

NPRG041 – C++

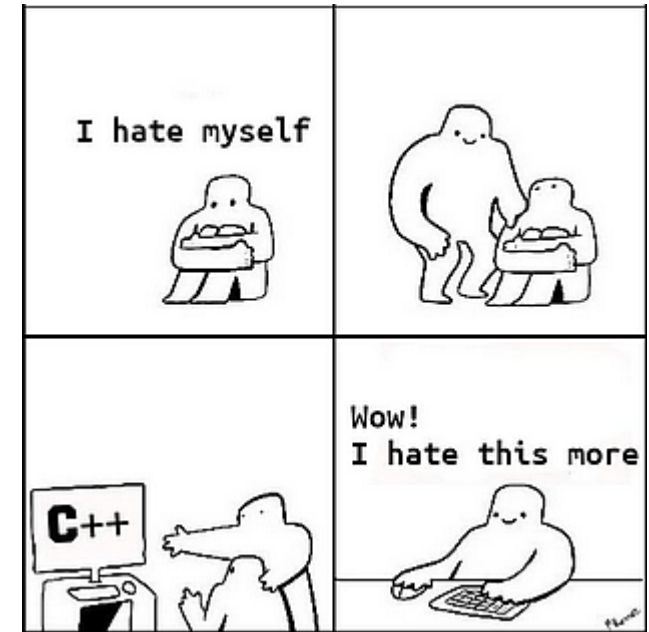
cvičení – Jiří Klepl

mattermost: ulita/2425ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

Klepl@d3s.mff.cuni.cz

Agenda

- Modularita, objekty
- Vlastnictví a RAI
- Pointery a reference
 - Raw pointery
 - `std::unique_ptr`
 - `std::shared_ptr`
- `std::move`



Správa paměti: RAI

- Život C++ objektu začíná konstruktorem a končí destruktorem
 - Konstruktor: může alokovat paměť, otevřít soubor, atd.
 - Destruktor: dealokuje paměť, zavírá soubor, atd.
- Objekt „umírá“ při
 - Konec scope, ve kterém byl definován
 - Smrt rodičovského objektu
 - `std::vector<T>` zabíjí svoje objekty
 -  `std::span<T>` ne (je jen reference na pole)

```
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
```

Raw (observer) pointers

- Definice: **T* ptr = &object**

- Operátory & (získání adresy) a * (dereference) jsou opaky

💀 Nepíšeme **ptr = new Object**

💀 Raw pointers by nikdy neměly vlastnit objekty, pouze observovat

⚠️ Pointerová aritmetika (spíše nepoužívat)

```
std::vector<int> vec{0, 1, 2, 3, 4};  
int* beg = &vec[0];  
int* end = &vec[vec.size() - 1];  
for (; beg < end; ++beg, --end) {  
    std::swap(*beg, *end);  
}  
for (int i : vec) {  
    std::println("{} ", i);  
}
```

Funkce na prohození objektů

Smart pointers

```
#include <memory>
```

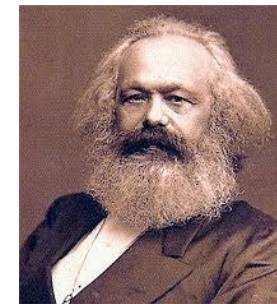
- **unique_ptr<Object>**

- Reprezentuje unikátní vlastnictví objektu na haldě
 -  nejde kopírovat
- Automaticky dealokuje paměť
 - Pokud ten pointer umře / nastavíme ho na **nullptr**



- **shared_ptr<Object>**

- Reprezentuje sdílené vlastnictví objektu na haldě
 -  má schovaný reference-counter vlastníků a automaticky dealokuje, pokud 0
-  práce s ním je dražší



Smart pointers

```
#include <memory>
```

- **unique_ptr<Object>**
 - Vyrobení objektu na haldě: **pointer = std::make_unique<Object>(parametry)**
- **shared_ptr<Object>**
 - Vyrobení objektu na haldě: **pointer = std::make_shared<Object>(parametry)**
- Obojí se chová jako normální pointer
 - ***ptr** dereference
 - **ptr->obj_method()** volání metod
- Převedení na observer pointer **Object***: **ptr.get()**
- Předání vlastnictví objektu pomocí: **dst = std::move(src)**
 - **dst** ukazuje na objekt, **src** má hodnotu **nullptr** (nic)

std::move(objekt)

```
#include <utility>
```

- Tato funkce nic nedělá, jen objekt castne na **T&&** referenci
 - T&& typicky ukazuje na *temporary* objekty
 - Např. návratová hodnota funkce ve výrazu: **obj = func()**
 - T&& se jmenuje **rvalue reference** (T& je lvalue reference)
 - r jako right/return (návratová hodnota **func()**); l jako left (**obj**)
 - Značí, že objekt má být chápán jako *temporary*
 - Funkce, co jej přijme, ho může volně „vykrást“ – převzít jeho data
 - Díky **std::move** můžeme zavolat **T&&** overload funkce namísto **T&**
 - **func(T& obj)** bude obj typicky kopírovat/číst nebo nějak modifikovat
 - **func(T&& obj)** převezme data z obj a obj nechá prázdný/nepoužitelný
- ⚠ move-ování typicky negarantuje vyprázdnění movovaného obj.

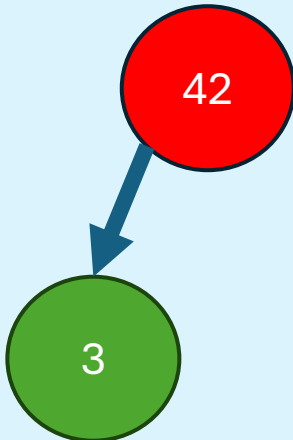
Příklad: bst (binární vyhledávací strom)

 nejsou požadavky na vyvažování

- Zadání: dokončete implementaci zdrojového kódu bst.cpp
 1. Program čte **stdin** nebo **soubor** (pokud je zadán v cmd argumentu)
 2. Na každém NEPRÁZDNÉM řádku vstupu je příkaz
 - **insert** <rest of the line after ws> ← přidá nový node
 - **remove** <rest of the line after ws> ← odebere node (rozebráno na dalších slidech)
 - **print** ← vytiskne obsah stromu v In-Order pořadí (prvky indentované /t podle hloubky od 0)
 - **clear** ← smaže obsah stromu
 3. Konec vstupu/error ukončí program
- Strom bude uložený pomocí `std::unique_ptr`
- <rest of the line after ws>: to, co přečte: `std::getline(input, str)`

