ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Rozšíření kontejnerů standardní knihovny o notifikace
Student:	Michal Drabina
Vedoucí:	Ing. Jan Trávníček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Nastudujte standardní knihovnu jazyka C++ se zaměřením na kontejnery.
Navrhněte množinu notifikací reflektujících změny obsahu v instancích standardních C++ kontejnerů.
Navrhněte způsob uložení posluchačů notifikací v instancích standardních C++ kontejnerů.
Implementujte sadu rozšířených standardních C++ kontejnerů podporujících notifikace.
Implementace musí být odolná vůči vyhozeným výjimkám v zaregistrovaných posluchačích notifikací.
Otestujte funkčnost implementace.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 25. listopadu 2018

Bakalářská práce

Rozšíření kontejnerů standardní knihovny o notifikace

Michal Drabina

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jan Trávníček, Ph.D.

15. května 2019

Poděkování

Chtěl bych poděkovat Janu Trávníčkovi, vedoucímu mé práce, za trpělivost, kterou měl při vedení mé práce, přestože jsem byl stále pozadu. Dále bych chtěl poděkovat svým rodičům, kteří mě v průběhu studia podporovali. Poděkování patří také mé přítelkyni, která mi byla psychickou oporou v době psaní práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2019 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Michal Drabina. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Drabina, Michal. *Rozšíření kontejnerů standardní knihovny o notifikace*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zabývá návrhem a implementací rozšíření kontejnerů standardní knihovny C++ o oznámení pokusů o změnu jejich obsahu. Na základě výsledků vyhodnocení posluchači se operace provede nebo zruší. Implementace je otestována z pohledu funkcionality a poždavků na typy objektů uložených v kontejneru.

Klíčová slova C++, rozšíření, standardní knihovna, kontejnery, změna obsahu, notifikace, validní stav, kontrola operace

Abstract

This thesis deals with design and implementation of extension of C++ standard libary containers with notification about attepts to alter their content. Based on results of evaluation by listeners the operation is performed or cancelled. Implementation is tested from the perspective of functionality and type requirements imposed on elements of containers.

Keywords C++, extension, standard library, containers, content change, notifications, valid state, operation check

Obsah

Úvod 1
Struktura práce . . . 2

1 coruja 3
1.1 Množina notifikací . . . 3
1.2 Implementace . . . 4

2 Analýza a návrh 7
2.1 Funkční požadavky . . . 7
2.2 Nefunkční požadavky . . . 8
2.3 Množina notifikací . . . 8
2.4 Notifikace vložení . . . 9
2.5 Notifikace odebrání . . . 9
2.6 Notifikace nahrazení . . . 10
2.7 Notifikace změny hodnoty . . . 10
2.8 Přístup k prvkům kontejneru . . . 10
2.9 Uložení posluchačů v instancích kontejnerů . . . 11
2.10 Kontejnery . . . 12
2.11 Systém notifikací . . . 12

3 Implementace 15
3.1 Shared proxy . . . 15
3.2 Třída notifier . . . 15
3.3 Třída connection . . . 17
3.4 Třídy reprezentující notifikace . . . 17
3.5 Třída notifications_base . . . 20
3.6 Práce s prvky kontejnerů . . . 21
3.7 Pomocné struktury . . . 24
3.8 Kontejnery . . . 25
3.9 Testování . . . 30

Závěr **31**

Literatura **33**

A Seznam použitých zkratek **35**

B Obsah přiloženého CD **37**

Seznam obrázků

1.1 Diagram tříd kontejnerů coruja 6

Úvod

Kontejnery jsou jednou ze základních částí objektově orientovaných programů. Jsou to objekty, které spravují úložiště pro své prvky a poskytují rozhraní pro práci s nimi . Na zakladě datové struktury, kterou daný kontejner implementuje se liší v přístupu k prvkům, možnostmi vkládání a mazání prvků a také asymptotickou složitostí jednotlivých operací.

Operace nad kontejery a jejich prvky často obklopuje kód, který přímo souvisí s těmito operacemi. Typicky jsou to kontroly, které zaručují, že obsah kontejneru po dané operaci zůstane ve validním stavu. Dále je to zpracování nového obsahu, čímž se ale tato práce nezabývá. Validita daných operací může záviset jak na aktuálním stavu kontejneru, tak na jiných objektech, což může vést ke zbytečnému posílání těchto závislostí programem. Při nesprávném návrhu může rozšiřování programu vést k znepřehlednění kódu, jeho duplikaci a k nucenému refaktorování. Jakékoli seskupování kódu pak neumožňuje dynamicky měnit, jaké kontroly se provádějí.

Rozhodl jsem se proto vytvořit knihovnu rozšiřující kontejnery standardní knihovny C++ o blokující notifikace při pokusu o změnu obsahu. Kontroly budou uloženy v instancích kontejnerů ve formě posluchačů notifikací, které bude možné dynamicky přidávat a odebírat. Operace se provede pouze, pokud bude schválena všemi posluchači.

Jako první je třeba správně definovat množinu notifikací, která reflektuje změny obsahu na základě operací nad kontejnery a jejich prvky. Dalším cílem je implementace systému notifikací, který bude jednoduché používat a implementace rozhraní jednotlivých kontejnerů. Cílem práce není implementace datových struktur. Nakonec je nutné otestovat jak funkčnost rozšíření, tak požadavky na typy objektů uložené v kontejnerech. Tyto požadavky by se pokud možno neměly lišit od požadavků původních kontejnerů.

Výsledná knihovna zpřehlední a zjednoduší práci s kontejnery s minimálním dopadem na stávající kód.

Struktura práce

V první kapitole bude zanalyzována knihovna, která implementuje notifikace při změně obsahu kontejneru a změně hodnoty objektu. Další kapitola se bude věnovat požadavkům na knihovnu a návrhu řešení. Poté bude diskutována samotná implementace a její testování. Na závěr bude zhodnoceno splnění cílů práce a definováno, jakým směrem by se mohl ubírat další vývoj knihovny.

Kapitola 1

coruja

Coruja je knihovna, implementující notifikace při použití metod měnících obsah kontejneru nebo změně hodnoty objektu. Využívá standard C++11. Knihovna bude analyzována z pohledu funkcionalit, návrhu a implementace.

„*Je to alternativní řešení návrhového vzoru Observer pomocí signálů a slotů* [...]*. STL kontejnery jako std::vector jsou upraveny jako pozorovatelné, to znamená, že posluchačům může být nahlášeno, že byl přidán nebo odebrán prvek.*“ [1][překlad autora] Dále knihovna poskytuje možnost, aby posluchačům byla oznámena změna obsahu proměnné. Notifikace nijak neinteragují se samotnými operacemi.

1.1 Množina notifikací

Knihovna implementuje tři typy notifikací:

- *After insert*
- *Before erase*
- *After change*

První dvě zmiňované fungují nad operacemi kontejnerů, poslední zmíňená je implementována zcela nezávisle na kontejnerech a lze ji použít pro jakýkoli objekt.

1.1.1 After insert

Posluchači jsou voláni po vložení prvků do kontejneru. Užitečná notifikace, která umožňuje automaticky zpracovat nový stav obsahu kontejneru. Díky informaci o tom, které prvky byly přidány je možné zpracovat pouze tyto prvky.

