NPRG041 – 2021/22 ZS – Cvičení MS – **Domácí úkol 2**

Gumové pole

Pro potřeby zamýšlené implementace grafu je nejprve potřeba naprogramovat vlastní šablonovaný kontejner gumového pole `Array<T>`, který umožní ukládání prvků libovolného datového typu `T` způsobem zaručujícím neměnnost umístění všech prvků po celou dobu jejich existence. Jinými slovy při manipulaci s tímto kontejnerem (např. při vkládání nebo odebírání prvků) nikdy nedojde k nutnosti přemisťování existujících prvků v paměti, a tedy k zneplatnění uživatelem držených odkazů na existující prvky.

Z hlediska vnitřní organizace takového gumového pole použijte standardní vektor `std::vector`, který bude udržovat chytré unikátní ukazatele `std::unique_ptr<T[]>` ukazující na dynamicky alokované bloky pro vlastní ukládání jednotlivých prvků, technicky realizované jako tradiční pole fixní velikosti. Ta se předá jako argument parametrického konstruktoru gumového pole. Není-li specifikována, předpokládejte výchozí velikost `10`.

Pro manipulaci s prvky naprogramujte minimálně následující základní funkce:

- `void push_back(const T& item)`: vloží na konec gumového pole nový prvek; v případě potřeby alokuje nový vnitřní blok
- `void pop_back()`: odebere prvek vyskytující se na konci gumového pole; v případě potřeby dealokuje již nepotřebný vnitřní blok; lze volat jen na neprázdné gumové pole
- `T& at(size_t index) const`: vrátí referenci na prvek vyskytující se na uvedené pozici
- `T& operator[](size_t index) const`: totéž

S ohledem na zamýšlenou funkcionalitu grafu je dále nutné naprogramovat následující konstruktory a operátory umožňující manipulaci s gumovými poli jako celky:

- `Array(const Array<T>& other)`: standardní copy constructor
- `Array(Array<T>&& other) noexcept`: standardní move constructor
- `Array<T>& operator=(const Array<T>& other)`: standardní copy assignment operátor
- `Array<T>& operator=(Array<T>&& other) noexcept`: standardní move assignment operátor

Posledním krokem je implementace dopředného iterátoru `std::forward_iterator_tag`, který bude umožňovat iteraci přes jednotlivé prvky gumového pole. To konkrétně znamená definovat minimálně následující funkce iterátoru:

- `bool operator!=(const iterator& other) const`: otestuje nerovnost dvou iterátorů, tedy že ukazují na různé prvky v daném gumovém poli
- `iterator& operator++()`: posune iterátor na další prvek gumového pole (je-li takový)
- `T& operator*() const`: zpřístupní prvek gumového pole, na který iterátor aktuálně ukazuje

Do třídy gumového pole pak přidejte poslední dvě členské funkce zpřístupňující tyto iterátory pro běžné použití:

- `iterator begin() const`: vrátí iterátor ukazující na začátek gumového pole, tedy na první prvek (je-li takový)
- `iterator end() const`: vrátí iterátor ukazující na konec gumového pole, tedy za poslední prvek (je-li takový)

Reprezentace grafu

Hlavním cílem tohoto domácího úkolu je naprogramovat šablonovanou třídu umožňující reprezentaci grafu, a to orientovaného i neorientovaného. V obou případech platí, že mezi libovolnou dvojicí vrcholů může (v daném směru) existovat nejvýše jedna hrana, uvažujeme tedy jen obyčejný graf, nikoli multigraf. Důležité je, že na každý vrchol i hranu potřebujeme umět navázat další libovolné informace, které si u nich chceme uložit.

Třída pro graf jako takový `Graph<NData, EData>` bude poskytovat základní rozhraní pro oba typy grafů. Bude však abstraktní, nebude tedy od ní možné vytvářet instance. To bude možné až pro konkrétní odvozenou variantu orientovaného grafu `DirectedGraph<NData, EData>` a analogicky neorientovaného grafu `UndirectedGraph<NData, EData>`. Ve všech případech (stejně jako u dalších dále uvedených tříd) popisuje šablonový argument `NData` libovolný datový typ použitý právě pro reprezentaci informací navázaných na jednotlivé vrcholy a analogicky `EData` pro hrany. Tyto typy mohou, ale stejně tak nemusí být stejné.