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

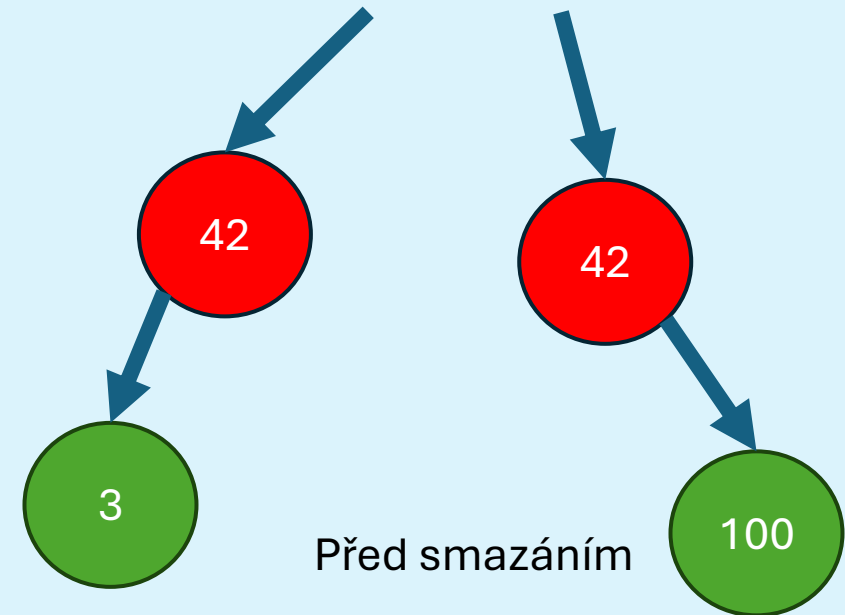
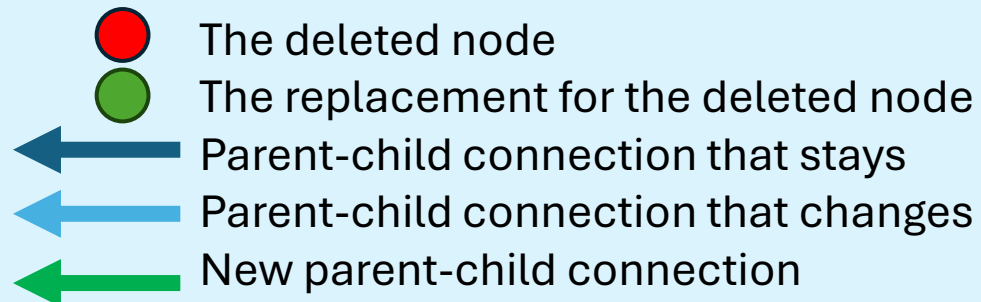


Remove Node – first two easy cases

- The deleted node has at most 1 child

➔ Just replace the node with the child

➔ Or nullptr if it has none

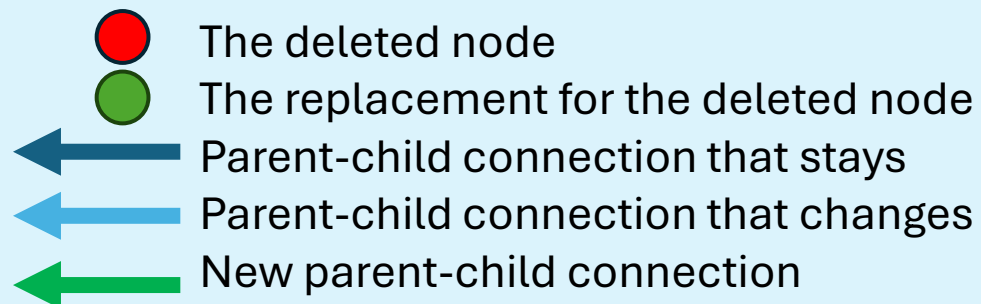
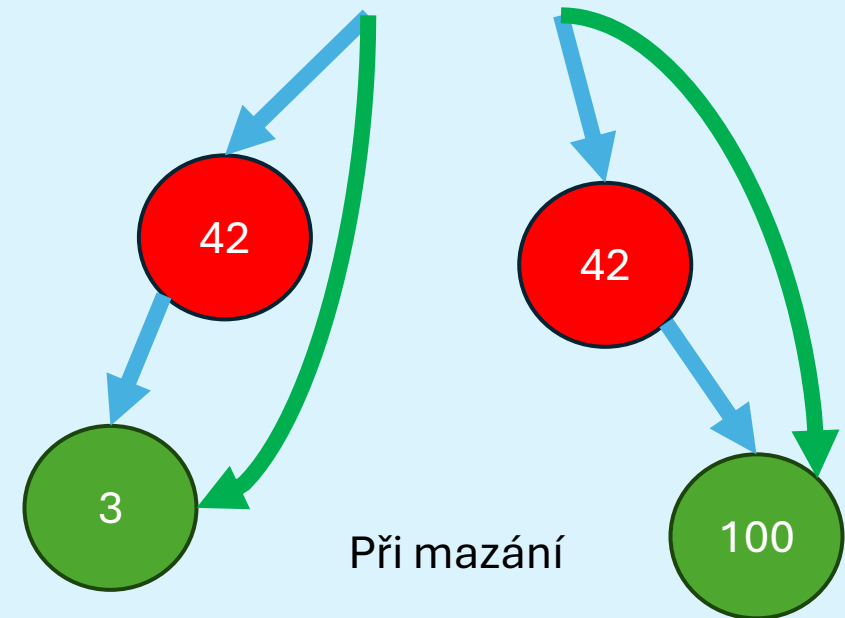


