NPRG041 – 2023/24 ZS – Cvičení MS – **Malý úkol C10**

Matice

Cílem tohoto úkolu je navrhnout a naprogramovat třídu umožňující reprezentaci matic obsahujících prvky libovolného číselného typu, stejně jako nad nimi následně naprogramovat vybrané základní operace.

Třídu matice konkrétně implementujeme jako šablonovanou, s parametry v podobě `template<typename element, size_t height, size_t width>`, kde `element` je datový typ prvku (např. `int`, ale také i vhodná uživatelsky definovaná třída implementující požadované chování), `height` je výška matice (počet řádků) a `width` šířka (počet sloupců).

Vnitřní úložiště pro prvky matice realizujeme pomocí privátní položky využívající kontejnerového pole `std::array`, a to v takové podobě, že celou matici rozvineme do jedné ploché lineární struktury. Jinými slovy toto pole nesmí obsahovat další vnořená pole, ale rovnou už jednotlivé prvky. Konkrétně předpokládejme indexovou aritmetiku, kdy prvek na logických souřadnicích `[row][column]` v tomto interním poli umístíme na pozici `[row * width + column]`.

Pro konstrukci matice nabídneme jediný parametrický konstruktor v podobě `Matrix(const element& value = 0)`, kdy uvedeným vzorovým prvkem `value` vyplníme všechny pozice vytvořené matice. Pro základní práci s maticí nabídneme následující členské funkce:

- `const element& get(size_t row, size_t column) const`: vrátí referenci na prvek na uvedených logických souřadnicích, tedy na řádku `row` a sloupci `column`
- `void set(size_t row, size_t column, const element& value)`: nastaví novou hodnotu prvku na uvedených logických souřadnicích
- `void print(std::ostream& stream = std::cout) const`: vypíše obsah celé aktuální matice do daného výstupního streamu, a to v následující podobě: nejprve do vnějšího páru hranatých závorek umístíme čárkami a mezerami oddělený výčet jednotlivých řádků matice, každý z nich pak opět zapouzdříme do vnitřního páru hranatých závorek a jeho jednotlivé prvky opět oddělíme čárkami a mezerami; pro úplnost dodejme konkrétní příklad: `[[1, 2], [3, 4], [5, 6]]`; na konci žádný konec řádku nepřidáváme; prvky samotné vypisujeme pomocí jejich operátoru `<<`

Obecně se předpokládá, že budeme vyžadovat přístup jen k prvkům na validních souřadnicích, tedy těch nepřekračujících skutečné rozměry matice. V opačném případě bude chování dotčených funkcí nedefinováno. Pokud jde o vypisování matic, pro zvýšení komfortu přidáme i příslušnou globální funkci operátoru `<<` pro zápis do streamu.

Dále implementujeme následující dva operátory pro preinkrementaci a postinkrementaci, a to formou členských funkcí nad maticí. V obou případech inkrementací myslíme to, že aktuální hodnotu každého jednotlivého prvku matice zvýšíme o 1. Přesněji řečeno, že nad každým prvkem zavoláme jeho vlastní operaci preinkrementace resp. postinkrementace. Nezapomeňme totiž, že nad prvky matice z hlediska jejich typu nemáme plnou kontrolu, takže musíme zachovat zamýšlenou sémantiku (jakkoli neobvyklé a divné by to totiž mohlo být, operace inkrementace a přičtení 1 by totiž mohly mít rozdílné významy).

- `Matrix& operator++()`: provedeme preinkrementaci matice, tedy vrátíme referenci na stávající matici s novými hodnotami prvků po provedení jejich inkrementace
- `Matrix operator++(int)`: provedeme postinkrementaci matice, tedy vrátíme nově vytvořenou kopii matice s původními hodnotami před provedením vlastní inkrementace hodnot stávající matice; hodnota předaného parametru se ignoruje

Představme si nyní, že bychom chtěli uživatelům našich matic nabídnout rozhraní, pomocí kterého by mohli používat operátor hranatých závorek pro přístup ke konkrétním prvkům matice, a to na základě skutečných pozic takových prvků v našem interním úložišti. Např. `matrix[4]` by v matici se 3 řádky a 2 sloupci odpovídalo přístupu k prvku na logických souřadnicích `[2][0]`, tedy k prvku ve třetím řádku a prvním sloupci.