1.1.2 Before erase

Oznámení o vymazání prvků z kontejneru ještě před provedením operace. Vzhledem k nemožnosti jakékoli signalizace neúspěchu, není možné operaci zastavit. To znamená, že notifikaci nelze použít jako kontrolní mechanizmus. Při zpracovávání oznámení posluchači je obsah kontejneru stále v původním stavu, což omezuje i možnost zpracování nového stavu obsahu. Jediné využití leží ve zpracování prvků, které budou odstraněny.

1.1.3 After change

Pomocí této notifikace je oznamován nový stav obsahu proměnné. Podobně jako u *After insert* je přínosem možnost zpracování nového obsahu.

1.2 Implementace

Tato sekce je zaměřena na analýzu implementace systému notifikací, posluchačů a samotnych kontejnerů.

1.2.1 Signály a sloty

Podle názvu by se mohlo zdát, že knihovna bude používat systém podobný tomu, který využívá knihovna Qt.[2] Takový systém je ale zbytečně složitý nástroj pro potřeby této knihovny. Signál je v tomto případě šablona třídy, která obsahuje list funkcí, posluchačů, které autor nazývá sloty. Signál poskytuje rozhraní pro vytvoření spojení se slotem a zavolání všech slotů. Pro reprezentaci spojení se využívá další třída s názvem *connection*, pomocí které lze spojení ukončit nebo zablokovat daný slot. Zablokovaný slot je stále uložen v seznamu, ale do odblokování není při notifikaci volán.

Zvolený systém je naprosto dostačující a možnost blokování slotů je zajímavou funkcionalitou navíc.

1.2.2 Notifikace

Notifikace jsou signály, kterým je jako parametr šablony předán předpis funkce, který musí sloty splňovat. Jednotlivé objekty, které je využívají si definují vlastní notifikace podle členských typů a udržují si je jako členské proměnné. Tento přístup by mohl v závislosti na návrhu zbytku knihovny vést k duplikaci kódu.

1.2.2.1 Notifikace nad kontejnery

Posílají do svých posluchačů referenci na kontejner a díky faktu, že všechny obě definované notifikace pracují nad prvky, které existují v kontejneru, je možné posluchačům posílat rozsah iterátorů identifikující prvky, se kterými operace

pracuje. Pro sekvenční kontejnery tento přístup funguje dobře, jelikož všechny operace přidání nebo odebrání prvku vždy pracují nad nepřerušovaným rozsahem prvků. Problém nastává u asociativních kontejnerů, které prvky ukládají na základě porovnávací funkce. V tomto případě se po vložení prvků do kontejneru tyto prvky zpětně dohledávají a notifikují se po jednom, což přidává operacím režii navíc.

1.2.2.2 Notifikace nad objektem

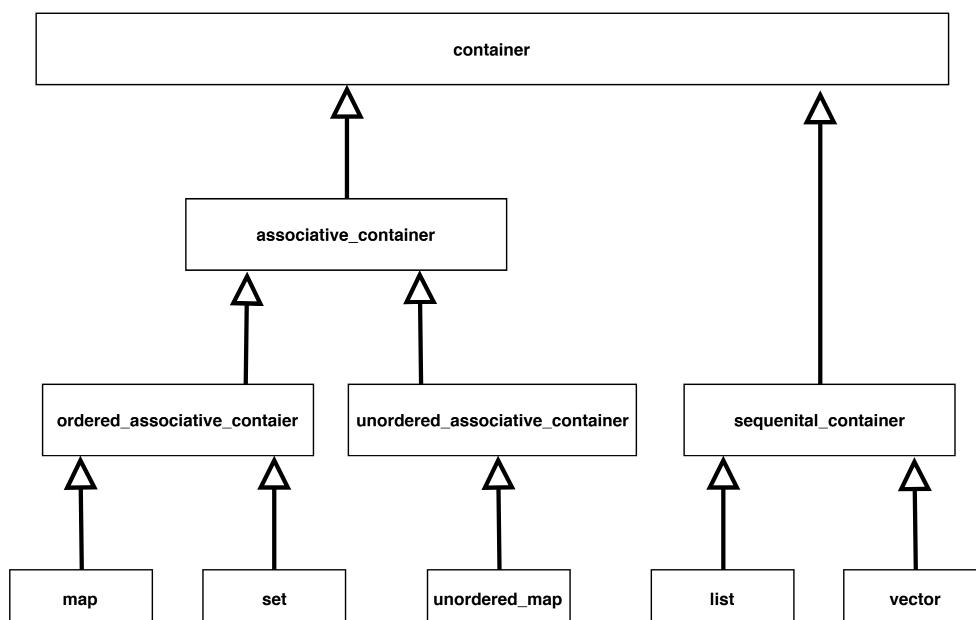
Pro využití této notifikace je třeba objekt obalit do poskytnuté třídy. Posluchačům notifikací po změně hodnoty objektu je k dispozici reference na instanci této obalující třídy. Ta implementuje vlastní operátor =, který volá operátor = obaleného objektu a poté samotnou notifikaci. Funkcionalita je tedy omezena pouze na typy s implementovaným operátorem =. Jiné nekonstantní operace nad tímto objektem ale provádět nezle. Třída také poskytuje přístup ke konstatní referenci obalovaného objektu. Požadavky a omezení obalující třídy mohou být problémem pro reálné využití.

1.2.3 Kontejnery

Kontejnery jsou implementovány jako šablony obalových tříd, kterým je možné jako jeden z parametrů šablony předat, jaký kontejner bude třída obalovat. Způsob jakým je šablona napsána ale částečně omezuje, jaké kontejnery lze použít. Omezení spočívá v tom, že použitý kontejner musí být šablona třídy se stejným počtem parametrů šablony, jako kontejnery standardní knihovny.

Duplikaci kódu je zamezeno využitím dědičnosti. Hierarchie dedičnosti kopíruje rozdělení kontejnerů podle vlastností, jelikož jednotlivé kategorie kontejnerů mají více společných metod. Zvolená množina notifikací tomuto návrhu nebrání, protože není třeba jakkoli měnit samotné operace, pouze se přidává volání slotů před nebo po dané operaci. Jeden z kontejnerů, který by do tohoto návrhu nezapadal, *forward_list*, není implementován. Tento návrh také zabraňuje problému s duplikací kódu vycházející ze způsobu implementace notifikací.

Základní třída *container* implementuje rozhraní společné pro všechny kontejntery. Přebírá členské typy z obalovaného kontejneru. Tyto typy je však třeba propagovat až do tříd reprezentující jednotlivé kontejnery. Definuje typy notifikací a jejich instance ukládá je jako své členské proměnné.



Obrázek 1.1: Hierarchie dědičnosti kontejnerů v knihovně coruja

Kapitola 2

Analýza a návrh

Ke správnému návrhu jakéhokoli softwaru je třeba znát požadavky na něj kladené. Tyto požadavky lze rozdělit na dvě základní skupiny: *funkční požadavky* a *nefunkční požadavky*. Funkční požadavky popisují, jaké chování a funkce jsou od díla očekávány. Nefunkční požadavky kladou nároky a omezení na využité technologie a postupy. Jasné definování požadavků vede k jednodušším rozhodnutím ve fázi návrhu. Kvalitní návrh zase šetří čas.

2.1 Funkční požadavky

F1: Notifikace při změně obsahu. Kontejnery musí posluchačům oznamovat veškeré pokusy o změnu jejich obsahu. Požadavek se vztahuje nejen na operace nad kontejnerem samotným, ale také na pokusy o změnu hodnoty některého z prvků.

F2: Notifikace před operací. Veškeré notifikace musí proběhnout před uskutečněním dané operace.