Třída grafu bude z hlediska vnitřních datových položek obsahovat dvě komponenty, a to instanci třídy `Nodes<NData, EData>` řešící veškerou funkcionalitu týkající se vrcholů a analogicky instanci třídy `Edges<NData, EData>` starající se o hrany. Každý jednotlivý uzel bude realizován jako instance třídy `Node<NData>` a hrana jako instance třídy `Edge<NData, EData>`, a to bez ohledu na to, jestli je, nebo není orientovaná. Jinými slovy hrany samy o sobě tento fakt rozlišovat nepotřebují.

Vrcholy a hrany

Každý vrchol i hrana mají přidělený unikátní identifikátor, a to z množiny $\mathbb{N}_0$, tedy přirozených čísel včetně nuly. Pokud graf obsahuje $n \in \mathbb{N}_0$ vrcholů, platí, že jejich identifikátory pokrývají všechna čísla od 0 do $n - 1$. Jinými slovy tyto identifikátory na sebe souvisle navazují, žádné nepřeskakujeme. Totéž analogicky (odděleně) platí pro hrany. Technicky pro identifikátory použijeme typ `size_t`. Konečně dodejme, že graf bude umožňovat pouze přidávat nové vrcholy a hrany, ty existující není možné odebírat.

Pokud jde konkrétně o třídu pro vrchol, je potřeba implementovat následující rozhraní:

- `size_t getId() const`: vrátí hodnotu identifikátoru daného vrcholu
- `NData& getData()`: vrátí referenci na datový obsah navázaný na daný vrchol

V případě třídy pro hranu je rozhraní analogické, jen přidáme možnost přistupovat k vrcholům definujícím danou hranu:

- `size_t getId() const`: vrátí hodnotu identifikátoru dané hrany
- `EData& getData()`: vrátí referenci na datový obsah navázaný na danou hranu
- `Node<NData>& getSource() const`: vrátí referenci na první z vrcholů
- `Node<NData>& getTarget() const`: vrátí referenci na druhý z vrcholů

Pro vrcholy i hrany je dále potřeba naprogramovat operátory, pomocí kterých je budeme schopni vypsat do streamu:

- `std::ostream& operator<<(std::ostream& os, const Node<NData>& node)`
- `std::ostream& operator<<(std::ostream& os, const Edge<NData, EData>& edge)`

V obou případech je potřeba respektovat následující formát výstupu. Máme-li vrchol s identifikátorem n, serializujeme jej do řetězce `node (n {data})`, kde `data` (tedy řetězec mezi uvedeným párem složených závorek, který už pro zjednodušení žádné další takové zanořené závorky obsahovat nemůže) reprezentuje serializovaný datový obsah. Předpokládá se, že každý typ `NData`, který pro datový obsah vrcholů chceme použít, sám o sobě implementuje příslušný `operator<<`, a tedy se umí sám tímto způsobem vypsat. Podobně pro hranu s identifikátorem m mezi vrcholy i a j bude výstup v podobě `edge (i)-[m {data}]->(j)`, kde `data` opět tvoří serializovaný obsah instance `EData` získaný příslušným operátorem `<<`.

Pokud bychom například měli graf `DirectedGraph<std::string, std::string>`, mohl by obsahovat vrchol `node (1 {jedna})` nebo hranu `edge (1)-[3 {jedna-ctyri}]->(4)`.

Komponenty grafu

Třída grafu zpřístupní datové komponenty pro vrcholy a hrany následujícím způsobem:

- `Nodes<NData, EData>& nodes()`: vrátí referenci na komponentu starající se o vrcholy
- `Edges<NData, EData>& edges()`: analogicky pro hrany

K detailní funkcionalitě obou těchto komponent se ještě vrátíme, teď se zaměříme na základní principy a hlavně datový obsah, který si prostřednictvím nich musíme pamatovat. Pokud jde o třídu `Nodes<NData, EData>`, bude obsahovat gumové pole objektů vrcholů, tedy `Array<Node<NData>>`. Předpokládá se, že všechny vrcholy v tomto poli budou uspořádány přesně v pořadí jejich identifikátorů. Jinými slovy vrchol v gumovém poli na pozici n má identifikátor n. Analogicky třída `Edges<NData, EData>` bude obsahovat gumové pole objektů hran, opět uspořádaných podle jejich identifikátorů.

