NPRG041 – 2023/24 ZS – Cvičení MS – **Velký úkol A02**

Grafy

Gumové pole

Pro potřeby zamýšlené implementace grafu je nejprve potřeba naprogramovat vlastní šablonovaný kontejner gumového pole `lib::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 realokace vnitřního úložiště. To celkově přinese nejenom šetrnější práci s dynamicky alokovanou pamětí, současně ale nikdy nedojde k potřebě přemisťování existujících prvků v paměti (v lepším případě přemisťováním, v horším kopírováním dle schopností těchto prvků), a tedy k zneplatnění uživatelem držených referencí nebo ukazatelů na tyto prvky. Obě tyto vlastnosti budou pro náš graf klíčové.

Rozsah funkcionality kontejneru gumového pole a přidružených iterátorů se předpokládá na úrovni posledních dvou malých domácích úkolů. V praktické rovině však není potřeba mít tuto implementaci zcela kompletní, postačí mít k dispozici jen ty funkce, které opravdu v rámci grafu používat budete chtít. Zejména v případě iterátorů nejspíše postačí dosažení jen kategorie dopředných iterátorů. V případě potřeby je naopak možné stávající funkcionalitu pole ještě rozšířit. Konkrétně by se mohl hodit např. další konstruktor v podobě `Array(size_t count, const element& value)`, který vytvoří novou instanci gumového pole obsahujícího `count` kopií vzorového prvku `value`, případně i globální funkci `void swap(Array<element>& a, Array<element>& b)`, která pomocí jednotlivých přesunů navzájem prohodí všechny vnitřní komponenty obou instancí gumových polí, a tedy prohodí obě pole jakožto celky.

Reprezentace grafu

Hlavním cílem tohoto domácího úkolu je naprogramovat šablonovanou třídu pro 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 rovněž je, že na každý vrchol i hranu potřebujeme umět navázat další libovolné informace, které si u nich budeme chtít uložit.

Třída pro graf jako takový `Graph<NData, EData>` bude poskytovat základní společné rozhraní pro obě uvedené varianty grafů. Bude však abstraktní, nebude od ní možné vytvářet instance. To bude možné až pro konkrétní odvozenou variantu orientovaného grafu `DirectedGraph<NData, EData>` a 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.

Třída grafu bude z hlediska logického členění obsahovat dvě oddělené komponenty, a to instanci třídy `Nodes<NData, EData>` pro řešení veškeré funkcionality týkající se vrcholů grafu a analogicky instanci třídy `Edges<NData, EData>` pro hrany. Každý jednotlivý uzel bude realizován jako instance třídy `Node<NData>` a hrana jako instance třídy `Edge<EData>`, a to bez ohledu na to, jestli je, nebo není orientovaná. Jinými slovy hrany samy o sobě tento fakt rozlišovat nepotřebují.

Přestože velká část návrhu celého řešení nebude nijak omezena, některé aspekty očekávaného rozhraní je jako obvykle nutné dodržet. V řadě případů budeme navíc potřebovat varianty funkcí pro modifikovatelné, ale i konstantní grafy nebo jejich součásti. Abychom hlavičky takových funkcí nemuseli v dalším textu neustále zbytečně zdvojovat, jen je označíme pomocí symbolu [♠].

Vrcholy a hrany

Každý vrchol i hrana mají přidělený unikátní identifikátor, a to z množiny $\mathbb{N}_0$, tedy přirozených čísel včetně nuly. Tyto identifikátory jsou imutabilní, tedy jakmile jsou jednou přiděleny, nikdy později se už změnit nemohou. 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`, v kódu však budeme pracovat s aliasem `Identifier` (pro usnadnění případné budoucí změny). Konečně dodejme, že graf bude umožňovat pouze přidávání nových vrcholů a hran (a to postupně v pořadí právě podle jejich identifikátorů), ty existující odebírat nebudeme chtít.

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

- `Identifier 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 určujícím danou hranu:

- `Identifier getId() const`: vrátí hodnotu identifikátoru dané hrany
- `Identifier getSource() const`: vrátí identifikátor zdrojového vrcholu
- `Identifier getTarget() const`: vrátí identifikátor cílového vrcholu
- `EData& getData()` [♠]: vrátí referenci na datový obsah navázaný na danou hranu

Pro vrcholy i hrany dále naprogramujeme operátory `<<`, pomocí kterých je budeme schopni vypsat do libovolného výstupního streamu:

- `std::ostream& operator<<(std::ostream& stream, const Node<NData>& node)`
- `std::ostream& operator<<(std::ostream& stream, const Edge<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) reprezentuje serializovaný datový obsah. Ať už bude vypadat jakkoli, nikdy nebude obsahovat další takové zanořené složené závorky. Předpokládá se, že každý typ `NData`, který pro datový obsah vrcholů budeme chtít použít, rovněž implementuje příslušný operátor `<<`, a tedy se sám umí tímto způsobem vypsat. Podobně pro hranu s identifikátorem m mezi vrcholy s identifikátory i a j budeme výstup očekávat v podobě `edge (i)-[m {data}]->(j)`, kde `data` opět tvoří serializovaný obsah instance `EData` získaný příslušným operátorem `<<`. Vypsání jednotlivých vrcholů a hran nebudeme doplňovat koncem řádku.