F3: Blokující notifikace. Posluchači notifikací musejí mít možnost zablokovat požadovanou operaci. To znamená, že pokud posluchač vyhodnotí danou operaci jako nevalidní, obsah objektu se nesmí změnit.

F4: Dynamická správa notifikací. Knihovna musí podporovat změny seznamu posluchačů za běhu programu. Základním požadavkem je možnost přidávání a odebírání posluchačů.

F5: Udržování celistvosti operace. Pokud operace využívá více druhů notifikací nebo postupné volání notifikací na různé části dat, pak selhání jakékoli z částí oznamovacího procesu musí vest ke zrušení celé operace.

F6: Podpora posluchačů se stavem. Systém notifikací musí podporovat posluchače, které udržují stav a při zpracování notifikací jej mění.

To znamená, že pokud operaci odmítne některý z posluchačů volaný později, notifikační systém musí doručit informaci o tom, že ohlášená operace neproběhne.

F7: **Odolost vůči vyjímkám.** Implementace musí být odolná vůči vyjímkám vyhozeným v posluchačích notifikací. To znamená, že pokud posluchač vyhodí vyjímku, notifikace se nesmí provést a její zpracování musí proběhnout stejně, jako by operaci odmítl. Poté se vyjímka propaguje dále programem.

2.2 Nefunkční požadavky

N1: **Podpora a využití C++17.** Implementované kontejnery musejí podporovat stejnou množinu operací jako kontejnery standardní knihovny ze standardu C++17. To také znamená možnost využití funkcí jazyku z tohoto standardu.

N2: **Zachování požadavků na typy.** Operace kontejnerů s notifikacemi by neměly klást vyšší požadavky na typ ukládaných objektů než operace kontejnerů standardní knihovny.

N3: **Zachování asymptotické složitosti.** Operace nutné k získání dat potřebných pro notifikace by měly ovlivnit asymptotickou složitost operací v co nejmenším rozsahu.

2.3 Množina notifikací

Definice množiny notifikací je závislá jak na nutnosti reprezentovat jejich kombinací veškeré operace nad kontejnery a jejich prvky, ale také na omezeních jazyka C++.

Všechny operace nad kontejnery je možné dekomponovat na posloupnost přidávání a odebírání prvků. Změna hodnoty uloženého objektu je však problémem. Změna může nastat dvěma způsoby. Prvním je volání operátoru =, který stávající hodnotu objektu nahrazuje za hodnotu poskytnutého objektu. Druhým způsobem je změna hodnoty objektu způsobená voláním metody objektu nebo dokonce přímá změna členských proměnných.

Řešení první možnosti je přímočařejší. Tuto notifikaci lze popsat jako nahrazení hodnoty. Lze ji také využít k zjednodušení notifikací nad kontejnery, kde v některých případech namísto kombinace notifikací o přidání a odebrání prvku lze zavolat pouze notifikaci o nahrazení hodnoty.

Druhý způsob změny hodnoty bude oznamován pomocí notifikace *value change*.

Jednotlivé notifikace budou diskutovány ve vlastních podkapitolách. Výsledná množina notifikací je následovná:

- vložení *(insert)*
- odstranění *(erase)*
- nahrazení *(replace)*
- změna hodnoty *(value change)*

2.4 Notifikace vložení

Operace přidání prvků fungují jak nad jedním prvkem, tak nad rozsahem iterátorů nebo celým kontejnerem. Vzhledem k požadavku **F2** a nutnosti jednoznačně určit argumenty volání posluchačů bude tato notifikace fungovat pouze nad jedním prvkem. Pro využití jiného přístupu by bylo nutné prvky přesunout do vybrané struktury a poté teprve do kontejneru, což celé zpracování komplikuje. U části kontejnerů je k dispozici i informace o pozici, kam bude prvek vložen, což může být pro zpracování prováděné v posluchačích cenná informace.

2.4.1 Argumenty volání posluchačů

Argumenty volání jsou v tomto pořadí:

- konstantní reference na kontejner
- konstatní reference na objekt, stejného typu jako prvky kontejneru, který má být vložen
- *pouze u některých kontejnerů:* konstatní iterátor na pozici, které bude použita k vložení prvku podle pravidel kontejneru

2.5 Notifikace odebrání

Díky požadavku **F2** je návrh argumentů volání posluchačů této notifikace jednoduchý. Prvky jsou při notifikaci uloženy v kontejneru, takže je možné je definovat pomocí iterátorů. Většina kontejnerů podporuje mazání prvků v rozsahu dvou iterátoru, což můžeme převést i argumenty posluchačů notifikací. Operace, které nepracují se sekvencí iterátoru budou volat posloupnost notifikací *erase*.

2.5.1 Argumenty volání posluchačů

Argumenty volání jsou v tomto pořadí:

- konstantní reference na kontejner
- konstatní iterátor reprezentující začátek rozsahu

- konstatní iterátor reprezentující konec rozsahu

Prvky, které budou odstraněny jsou reprezentovány rozsahem [první iterátor, poslední iterátor)

2.6 Notifikace nahrazení

Její primární využití je při volání operátoru = na objekt uložený v kontejneru. Vzhledem k nutnosti jednotného předpisu funkce pro posluchače je možné zaměřit se pouze na operátor =, který jako argument volání přijímá objekt stejného typu, jako objekt uložený v kontejneru. Způsob řešení volání posluchačů bude diskutován v podkapitole 2.8.

2.6.1 Argumenty volání posluchačů

Argumenty volání jsou v tomto pořadí:

- konstantní reference na kontejner
- konstatní iterátor na prvek kontejneru, který má být nahrazen
- konstantní reference na objekt, který má nahradit daný prvek

2.7 Notifikace změny hodnoty

Narozdíl od ostatních notifikací v tomto případě oznámení před operací není možné, protože před provedením operace neexistuje způsob, jakým se dostat k nové hodnotě objektu. Způsob jakým budou splněny požadavky **F1**, **F3** a **N2** bude diskutován v podkapitole 2.8.

2.7.1 Argumenty volání posluchačů

Argumenty volání jsou v tomto pořadí:

- konstantní reference na kontejner
- konstatní iterátor na prvek kontejneru, který byl změněn

2.8 Přístup k prvkům kontejneru

Ke splnění požadavku **F1** je nutné zamezit přímému přístupu rozhraní objektů uložených v kontejnerech a veškeré požadované operace nad objektem zpracovávat jinak. To znamená nutnost veškeré reference získané z kontejneru obalit do třídy, která tuto funkcionalitu bude zajišťovat.

Pro notifikaci nahrazení je nutné implementovat operátor =. Neurčitost obalovaného typu v tomto případě nepředstavuje problém. Jediný operátor =,

který je třeba implementovat je ten, který přijímá druhý objekt obalovaného typu. Operátor přijímající jiný typ, než ten obalovaný, by vyžadoval podporu dalších operací, což je v rozporu s požadavkem **N2**. Vzhledem ke znalosti předpisu tohoto operátoru pak může být jeho volání realizováno přes obalovou třídu, která zajistí volání notifikací.

Pro notifikaci změny hodnoty je třeba navrhnou řešení, které bude splňovat, nebo alespoň v co nejmenší míře porušovat požadavky **F1**, **F3** a **N2**.

