NPRG041 – 2025/26 ZS – Cvičení MS – **Malý úkol C10**

Gumové pole I

V rámci tohoto a následujícího úkolu navrhneme implementaci vlastního generického kontejneru, a to konkrétně kontejneru gumového pole. Jeho implementace bude do značné míry vycházet z principů a chování standardního kontejneru vektoru, některé jeho aspekty však záměrně navrhneme výrazně odlišným způsobem. Inspirujeme se i veřejným rozhraním vektoru, implementujeme však jen určitý výběr jeho základních nebo zajímavých metod, protože implementace všech by pro nás nebyla dostatečně přínosná. Konkrétně v tomto úkolu se nejprve zaměříme na vnitřní úložiště gumového pole, jeho základní funkcionalitu a kopírovací a přesouvací konstruktory a operátory přiřazení. Příště se podíváme na konsolidaci kontejneru, pokročilé kontejnerové konstruktory, pokročilé způsoby vkládání nových prvků a hlavně naprogramujeme vlastní iterátory.

Cílem u našeho gumového pole je, abychom zachovali možnost průběžně vkládat nové prvky nebo naopak ty existující odebírat, vždy však jen na jeho konci. To jinými slovy znamená, že celé gumové pole bude vždy logicky souvislé bez jakýchkoli děr. Oproti standardnímu vektoru však vnitřní úložiště naprogramujeme tak, abychom se obešli bez nutnosti mít v paměti alokovaný prostor pro vlastní prvky, který by musel být také fyzicky souvislý. Díky tomu nebudeme potřebovat časově náročné realokace a navíc budeme schopni garantovat neměnnost umístění vložených prvků po celou dobu jejich existence v kontejneru. Zcela tedy eliminujeme riziko zneplatnění ukazatelů, referencí nebo iterátorů na jednotlivé prvky pole po celou dobu jejich životního cyklu, což standardní kontejner vektoru není schopen zaručit.

Třída gumového pole, nazvěme ji jednoduše `Array`, bude mít jediný šablonový parametr `T`, a to datový typ prvku. Může jít o číselné typy, stejně jako jakékoli uživatelsky definované třídy (o kterých toho předem nemůžeme předpokládat mnoho), nebo i ukazatele na ně (ať už céčkové nebo chytré).

Vnitřní úložiště realizujeme dvouúrovňově, kdy na první úrovni nejprve použijeme standardní vektor `std::vector` obsahující céčkové ukazatele na bloky druhé úrovně. Blokem druhé úrovně pak myslíme obyčejné dynamicky alokované céčkové pole jednotlivých prvků. Všechny tyto bloky budou mít stejnou, předem zvolenou fixní velikost, která zůstane po celou dobu existence daného gumového pole neměnná. To nám pro přístup ke konkrétním prvkům umožní používat přímočarou indexovou aritmetiku, kdy prvek na logické pozici i umístíme do bloku (i/s) na jeho fyzickou pozici $(i \bmod s)$, kde s je zvolená velikost bloku (měřená počtem slotů pro prvky).

Nově vytvořený kontejner gumového pole bude prázdný, nebude tedy obsahovat žádný prvek a hlavně ani žádný vnitřní blok. Ty pak budou dynamicky přidávány nebo odebírány dle potřeby, a to v návaznosti na požadavky vkládání nebo naopak odebírání jednotlivých prvků jako takových. Potřebné vnitřní mechanismy nastavíme tak, abychom vždy měli alokovaný přesně takový počet bloků, který je pro uložení aktuálního počtu prvků nezbytně nutný. Šlo by samozřejmě použít i jiné nebo chytřejší strategie, my si však nechceme naši situaci zbytečně komplikovat.

Nyní se zaměříme na popis detailů očekávané funkcionality a rozhraní konkrétních členských nebo i globálních funkcí. Začneme základním konstruktorem (později přidáme další) a destruktorem:

- `Array(size_t block_size)`: vytvoří prázdnou instanci gumového pole; jediný předaný parametr udává požadovanou velikost bloku; není-li uvedena, budeme předpokládat výchozí velikost 10 prvků
- `~Array() noexcept`: provede destrukci gumového pole

Dále nabídneme následující tři členské funkce, pomocí kterých získáme základní informace o aktuálním obsahu a kapacitě kontejneru:

- `inline size_t size() const`: vrátí velikost pole, tedy počet prvků do něj aktuálně vložených
- `inline bool empty() const`: vrátí `true` právě tehdy, pokud pole aktuálně neobsahuje žádný prvek
- `inline size_t capacity() const`: vrátí velikost alokovaného prostoru vnitřního úložiště, tedy počet prvků, které se vejdou do všech aktuálně alokovaných vnitřních bloků

Za účelem přidání nového prvku na konec gumového pole resp. odebrání posledního prvku z konce gumového pole nabídneme následující metody. V případě přidávání pak konkrétně nabídneme kopírovací i přesouvací (vykrádací) variantu. Rovněž nabídneme i metodu na smazání kompletního obsahu pole.