Třída hran však bude obsahovat také matici sousednosti, kterou budeme využívat k rychlému nalezení hrany na základě znalosti identifikátorů dvojice vrcholů, které by měla propojovat. Tato matice bude technicky obsahovat jen ukazatele na příslušné hrany, pro její implementaci můžete využít třeba standardní kontejner `std::vector`, gumové pole tedy v tomto případě používat nemusíte (není k tomu důvod). Jen zopakujme, že pozice (i, j) odpovídá hraně vedoucí z vrcholu s identifikátorem i do vrcholu j. V případě neorientovaného grafu musí platit, že matice je symetrická podle hlavní diagonály. Pro úplnost ještě dodejme, že chceme umět pracovat i se smyčkami, tedy hranami začínajícími a končícími ve stejném vrcholu.

Serializace grafu

Graf jako celek, myšleno jeho vrcholy a hrany, musíme být schopni vypsat na standardní výstup (nebo i do jiného streamu). Platí, že vždy nejprve vypíšeme všechny vrcholy, a to ve vzestupném pořadí podle jejich identifikátorů, teprve pak všechny hrany, opět vzestupně podle identifikátorů. Vždy jeden vrchol resp. jedna hrana na řádek. Podle očekávání bude vrcholy umět vypsat třída `Nodes<NData, EData>`, naopak hrany třída `Edges<NData, EData>`. V obou případech toho dosáhneme pomocí následujícího stejného rozhraní:

- `void print(std::ostream& os = std::cout) const`: vypíše komponentu vrcholů resp. hran

Dodáme ale i analogickou implementaci operátorů zápisu do streamu:

- `std::ostream& operator<<(std::ostream& os, const Nodes<NData, EData>& nodes)`
- `std::ostream& operator<<(std::ostream& os, const Edges<NData, EData>& edges)`

Následně do třídy grafu jako celku už snadno doplníme následující funkce:

- `void print(std::ostream& os = std::cout) const`: vypíše graf (tedy všechny vrcholy a následně všechny hrany) do uvedeného streamu
- `void print(const std::string& filename) const`: vypíše graf do výstupního souboru s uvedeným názvem

Serializace ukázkového grafu by pak mohla vypadat např. takto:

```
node (0 {nula})
node (1 {jedna})
node (2 {dva})
edge (0)-[0 {nula-jedna}]->(1)
edge (0)-[1 {nula-dva}]->(2)
```

Třída hran bude mít pro účely ladění a testování ještě následující funkci, která bude umět na vyžádání vypsat obsah již diskutované matice sousednosti:

- `void printMatrix(std::ostream& os = std::cout) const`: vypíše matici sousednosti do daného streamu; každý řádek matice vypíšeme na samostatný řádek výstupu; hodnoty (jednotlivé sloupce) v řádku matice pak vypisujeme za sebou a oddělujeme je symbolem `|`; pokud na dané pozici je hrana, vypíšeme její identifikátor; v opačném případě vypíšeme symbol `-`

Serializace matice sousednosti pro předchozí graf (ve variantě neorientovaného grafu) pak bude vypadat konkrétně takto:

```
-|0|1
0|-|-
1|-|-
```

Vkládání vrcholů a hran

Nově zkonstruovaný graf bude vždy prázdný, tedy nebude obsahovat žádný vrchol ani hranu. Ty pak můžeme přidávat jednotlivě pomocí funkcí, na které se teď zaměříme.

Komponenta pro vrcholy konkrétně nabídne následující rozhraní:

- `Node<NData>& add(size_t id, NData data)`: vloží do grafu nový vrchol s uvedeným identifikátorem a navázanými daty; v tuto chvíli se předpokládá, že uvedený identifikátor je validní, tedy odpovídá první další dosud nevyužité pozici v gumovém poli; pozor, že vložení nového vrcholu samozřejmě vynutí úpravy matice sousednosti; vrátí se reference na vytvořený objekt vrcholu
- `Node<NData>& add(NData data)`: totéž, jen identifikátor vkládaného vrcholu implicitně určíme jako další volný

Komponenta pro hrany analogicky nabídne následující funkce:

- `Edge<NData, EData>& add(size_t id, size_t source, size_t target, EData data)`: vloží do grafu novou hranu se specifikovaným identifikátorem a navázanými daty, a to hranu spojující uvedenou dvojici vrcholů; opět pro tuto chvíli předpokládáme korektnost všech argumentů; vrátí se reference na vytvořený objekt hrany
- `Edge<NData, EData>& add(size_t source, size_t target, EData data)`: totéž, jen identifikátor vkládané hrany opět implicitně určíme jako další volný

Pokud by se z nějakého důvodu stalo, že nový vrchol nebo novou hranu není možné do grafu vložit (např. z důvodu nepodařené alokace paměti ve standardním kontejneru nebo i našem gumovém poli), je potřeba takovou situaci korektně ošetřit a docílit atomicity. Tedy že operace vkládání se buď celá podaří, nebo nikoli. Tedy v případě dílčího neúspěchu je nutné všechny struktury grafu navrátit do stavu, který bude opět konzistentní.