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

Komponenty grafu

Každý graf musí z hlediska svého datového obsahu uchovávat minimálně následující tři datové položky: 1) vnitřní úložiště pro instance vrcholů, 2) vnitřní úložiště pro instance hran a dále také 3) matici sousednosti, kterou budeme používat k rychlému nalezení hran na základě znalosti identifikátorů dvojic vrcholů, které by měly propojovat. Ve všech těchto třech případech předpokládáme, že se bude používat výhradně naše gumové pole `lib::Array`, nikoli standardní kontejner vektoru `std::vector`. Kromě uvedených položek je samozřejmě v případě potřeby možné uchovávat i další potřebné údaje.

Třída grafu zpřístupní 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

Pokud jde o umístění výše uvedených datových položek, nabízejí se dvě základní řešení: buď je všechny umístíte na jedno místo přímo do třídy grafu `Graph<NData, EData>`, nebo je můžete rozdělit a odpovídajícím způsobem umístit až do tříd obou uvedených komponent `Nodes<NData, EData>` resp. `Edges<NData, EData>`. Obě varianty přitom nabízí určité výhody, mají však i svá úskalí. Dobře si proto rozmyslete, který přístup je vám bližší. V každém případě musí být obě komponenty zachovány, i kdyby to bylo jen kvůli zapouzdření očekávané funkcionality, kterou jednoduše nelze poskytovat jen na jednom místě přímo v grafu jako takovém.

Pokud jde o kontejner gumového pole objektů vrcholů, 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. Totéž bude analogicky platit i pro kontejner gumového pole objektů hran.

Matice sousednosti bude technicky obsahovat ukazatele na příslušné hrany. Jen poznamenejme, že pozice (i, j) odpovídá hraně vedoucí z vrcholu s identifikátorem i do vrcholu s identifikátorem j. V případě neorientovaného grafu musí platit, že matice je symetrická podle hlavní diagonály. Smyčky, tedy hrany začínající i končící ve stejném vrcholu, jsou rovněž podporovány.

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 jejich identifikátorů. Vždy jeden vrchol resp. jedna hrana na řádek, každý pak ukončíme pomocí `std::endl`. 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& stream = std::cout) const`: vypíše komponentu vrcholů resp. hran

Dodáme ale i analogickou implementaci operátorů zápisu do streamu:

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

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

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

Serializace ukázkového grafu by pak mohla vypadat např. takto:

```
node (0 {zero})
node (1 {one})
node (2 {two})
edge (0)-[0 {zero-one}]->(1)
edge (0)-[1 {zero-two}]->(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& stream = std::cout) const`: vypíše obsah matice 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) by pak vypadala 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(Identifier id, const 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(Identifier id, NData&& data)`: totéž analogicky
- `Node<NData>& add(const NData& data)`: totéž, jen identifikátor nově vkládaného vrcholu implicitně určíme jako další volný
- `Node<NData>& add(NData&& data)`: totéž analogicky

Komponenta pro hrany analogicky nabídne následující funkce:

- `Edge<EData>& add(`
  `Identifier id, Identifier source, Identifier target, const 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<EData>& add(`
  `Identifier id, Identifier source, Identifier target, EData&& data`
  `)`: totéž analogicky
- `Edge<EData>& add(`
  `Identifier source, Identifier target, const EData& data`
  `)`: totéž, jen identifikátor vkládané hrany opět implicitně určíme jako další volný
- `Edge<EData>& add(`
  `Identifier source, Identifier target, EData&& data`
  `)`: totéž analogicky

Pokud by se z nějakého důvodu stalo, že nový vrchol nebo novou hranu není možné do grafu vložit (z důvodu nepodařené alokace paměti v rámci našeho kontejneru gumového pole nebo kvůli nevydařenému kopírování instancí navázaných dat), je potřeba takovou situaci korektně ošetřit a docílit atomicity. Tedy že operace vkládání se buď celá podaří, nebo nikoli. V případě dílčího neúspěchu je tedy nutné všechny struktury grafu navrátit do stavu, který bude opět konzistentní.

Import grafu

Pro snadnější vytváření grafů implementujeme i funkce, pomocí kterých budeme moci obsah grafu načíst a 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, stejně jako je libovolně prokládat již diskutovanými funkcemi na manuální přidávání jednotlivých vrcholů nebo hran.

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

- `void import(std::istream& stream = std::cin)`: importuje vrcholy a hrany z daného 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 jen vrcholy a pak už 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, musíme 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`.

Předpokládá se, že při realizaci importu budeme vnitřně používat již dříve uvedené funkce na přidávání jednotlivých vrcholů a hran. Případné zcela prázdné řádky se přeskočí. Pro ilustraci uveďme validní obsah souboru, který i přes promíchané pořadí vrcholů a hran odpovídá našemu prvnímu příkladu:

```
node (0 {zero})
node (1 {one})
```

```
edge (0)-[0 {zero-one}]->(1)
node (2 {two})
edge (0)-[1 {zero-two}]->(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 vrcholů v grafu
- `bool exists(Identifier id) const`: otestuje existenci vrcholu aneb vrátí `true`, pokud v grafu existuje vrchol s uvedeným identifikátorem, jinak `false`
- `Node<NData>& get(Identifier id)` [♠]: vrátí referenci na vrchol s uvedeným identifikátorem; pro tuto chvíli předpokládejme, že můžeme přistupovat jen k existujícím vrcholům
- `Node<NData>& operator[](size_t id)` [♠]: totéž

Analogicky u třídy pro komponentu hran připravíme následující funkce:

- `size_t size() const`: vrátí aktuální počet hran v grafu
- `bool exists(Identifier id) const`: otestuje existenci dané hrany aneb vrátí `true`, pokud v grafu existuje hrana s uvedeným identifikátorem, jinak `false`
- `bool exists(Identifier source, Identifier target) const`: totéž, jen pro detekci existence hrany mezi uvedenou dvojicí vrcholů určených jejich identifikátory
- `Edge<EData>& get(Identifier id)` [♠]: vrátí referenci na hranu s daným identifikátorem; pro tuto chvíli opět předpokládejme jen validní identifikátor
- `Edge<EData>& get(Identifier source, Identifier target)` [♠]: totéž, jen pro danou dvojici vrcholů podle jejich identifikátorů

Do komponenty hran dále přidáme i operátor `[]`, který nám umožní přistupovat ke konkrétním hranám. Nikoli však jednoúrovňově pomocí identifikátorů takových hran, ale dvouúrovňově pomocí identifikátorů dvojice odpovídajících vrcholů `source` resp. `target` (v tomto pořadí). 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 pro komponentu vrcholů (analogicky i pro hrany) tedy potřebujeme definovat následující dvě dvojice funkcí se zřejmým významem:

- `typename lib::Array<Node<NData>>::iterator begin()`
- `typename lib::Array<Node<NData>>::iterator end()`
- `typename lib::Array<Node<NData>>::const_iterator begin() const`
- `typename lib::Array<Node<NData>>::const_iterator end() const`

Manipulace s grafy

Nad třídou grafu je dále potřeba naprogramovat standardní kopírovací a přesouvací (vykrádací) konstruktory a operátory přiřazení, díky nimž budeme schopni manipulovat s grafy jako celky. Pro jistotu dodejme, že každý vrchol i hrana jsou v logickém i fyzickém vlastnictví grafu, do kterého patří. Pokud tedy kopírujeme grafy jako celky, musíme kopírovat i veškerý jejich obsah.

Chybové situace

Doteď jsme u většiny funkcí předpokládali, že jejich parametry byly validní a ani nic jiného se nemůže pokazit. V této části popíšeme, jak všechny dotčené části kódu upravíme takovým způsobem, abychom mohli většinu krajních a chybových situací detekovat a také korektně ošetřit.

Za tímto účelem vytvoříme jednoduchou hierarchii vlastních výjimek. Abstraktním předkem bude naše vlastní třída `Exception`. Poskytne jedinou funkci, a to `const char* message() const`, jejímž účelem bude vrátit konkrétní textový popis vzniklé chyby. Z praktických důvodů však budeme potřebovat rozlišit dva druhy konkrétních odvozených výjimek, a to pro výjimky se statickými textovými zprávami (známými v době kompilace) a dynamicky zkonstruovanými zprávami (poskládanými až za běhu).