- `void push_back(const T& item)` [♦]: přidáme nový prvek na aktuální konec gumového pole; prvek jako takový fyzicky umístíme na první dosud nevyužitou pozici v aktuálním (tedy posledním) vnitřním bloku; přesněji řečeno na tuto pozici umístíme kopii předaného prvku a staneme se jejím vlastníkem; pokud už v aktuálním bloku žádné další volné místo není, přidáme nejdříve blok nový
- `void push_back(T&& item)` [♦]: stejné chování jako u předchozí varianty, jen předpokládáme prvek předaný rvalue referencí, a tedy jej na příslušnou pozici jen přesuneme pomocí `std::move`, tedy bez jeho kopírování; opět se staneme jeho vlastníkem a volající už původní instanci nesmí dále používat
- `void pop_back()`: odebereme poslední prvek z aktuálního konce gumového pole; jelikož jsme v každém případě byli jeho vlastníkem, musíme se postarat o jeho korektní ručně vyvolanou destrukci; pokud se uvolněním prvku stal poslední vnitřní blok zcela nevyužívaný, odebereme jej také
- `void clear()`: kontejner vyprázdníme aneb odebereme všechny jeho prvky

Už v tuto chvíli by mělo být intuitivně zřejmé, že přidávání nových prvků se kvůli možnosti selhání dynamické alokace v našem vnitřním úložišti nemusí vždy podařit. Obecně je čistě na nás, jakým způsobem se takové situace rozhodneme řešit, pokud vůbec. Nadále ale chceme usilovat o respektování principů používaných ve standardních kontejnerech. To konkrétně znamená, že zajistíme takzvanou strong exception guarantee. Jinými slovy zaručíme atomicitu potenciálně rizikových operací aneb v případě jejich selhání navrátíme vnitřní obsah celého gumového pole do původního stavu platného před voláním příslušné funkce. Pokud jde o použitelnost kontejneru, jiné chování si ostatně ani rozumně představit nejde. Tento princip tedy dodržíme u všech našich metod, u kterých riziko selhání hrozí. A jelikož jich bude více, pro větší přehlednost je vždy označíme již uvedeným symbolem [♦].

Z praktického pohledu na věc bude nejprve vhodné samostatně implementovat dvojici pomocných interních metod, pomocí kterých budeme schopni v případě potřeby přidat nový vnitřní blok (dynamicky jej alokovat a zapamatovat si jej) resp. existující (poslední a již nevyužívaný) blok odebrat (dealokovat jej a zapomenout jej).

Pokud jde o dynamickou alokaci jako takovou, nebudeme používat obvyklé operátory `new` a `delete` (resp. jejich varianty pro pole), protože jejich využití automaticky a nevyhnutelně znamená provedení alokace a volání konstruktoru resp. volání destruktoru a provedení dealokace. Místo nich budeme pracovat s nízkoúrovňovou dynamickou alokací, v rámci které je možné oba aspekty od sebe navzájem oddělit.

Nový vnitřní blok konkrétně alokujeme pomocí funkce `void* ::operator new(size_t size)`. Jejím jediným parametrem je velikost požadované paměti v počtu bytů. Pokud se alokace podaří, dostaneme validní céčkový ukazatel na začátek alokovaného místa, jen si jej přetypujeme z `void*` na `T*`. Pokud se alokace nepodaří, je vyvolána obvyklá výjimka `std::bad_alloc`. Pokud naopak existující vnitřní blok chceme dealokovat, použijeme funkci `void ::operator delete(void* ptr)`, přičemž jen předáme ukazatel na začátek takového bloku. Obě uvedené funkce najdeme v knihovně `<new>`.

V rámci vkládání nového prvku do příslušného volného slotu v aktuálně posledním bloku potřebujeme zavolat odpovídající konstruktor prvku. K tomu nám poslouží placement new operátor. Konkrétně jen využijeme konstrukce `new (target) T(...)`, kde `target` je ukazatel na místo, kde taková instance prvku má vzniknout (místo samotné už alokováno bylo), a `...` nahradíme konkrétními hodnotami parametrů zamýšleného konstruktoru prvku, což v našem případě bude buď jeho kopírovací, nebo přesouvací (vykrádací) konstruktor. Při odebírání prvku naopak musíme explicitně zavolat jeho destruktor, což snadno uděláme pomocí konstrukce `target->~T()`.

Jak už jsme popsali, funkce na přidávání prvků, stejně jako na přidávání bloků, musí z pohledu očekávané konzistence zaručit atomické chování. Tedy že se buď zamýšlený záměr podaří realizovat zcela úspěšně, nebo i při jen dílčím neúspěchu efekt již provedených akcí kompenzujeme a o celkovém neúspěchu informujeme vhodnou výjimkou.