Import grafu

Pro snazší vytváření grafů implementujeme i funkce, pomocí kterých budeme moci graf importovat ze vstupního souboru (nebo i jiného streamu). V tomto případě platí, že importovat vrcholy a hrany můžeme jen do již vytvořené instance grafu. Také platí, že importovací funkce je možné volat opakovaně, a tedy instanci grafu poskládat např. z více vstupních souborů obsahujících jednotlivé menší části celého grafu.

Funkce pro import definujeme na úrovni třídy pro graf jako celek:

- `void import(std::istream& is = std::cin)`: importuje vrcholy a hrany z uvedeného vstupního streamu
- `void import(const std::string& filename)`: importuje vrcholy a hrany ze vstupního souboru s uvedeným názvem

Ve vstupních datech mohou být vrcholy a hrany libovolně promíchány. Nemusí tedy platit, že jsou nejprve uvedeny vrcholy a pak jen hrany. Vždy ale platí, že každý řádek obsahuje jen jeden vrchol, nebo jednu hranu. Současně se také můžeme spolehnout na to, že záznam každého vrcholu i hrany je syntakticky správně formovaný a odpovídá struktuře, kterou jsme popisovali u příslušných funkcí na serializaci. Pokud jde o datový obsah navázaný na vrcholy resp. hrany, musíte jej načíst z řetězce mezi párem složených závorek, a to výhradně pomocí operátoru `>>`. Jinými slovy předpokládáme, že každý konkrétní datový typ, který v tomto smyslu chceme u vrcholů nebo hran používat, musí implementovat příslušnou funkci `std::istream& operator>>(std::istream& is, NData& data)` resp. analogicky pro `EData`.

Pro ilustraci uveďme validní obsah souboru, který i přes promíchané pořadí vrcholů a hran odpovídá našemu prvnímu příklad:

```
node (0 {nula})
node (1 {jedna})
edge (0)-[0 {nula-jedna}]->(1)
node (2 {dva})
edge (0)-[1 {nula-dva}]->(2)
```

Přístup k vrcholům a hranám

Třída pro komponentu vrcholů nabídne následující funkce, pomocí kterých budeme schopni přistupovat ke konkrétním vrcholům nebo se nad nimi dotazovat:

- `size_t size() const`: vrátí aktuální počet existujících vrcholů
- `bool exists(size_t id) const`: otestuje existenci vrcholu aneb vrátí `true`, pokud v grafu existuje vrchol s uvedeným identifikátorem, jinak `false`
- `Node<NData>& get(size_t id) const`: vrátí referenci na objekt vrcholu s uvedeným identifikátorem; pro tuto chvíli předpokládejme, že můžeme přistupovat jen k existujícím vrcholům
- `Node<NData>& operator[](size_t id) const`: totéž

Analogicky u třídy pro komponentu hran připravíme následující funkce:

- `size_t size() const`: vrátí aktuální počet existujících hran
- `exists(size_t id) const`: otestuje existenci hrany aneb vrátí `true`, pokud v grafu existuje hrana s uvedeným identifikátorem, jinak `false`
- `exists(size_t source, size_t target) const`: totéž, ale tentokráte se detekuje existence hrany mezi uvedenou dvojicí vrcholů určených jejich identifikátory
- `Edge<NData, EData>& get(size_t id) const`: vrátí referenci na objekt hrany s uvedeným identifikátorem; pro tuto chvíli opět předpokládejme jen validní identifikátor
- `Edge<NData, EData>& get(size_t source, size_t target) const`: totéž, ale opět pro uvedenou dvojici vrcholů podle jejich identifikátorů

Nad komponentou hran je dále ještě potřeba implementovat operátor hranatých závorek, který by nám umožňoval přistupovat ke konkrétním hranám. V tomto případě ale nechceme používat identifikátory hran, ale dvojice identifikátorů odpovídajících vrcholů, podobně jako je tomu u poslední uvedené funkce `get(source, target)`. Chceme tedy umět používat kód ve tvaru `graph.edges()[source][target]`.