Remove Node – first two easy cases

- The deleted node has at most 1 child

➔ Just replace the node with the child

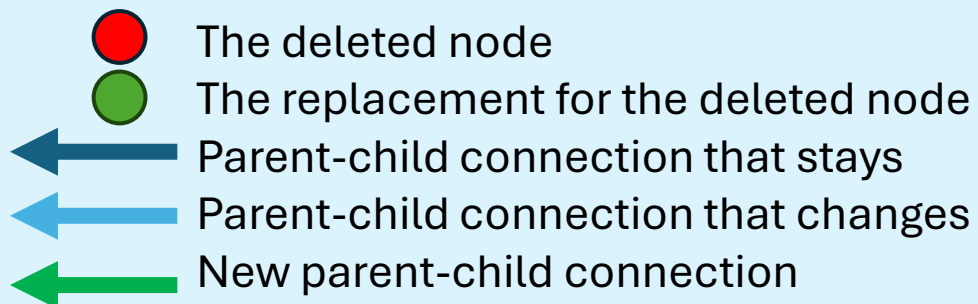
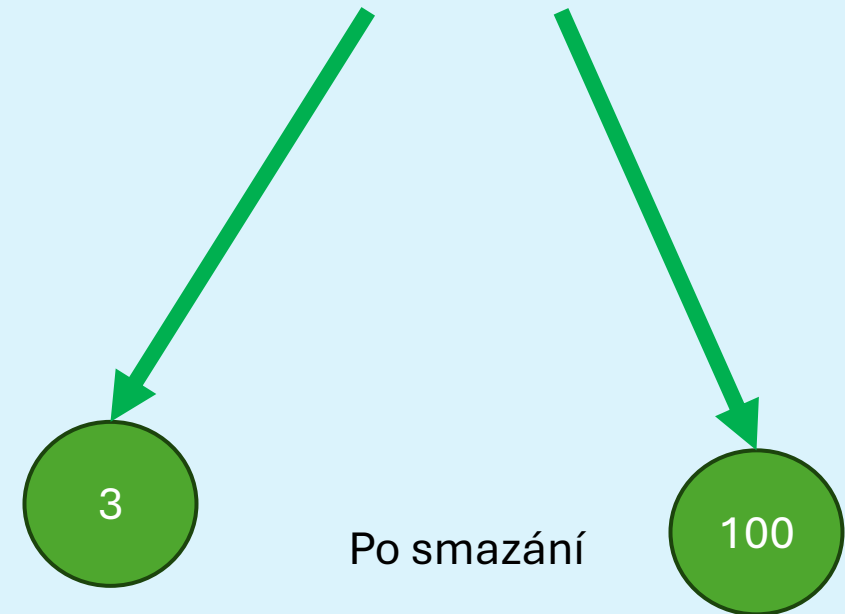
➔ Or nullptr if it has none



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

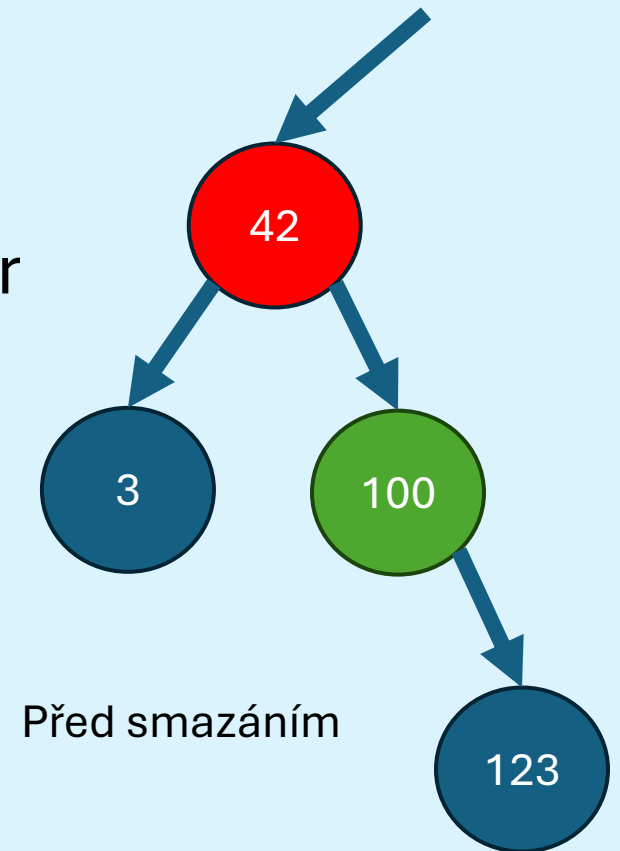
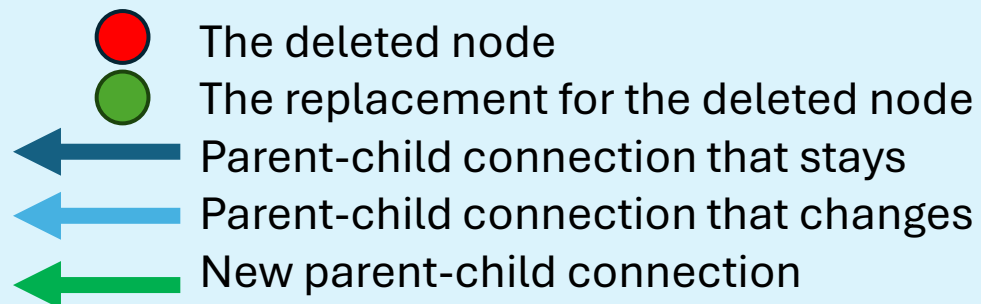


Remove Node – the second case

- 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

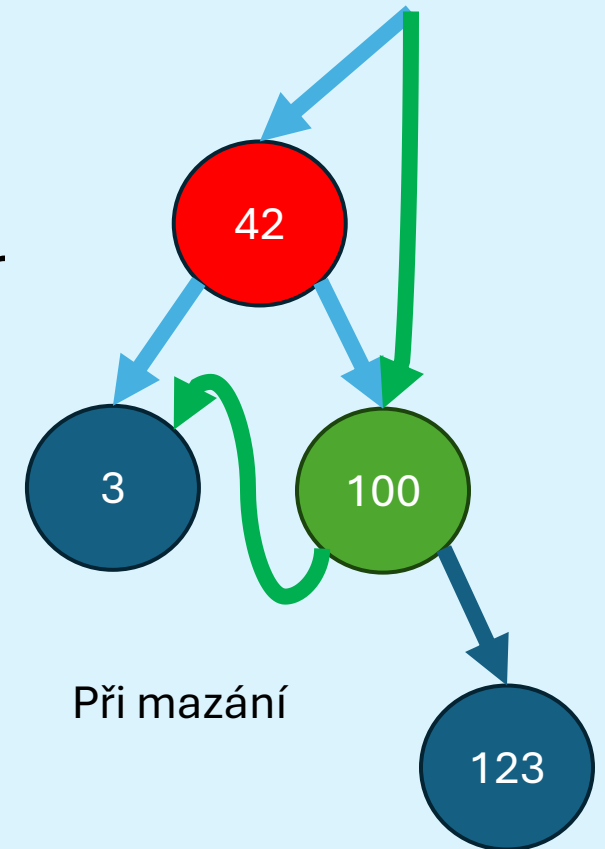
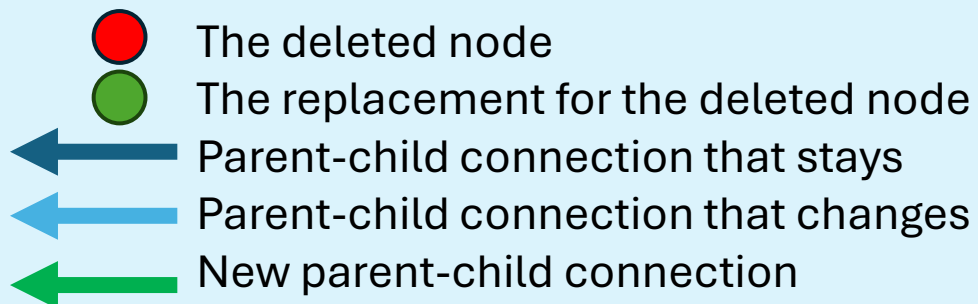
➔ Replace the node with the successor



