NPRG041 – 2024/25 ZS – Cvičení MS – **Malý úkol C11**

Gumové pole II

V rámci tohoto úkolu dokončíme naši rozpracovanou reprezentaci vlastního kontejneru gumového pole. Nejprve převezmeme veškerý stávající kód a rozšíříme jej o implementaci pokročilejších konstruktorů a operátorů přiřazení. Následně pro tento kontejner naprogramujeme i vlastní iterátory, a to na úrovni nejvyšší kategorie iterátorů s náhodným přístupem.

Pokud jde o zmíněné pokročilé konstruktory a operátory přiřazení, implementujeme následující čtveřici standardních funkcí:

- `Array(const Array<Element>& other)`:
  kopírovací konstruktor (copy constructor)
- `Array(Array<Element>&& other) noexcept`:
  přesouvací (vykrádací) konstruktor (move constructor)
- `Array<Element>& operator=(const Array<Element>& other)`:
  kopírovací operátor přiřazení (copy assignment)
- `Array<Element>& operator=(Array<Element>&& other) noexcept`:
  přesouvací (vykrádací) operátor přiřazení (move assignment)

Zdůrazněme, že (nejenom) u kopírovacího operátoru přiřazení opět očekáváme zajištění atomického chování, tedy že v případě selhání zachováme celý původní obsah (myšleno gumového pole, do kterého bylo přiřazováno). V obou operátorech přiřazení dále očekáváme preventivní ochranu řešící situace, kdy bychom se pokoušeli provést přiřazení sami do sebe. V návaznosti na vykrádací konstruktor a operátor přiřazení ještě implementujeme globální funkci `void swap(Array<Element>& array_1, Array<Element>& array_2) noexcept`. Jejím prostřednictvím budeme (nejenom my sami) schopni snadno prohazovat obsah dvou instancí gumových polí.

Vlastní iterátor pro kontejner gumového pole naprogramujeme v podobě veřejné vnitřní šablonované třídy `iterator_base<bool Constant>` v rámci třídy našeho pole. Jediným jejím parametrem `Constant` bude příznak `false` resp. `true`, zda má iterátor pracovat nad modifikovatelným nebo konstantním gumovým polem. Lépe řečeno, zda má vracet reference na modifikovatelné nebo konstantní prvky. Z praktického pohledu totiž potřebujeme nabídnout obě varianty chování, přístup pomocí šablony nám pak pomůže vyvarovat se zbytečné duplikace kódu.

Abychom však uživatele našich iterátorů zbytečně nezatěžovali vnitřními detaily, nabídneme definici dvou veřejných typových aliasů v podobě `iterator` pro `iterator_base<false>` resp. `const_iterator` pro `iterator_base<true>`. Třída gumového pole pak nabídne následující standardní metody pro zpřístupnění iterátorů na první resp. za poslední prvek pole:

- `iterator begin()`, `const_iterator begin() const` a `const_iterator cbegin() const`:
  vrátí instanci iterátoru v modifikovatelné resp. konstantní variantě, který bude ukazovat na první prvek našeho gumového pole (je-li takový, jinak za konec)
- `iterator end()`, `const_iterator end() const` a `const_iterator cend() const`:
  analogicky vrátí instanci iterátoru ukazujícího za aktuální konec našeho pole

Nyní se podívejme na implementaci samotné třídy bázového iterátoru. Abychom obě varianty našich iterátorů dokázali používat i ve spojitosti se standardními algoritmy, je nejprve nutné jejich chování a možnosti popsat pomocí následujících veřejných typových aliasů. Tagy pro jednotlivé kategorie iterátorů najdeme v knihovně `<iterator>`:

- `using iterator_category = std::random_access_iterator_tag`
- `using value_type = Element`
- `using pointer = Element*`
- `using reference = Element&`
- `using difference_type = std::ptrdiff_t`

Položky `value_type`, `pointer` a `reference` však v této podobě budou fungovat jen pro modifikovatelnou variantu. Nahradíme je tedy použitím konstrukce `std::conditional_t<bool B, class T, class F>`, která jednoduše řečeno vybere typ `T` resp. `F` podle skutečné `true` / `false` hodnoty parametru `B`. Konstrukci samotnou najdeme v knihovně `<type_traits>`. Stejný trik pak můžeme použít i pro výběr správného typu pro uchování si reference nebo ukazatele na samotné gumové pole v rámci datových položek, které si budeme muset pamatovat.

