NPRG041 – 2025/26 ZS – Cvičení MS – **Malý úkol C11**

Gumové pole II

V rámci tohoto úkolu dokončíme naši rozpracovanou reprezentaci vlastního kontejneru gumového pole. Veškerý stávající kód převezmeme a postupně jej rozšíříme o implementaci pokročilejších kontejnerových konstruktorů, pokročilejších metod na vkládání nových prvků a rovněž i dalších pomocných metod pro manipulaci s prvky. V neposlední řadě pro toto pole naprogramujeme i vlastní iterátory, a to na úrovni kategorie iterátorů s náhodným přístupem.

Implementaci našeho pole však nejprve upravíme takovým způsobem, aby jeho použití bylo uživatelsky přívětivé a současně bylo kompatibilní se standardními knihovnami. To zásadním způsobem rozšíří potenciál jeho využití. Aneb už nepůjde jen o nějaké naše proprietární řešení, ale o plnohodnotný kontejner, se kterým zvládnou pracovat i standardní algoritmy nebo rozsahy. Těm se ostatně budeme věnovat hned v dalším, tedy posledním malém úkolu.

Začněme s umístěním celé implementace gumového pole do vlastního jmenného prostoru s názvem `lib`. Dále do třídy pole jako takové přidáme řadu veřejných typových aliasů, prostřednictvím kterých budou ostatní schopni zjistit nezbytné informace o typových rysech námi spravovaných elementů:

- `using value_type = T;`
- `using size_type = std::size_t;`
- `using difference_type = std::ptrdiff_t;`
- `using reference = T&;`
- `using const_reference = const T&;`
- `using pointer = T*;`
- `using const_pointer = const T*;`

V souladu s předchozím úkolem šablonový parametr `T` označuje typ našich prvků. Později navíc přidáme ještě další dva takové aliasy, ty se budou týkat našich iterátorů.

Další novinkou bude systematické použití konceptů a navazujících mechanismů za účelem větší kontroly nad konkrétními typy `T`, které jsou pro naše prvky smysluplně použitelné, ať už na úrovni pole jako celku nebo jeho konkrétních vybraných metod. Za tímto účelem navrhneme vlastní koncept `ArrayElement`, který zajistí alespoň jeden způsob konstruovatelnosti `std::copy_constructible`, `std::move_constructible` nebo `std::default_initializable` a současně destruovatelnost `std::destructible`. Všechny zmíněné standardní koncepty najdeme v knihovně `<concepts>`. Nově tak díky tomu pro naše gumové pole budeme moci uvažovat deklaraci `template<ArrayElement T> class Array`.

Nyní se zaměříme na tři nové konstruktory, které umožní pokročilé vytváření nových instancí gumových polí s neprázdným obsahem již od okamžiku jejich vzniku. Ve všech případech budeme uvažovat výchozí velikost bloku rovnou 10, symbolem [♦] pak připomínáme nutnost zajištění jejich atomicity.

- `Array(size_t count, const T& item)` [♦]: vytvoří novou instanci pole, která bude obsahovat `count` kopií předloženého vzorového prvku `item`
- `Array(std::initializer_list<T> items)` [♦]: vytvoří novou instanci pole, do které vložíme kopie prvků, které jsou připraveny v předloženém inicializačním seznamu (knihovna `<initializer_list>`)
- `template<std::input_iterator InputIterator>`
  `Array(InputIterator begin, InputIterator end)` [♦]: vytvoří nové pole, do kterého vložíme kopie prvků, které patří do rozsahu [`begin`, `end`) nad nějakou kolekcí (jiným gumovým polem, jiným kontejnerem, ale třeba i céčkovým polem), který je určený uvedenou dvojicí iterátorů nebo i céčkových ukazatelů (knihovna `<iterator>`)

Dalším krokem je rozšíření stávajících možností na vkládání jednotlivých prvků do našeho pole. Aktuálně v tomto smyslu máme k dispozici dvojici funkcí `push_back`. Jelikož je jejich implementace prakticky identická, využitím relativně přímočarého triku pomocí univerzálních referencí a perfektního předávání parametrů je nejprve sloučíme dohromady, resp. nahradíme jednou jedinou šablonovou funkcí. Pomocí variadické šablony následně ještě naprogramujeme neméně užitečnou funkci `emplace_back`.

- `template<typename Source>`
  `requires std::constructible_from<T, Source&&>`
  `void push_back(Source&& item)` [♦]: pomocí kopírování nebo přesouvání (podle původní kategorie předané hodnoty ve smyslu lvalue resp. rvalue) vložíme na konec pole předložený nový prvek; pro zajištění obnovy jeho původní kategorie při jeho dalším předání použijeme funkci `std::forward` (knihovna `<utility>`)
- `template<typename... Args>`
  `requires std::constructible_from<T, Args&&...>`
  `T& emplace_back(Args&&... args)` [♦]: analogicky na konec pole vložíme nový prvek, v tomto případě vytvořený konstruktorem prvku, který očekává stejné parametry jako ty do této funkce předané; pro jejich předání opět použijeme stejný mechanismus perfektního předávání, jen je potřeba jej doplnit o expanzi balíčku parametrů, tedy použít konstrukci `std::forward<Args>(args)...`; nakonec ještě vrátíme modifikovatelnou referenci na takto vložený prvek

Vlastní iterátor pro kontejner gumového pole naprogramujeme v podobě veřejné vnitřní šablonové 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>`. Ne náhodou jsme zvolili právě tato jména, odpovídají totiž zbývajícím dvěma aliasům, jejichž definici jsme slíbili už v úvodu. Třída gumového pole pak nabídne následující standardní metody pro zpřístupnění iterátorů na první resp. za poslední prvek pole. Všechny implementujeme jako inline metody a pro větší přehlednost navíc spojíme jejich deklarace rovnou s definicemi:

- `inline iterator begin()` / `const_iterator begin() const` / `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)
- `inline iterator end()` / `const_iterator end() const` / `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 s již zmíněnými standardními algoritmy, je opět nepřekvapivě nutné jejich chování a možnosti popsat pomocí řady veřejných typových aliasů. Tagy pro jednotlivé kategorie iterátorů najdeme v knihovně `<iterator>`.