Remove Node – the second case

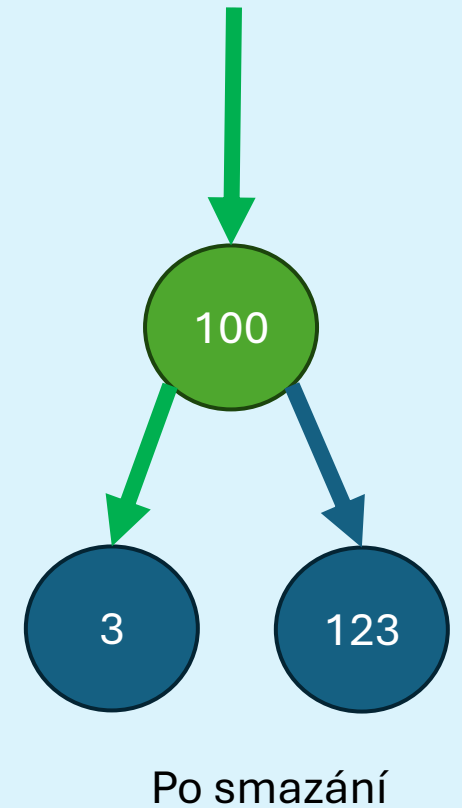
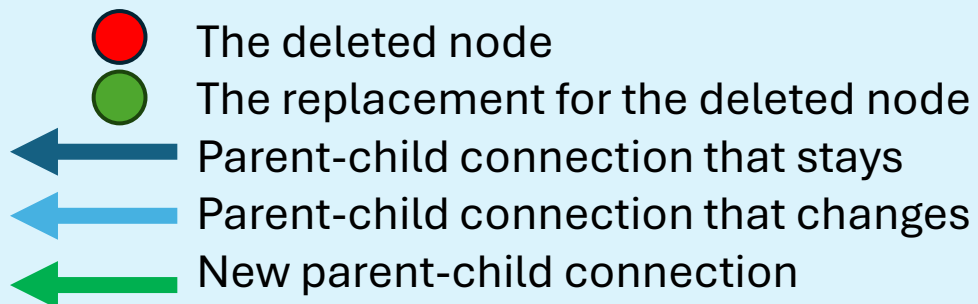
- The deleted node has two children
 - And the right child is the successor (has no left child)

- ➔ 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

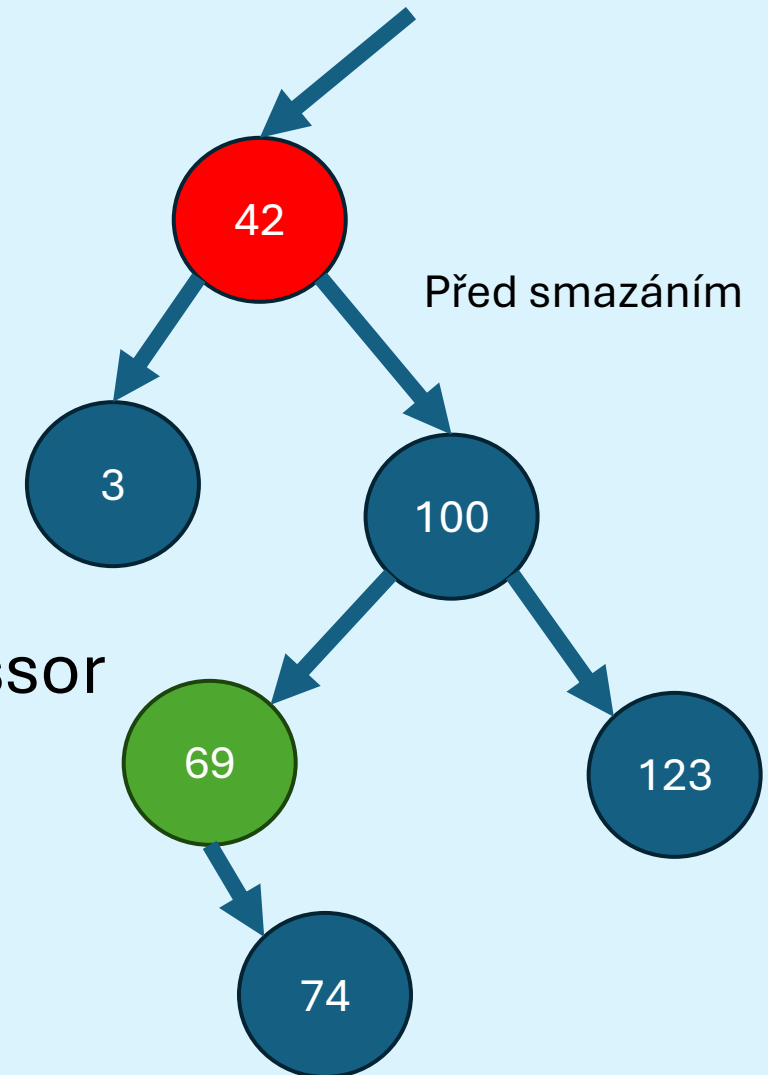
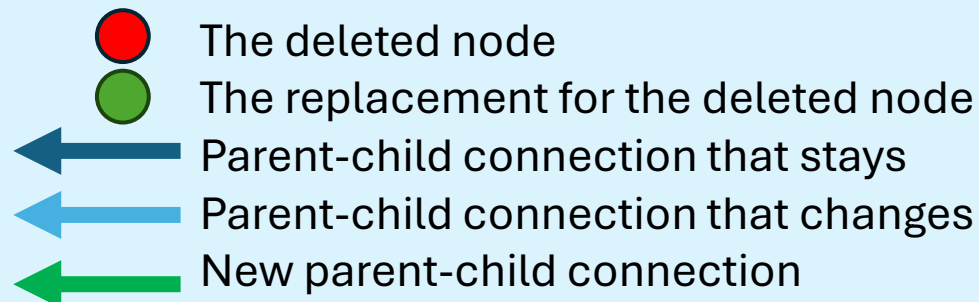