Při zaměření se na požadavek **F3** se nabízí před provedením operace zkopírovat stávající objekt do proměnné, po provedení metody notifikovat změnu stavu a při neúspěchu vrátit objektu původni hodnotu. Tento přístup však porušuje požadavek **N2**, objekt by musel být zkonstruovatelný kopírováním a také podporovat přiřazení kopií. Vzhledem k neurčitosti typu objektu je nutné definovat rozhraní pro práci s ním bez kladení jakýchkoli požadavků. Uživatel knihovny však ví, s jakým typem pracuje, proto je možné přenést povinnost splnění požadavku na něj. Notifikace oznamuje již proběhlou změnu hodnoty a uživatel bude muset poskytnout funkci, která tuto změnu zvráti, pokud některý z posluchačů změnu odmítne.

Třída bude pracovat s jedním nebo dvěma volatelnými objekty, které jako jediný argument přijmou nekonstantní referenci na obalovaný objekt. V těchto voláních může uživatel pracovat s objektem bez jakýchkoli omezení. První objekt je určen pro změnu hodnoty prvku. Druhý objekt je určen pro zvrácení dané změny. Po volání prvního objektu je zavolána notifikace změny hodnoty. Pokud tato notifikace selže, je zavolán druhý objekt ke zvrácení změny hodnoty.

Jedno z prozkoumávaných řešení bylo také obalení volání členských funkcí.[3] Tento přístup je elegantnější, jelikož zachovává rozhraní obalovaného objektu, ale nedovoluje zvrátit operaci, ani oznámit výsledek volání posluchačů. Největší problém je však zachování přístupu k členským proměnným.

2.9 Uložení posluchačů v instancích kontejnerů

Způsob uložení posluchačů musí splňovat funkční požadavky **F4** a **F6**. Toho lze dosáhnout uložením posluchačů do kontejneru podporujícího iterování oběma směry. Pro využití kontejneru musí být jednoznačně určen typ ukládaných objektů. To znamená, že pro každou notifikaci, kterou kontejner podporuje musí definovat předpis funkce. Předpis funkce pro posluchače oznámení o operaci musí být až na návratový typ stejný, jako předpis funkce pro posluchače oznámení o neprovedení dříve oznámené operace.

Pro splnění **F3** využijeme *bool* jako návratový typ pro posluchače oznámení operace všech notifikací.

2.10 Kontejnery

Poznatky z analýzy knihovny coruja vedou k využití dědičnosti kvůli omezení duplikace kódu při implementaci rozhraní kontejnerů. Na tuto knihovnu jsou však kladeny jiné požadavky. Narozdíl od analyzované knihovny je k některým operacím třeba přidat kód pro získání objektů potřebných k provedení notifikace. Jednotlivé kontejnery pak pro tyto operace kladou různé nároky na typ uložených objektů, kterým se musí zpracování přizpůsobit. Využití polymorfismu pak přidává zbytečnou režii. Přesto by se menší části rozhraní daly rozdělit do hierarchické struktury, ale z důvodu zjednodušení prvotní implementace této možnosti nebude využito. To znamená, že každá třída reprezentující kontejner podporující notifikace, bude implementovat celé rozhraní daného kontejneru. Tomuto rozhodnutí musíme přizpůsobit také návrh systému notifikací.

2.10.1 Kontejnery standardní knihovny a jejich notifikace

- **array**:

  replace, value change

  Prvky nelze přidávat ani odebírat, je možné měnit jejich hodnotu.

- **vector, deque, list, forward_list**:

  insert, erase, replace, value change

  Prvky lze přidávat, odebírat a měnit jejich hodnotu

- **set, multiset, unordered_set, unordered_multiset**:

  insert, erase, replace

  Prvky lze přidávat a odebírat. Měnit jejich hodnotu nelze, ale notifikace replace se dá využít při notifikování odstranění klíče a jeho následné přidání v rámci jedné operace.

- **map, multimap, unordered_map, unordered_multimap**:

  insert, erase, replace, value change

  Prvky lze přidávat, odebírat a měnit hodnotu mapovaného objektu.

2.11 Systém notifikací

Vzhledem k navrhnuté množině notifikací a zvolenému způsobu implementace kontejnerů je nutné zajistit jednoduchý způsob, jakým specifikovat notifikace, které bude kontejner používat. Části kódu použité v textu budou zjednodušeny pouze pro demonstrování diskutované problematiky.

Jednou z možností je definice objektů zajišťujících funkcionalitu notifikací jako členských proměnných. Tento přístup splňuje požadavek na specifikaci

podmnožiny notifikací využívaných kontejnerem. Problémem je však nutnost definice rozhraní pro práci s posluchačemi v každém kontejneru. Implementace *getterů* těchto objektů pak zbytečně porušuje zapouzdření objektu.

Druhou možností je využití dědičnosti. Podporované notifikace jsou definovány jako seznam základních tříd. Rozhraní pro práci s posluchači je zděděno z těchto základních tříd. Není nutné využití polymorfismu, což znamená, že tento přístup nepřidává režii kvůli využívání tabulky virtuálních metod.

Další užitečnou funkcionalitou je možnost provádět operace jako vymazaní všech posluchačů nebo výměnu všech posluchačů nad všemi notifikacemi, které kontejner používá. Toho je možné dosáhnout přidáním další třídy do hierarchie dědičnosti mezi samotný kontejner a jednotlivé notifikace. Tato třída bude implementována jako variadická šablona třídy, která jako parametry šablony přijme třídy notifikací, od kterých bude dědit. Tento přístup zachová možnost jednoduché specifikace

Jednotlivé notifikace podporují stejné operace s posluchači a s vyjímkou *value change* také stejný proces notifikování. Opět se tedy nabízí využití dědičnosti. V tomto případě však třídy reprezentující jednotlivé notifikace budou od základní třídy dědit pouze implementaci a veřejné rozhraní budou implementovat samy. To znamená, že není nutné využívat polymorfismu.

KAPITOLA 3

Implementace

Tato kapitola se bude zabývat implementačními detaily, kterými bylo dosaženo požadovaných vlastností částí knihovny popsaných v minulé kapitole. Budou také popsány změny v chování kontejnerů a důvody, které vedly k těmto změnám.

Celá knihovna je implementována pouze v hlavičkových souborech bez jakýchkoli vnějších závislostí.

Veškeré ukázky kódu budou upraveny a omezeny pouze na části diskutované v textu.

3.1 Shared proxy

Velmi jednoduchá kostrukce, která je ale pro správné fungování celého systému zcela nezbytná.

```
template<typename Ty>
using shared_proxy = std::shared_ptr<Ty*>;
```

Vzhledem k nutnosti volat notifikace z jiných tříd, než je samotný kontejner, je potřeba mít ke kontejneru v těchto třídách přístup. K tomu by bylo možné využít pouze jednoduchý pointer. Problém ale nastává při využití metody swap. Ta neinvaliduje reference na objekty uložené v kontejneru. V tomto případě je nutná možnost pointer vyměnit, k čemuž je nutný další stupeň indirekce.

3.2 Třída notifier

Základní třída jednotlivých notifikací. Je určena k chráněnému dědění. Parametry šablony se používají jako argumenty předpisu funkce, který musejí posluchači splňovat. Typy se jí tedy předávají včetně konstantnosti a referencí.

Posluchači se udržují v instancích *std::function*. K reprezentaci páru posluchače oznámení o operaci a posluchače o neprovedení dané operace se používá *std::pair*. Tyto objekty se ukládají do kontejneru *std::list*. Ten svými vlastnostmi nejblíže splňuje požadavky kladené na operace nad posluchači. Třída implementuje metody pro přidání posluchačů a jejich notifikaci způsobem, který splňuje funkční požadavky **F3**, **F6** a **F7**. Respektuje také pořadí volání posluchačů. Oznamování o neprovedení oznámené operace se tedy provádí v opačném pořadí, než původní oznámení. Dále také poskytuje možnost vymazat všechny posluchače nebo je vyměnit s jinou instancí třídy *notifier*.

