& a T&&
  - Overload s T& bude typicky data kopírovat
  - Overload s T&& bude typicky data move-ovat, zabráníme kopírování

Rvalue reference: Object&&

- Reference na temporary objekty
  - Čerstvě vyrobené objekty: `std::vector<int>{1, 2, 3}`
  - Výsledky různých operací: `string1 + string2`
  - `std::move(objekt)` <- tímhle naznačíme, že data objektu předáváme pryč
- **Velký chyták:** **`auto&&`** není nutně rvalue reference – je forwarding
  - Pokud do **`auto&&`** přiřadíme temporary (rvalue), tak to opravdu je rvalue reference
  - Pokud přiřadíme třeba pojmenovaný objekt (lvalue), tak se **`auto&&`** převede na správnou referenci buďto **`const T&`** nebo jen **`T&`** podle toho, jestli objekt je **`const`**
  - To samé pak bude platit u templatů:
    `template<class T> void foo(T&& fwd_ref) { … }`

void (std::)swap(T &a, T &b)

#include <utility>

- Prohodí hodnoty a, b – využívá k tomu move a nebo specializaci swap
  - Specializaci swap definujte jako non-member funkci ve stejným namespacu (prostě vedle té třídy nebo jako frienda)
    - Standardně se společně s ní definuje metoda swap (ta vnější swap pak typicky jen volá tuto metodu)
- Nejbezpečnější použití weirdly-enough není std::swap(a, b), ale:
  using std::swap; // note: using v lokálním scopu
  swap(a, b);
  - **Proč?** Protože swap(a, b) mohlo mít specializaci definovanou u jejich třídy

Tuply, pairy a structured binding

- Dvě C++ třídy pro uspořádané n-tice prvků
  - n = 2: std::pair<First, Second> - lze vytvořit pomocí std::make_pair
  - n bez omezení: std::tuple<T1, T2, …> - lze vytvořit pomocí std::make_tuple
  - Obě třídy podporují vytváření pomocí {v1, v2, …, vn}
- Operátory <, >, ==, … definovány lexikograficky
- Structured binding:
  - Kopírování hodnot z tuplu/pair: auto [v1, v2, …, vn] = tuple_n;
  - Reference na hodnoty z tuplu/pair: auto&& [v1, v2, …, vn] = tuple_n;
- To samé, ale s již existujícími objekty: std::tie(a, b, …, x) = tuple_n
  - Lze využít na definici operátorů ve třídě:

```
bool operator==(const X& othr) const {
    return tie(f1_, std::f2_) == tie(othr.f1_, othr.f2));
}
```

#include <utility>

#include <tuple>