- `using iterator_category = std::random_access_iterator_tag;`
- `using value_type = T;`
- `using pointer = T*;`
- `using reference = T&;`
- `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 reference nebo ukazatele na samotné gumové pole v rámci datových položek, které si budeme muset pamatovat.

Parametrický konstruktor iterátoru navrhneme dle potřeby, jeho přesná podoba jinými slovy předepsaná není. S ohledem na již zmíněnou kompatibilitu je ale potřeba implementovat i veřejný defaultní konstruktor. Z povahy věci je zřejmé, že jím vytvořené instance nemohou obsahovat jakkoli smysluplné informace, to ale není podstatné. Jde jen o to, aby takový bezparametrický konstruktor k dispozici byl.

Pokud jde o očekávanou funkcionalitu, ve všech případech půjde o nejrůznější operátory, navíc značně triviální z hlediska jejich obsahu. Z důvodu zvýšení celkové přehlednosti a také usnadnění naší práce tedy

opět jejich deklarace rovnou spojíme s definicemi. Nejprve se zaměříme na implementaci povinných metod na úrovni dopředných iterátorů (tag `std::forward_iterator_tag`):

- `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 (tag `std::bidirectional_iterator_tag`):

- `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 (od začátku předpokládaný výsledný tag `std::random_access_iterator_tag`):

- `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 čísla záporného
- Analogicky `operator-`
- `difference_type operator-(const iterator_base& other) const`: spočítá rozdíl dvou iterátorů aneb vrátí počet prvků mezi naším a druhým 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 iterátor ve smyslu porovnávání `<` aneb 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 obou operandů. Toho je možné dosáhnout jen prostřednictvím globální funkce, tu je navíc možné definovat jen v rámci friend deklarace:

- `friend iterator_base operator+(difference_type n, const iterator_base& it)`

Z praktických důvodů je ještě vhodné umět provést přetypování iterátoru v modifikovatelné variantě na tu konstantní, tedy umět přetypovat `iterator` na `const_iterator` (samozřejmě jen v tomto směru). Potřebného chování dosáhneme přidáním členské funkce pro konverzní operátor v následující podobě:

- `operator iterator_base<true>() const`: vrátí nově vytvořenou instanci konstantního iterátoru, který bude ukazovat na stejnou pozici jako zdrojový iterátor

Dodejme, že odpovědnost za korektní používání iterátorů opět v plném rozsahu přenášíme na uživatele. To znamená, že se 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.

Nyní se opět vrátíme ke gumovému poli jako takovému a přidáme ještě další dvě funkce pro manipulaci s jeho prvky. Hodit se nám budou v posledním malém úkolu.

- `void resize(size_t size)`: změníme velikost gumového pole na novou velikost `size`; při zmenšení velikosti jednoduše odebereme odpovídající počet existujících prvků z konce pole; naopak v případě zvětšení velikosti nové prvky na konec přidáme; tyto nové prvky pak konkrétně vytvoříme pomocí jejich defaultního konstruktoru
- `iterator erase(const_iterator first, const_iterator last)`: z dané instance pole odebereme všechny prvky patřící do intervalu [`first`, `last`) určeného našimi iterátory; vrátíme modifikovatelný iterátor ukazující na první prvek za posledním smazaným prvkem (pokud vůbec nějaký odebrán byl); z důvodu snížení složitosti této operace předpokládáme, že naše prvky jsou přesouvatelné, navíc bez výjimek; díky tomu nemusíme (nebudeme) řešit případné problémy s dosažením atomicity, protože za takových okolností ani vzniknout nemohou

Na úplném konci úkolu ještě přidáme nejrůznější statické asserty v podobě `static_assert(condition, message)`, tedy kontroly prováděné v čase kompilace, pomocí kterých jsme schopni ověřit splnění specificky vyžadovaných schopností elementů na úrovni jednotlivých funkcí. Cílem těchto kontrol není náš kontejner jako celek dále omezovat, jen chceme identifikovat ty předpoklady, které už jsme stejně implicitně měli, a v případě jejich nenaplnění umožnit kompilátoru vypsání smysluplných chybových hlášek.

Konkrétní očekávané kontroly, tedy dotčené funkce kontejneru, kontrolované schopnosti elementů, ani chybové zprávy nijak předepsány nejsou a nebudou ani kontrolovány (protože to automatizovaně ani nejde). Zkuste tedy váš kód v tomto smyslu obohatit co nejlépe. Použít k tomu můžete zejména následující standardní koncepty nebo `requires` podmínky:

- `std::copy_constructible`
- `std::constructible_from`
- `std::assignable_from`
- `std::default_initializable`
- `requires(T& t, T&& s) { { t = std::move(s) } noexcept; }`

Opět odevzdejte pouze a jenom hlavičkový soubor `Array.h` a dodržte obvyklé požadavky na úkoly kladené. Cílem tohoto úkolu je demonstrovat schopnost návrhu pokročilých kontejnerových konstruktorů a vlastních iterátorů, práce s vnořenými šablonami, variadickými šablonami a perfektním předáváním, konverzními operátory, vlastním jmenným prostorem a také koncepty a statickými assert kontrolami.