Třída implementuje volání posluchačů oznámení o operaci a volání posluchačů o neprovedení oznámené operace také separátně, jelikož notifikace změny hodnoty potřebuje upravit samotný proces notifikace.

Odstranění posluchačů ze seznamu je možné přes instanci třídy *connection*, která reprezentuje připojení posluchače k notifikaci. Vzhledem k tomu, že třída *notifier* poskytuje metodu swap, je nutné využít shared proxy.

```
1  template<typename... Args>
2  class notifier
3  {
4  protected:
5      using op_listener_type = std::function<bool(Args...)>;
6      using undo_listener_type = std::function<void(Args...)>;
7      using listeners_pair = std::pair<op_listener_type,
       ↪  undo_listener_type>;
8
9  public:
10     using listeners_type = std::list<listeners_pair>;
11     using shared_proxy = utils::shared_proxy<listeners_type>;
12     using connection_type = connection<notifier>;
13
14     connection add_listener(op_listener_type&& listener,
       ↪  undo_listener_type&& undo_listener = nullptr);
15
16     bool notify(Args... args) const;
17     typename listeners_type::const_iterator notify_operation(Args...
       ↪  args) const;
18     void notify_undo(typename listeners_type::const_iterator
       ↪  failed_listener, Args... args) const;
19     void notify_undo(Args... args) const;
20
21     void clear_listeners();
22     void swap_listeners(notifier& other);
23
24     private:
25         listeners_type m_listeners;
26         shared_proxy   m_proxy;
27  };
```

3.3 Třída connection

Odpojení posluchače od notifikace je realizováno přes tuto třídu. Je také možné zkontrolovat, jestli je spojení stále aktivní. Další implementovanou funkcionalitou je změna posluchače oznámení o neprovedení oznámené operace.

K identifikaci posluchače se používá iterátor do seznamu posluchačů.

Původním záměrem bylo povolit kopírování instancí této třídy. Od něj se ale muselo upustit, jelikož implementace této funkcionality byla komplikovanější a její přítomnost není esenciální pro fungování systému. Instance třídy jsou tedy *move-only*.

```
1  template<typename Notifier>
2  class connection
3  {
4  private:
5      using shared_proxy = typename Notifier::shared_proxy;
6      using iterator = typename Notifier::listeners_type::iterator;
7      using undo_listener_type = typename
       ↪  Notifier::listeners_type::value_type::second_type;
8
9  public:
10     connection(const shared_proxy& proxy, iterator listener);
11
12     connection(connection&&) = default;
13     connection& operator=(connection&&) = default;
14
15     bool active();
16     void disconnect();
17     void change_undo_listener(undo_listener_type&& undo_listener);
18
19     private:
20         shared_proxy m_proxy;
21         iterator     m_listener;
22 };
```

3.4 Třídy reprezentující notifikace

Všechny třídy reprezentující notifikace jsou implementovány jako šablony tříd, které chráněně dědí od třídy *notifier*. Jako parametry šablony přijímají typy, které potřebují k definici argumentů volání posluchačů. Chráněné dědění bylo zvoleno kvůli tomu, že kontejnery mohou dědit od více notifikací zároveň. To by mohlo při veřejném dědění vést k nejednoznačnosti volaných metod. Třídy notifikací tedy implementují vlastní veřejné rozhraní pro práci s posluchači. Jmenovitě přidávání posluchačů, výměna posluchačů s jinou instancí notifikace stejného druhu a vymazání všech posluchačů. Dále pro kontejnery implementují chráněné rozhraní pro samotné oznamování.

3.4.1 Třída insert

Reprezentuje notifikaci přidání prvku do kontejneru. U některých kontejnerů je možné posluchačům předat informaci o tom, kam bude prvek přidán. Ostatní tuto informaci neposkytují. Univerzálnost třídy v tomto směru je zajištěna její realizací jako variadické šablony třídy.

Posluchače je možné notifikovat pouze o vložení jednoho prvku. Kontejnery však často pracují s množinou prvků. Třída proto implementuje i metody pro postupné oznamování prvků z rozsahu iterátorů. Stejně jako volání jednotlivých posluchačů je dodrženo pořadí volání notifikací jednotlivých prvků. Implementace této funkcionality splňuje požadavkek **F5**.

```
template<typename Container, typename T, typename... ConstIterator>
class insert : protected notifier<const Container&, const T&,
↪ ConstIterator...>
{
...
protected:
    /**
     * This overload participates in overload only if ConstIterator
    ↪ isn't an empty pack
     */
    template<typename PosIterator, typename ValIterator>
    bool insert_called(ValIterator val_first, ValIterator val_last,
    ↪ PosIterator pos_first);

    /**
     * This overload participates in overload only if ConstIterator
    ↪ isn't an empty pack
     */
    template<typename PosIterator, typename ValIterator>
    void insert_undo(ValIterator val_fail, ValIterator val_first,
    ↪ PosIterator pos_fail);

    /**
     * This overload participates in overload only if ConstIterator is
    ↪ an empty pack
     */
    template<typename ValIterator>
    bool insert_called(ValIterator val_first, ValIterator val_last);

    /**
     * This overload participates in overload only if ConstIterator is
    ↪ an empty pack
     */
    template<typename ValIterator>
    void insert_undo(ValIterator val_fail, ValIterator val_first);
...
}
```

3.4.2 Třída replace

Replace realizuje oznámení o nahrazení hodnoty prvku kontejneru. Stejně jako u notifikace *insert* je vhodné implementovat možnost práce s více prvky najednou.

```
template<typename Container, typename ConstIterator, typename T>
class replace : protected notifier<const Container&, ConstIterator,
↪ const T&>
{
...
protected:
    template<typename PosIterator, typename ValIterator>
    bool replace_called(PosIterator pos_first, ValIterator val_first,
    ↪ ValIterator val_last);

    template<typename PosIterator, typename ValIterator>
    void replace_undo(PosIterator pos_fail, ValIterator val_fail,
    ↪ ValIterator val_first);
...
};
```

3.4.3 Třída value_change

Změna hodnoty prvku uloženého v kontejneru je oznamována touto notifikací. Vzhledem k volání posluchačů až po provedení operace, přijímá metoda realizující volání posluchačů kromě iterátoru na změněnou hodnotu také funkci, která bude zavolána při odmítnutí nové hodnoty některým z posluchačů. Ta by měla zvrátit provedenou změnu hodnoty. Až poté jsou zavolány posluchače oznámení o neprovedení oznámené operace.

