NPRG041 – 2024/25 ZS – Cvičení MS – **Malý úkol C08**

Matice

Cílem tohoto úkolu je navrhnout a naprogramovat třídu umožňující reprezentaci matic obsahujících prvky libovolného číselného typu a následně nad takovými maticemi naprogramovat vybrané základní operace.

Třídu matice konkrétně naprogramujeme v podobě šablonové třídy s následujícími parametry `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 fyzickou 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:

- `Element& get(size_t row, size_t column)`: vrátí modifikovatelnou referenci na prvek umístěný na uvedených logických souřadnicích v modifikovatelné matici, tedy na řádku `row` a sloupci `column`
- `const Element& get(size_t row, size_t column) const`: totéž pro konstantní referenci a matici
- `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. Žádné kontroly vstupních parametrů tedy provádět nebudeme a plnou odpovědnost za korektní používání matic přenášíme na volající. 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, divné a nedoporučované by to totiž mohlo být, operace inkrementace a přičtení 1 by totiž mohly mít rozdílné významy, stejně jako by se vzájemně mohly lišit obě varianty chování inkrementace).

- `Matrix& operator++()`: provede preinkrementaci matice, tedy vrátí referenci na stávající matici s novými hodnotami prvků po provedení jejich inkrementace
- `Matrix operator++(int)`: provede postinkrementaci matice, tedy vrátí 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, analogicky jako u našich `get` funkcí.

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`
  `)`: vrátí 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`
  `)`: vrátí kopii matice `matrix`, kde hodnotu každého prvku vynásobíme parametrem `factor`
- `Matrix<Element, Height, Width> operator+(`
  `const Matrix<Element, Height, Width>& matrix_1,`
  `const Matrix<Element, Height, Width>& matrix_2`
  `)`: sečte dvě matice `matrix_1` a `matrix_2` stejných rozměrů
- `Matrix<Element, Height, Width> operator*(`
  `const Matrix<Element, Height, Depth>& matrix_1,`
  `const Matrix<Element, Depth, Width>& matrix_2`
  `)`: vynásobí dvě matice `matrix_1` a `matrix_2` vzájemně kompatibilních rozměrů

Na úplný závěr ještě celou implementaci naší třídy matice upravíme a rozšíříme takovým způsobem, aby podporovala mechanismus copy-on-write. Chceme tedy zajistit, aby více instancí třídy matice mohlo sdílet stejný obsah a ke skutečnému kopírování matic ve smyslu kopírování jejich datového obsahu docházelo až v okamžiku, kdy to bude nevyhnutelné. Z praktického pohledu je vhodné k řešení tohoto požadavku přistoupit až tehdy, kdy všechnu ostatní funkcionalitu matice už budeme mít naprogramovanou a řádně odladěnou. A nebude to vlastně nic komplikovaného.

Základní myšlenka spočívá v tom, že třída matice už nadále nebude přímo obsahovat datovou položku pole `std::array` pro vnitřní úložiště. Namísto toho bude obsahovat jen chytrý ukazatel na něj. Konkrétně pak sdílený chytrý ukazatel, který je přesně pro takovou funkcionalitu sdílení navržen. Vnitřní pole úložiště tedy bude vyčleněno mimo třídu matice a bude vznikat dynamickou alokací.

Budeme-li chtít existující instanci matice nakopírovat, nakopíruje se ve skutečnosti jen její obálka s chytrým ukazatelem, vnitřní úložiště zůstane sdílené. A přesně takové chování nám zajistí překladačem automaticky vygenerovaný kopírovací konstruktor a operátor přiřazení pro matice. Ty totiž jen slepě kopírují jednotlivé položky. V našem případě sdílený ukazatel, který nakopírováním umožní sdílení cíle. Tedy přesně to, co potřebujeme.

Kdykoli nad nějakou maticí zavoláme libovolnou operaci, která má vést k modifikaci jejího obsahu, budeme muset zajistit, že před samotnou realizací takové modifikující akce nejprve oddělíme její datový obsah. Tedy pokud její vnitřní úložiště skutečně je v daný okamžik sdíleno s nějakou další jednou nebo i více maticemi najednou, vytvoříme novou instanci vnitřního úložiště jakožto kopii aktuálního obsahu toho stávajícího. Staneme se jeho výhradním vlastníkem, a můžeme tedy začít se zamýšlenými modifikacemi, protože ostatní takové matice už jimi ovlivněny nebudou.

Kvůli zajištění tohoto mechanismu navrhneme vnitřní pomocnou funkci třídy matice. Jejím úkolem bude provést test nutnosti separace a její případné provedení. Pro test samotný využijeme metodu `use_count()` nad sdíleným ukazatelem, která vrací aktuální počet realizovaných sdílení. Naši separační metodu pak zavoláme v rámci každé naší operace nad maticí, která modifikace provádí. Nezapomeňme ale, že se to týká i těch metod, které sice samy od sebe žádné modifikace neprovádí, vrací ale modifikovatelné reference na prvky matice. Jejich poskytnutím totiž uživatelům dáváme možnost takové modifikace provést, aniž bychom to následně mohli jakkoli zjistit a ošetřit. Na to už by jinými slovy bylo pozdě.

Uvědomme si rovněž, že implementací celého mechanismu líného chování s odloženým kopírováním jsme způsobili, že poskytnuté reference na prvky matice už nemají trvalý charakter. A to bez ohledu na to, zda šlo o modifikovatelné reference nebo ty konstantní. Každá diskutovaná modifikační akce je totiž může zneplatnit. Těchto důsledků si musíme být vědomi aneb vnímat získané reference jen jako reference určené k okamžitému použití. Takové chování by nás ale současně nemělo překvapit, protože u většiny standardních kontejnerů včetně např. `std::vector` rovněž hrozí zneplatnění veškerých referencí, ukazatelů nebo iterátorů po provedených modifikacích.

Oproti běžným zvyklostem je nutné veškerý kód šablonových 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 a použitím šablonových tříd a funkcí, vnitřními třídami, kontejnerem pole `std::array`, přetypováním `const_cast` a také implementací dalších vlastních operátorů, konkrétně aritmetických a operátoru indexace.