Nejprve se zaměříme na implementaci povinných metod na úrovni dopředných iterátorů:

- `bool operator==(const iterator_base& other) const`: otestuje rovnost našeho iterátoru vůči jinému iterátoru nad stejným gumovým polem, tedy že oba ukazují na stejnou logickou pozici
- `bool operator!=(const iterator_base& other) const`: otestuje rozdílnost obou iterátorů, tedy že ukazují na rozdílné pozice
- `iterator_base& operator++()`: preinkrementuje náš iterátor aneb posune aktuální logickou pozici o 1 dál směrem dopředu
- `iterator_base operator++(int)`: analogicky provede postinkrementaci
- `reference operator*() const`: dereferencuje náš iterátor aneb vrátí referenci na prvek, na který iterátor aktuálně ukazuje
- `pointer operator->() const`: vrátí céčkový ukazatel na prvek, na který iterátor aktuálně ukazuje

Nad rámec již uvedených operátorů přidáme následující dva, abychom dosáhli úrovně obousměrných iterátorů:

- `iterator_base& operator--()`: provede predekrementaci iterátoru
- `iterator_base operator--(int)`: analogicky provede postdekrementaci iterátoru

Následně rozšíříme nabízenou funkcionalitu až na úroveň iterátorů s náhodným přístupem:

- `iterator_base operator+(difference_type n) const`: vrátí novou instanci iterátoru, který bude ukazovat na pozici posunutou o příslušný počet prvků, a to dopředu v případě kladného čísla a dozadu v případě záporného
- Analogicky `operator-`
- `difference_type operator-(const iterator_base& other) const`: spočítá rozdíl dvou iterátorů aneb vrátí počet prvků mezi druhým a naším iterátorem
- `iterator_base& operator+=(difference_type n)`: posune náš iterátor o příslušný počet prvků dopředu a vrátí referenci na něj
- Analogicky `operator-=`
- `reference operator[](difference_type n) const`: vrátí referenci na prvek gumového pole, který se vyskytuje o příslušný počet pozic dopředu relativně vůči aktuální pozici, na kterou náš iterátor aktuálně ukazuje
- `bool operator<(const iterator_base& other) const`: porovná náš a druhý předaný iterátor aneb konkrétně detekuje, zda náš iterátor ukazuje na nižší pozici než druhý předaný iterátor
- Analogicky `operator<=`, `operator>` a `operator>=`

Všechny výše uvedené operátory jsme implementovali pomocí členských funkcí naší třídy iterátoru. Podporovat ale musíme i výrazy ve tvaru `n + it`, tedy posun iterátoru `it` o příslušný počet pozic `n` dopředu, ovšem ve variantě s prohozeným pořadím operandů. Toho je možné dosáhnout jen prostřednictvím globální funkce `iterator_base operator+(difference_type n, const iterator_base& it)`.

Z praktických důvodů je ještě vhodné umět provést konverzi iterátoru v modifikovatelné variantě na tu konstantní, tedy umět přetypovat `iterator` na `const_iterator`, a to samozřejmě opravdu jen v tomto směru. Potřebného chování dosáhneme přidáním členské funkce pro konverzní operátor v podobě `operator iterator_base<true>() const`. Jediným úkolem je vrátit nově vytvořenou instanci příslušného iterátoru.

Dodejme, že odpovědnost za korektní používání iterátorů přenášíme v plném rozsahu na uživatele. To znamená, že se za všech okolností musí vyvarovat například následujících situací: používání iterátorů napříč různými instancemi gumových polí, používání zneplatněných iterátorů (vlivem provedených operací modifikujících dané gumové pole), přistupování na neplatné pozice (např. za koncem gumového pole),

dereferencování neplatných pozic apod. V takových a podobných případech bude chování našich iterátorů nedefinované a patřičné situace nijak detekovat ani ošetřovat nebudeme.

Na závěr přemístíme veškerou implementaci našeho gumového pole i iterátorů do vlastního jmenného prostoru s názvem `lib`.

Opět odevzdejte pouze a jenom hlavičkový soubor `Array.h` a dodržte obvyklé požadavky na úkoly kladené. Cílem tohoto úkolu je demonstrovat schopnost návrhu a práce s copy a move konstruktory a operátory přiřazení, návrh vlastních iterátorů, práci s vnořenými šablonami, konverzními operátory a také vlastním jmenným prostorem.