NPRG041 – 2022/23 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 vektoru, tedy chceme mít schopnost do pole dynamicky vkládat nové prvky nebo z něho naopak odebírat ty existující. 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 souvislý. Díky tomu budeme schopni garantovat neměnnost umístění těchto prvků po celou dobu jejich existence v kontejneru.

Konkrétně v tomto úkolu navrhneme kromě již nastíněného vnitřního úložiště také základní funkcionalitu, příště navrhneme pokročilé konstruktory a operátory přiřazení, stejně jako vlastní iterátor.

Třída gumového pole, nazvěme ji jednoduše `Array`, bude mít jediný šablonový parametr `element`, a to datový typ prvků. Může jít o číselné typy, stejně jako jakékoli uživatelsky definované třídy, o kterých toho předem nemůžeme předpokládat mnoho.

Vnitřní úložiště realizujeme dvouúrovňově, kdy na první úrovni nejprve použijeme standardní vektor `std::vector` obsahující chytré ukazatele na bloky druhé úrovně. Z povahy věci konkrétně použijeme unikátní chytré ukazatele `std::unique_ptr`, opět z knihovny `<memory>`. Blokem druhé úrovně pak myslíme obyčejné dynamicky alokované céčkové pole jednotlivých prvků.

Nově vytvořený kontejner gumového pole bude podle očekávání prázdný, tedy nebude obsahovat ani žádný vnitřní blok. Ty pak budou dynamicky přidávány nebo odebírány dle potřeby, a to v návaznosti na požadavky vkládání nebo naopak odebírání prvků jako takových. Konstruktor nabídneme jediný, a to v podobě `Array(size_t block_size)`, kde předáme konkrétní požadovanou velikost vnitřních bloků. Není-li uvedena explicitně, budeme předpokládat výchozí velikost 10 prvků. V každém případě platí, že tato velikost je pro dané gumové pole neměnná. Tedy stejná po celou dobu jeho existence.

Pro přidávání a odebírání prvků nabídneme na logické úrovni dvě metody. Konkrétně přidání nového prvku na konec gumového pole a naopak odebrání posledního prvku z konce gumového pole. To jinými slovy znamená, že celé gumové pole bude vždy logicky souvislé bez jakýchkoli děr. Vše technicky realizujeme následujícími metodami:

- `void push_back(const element& item)`: přidáme nový prvek na logický konec gumového pole; prvek jako takový fyzicky umístíme na první dosud nevyužitou pozici v aktuálním (tedy posledním) interním bloku; přesněji řečeno na tuto pozici samozřejmě umístíme kopii předaného prvku a staneme se vlastníkem této kopie; 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; to znamená, že na příslušnou pozici předaný prvek přesuneme, staneme se jeho vlastníkem a volající už původní prvek nesmí dále používat
- `void pop_back()`: odebereme poslední prvek z logického 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 interní blok zcela nevyužívaný, odebereme jej také

Dále nabídneme následující dvě praktické metody, užitečné zejména pro účely případného ladění:

- `size_t size() const`: vrátíme aktuální velikost gumového pole, tedy počet prvků do něj vložených; metodu implementujeme jako `inline` metodu
- `void print(std::ostream& os = std::cout) const`: vypíšeme obsah celého gumového pole, a to ve formátu `[1, 2, 3, 4, 5]`; jinými slovy vypíšeme pár hranatých závorek a do něj umístíme čárkami a mezerami oddělený výčet jednotlivých prvků; každý prvek sám o sobě vypíšeme pomocí operátoru `<<`; z toho plyne požadavek, že můžeme vytvářet gumová pole jen takových prvků, které takový operátor implementují

Pro zvýšení uživatelského komfortu přidáme i příslušnou globální funkci operátoru `<<` pro zápis celého gumového pole do streamu. Půjde samozřejmě jen o přesměrování volání na právě vytvořený `print`. Pro přístup k jednotlivým prvkům 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 kontejneru nebo minimálně vhodných systémových podmínkách vůbec nastat neměly. Za tímto účelem navrhneme hierarchii tříd výjimek. Konkrétně předpokládáme třídu `Exception` a od ní odvozené varianty `OutOfRangeException`, `UnavailableMemoryException` a `EmptyArrayException`. Konstruktory nabídneme dva: `Exception(const std::string& text)` a `Exception(std::string&& text)`, kde jen v obou případech očekáváme textovou hlášku blíže popisující nastalou chybovou situaci. Z hlediska dalšího rozhraní už nabídneme jen metodu na získání této hlášky pomocí `const std::string& what() const`.

Očekává se návrh rozumného obsahu těchto hlášek, jejich konkrétní podoba však předepsána není. Potřebujeme jen zajistit následující chování:

- `OutOfRangeException`: vyhodíme tehdy, když v jakékoli přístupové metodě `at` nebo `[]` požadujeme nevalidní logickou pozici
- `EmptyArrayException`: vyhodíme, pokud se snažíme volat metodu `pop_back` nad prázdným polem
- `UnavailableMemoryException`: vyhodíme v situaci, kdy se v návaznosti na požadavek přidávání nového prvku pomocí metod `push_back` nepodaří dynamicky alokovat nový vnitřní blok, tedy v reakci na výjimku `std::bad_alloc`

Zdůrazněme, že zejména v posledních dvou uvedených případech je ještě potřeba zajistit, že celý kontejner gumového pole zůstane vždy vnitřně korektní aneb že se naše metody z pohledu konzistence kontejneru budou chovat atomicky.

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. Současně je cílem vyzkoušet si práci s unikátními chytrými ukazateli a vlastními výjimkami.