Na této ukázce je také vidět, že při vyhození vyjímky některým z posluchačů operace opravdu neproběhne. Navíc i v bloku *catch* je snaha dodržet požadavek **F6**.

```
template<typename Container, typename ConstIterator>
class value_change : protected notifier<const Container&,
↪ ConstIterator>
{
private:
...
    template<typename C, typename T, typename B>
    friend class reference_wrapper;
protected:
    bool value_changed(ConstIterator it, std::function<void()>
    ↪ rollback = nullptr) const {
        typename base::listeners_type::const_iterator failed_it;
        try {
            failed_it = this->notify_operation(*static_cast<const
            ↪ Container*>(this), it);
```

```
13          } catch (...) {
14              rollback();
15              this->notify_undo(failed_it, *static_cast<const
                ↪  Container*>(this), it);
16
17              throw;
18          }
19
20          if (!this->check_success(failed_it)) {
21              rollback();
22              this->notify_undo(failed_it, *static_cast<const
                ↪  Container*>(this), it);
23
24              return false;
25          }
26
27          return true;
28      }
29  }
```

3.4.4 Třída erase

Erase reprezentuje notifikaci odstranění prvků z kontejneru. Jako jediná z notifikací neimplementuje žádnou funkcionalitu navíc oproti základní třídě *notifier*.

```
1  template<typename Container, typename ConstIterator>
2  class erase : protected notifier<const Container&, ConstIterator,
   ↪  ConstIterator>;
```

3.5 Třída notifications_base

Třída tvoří mezičlánek v hierarchii dědičnosti mezi kontejnerem a využívanými notifikacemi. Poskytuje veřejné rozhraní pro práci se všemi posluchači napříč notifikacemi. Původním záměrem bylo podporovat operaci *merge*, problémem však bylo udržování správného stavu *shared_proxy* v instancích třídy *connection*. Podporovanými operacemi jsou *clear* a *swap*.

```
1  template<typename... Notifications>
2  class notifications_base : public Notifications...;
```

Díky tomuto způsobu definice třídy je možné definovat množinu notifikací používaných kontejnerem následovně:

```
1  class vector : public notifications_base<insert<...>, erase<...>,
   ↪  replace<...>>
```

Iterování nad jednotlivými notifikacemi je řešeno přístupem viděným ve funkcionálním programování.

```
1  ...
2      void clear_listeners() { clear_listeners_impl<Notifications...>();
       ↪  }
3  
4      template<typename Last>
5      void clear_listeners_impl() {
6          Last::clear_listeners();
7      }
8  
9      template<typename First, typename Second, typename... Rest>
10     void clear_listeners_impl() {
11         First::clear_listeners();
12         clear_listeners_impl<Second, Rest...>();
13     }
```

3.6 Práce s prvky kontejnerů

Metody kontejnerů a iterátorů vracející reference nebo ukazatele na uložené prvky jsou upraveny tak, aby vracely instance tříd zajišťující notifikaci posluchačů při změně hodnoty daných prvků. Instance těchto tříd nejsou nikde uloženy, což znamená, že se při každém volání daných metod konstruuje nová instance.

3.6.1 Reference

Pro práci s referencemi objektů implementuje knihovna šablony obalujících tříd. Konstatní reference je pro zachování konzistentního rozhraní obalována i přesto, že přes ni není možné objekt upravovat.

Místo reference tyto třídy udržují iterátor, jelikož je potřeba i jako parametr posluchačů. Konstatní verze udržují pouze tento iterátor. Nekonstatní pak navíc potřebují přístup ke kontejneru, k čemuž je využito *shared_proxy*. Práce s obalovaným objektem je možná přes metodu *invoke*.

Pro zjednodušení kódu jsou konstantní varianty obalujících tříd využity jako základní třídy nekonstatních variant a jejich rozšíření.

```
1  ...
2  template<typename Ty>
3  void invoke(Ty&& callable) const {
4      auto function = std::function(std::forward<Ty>(callable));
```

```
5       static_assert(std::is_same_v<decltype(function),
        ↪   std::function<void(const T&)>> ||
        ↪   std::is_same_v<decltype(function), std::function<void(const
        ↪   T)>> || std::is_same_v<decltype(function),
        ↪   std::function<void(T)>>, "In
        ↪   const_reference_wrapper::invoke(Ty) const: Ty can only be
        ↪   Callable with signature void(const T&), void(const T) or
        ↪   void(T)");
6
7       function(data());
8   }
9   ...
```

Metoda *invoke* konstatní verze *reference wrapperu* přijímá volatelný objekt. Povolený předpis volání tohoto objektu je vidět na úryvku kódu výše. V tomto případě by bylo možné přijímat rovnou std::function, ale tento přístup by změnil chování oproti nekonstantní variantě.

```
1   ...
2   using base::invoke;
3
4   template<typename Ty>
5   bool invoke(Ty&& callable, Ty&& rollback) {
6       std::function function(std::forward<Ty>(callable));
7       static_assert(std::is_same_v<decltype(function),
        ↪   std::function<void(T&)>>, "reference_wrapper::invoke(Ty, Ty):
        ↪   Ty can only Callable with signature void(T&)");
8
9       return invoke_nonconst(std::move(function),
        ↪   std::forward<Ty>(rollback));
10  }
11  ...
```

Pro správné fungování ADL je nutné deklarovat použití metody invoke ze základní třídy, kterou je konstatní varianta *reference wrapperu*.

Nekonstatní varianta navíc přijímá druhý volatelný objekt, jehož úkolem je zvrátit změny provedené tím prvním.

Pro správnou funkci této metody není možné využít chování třídy std::function. Při její konstrukci z volatelného objektu se kontroluje, jestli je možné předpis funkce tohoto objektu převést na předpis funkce std::function. To znamená, že následující kód bude fungovat.

```
1   std::function<int(int&)> = [](const int x){
2       return x == 1;
3   }
```

Takové chování je problémem, jelikož v nekonstatní verzi *reference wrapperu* je nutné se omezit pouze na funkce přebírající referenci na objekt. Toho

je dosaženo pomocí *compile time* assertu. V konstatní verzi by tento problém nebylo třeba řešit, ale z důvodu konzistence chování je implementována stejně.

Rozšíření nekonstatní reference implementuje operátor =, pro využití notifikace *replace*. Pro jejich použití je nutné, aby obalovaný typ implementoval daný operátor.

```
1  replaceable_reference_wrapper& operator=(const T& new_value);
2  replaceable_reference_wrapper& operator=(T&& new_value);
```

Pro kontejnery ukládající klíč a hodnotu jako *map* jsou implementovány specializace těchto tříd. Získávání daných hodnot z iterátoru se liší a pro reprezentaci celé hodnoty byly přidány členské proměnné tak, aby rozhraní odpovídalo tomu od *std::pair*.

```
1  template<typename Map>
2  class map_reference_wrapper : public reference_wrapper<Map, typename
   ↪   Map::value_type>
3  {
4  ...
5      key_wrapper<Map>               first;
6      mapped_reference_wrapper<Map> second;
7  };
```

3.6.2 Pointer

Ukazatel na prvek kontejneru je možné získat pouze z operátoru -> iterátoru. Místo ukazatele na prvek je vrácena instance třídy *pointer_proxy*, která v sobě udržuje *reference wrapper* na prvek a která implementující operátor ->. Volání operátoru se zřetězí. Výsledkem je tedy stejné rozhraní jako při vracení reference.

```
1  template<typename Wrapper>
2  class pointer_proxy
3  {
4  public:
5      pointer_proxy(const Wrapper& wrapper) : m_wrapper(wrapper) {}
6
7      Wrapper* operator->() { return &m_wrapper; }
8
9  private:
10     Wrapper m_wrapper;
11 };
```

3.6.3 Iterátory

Musejí být opraveny také návratové typy metod iterátorů kontejnerů. Stejně jako u *reference wrapperu*, konstatní iterátory udržují pouze obalovaný iterátor, nekonstatní verze pak také *shared_proxy* na kontejner ke kterému patří.