V první kategorii bude jediná odvozená třída, a to `MemoryException`. Důvodem je, že bude vyvolávána při nedostatku paměti pro dynamickou alokaci. V takovém okamžiku samozřejmě není možné vyhazovat takové výjimky, které by ji samy potřebovaly. Ve druhé kategorii pak budou všechny ostatní odvozené typy výjimek, konkrétně `IdentifierException`, `ElementException`, `ConflictException` a `FileException`.

Následující přehled obsahuje konkrétní ošetřované situace, jejich pořadí, očekávané typy výjimek, stejně jako očekávané zprávy. Ty pak obsahují zástupné symboly ve tvaru `$něco`, které jen nahradíme za příslušnou konkrétní hodnotu:

- Přístup ke konkrétnímu vrcholu pomocí funkce `get` nebo operátoru `[]` na základě jeho identifikátoru:
  - `ElementException("Node with identifier $id does not exist");`
    pokud se přistupuje k vrcholu, který neexistuje
- Přidání nového vrcholu pomocí funkce `add`:
  - `IdentifierException("Invalid node identifier $id requested");`
    pokud je použit neplatný (nenavazující) identifikátor
  - `ConflictException("Node with identifier $id already exists");`
    pokud je použit identifikátor již existujícího vrcholu
  - `MemoryException("Unavailable memory for a new node in the nodes container");`
    pokud není možné vložit objekt vrcholu do gumového pole vrcholů
  - Původní výjimka v případě, že selže vytváření kopie navázaných dat
  - `MemoryException("Unavailable memory for the adjacency matrix extension");`
    pokud není možné do matice sousednosti přidat nový sloupec a řádek
- Přístup ke konkrétní hraně pomocí funkce `get` na základě jejího identifikátoru:
  - `ElementException("Edge with identifier $id does not exist");`
    pokud se přistupuje k hraně, která neexistuje
- Přístup ke konkrétní hraně pomocí funkce `get` nebo operátoru `[]` na základě dvojice vrcholů:
  - `ElementException("Source node with identifier $source does not exist");`
    pokud zdrojový vrchol neexistuje
  - `ElementException("Target node with identifier $source does not exist");`
    pokud cílový vrchol neexistuje
  - `ElementException("Edge between nodes $source and $target does not exist");`
    pokud se přistupuje k neexistující hraně mezi jinak validní dvojicí vrcholů
- Test existence hrany pomocí funkce `exists` na základě dvojice vrcholů:
  - `ElementException("Source node with identifier $source does not exist");`
    pokud zdrojový vrchol neexistuje
  - `ElementException("Target node with identifier $source does not exist");`
    pokud cílový vrchol neexistuje
- Přidání nové hrany pomocí funkce `add`:
  - `IdentifierException("Invalid edge identifier $id requested");`
    pokud je použit neplatný (nenavazující) identifikátor
  - `ConflictException("Edge with identifier $id already exists");`
    pokud hrana s uvedeným identifikátorem již existuje

  - `ElementException("Source node with identifier $source does not exist");`
    pokud zdrojový vrchol neexistuje
  - `ElementException("Target node with identifier $source does not exist");`
    pokud cílový vrchol neexistuje
  - `ConflictException("Edge between nodes $source and $target already exists");`
    pokud už mezi danou dvojicí vrcholů hrana existuje
  - Původní výjimka v případě, že selže vytváření kopie navázaných dat
  - `MemoryException("Unavailable memory for a new edge in the edges container");`
    pokud není možné vložit objekt hrany do gumového pole hran

- Vypsání grafu do uvedeného souboru pomocí funkce `print`:
  - `FileException("Unable to open output file $filename");`
    pokud není možné otevřít daný výstupní soubor

- Import grafu z uvedeného souboru pomocí funkce `import`:
  - `FileException("Unable to open input file $filename");`
    pokud není možné otevřít daný vstupní soubor

Závěrečné pokyny

Odevzdejte všechny vytvořené zdrojové soubory (patrně jen `Graph.h` a `Array.h`) kromě souboru `Main.cpp`, který již součástí připraveného testu je. Obsahuje jedinou direktivu `#include <Graph.h>` a dále funkci `main`, která bude průběh testu jako obvykle řídit.

Cílem úkolu je ukázat schopnost práce s konstrukty, se kterými jsme se 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, ukazatele, operátory, iterátory nebo výjimky.

Předložená implementace pochopitelně musí být korektní a stabilní, kompilace musí proběhnout bez jakýchkoli varování. 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 parametrů hodnotou nebo referencí, kvalita návrhu tříd a použití dědičnosti a virtuálních metod, zbytečné neopakování stejných fragmentů kódu, používání pojmenovaných konstant, ošetřování chybových situací stejně jako využívání standardních knihoven, kontejnerů nebo funkcí.