Remove Node – the scary case

- 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

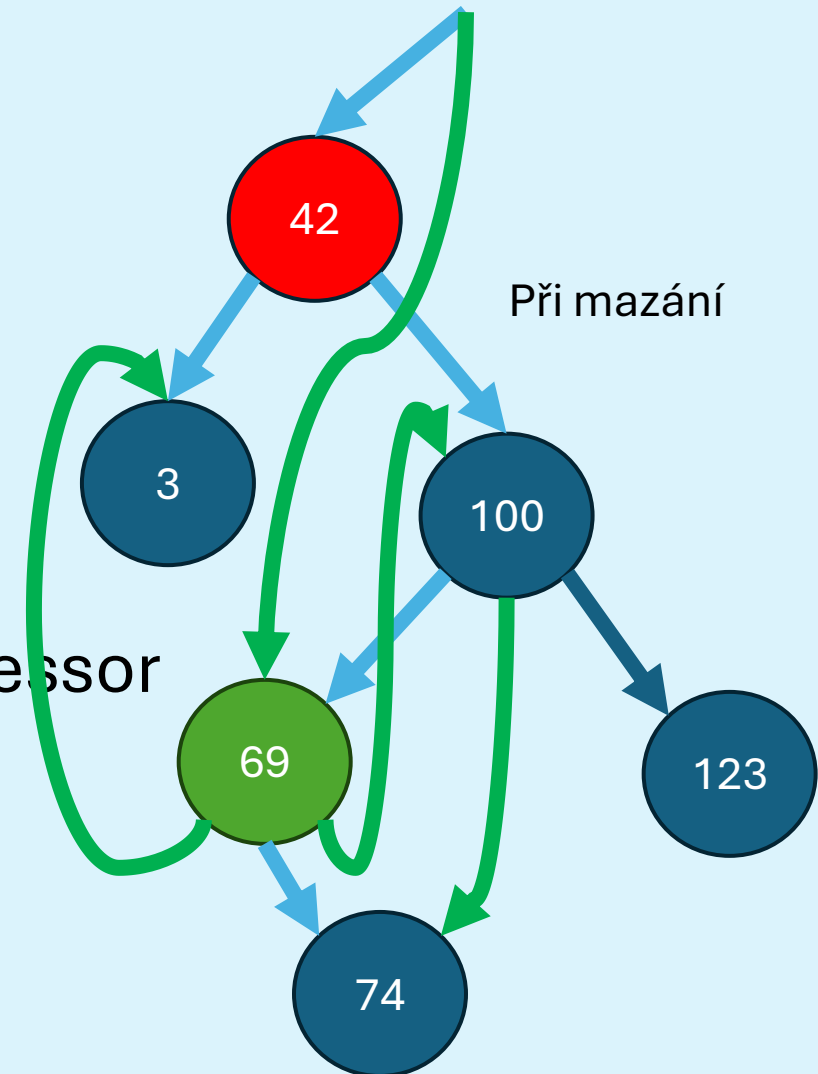
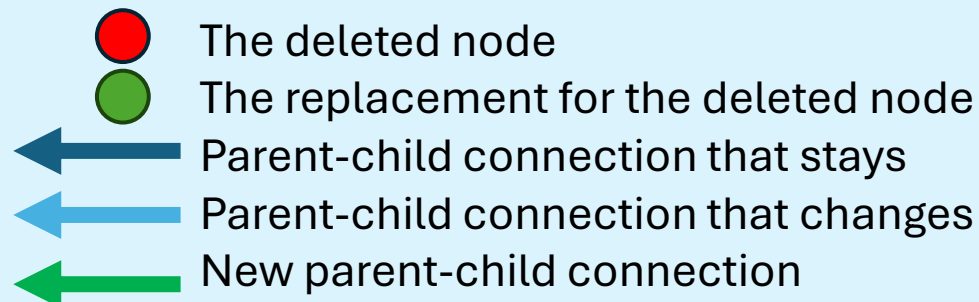


Remove Node – the scary case

- 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

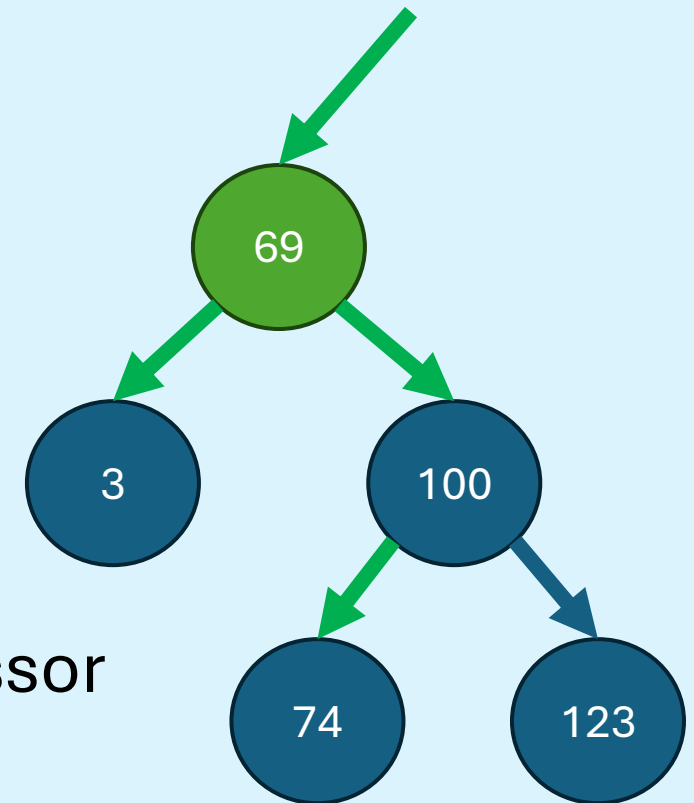
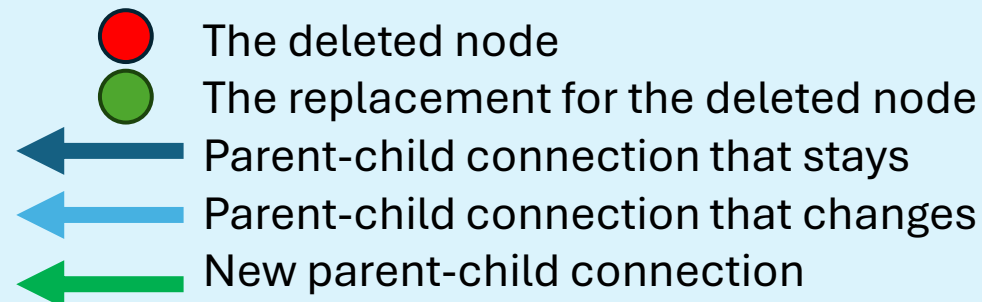