Iterátory definují členský typ *iterator_category*, přestože z důvodu používání obalujících tříd pro reference a ukazatele zcela nesplňují standard. Bez jeho definování by však algoritmy pracující s iterátory nevěděly, jaké operace mohou použít.

Oproti standardní knihovně implementují iterátory navíc metody pro získání obalovaného iterátoru a reference. Tato funkcionalita je využívána v kódu knihovny.

3.7 Pomocné struktury

Opakujícím se problémem při implementaci kontejnerů byla iterace nad objekty. Pro jednodušší a efektivnější implementaci byly vytvořeny tyto struktury a třídy.

3.7.1 Iterace nad jedním objektem

Jedním z problémů je nutnost iterace nad jedním objektem. U sekvenčních kontejnerů je možno přidat n kopií jednoho objektu. K využití hromadného oznámení poskytovaného notifikační třídou, je nutné předat rozsah iterátorů na objekty, které mají být přidány. K tomu slouží *count_ref_iterator*. Udržuje referenci na daný objekt a počítadlo. Dva iterátory jsou si rovny, pokud je hodnota jejich počítadel stejná.

Další abstrakce je prováděna v jednotlivých kontejnerech, kdy je někdy nutné mít k dispozici objekt definující rozsah iterátoru. K tomu v tomto případě slouží následující struktura:

```
1  template<typename Size, typename Difference, typename T>
2  struct count_ref_range
3  {
4      using iterator = helper_iterators::count_ref_iterator<Size,
       ↪  Difference, T>;
5
6      iterator begin() const { return iterator{0, m_ref}; }
7      iterator end() const { return iterator{m_count, m_ref}; }
8
9      Size     m_count;
10     const T& m_ref;
11 };
```

3.7.2 Neměnný iterátor

Při využití notifikace o přidání více prvků najednou , přijímají metody iterátor, pomocí kterého se iteruje nad iterátory do kontejneru, které reprezentují místo vložení. Jsou však případy, kdy tato pozice je konstantní. Místo vytvoření kontejneru s určitým počtem kopií stejného iterátoru se využívá struktura

constant_iterator_wrapper, chovající se jako iterátor. Iterující volání nejsou předány do obalovaného iterátoru a při dereferenci je předán právě tento iterátor.

3.7.3 Obal iterátoru

Vzhledem k implementaci notifikací k podpoře sekvenčních i asociativních kontejnerů je při hromadném oznámení nahrazení prvků nutné iteratovat nad iterátory do kontejneru. U sekvenčních kontejnerů se však nahrazení může dít pouze v prvcích jdoucích za sebou. Místo definice kontejneru s touto sekvencí iterátorů se využívá struktura *iterator_wrapper*, která jednodušše obalí iterátor do další vrstvy chovající se jako iterátor. Iterující volání jsou propagovány do obalovaného iterátoru a při dereferenci je vrácen právě tento iterátor.

3.7.4 Reverse range

Často je třeba iterovat nad všemi prvky kontejneru v opačném pořadí, než normálně. To je možné vyřešit přes normální *for* cyklus. Iterace nad celým kontejnerem se ale v moderním C++ řeší pomocí *foreach* cyklu. Pro zachování konzistence programovacího stylu je implementována tato třída:

```
1  template<typename Ty>
2  class reverse_range
3  {
4  public:
5      reverse_range(Ty& container) : m_underlying{container} {}
6      auto begin() { return m_underlying.rbegin(); }
7      auto end() { return m_underlying.rend(); }
8
9  private:
10     Ty& m_underlying;
11 };
```

3.8 Kontejnery

Při kombinaci všech doposud popsaných částí, je možné definovat třídu kontejneru. Definice je demonstrována na kontejneru *vector*.

```
1  template<typename WrappedVector>
2  class vector;
3
4  // for iterators
5  template<typename WrappedVector>
6  class vector_traits
7  {
8  public:
9      using container = vector<WrappedVector>;
```

```
10      using wrapped_type = WrappedVector;
11      using wrapped_iterator = typename wrapped_type::iterator;
12      using wrapped_const_iterator = typename
        ↪   wrapped_type::const_iterator;
13  };
14
15  // for class definitions
16  template<typename WrappedVector>
17  class vector_types
18  {
19  private:
20      using traits = vector_traits<WrappedVector>;
21
22  public:
23      using wrapped_type = typename traits::wrapped_type;
24      using iterator = iterators::random_access_iterator<traits>;
25      using const_iterator =
        ↪   iterators::const_random_access_iterator<traits>;
26      using value_type = typename WrappedVector::value_type;
27  };
28
29  template<typename WrappedVector>
30  using vector_notifications = notifications_base<
31      insert<vector<WrappedVector>, typename
        ↪   vector_types<WrappedVector>::value_type, typename
        ↪   vector_types<WrappedVector>::const_iterator>,
32      erase<vector<WrappedVector>, typename
        ↪   vector_types<WrappedVector>::const_iterator>,
33      replace<vector<WrappedVector>, typename
        ↪   vector_types<WrappedVector>::const_iterator, typename
        ↪   vector_types<WrappedVector>::value_type>,
34      value_change<vector<WrappedVector>, typename
        ↪   vector_types<WrappedVector>::const_iterator>>;
35
36  template<typename WrappedVector>
37  class vector : public vector_notifications<WrappedVector>
38  {
39  ...
40      shared_proxy m_proxy;
41      wrapped_type m_data;
42  }
```

Kontejnery, které nevyužívají notifikaci *value_change* nepotřebují udržovat instanci třídy *shared_proxy*.

Všechny kontejnery implementují metodu *container*, která poskytuje přístup k referenci na obalovaný kontejner. Ta může být užitečná pokud existuje část kódu, ve které uživatel nechce být notifikován.

Návratové typy a hodnoty metod využívajících notifikace kromě operátorů=, jsou upraveny tak, aby bylo možné indikovat neuspěch operace na základě no-

tifikací.

Veškeré notifikování je prováděno pouze, pokud operace může být provedena na základě podmínek, které definuje standardní knihovna. Například, pokud *set* již obsahuje daný klíč, volání posluchačů neproběhne.

Potřeba disponovat zkonstruovanými objekty pro notifikování znemaná, že veškeré záruky o nepoužití objektů předaných metodám, které standardní knihovna definuje končí momentem notifikace.

Popsané asymptotické složitosti nezahrnují proces notifikace, pouze proces získávání dat potřebných pro oznámení.

Kontejnery jsou implementovány po vzoru kontejnerů standardní knihovny. Nejsou však omezeny pouze na ně. Jako parametr šablony lze kontejneru předat jakýkoli kontejner, který se chováním a rozhraním podobá tomu ze standardní knihovny.

3.8.1 array

Operace pracující s tímto kontejnerem jsou ve standardní knihovně definovány jako *constexpr*. V této knihovně to vzhledem k udržování seznamu posluchačů a jejich volání není možné. Specifikátory constexpr jsou z rozhraní array odstraněny.

3.8.2 Konstruktory a operátory =

K rozhraní převzatého ze standardní knihovny jsou přidány konstruktory a operátory = pro obalovaný typ.

Všechny operátory =, až na *copy* a *move*, využívají notifikace. S vyjímkou kontejneru *array*, kde se využívá pouze notifikace *replace*, jsou to kombinace notifikací *replace*, *insert* a *erase*.

3.8.3 assign

Metoda *assign* je oznamována stejným způsobem, jako operátor =.