Možnost selhání alokace nového bloku pomocí funkce `::operator new` už jsme diskutovali. Uvědomme si ale, že selhat může i operace `push_back`, pomocí které ukazatel na takový nově alokovaný blok vkládáme do vektoru na první úrovni vnitřního úložiště. Při vkládání nového prvku pak může selhat jeho konstruktor, což se pozná tím, že vyhodí nějakou výjimku. A ta může být skutečně naprosto libovolná. Poznamenejme rovněž, že konstrukce nového objektu (ať už jakéhokoli) se považuje za úspěšnou, jedině pokud proběhla úspěšně celá až do úplného konce. Pokud ne, kompilátor se sám postará o destrukci jeho případných datových položek nebo předků (pokud již byly úspěšně inicializovány), a destrukci objektu jako celku tedy sami nevyvoláváme a ani tak učinit nesmíme (protože jednoduše ani nevznikl).

Pro přístup k jednotlivým prvkům gumového pole dále naprogramujeme následující dvě dvojice metod se zjevným významem:

- `T& at(size_t index)`
- `const T& at(size_t index) const`
- `T& operator[](size_t index)`
- `const T& operator[](size_t index) const`

Další částí tohoto úkolu je ošetření krajních situací, které by při správném využívání našeho kontejneru ze strany uživatele vůbec nastat neměly. Přesněji řečeno nabídneme možnost aktivovat jakýsi ladicí režim, v rámci kterého takové situace detekovat a příslušné výjimky vyvolávat budeme. Pokud ale tento režim aktivní nebude, což chápeme jako výchozí variantu pro produkční prostředí, takové kontroly vůbec provádět nebudeme (a příslušné fragmenty kódu se ani součástí zkompilované aplikace nestanou).

Technické řešení je snadné. Dotčený kód v rámci implementace našeho gumového pole umístíme do podmíněně kompilovaných sekcí pomocí direktiv `#ifdef` a `#endif` (eventuálně dalších podobných). Uživatel pak v případě zájmu ladicí režim aktivuje direktivou `#define`. Konkrétně budeme pro tyto účely používat makro s názvem `ARRAY_DEBUG_MODE` a ošetřovat budeme následující situace:

- `std::out_of_range("Invalid index")`: pokud v libovolné přístupové metodě `at` nebo `[]` volající požaduje neplatný index prvku; v případě funkcí `at` tuto výjimku vyhodíme vždy (bez ohledu na ladicí režim), u operátorů `[]` naopak jen v ladicím režimu (cílem tohoto nesymetrického přístupu je poskytnout stejné chování, jako nabízí standardní kontejnery)
- `std::invalid_argument("Empty array")`: pokud se volající snaží provést metodu `pop_back` nad prázdným gumovým polem (jen v ladicím režimu)

Obě používané standardní výjimky najdeme v knihovně `<stdexcept>`. Zdůrazněme, že smyslem našich podmíněných kontrol není řešit korektnost naší implementace gumového pole, ale nabídnout uživatelům tohoto pole lepší možnosti pro ladění jejich programů.

Nakonec ještě implementujeme čtveřici speciálních členských funkcí, konkrétně kopírovací a přesouvací konstruktory a operátory přiřazení. Jejich výchozí implementace generovaná kompilátorem, podobně jako u destruktoru, by totiž nevyhovovala našim potřebám. Nejenom v rámci implementace některých z nich potom bude užitečná i globální funkce `void swap(Array<T>& array_1, Array<T>& array_2) noexcept`, jejímž úkolem bude vzájemně prohodit dvě instance gumových polí.

- `Array(const Array& other)` [♦]: kopírovací konstruktor
- `Array(Array&& other) noexcept`: přesouvací (vykrádací) konstruktor
- `Array& operator=(const Array& other)` [♦]: kopírovací operátor přiřazení
- `Array& operator=(Array&& other) noexcept`: přesouvací (vykrádací) operátor přiřazení

Pokud jde specificky o přesouvací konstruktor a operátor přiřazení, vykradení zdrojového gumového pole provedeme tak, že jeho výsledný stav musí odpovídat validnímu zcela prázdnému poli. Současně ještě do obou těchto funkcí přidáme preventivní ochranu kvůli situacím, kdy bychom se pokoušeli provést přiřazení sami do sebe.

Veškerý kód umístěte do jediného hlavičkového souboru s názvem `Array.h`, ten také jako jediný očekávaný soubor odevzdejte. O řízení průběhu testu se opět postará předpřipravená funkce `main`.

Úkol se zaměřuje na schopnost návrhu vlastního jednoduchého kontejneru s netriviálním uspořádáním vnitřního úložiště pro jeho jednotlivé prvky. To v praktické rovině znamená práci s nízkoúrovňovou alokací v podobě funkcí `::operator new` a `::operator delete`, přímou konstrukcí objektů (pomocí placement new operátoru) a jejich destrukcí (manuálním voláním destruktorů), dále návrh vlastních kopírovacích a přesouvacích konstruktorů a operátorů přiřazení, vyhazování standardních výjimek a využívání podmíněně kompilovaných sekcí kódu. Cílem úkolu je rovněž pokrytí obecných principů týkajících se úrovní výjimkové bezpečnosti, pravidel vykrádání zdrojů, dopadů Rule of Five a implicitně generovaných funkcí.