Vlastně bychom takto jen zprostředkovali přesně takové chování operátoru `[]`, které již vnitřní kontejner pole nabízí. Za tímto účelem by stačilo, pokud bychom v naší třídě matice nabídli metody v podobě `element& operator[](size_t position)` a `const element& operator[](size_t position) const`. Dvě oddělené

varianty proto, abychom použití takového operátoru dovolili nad modifikovatelnými i nemodifikovatelnými instancemi matic.

Je zřejmé, že implementace takovéto dvojice operátorů by byla přímočará, určitě by ale nebyly přívětivé z uživatelského hlediska. Implementovat je tedy nebudeme, namísto nich naopak navrhneme operátory hranatých závorek, které nám umožní přístup ke konkrétním prvkům na základě našich logických souřadnic.

Chceme tedy mít možnost dvouúrovňové indexace, nejprve podle konkrétního řádku, následně podle konkrétního sloupce. Např. `matrix[2][0]`. Aby něco takového bylo možné, bude nejprve nutné navrhnout pomocnou třídu, prostřednictvím které si nejprve požadavek přístupu k vybranému řádku zapamatujeme a teprve následně požadavek přístupu i ke konkrétnímu sloupci zrealizujeme. Aneb operátory `[]` na první úrovni (jakožto metody na třídě samotné matice) nejprve vrátí instanci naší pomocné třídy, teprve následně operátory `[]` na druhé úrovni (jakožto metody na pomocné třídě požadavku) zpřístupní požadovaný prvek jako takový.

Abychom se naučili další nové techniky, zmíněnou pomocnou třídu požadavku naprogramujeme jako vnitřní privátní třídu třídy matice. Z hlediska datových položek bude obsahovat konstantní referenci na matici (`const Matrix&`) a číslo požadovaného řádku. Příslušný parametrický konstruktor záměrně uděláme jako privátní a třídě matice jej zpřístupníme pomocí konstrukce `friend`. Pro odebrání ochrany konstantnosti při přístupu ke konkrétnímu prvku použijeme mechanismus přetypování `const_cast`.

Nakonec ještě implementujeme následující operace nad maticemi, tentokráte ve všech případech pomocí globálních funkcí:

- `Matrix<element, height, width> operator+(`
  `const Matrix<element, height, width>& matrix,`
  `const element& increment`
  `)`: vrátíme kopii matice `matrix`, kde k hodnotě každého prvku přičteme parametr `increment`
- `Matrix<element, height, width> operator*(`
  `const Matrix<element, height, width>& matrix,`
  `const element& factor`
  `)`: vrátíme kopii matice `matrix`, kde hodnotu každého prvku vynásobíme parametrem `factor`
- `Matrix<element, height, width> operator+(`
  `const Matrix<element, height, width>& matrix1,`
  `const Matrix<element, height, width>& matrix2`
  `)`: sečteme dvě matice `matrix1` a `matrix2` stejných rozměrů
- `Matrix<element, height, width> operator*(`
  `const Matrix<element, height, depth>& matrix1,`
  `const Matrix<element, depth, width>& matrix2`
  `)`: vynásobíme dvě matice `matrix1` a `matrix2` vzájemně kompatibilních rozměrů

Oproti běžným zvyklostem je nutné veškerý kód šablonovaných tříd a funkcí umístit do hlavičkových souborů. Předpokládejme, že v našem případě bude jeden jediný a bude se jmenovat `Matrix.h`. Jakkoli to obecně není potřeba, pro rozšíření našich dovedností opět důsledně oddělíme definice obou našich tříd (tedy že definici vnitřní pomocné třídy požadavku uvedeme mimo tělo třídy matice) a taktéž oddělíme definice všech jednotlivých členských funkcí obou našich tříd od jejich deklarací (tedy že jejich implementace uvedeme mimo tělo příslušné třídy). K tomu bude v určitých komplikovanějších případech kvůli závislým jménům, u nás konkrétně při popisu návratových typů, nutné použít klíčové slovo `typename`.

Odevzdejte pouze a jenom již zmíněný soubor `Matrix.h`. Jako obvykle předpokládejte, že hlavní soubor `Main.cpp` s funkcí `main` je již součástí připraveného testu. Cílem tohoto úkolu je seznámit se s návrhem šablonovaných tříd a funkcí, vnitřními třídami, kontejnerem pole `std::array`, přetypováním `const_cast` a také implementací dalších vlastních operátorů, konkrétně aritmetických a operátoru indexace.