Remove Node – the scary case

- 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

Počítání Oveček



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

- Program čte

- stdin (std::cin), pokud nedostane cmd argumenty
- jinak pro každý argument otevře soubor s danou cestou

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

- Tipy

- Nejspíš využijeme čtení znak po znaku:
- Nechceme ukládat celý obsah souboru

```
void func(std::istream& in) {  
    for (;;) {  
        char c = in.get();  
        // check if successful:  
        if (in.fail()) break;  
        process(c);  
    }  
}
```

! Detekce chyb/EOF

Ted' už víme, že c je platný znak

```
for (auto&& file : files) {  
    std::ifstream in(file);  
    if (!in.good()) ...  
    func(in);  
}
```

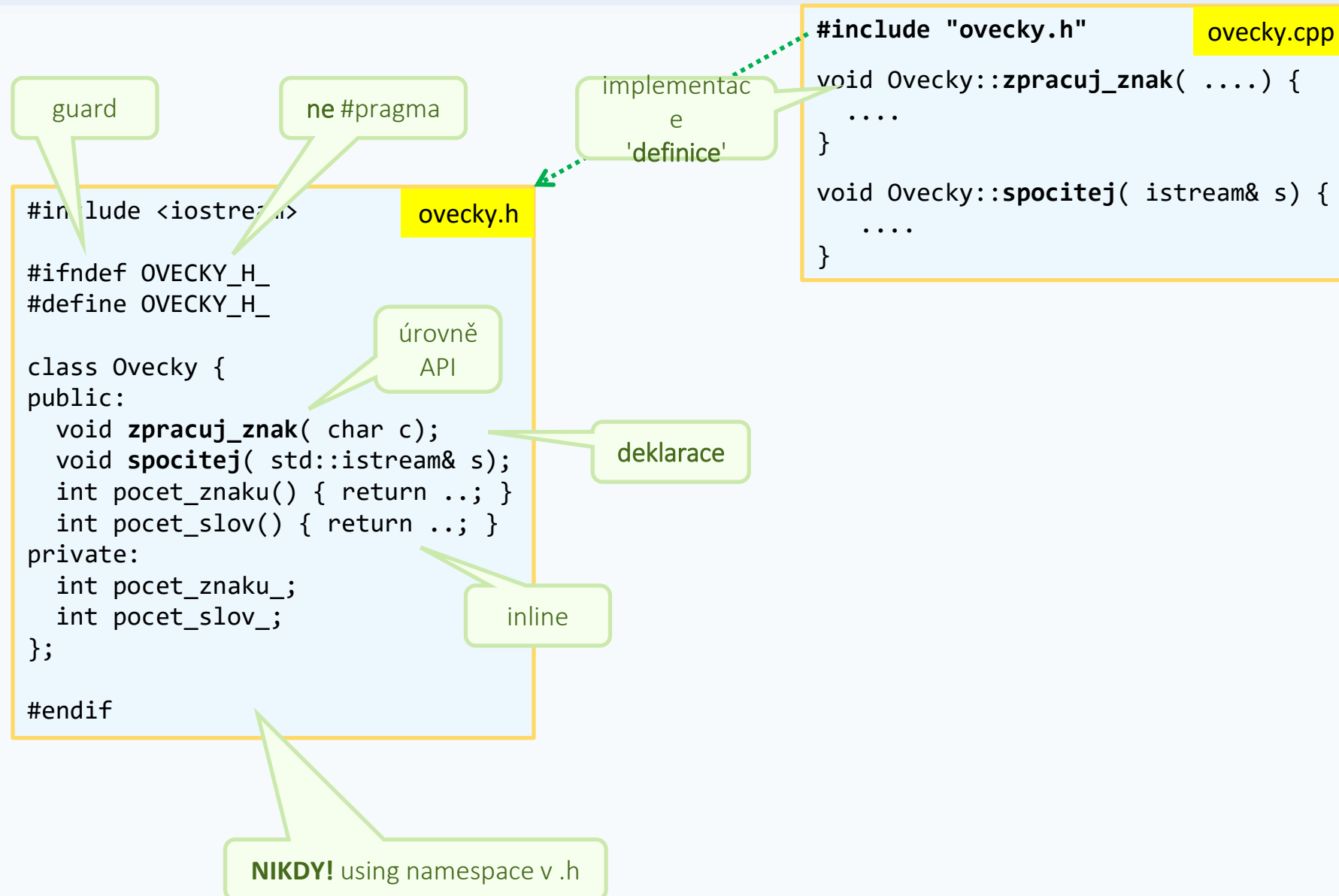
! Detekce chyb

Požadavky na řešení

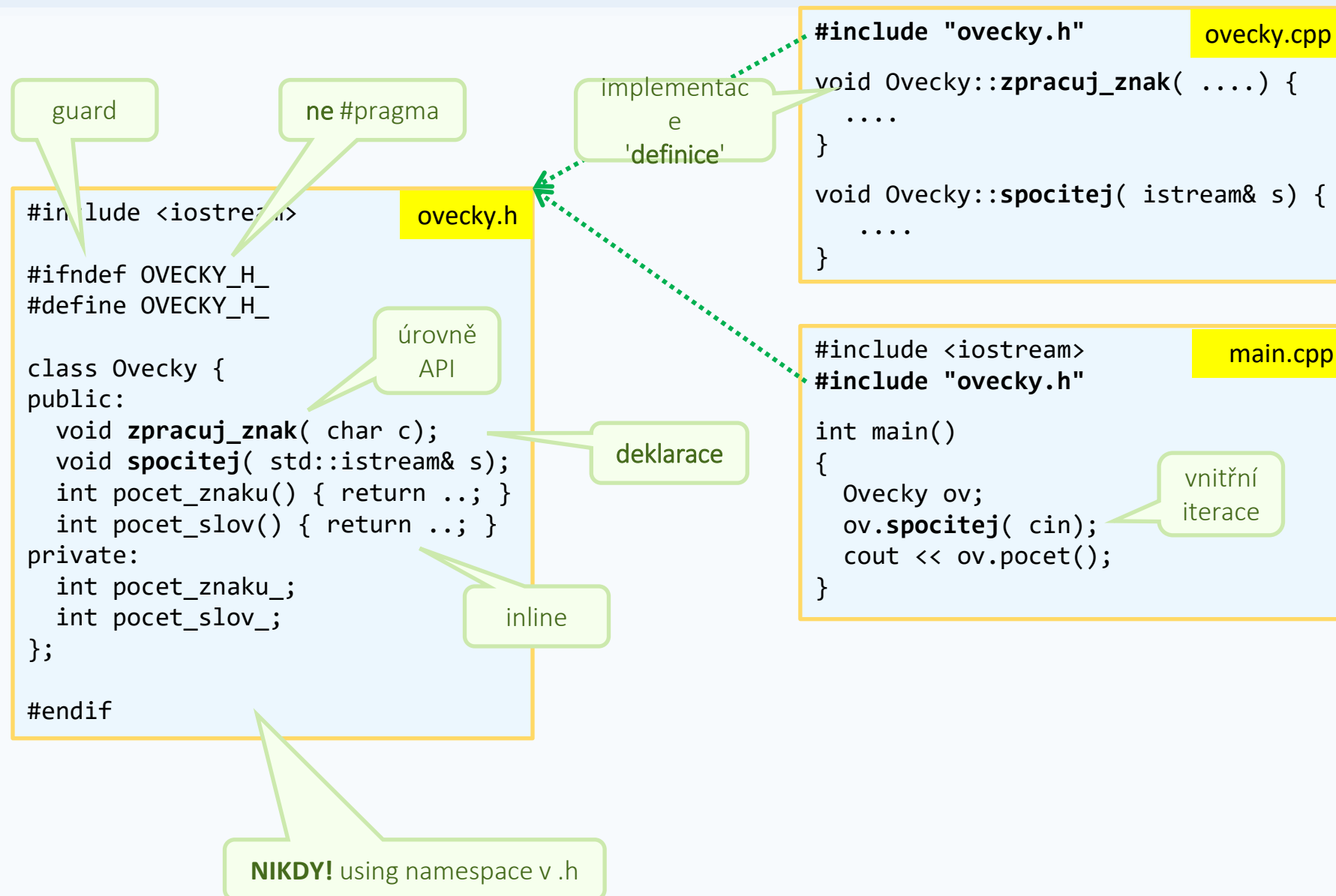
- Modularita
 - Stav programu uložený v objektu nějaké třídy
 - Rozumné rozdělení na .cpp a .hpp soubory
 - V .hpp budou definice API, třídy
 - ⚠ Nezapomenout na header-guardy (ukázka na dalším slidu)
 - V .cpp budou implementace funkcí („one-liner“ funkce mohou být přímo ve třídě)
 - Dlouhé funkce rozděleny na více funkcí
 - Funkce dělá jednu akci
- Kvalita
 - Řešení používá věci podle cvičení, přednášek, cppreference, ...
 - ⚠ Kontrola chyb načítání souborů, ...
 - ⚠ Datové položky ve třídách inicializovány pomocí initializer-listů:

```
Trida(int argument) : polozka_{argument}, { /* konstruktor */ }
```