Iterátory nad grafem

Komponenty vrcholů a hran dále nabídnou iterátory umožňující iterovat nad jednotlivými vrcholy resp. hranami v grafu. Pochopitelně není nutné implementovat nic nového, stačí jen následujícím způsobem zpřístupnit iterátory z gumového pole.

Konkrétně tedy nad komponentami vrcholů a hran potřebujeme definovat následující dvě dvojice funkcí se zřejmým významem:

- `typename Array<Node<NData>>::iterator begin() const`
- `typename Array<Node<NData>>::iterator end() const`
- `typename Array<Edge<NData, EData>>::iterator begin() const`
- `typename Array<Edge<NData, EData>>::iterator end() const`

Manipulace s grafy

Nad třídou grafu je dále potřeba naprogramovat následující konstruktory a operátory umožňující manipulaci s grafy jako celky:

- `Graph(const Graph& other)`: standardní copy constructor
- `Graph(Graph&& other) noexcept`: standardní move constructor
- `Graph& operator=(const Graph& other)`: standardní copy assignment operátor
- `Graph& operator=(Graph&& other) noexcept`: standardní move assignment operátor

Chybové situace

Doteď jsme u většiny funkcí předpokládali, že jejich argumenty jsou validní a ani nic jiného se nemůže pokazit. V následující části všechny dotčené části kódu upravíme takovým způsobem, abychom mohli většinu zajímavých krajních a chybových situací detekovat a korektně ošetřit.

Za tímto účelem vytvoříme jednoduchou hierarchii vlastních výjimek. Abstraktním předkem bude třída `Exception`. Definovat bude jedinou funkci, a to `std::string& what()`, jejímž účelem bude vrátit konkrétní a podrobný textový popis vzniklé chyby. Od předka následně odvodíme následující odvozené typy výjimek: `NonexistingItemException`, `ConflictingItemException`, `InvalidIdentifierException`, `UnavailableMemoryException` a `FileProcessingException`.

Následující přehled obsahuje situace, ve kterých je potřeba příslušné typy výjimek vyhodit, a současně také konkrétní textové hlášky, které s nimi musíte asociovat. Ty pak obsahují zástupné symboly `$něco`, které nahradíte příslušnou konkrétní hodnotou.

- `Node<NData>& Nodes<NData, EData>::get(size_t id) const`:
  `Node<NData>& Nodes<NData, EData>::operator[](size_t id) const`:
  - `NonexistingItemException("Attempting to access a nonexisting node with identifier $id, only $size nodes are available");` pokud se přistupuje k vrcholu, který neexistuje
- `Node<NData>& Nodes<NData, EData>::add(size_t id, NData data)`:
  - `InvalidIdentifierException("Attempting to add a new node with invalid identifier $id, expected $size instead");` pokud je použit neplatný (nenavazující) identifikátor
  - `ConflictingItemException("Attempting to add a new node with identifier $id which already is associated with another existing node");` pokud je použit identifikátor již existujícího vrcholu
  - `UnavailableMemoryException("Unable to insert a new node record into the underlying container of nodes");` pokud není možné vložit objekt vrcholu do gumového pole vrcholů
  - `UnavailableMemoryException("Unable to extend the underlying adjacency matrix container for edges");` pokud není možné do matice sousednosti přidat nový sloupec a řádek
- `Edge<NData, EData>& Edges<NData, EData>::get(size_t id) const`:
  - `NonexistingItemException("Attempting to access a nonexisting edge with identifier $id, only $size edges are available");` pokud se přistupuje k hraně, která neexistuje
- `Edge<NData, EData>& Edges<NData, EData>::get(size_t source, size_t target) const`:
  - `NonexistingItemException("Attempting to access an edge between a nonexisting pair of nodes with identifiers $source and $target, only $size nodes are available");` pokud se přistupuje k hraně pro dvojici vrcholů, kde jeden, druhý nebo dokonce oba neexistují
  - `NonexistingItemException("Attempting to access a nonexisting edge between a pair of nodes with identifiers $source and $target");` pokud se přistupuje k neexistující hraně mezi jinak validní dvojicí vrcholů
- `bool Edges<NData, EData>::exists(size_t source, size_t target) const`:

  - `NonexistingItemException("Attempting to test the existence of an edge between a nonexisting pair of nodes with identifiers $source and $target, only $size nodes are available");` pokud jeden, druhý nebo dokonce oba zmíněné vrcholy neexistují

