NPRG041 – 2023/24 ZS – Cvičení MS – **Malý úkol C11**

Gumové pole I

V rámci poslední dvojice úkolů navrhneme implementaci vlastního generického kontejneru, a to konkrétně gumového pole. Jeho chování bude do značné míry vycházet ze standardního kontejneru vektoru, některé jeho aspekty však záměrně navrhneme a naprogramujeme jinak. Konkrétně v tomto úkolu se nejprve zaměříme na vnitřní úložiště takového gumového pole a jeho základní funkcionalitu, příště se podíváme na pokročilé konstruktory a operátory přiřazení, stejně jako naprogramujeme vlastní iterátor.

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.

Třída gumového pole, nazvěme ji jednoduše `Array`, bude mít jediný šablonový parametr `element`, 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.

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 umožní pro přístup ke konkrétním prvkům používat přímočarou indexovou aritmetiku.

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 pak konkrétně nastavíme tak, abychom vždy měli alokovaný právě takový počet bloků, který je pro uložení aktuálního počtu prvků nezbytně nutný. Šlo by samozřejmě použít i jiné chytřejší strategie, my si však nechceme naši situaci zbytečně komplikovat.

V první řadě se pojďme podívat na dva konstruktory, které ve třídě gumového pole nabídneme:

- `Array(size_t block_size)`: vytvoří prázdnou instanci gumového pole; předaný parametr udává požadovanou velikost bloku; není-li uvedena explicitně, budeme předpokládat výchozí velikost 10
- `Array(std::initializer_list<element> items)`: vytvoří novou instanci gumového pole, do které rovnou vložíme kopie prvků podle předloženého inicializačního seznamu; pokud jde o velikost bloku, v tomto případě automaticky předpokládáme výše uvedenou výchozí hodnotu

Nesmíme zapomenout ani na implementaci destruktoru `~Array() noexcept`, jehož úkolem bude zařídit korektní destrukci celého gumového pole. Dále nabídneme následující tři metody, které budou užitečné zejména pro účely případného ladění:

- `inline size_t size() const`: vrátí aktuální velikost gumového pole, tedy počet prvků do něj aktuálně vložených
- `inline size_t capacity() const`: vrátí aktuální velikost alokovaného prostoru vnitřního úložiště, tedy počet prvků, které se vejdou do všech aktuálně alokovaných vnitřních bloků
- `void print(std::ostream& stream = std::cout) const`: vypíše obsah celého gumového pole, a to ve formátu `[1, 2, 3, 4, 5]`; jinými slovy nejdříve vypíšeme pár hranatých závorek a do něj umístíme čárkami a mezerami oddělený výčet jednotlivých prvků; každý prvek sám o sobě vypíšeme pomocí jeho operátoru `<<`

Pro zvýšení uživatelského komfortu přidáme i příslušnou globální funkci operátoru `<<` pro zápis celého gumového pole do streamu.

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

- `void push_back(const element& 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(element&& item)`: stejné chování jako u předchozí varianty, jen předpokládáme prvek předaný rvalue referencí, a tedy jej na příslušnou pozici jen přesuneme pomocí `std::move` bez kopírování; opět se staneme jeho vlastníkem a volající už původní prvek 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

Z praktického pohledu na věc bude nejprve vhodné samostatně implementovat dvojici pomocných interních metod, pomocí kterých budeme v první řadě schopni 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 neoddělitelně 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 blok tedy alokujeme pomocí funkce `void* malloc(size_t size)`, jejímž jediným parametrem je velikost požadované paměti v počtu bytů. Pokud se alokace podaří, dostaneme céčkový ukazatel na začátek alokovaného místa, jen si jej přetypujeme z `void*` na `element*`. Pokud se alokace nepodaří, dostaneme `nullptr`. Pokud naopak existující vnitřní blok chceme dealokovat, použijeme funkci `void free(void* ptr)`, přičemž jen předáme ukazatel na začátek takového bloku.

V rámci vkládání nového prvku do příslušného volného slotu v aktuálně posledním bloku potřebujeme zavolat příslušný konstruktor daného prvku. K tomu nám poslouží placement new operátor. Konkrétně jen využijeme konstrukce `new (ptr) element(...)`, kde `ptr` 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 `ptr->~element()`.

Velmi obezřetní musíme být zejména při návrhu následujících funkcí: inicializační konstruktor, funkce na přidání nového vnitřního bloku a (obě) funkce na přidání nového prvku. Za všech okolností totiž potřebujeme z pohledu očekávané konzistence zařídit, že jejich chování bude atomické. Tedy že se buď zamýšlený záměr podaří realizovat zcela úspěšně, nebo při dílčím neúspěchu efekt již provedených akcí kompenzujeme a o celkovém neúspěchu informujeme vhodnou výjimkou.

Konkrétně při selhání alokace vnitřního bloku pomocí funkce `malloc` vyvoláme standardní výjimku `std::bad_alloc`. Selhat ale 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 vyhodil nějakou výjimku. A ta může být skutečně naprosto libovolná. Poznamenejme rovněž, že konstrukce nového objektu (ať už jakéhokoli) se považuje za úspěšnou, jedině pokud proběhne úspěšně celá až do úplného konce. Pokud ne, kompilátor se sám postará o destrukci jeho případných netriviálních datových položek a stejně tak i případných předků v hierarchii. Toho si potřebujeme být vědomi i v našem inicializačním konstruktoru.

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

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

Poslední 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 zkompilovaného kódu nestanou).

Technické řešení je pak snadné: ladicí režim nejprve v případě zájmu aktivujeme z céčkového souboru direktivou `#define __DEBUG__` a v hlavičkovém souboru s naší implementací gumového pole pak už jen používáme podmíněné sekce v podobě 1) `#ifdef __DEBUG__` pro otevření podmíněné sekce, 2) vlastní fragment našeho kódu a 3) `#endif` pro její ukončení. Konkrétně pak budeme ošetřovat následující situace:

- `std::out_of_range("Invalid index")`: pokud v jakékoli přístupové metodě `at` nebo `[]` volající požaduje neplatný index prvku
- `std::invalid_argument("Empty array")`: pokud se volající snaží provést metodu `pop_back` nad prázdným gumovým polem

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

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, práci s inicializačními seznamy, nízkoúrovňovou alokací a přímou konstrukci a destrukci objektů.