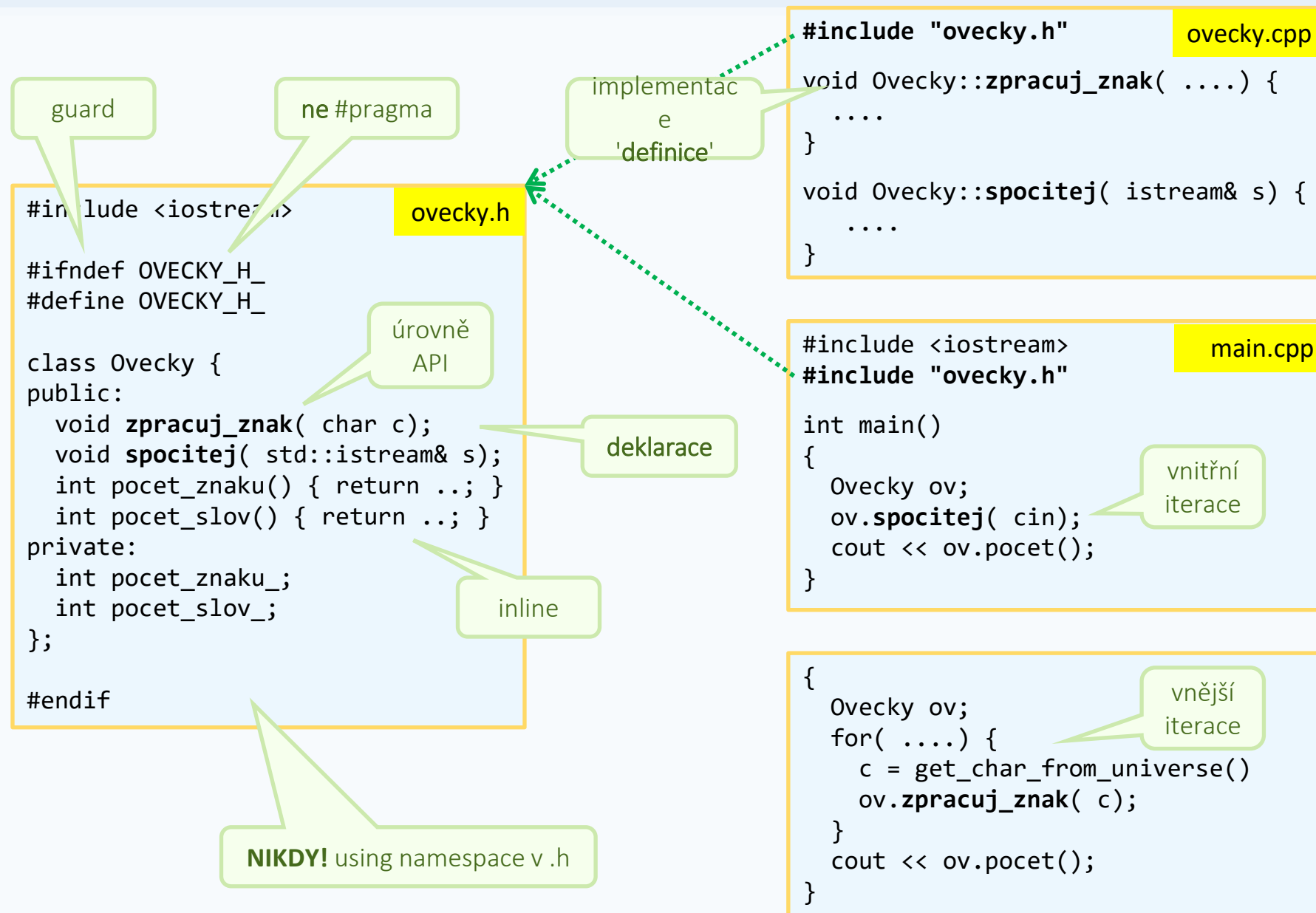
Modularita a zdrojové soubory



Modularita a zdrojové soubory



Modularita a zdrojové soubory



Inline a ne-inline metody

```
class Trida
{
    string slozitaFce( int x);
};
```

trida.h

```
#include "trida.h"
string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i<x+y; ++i) {
        ....
    }
}
```

trida.cpp

```
#include "trida.h"
{
    Trida q;

    s = q.slozitaFce( 1);
}
```

předání parametrů a volání
push 1
s = call q.slozitaFce
add esp, 8

Inline a ne-inline metody

```
class Trida
{
    string getResult () { return r; }
    string slozitaFce( int x);
};
```

trida.h

inline metoda

inline metoda
rozvinutí místo volání
`s = q.r`

```
#include "trida.h"

{
    Trida q;
    s = q.getResult();
    s = q.slozitaFce( 1);
}
```

```
#include "trida.h"
string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i<x+y; ++i) {
        ....
    }
}
```

trida.cpp

předání parametrů a volání
`push 1`
`s = call q.slozitaFce`
`add esp, 8`

Inline a ne-inline metody

```
class Trida
{
    string getResult () { return r; }
    string slozitaFce( int x);
    int inlineFce( int x)
    { int y = -1;
      for( i = 0; i < x+y; ++i)
        ....
    }
};
```

trida.h

inline metoda

```
#include "trida.h"

string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i < x+y; ++i) {
        ....
    }
}
```

trida.cpp

```
#include "trida.h"

{
    Trida q;
    s = q.getResult();
    s = q.slozitaFce( 1);
    z = q.inlineFce( 2);
}
```

inline metoda
rozvinutí místo volání
s = q.r

předání parametrů a volání
push 1
s = call q.slozitaFce
add esp, 8

y = -1
i = 0
loop:
if(i >= 2+y) goto ...
...
++i
goto loop
... ..

Inline a ne-inline metody

```
class Trida
{
    string getResult () { return r; }
    string slozitaFce( int x);
    int inlineFce( int x)
    { int y = -1;
      for( i = 0; i < x+y; ++i)
      ....
    }
};
```

trida.h

inline metoda

inline metoda
rozvinutí místo volání
s = q.r

```
#include "trida.h"

{
    Trida q;
    s = q.getResult();
    s = q.slozitaFce( 1);
    z = q.inlineFce( 2);
}
```

```
#include "trida.h"

string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i < x+y; ++i) {
        ....
    }
}
```

trida.cpp

předání parametrů a volání
push 1
s = call q.slozitaFce
add esp, 8

~~y = -1
i = 0
loop:
if(i >= 2+y) goto ...
...
++i
goto loop
... ..~~

složité metody v samostatném modulu
přehlednost rozhraní
separátní kompilace - čas
vygenerovaný kód

Inline a ne-inline metody

```
class Trida
{
    string getResult () { return r; }
    string slozitaFce( int x);
    int inlineFce( int x)
    { int y = -1;
      for( i = 0; i < x+y; ++i)
      ....
    }
};
```

trida.h

inline metoda

inline metoda
rozvinutí místo volání
s = q.r

```
#include "trida.h"

{
    Trida q;
    s = q.getResult();
    s = q.slozitaFce( 1);
    z = q.inlineFce( 2);
}
```

```
#include "trida.h"

string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i < x+y; ++i) {
        ....
    }
}
```

trida.cpp

předání parametrů a volání
push 1
s = call q.slozitaFce
add esp, 8

~~y = -1
i = 0
loop:
if(i >= 2+y) goto ...
...
++i
goto loop
... ..~~

složité metody v samostatném modulu
přehlednost rozhraní
separátní kompilace - čas
vygenerovaný kód

ale: **šablony!**
nutná definice při kompilaci
⇒ vše v headeru

Inline a ne-inline metody

```
class Trida
{
    string getResult () { return r; }
    string slozitaFce( int x);
    int inlineFce( int x)
    { int y = -1;
      for( i = 0; i < x+y; ++i)
      ....
    }
};
```

trida.h

inline metoda

inline metoda
rozvinutí místo volání
s = q.r

```
#include "trida.h"

{
    Trida q;
    s = q.getResult();
    s = q.slozitaFce( 1);
    z = q.inlineFce( 2);
}
```

```
#include "trida.h"

string Trida::slozitaFce( int x)
{
    int y;
    for( i = 0; i < x+y; ++i) {
        ....
    }
}
```

trida.cpp

předání parametrů a volání
push 1
s = call q.slozitaFce
add esp, 8

~~y = -1
i = 0
loop:
if(i >= 2+y) goto ...
...
++i
goto loop
... ..~~

složité metody v samostatném modulu
přehlednost rozhraní
separátní kompilace - čas
vygenerovaný kód

ale: **šablony!**
nutná definice při kompilaci
⇒ vše v headeru

~ C++20 Modules

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

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

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