- Opetárory `[]` pro přístup k hranám:
  - `NonexistingItemException("Attempting to access an edge outgoing from a nonexisting source node with identifier $source, only $size nodes are available");` pokud neexistuje vrchol, okdkud má hrana vést
  - `NonexistingItemException("Attempting to access an edge incoming to a nonexisting target node with identifier $target, only $size nodes are available");` pokud neexistuje vrchol, kam má hrana vést
  - `NonexistingItemException("Attempting to access a nonexisting edge between a pair of nodes with identifiers $source and $target");` pokud se přistupuje k neexistující hraně mezi jinak validní dvojicí vrcholů

- `Edge<NData, EData>& Edges<NData, EData>::add(size_t id, size_t source, size_t target, EData data)`:
  - `InvalidIdentifierException("Attempting to add a new edge with invalid identifier $id, expected $size instead");` pokud je použit neplatný (nenavazující) identifikátor
  - `ConflictingItemException("Attempting to add a new edge with identifier $id which already is associated with another existing edge");` pokud hrana s uvedeným identifikátorem již existuje
  - `NonexistingItemException("Attempting to add a new edge between a nonexisting pair of nodes with identifiers $source and $target, only $size nodes are available");` pokud jeden, druhý nebo dokonce oba zmíněné vrcholy neexistují
  - `ConflictingItemException("Attempting to add a new edge between a pair of nodes with identifiers $source and $target which already are connected with another existing edge");` pokud už mezi danou dvojicí vrcholů hrana existuje
  - `UnavailableMemoryException("Unable to insert a new edge record into the underlying container of edges");` pokud není možné vložit objekt hrany do gumového pole hran

- `void Graph<NData, EData>::print(const std::string& filename) const`:
  - `FileProcessingException("Unable to open an output file $filename");` pokud není možné otevřít daný výstupní soubor

- `void Graph<NData, EData>::import(const std::string& filename)` :
  - `FileProcessingException("Unable to open an input file $filename");` pokud není možné otevřít daný vstupní soubor

Doxygen dokumentace

Veškerý vytvořený kód je potřeba doplnit o kompletní a kvalitní programátorskou dokumentaci vytvořenou pomocí nástroje `Doxygen`. Předpokládejte konfigurační soubor, který kromě výchozích nastavení bude obsahovat direktivy `EXTRACT_PRIVATE = YES` a `EXTRACT_STATIC = YES`. Samotný konfigurační soubor však s projektem neodevzdávejte.

Závěrečné informace

V rámci projektu předpokládejte soubor `Main.cpp`, jehož obsahu bude odpovídat následující struktuře:

```
#include "Graph.h"
int main(int argc, char** argv) {
    ...
}
```

Tento soubor tedy bude obsahovat výhradně jen deklaraci na import jediného hlavičkového souboru `Graph.h` a také implementaci `main` funkce. Během práce na domácím úkolu si do funkce `main` (nebo i jinam v tomto souboru) doplňte jakýkoli další kód, pomocí kterého aplikaci odladíte. Váš `Main.cpp` soubor však do systému odevzdávat nebudete, Vaše řešení bude automaticky doplněno o jiný již předpřipravený `Main.cpp` soubor, pomocí kterého bude Vaše implementace automaticky otestována. Jinými slovy je potřeba odevzdat všechny zdrojové soubory s výjimkou souboru `Main.cpp`.

Cílem úkolu je ukázat schopnost práce s konstrukty, se kterými jsme se už od začátku semestru setkali. Kromě základních dovedností tedy jde zejména o práci s textovými soubory, streamy, návrh tříd, použití konstruktorů a destruktorů, dědičnost, virtuální metody, dynamickou alokaci, šablony, chytré ukazatele, operátory, iterátory nebo výjimky.

Předložená implementace pochopitelně musí být korektní a stabilní. Hodnotit se ale bude i celková kvalita kódu. Tedy zejména avšak ne výlučně organizace kódu do jednotlivých souborů, tříd a funkcí, použití hlavičkových souborů, pojmenovávání souborů, funkcí nebo i proměnných, celkový vizuální styl kódu a indentace, předávání argumentů hodnotou nebo referencí, kvalita návrhu tříd a použití dědičnosti a virtuálních metod, zbytečné neopakování stejného kódu, deklarace významných konstant, ošetřování chybových situací stejně jako využívání standardních knihoven, kontejnerů nebo funkcí.

Za úkol je možné získat až 25 bodů, každý započatý týden zpoždění bude penalizován 10 body.