Pro přetížení pracující se dvěma iterátory je nejprve zkounstruována nová instance obalovaného typu, na kterém je operace provedena. Poté je provedena metoda *swap* mezi tímto prozatimním kontejnerem a členskou proměnnou. Přístup byl zvolen z důvodu, že předané iterátory nemusí být dereferencovatelné na typ ukládaný v kontejneru, ale pouze na typ, který na něj lze převést. Pro notifikace však potřebujeme již zkonstruované objekty. Metoda *swap* pak zajišťuje efektivitu a neklade žádné další požadavky na typ ukládaných objektů.

3.8.4 clear

Operace je oznamována notifikací *erase*.

3.8.5 insert

Metoda využívá notifikace *insert*.

Stejně jako u metody *assign* je v přetížení přijímajícího dvojici iterátorů zkonstruována dočasná instance obalovaného typu, do kterého jsou prvky vloženy. U kontejnerů implementující metodu *splice* je tato metoda použita pro přesunutí prvků do obalovaného kontejneru. V ostatních kontejnerech je využito *std::move_iterator*.

Přetížení metody insert přijímající *node handle* mapy a jejich variant musí také zkonstruovat dočasnou instanci obalovaného typu do kterého se prvek vloží, jelikož jiným způsobem nelze získat celý ukládany objekt, pouze jeho části. *Node handle* je z tohoto kontejneru po volání posluchačů znovu extraktován.

3.8.6 insert_or_assign

Volá notifikace *insert* nebo *replace*.

Při existenci prvku je pro notifikaci nutné hodnotu zkonstruovat před přiřazením. K požadavkům na typy přidává nutnost, aby typ mapovaných objektů podporoval přiřazení z *rvalue*.

Pokud prvek neexistuje, vložení je provedeno pomocí *emplace_hint*.

3.8.7 emplace, emplace_hint

Operace je oznamována pomocí notifikace *insert*.

Asociativní konteinery využívají dočasnou instanci obalovaného typu, do které je hodnota zkonstruována. Poté je extrahován *node handle*, který je předán metodě *insert*.

Sekvenční kontejnery implementující metodu *splice* také využívají dočasný kontejner. Hodnota je poté vložena právě pomocí metody *splice*.

Ostatní kontejnery zkonstruují dočasný objekt ukládaného typu a ten pomocí *std::move* přesunou do obalovaného kontejneru.

3.8.8 try_emplace

Stejně jako *emplace*, využívá notifikaci *insert*.

Metoda konstruuje dočasnou instanci obalovaného typu, do které je hodnota zkonstruována. Poté je extrahován *node handle*, který je předán metodě *insert*.

3.8.9 erase

Operace využívá notifikaci *erase*.

3.8.10 push_front, push_back

Operace jsou oznamovány pomocí notifikace *insert*.

3.8.11 emplace_front, emplace_back

Další z metod, které využívají notifikaci *insert*.

Deque neimplementuje metodu *splice*, což v kombinaci s nutností zkonstruovat hodnotu před notifikací vede k přidání požadavku na typ ukládaného objektu. Využívá dočasné instance ukládaného typu se kterým je po povolení operace posluchačemi zavolána odpovídající varianta *push*. Typ ukládaných objektů musí být zkonstruovatelný z *rvalue*. Tento přístup klade na typ menší požadavky než využití meody *emplace*.

List a *forward_list* používají již implementovanou metodu *emplace*.

3.8.12 pop_front, pop_back

Metody využívají notifikaci *erase*.

3.8.13 swap

S vyjímkou kontejneru *array*, kde se využívá pouze notifikace *replace*, je používáná kombinace notifikací *replace*, *insert* a *erase*.

Ve standardní knihovně má tato metoda konstatní asymptotickou složitost. Vzhledem k nutnosti zmapovat jaké prvky se v kontejnerech nachází a následné volání notifikaci se složitost mění následovně:

- **Sekvenční kontejnery:** O(n + m), kde n, m jsou velikosti kontejnerů
- **Seřazené asociativní kontejnery:** O(m * log n + n * log m), kde n, m jsou velikosti kontejnerů
- **Neseřazené asociativní kontejnery:** Amortizovaná složitost O(m + n), kde n, m jsou velikosti kontejnerů

3.8.14 merge

K rozhraní převzatého ze standardní knihovny jsou přidány přetížení pro obalovaný typ.

Operace využívá kombinaci notifikací *replace* a *insert* pro cílový kontejner a *erase* pro kontejner zdrojový.

Kvůli nutnosti mít v iterátorech přístup k vlastnícímu kontejneru, jsou při použití této metody invalidovány iterátory na přesunuté prvky.

3.8.15 extract

Operace využívá notifikaci *erase*.

3.8.16 splice

Operace využívá notifikaci *insert* pro cílový kontejner a *erase* pro kontejner zdrojový.

Ze stejných důvodů jako u metody merge jsou iterátory na přesunuté prvky invalidovány, pokud zdrojový objekt není stejný, jako objekt cílový.

3.8.17 remove, remove_if

Metody používají notifikaci *erase*.

3.9 Testování

Při psaní softwaru je těžké, ne-li nemožné vyhnout se chybám. Proto je pro zajištění kvality nutné kód testovat. Testy sice nemohou prokázat neexistenci chyb, mohou však odhalit jejich existenci. Na testovacím kódu lze také pochopit, jakým způsobem mají být části kódu používány.

Veškeré testy jsou prováděny s třídami implementující pouze minimální zdokumentované požadavky pro jednotlivé operace. Pokud se testy zkompilují znamená to, že požadavky odpovídají dokumentaci.

K testování je využita knihovna Catch2.[4] Její hlavní vyhodou je, že je celá implementována v jednom hlavičkovém souboru, což ulehčuje import do projektu.

Závěr

Prvním cílem práce bylo definování množiny notifikací, která reflektuje změny obsahu kontejneru. Veškeré operace měnící obsah kontejneru se podařilo vyjádřit posloupností notifikací z definované množiny. To znamená, že tento cíl byl splněn.

Dalším cílem bylo navrhnout jednoduše použitelný systém notifikací. Tento cíl se také podařilo splnit. Definice notifikací spočívá pouze v deklaraci, které základní třídy mají být použity. Není třeba znovu implementovat rozhraní.

Hlavním cílem bylo implementovat kontejnery rozšiřující kontejnery standardní knihovny s minimálními změnami v chování. Tento cíl byl také splněn, i když určité změny byly nevyhnutelné. Pozitivem je, že implementace není vázána čistě na kontejnery standardní knihovny, ale lze ji použít i s jinými kontejnery.

Možnosti pro další vývoj knihovny:

- Sloučení společného rozhraní kontejnerů do společných tříd
- Implementace možnosti dočasného suspendování posluchačů a jejich následné reaktivace
- Optimalizace algoritmů
- Prozkoumání možnosti aby notifikace *insert* fungovala na dvojicí iterátorů obalovaného typu
- Prozkoumat možnosti jak zabránit invalidaci iterátoru při operacích *merge* a *splice*

Literatura

[1] COSME, Ricardo; Coruja: Readme.org In: *Github* [online]. 2018-09-14 [cit. 2019-05-15]. Dostupné z: `https://github.com/ricardocosme/coruja/blob/383dd2106085c831269e69660447f8e6f922fd92/README.org`.

[2] The Qt Company Ltd.; Signals & Slots. In: *Qt Documentation* [online]. ©2019 [cit. 2019-05-15]. Dostupné z: `https://doc.qt.io/qt-5/signalsandslots.html`.

[3] STROUSTRUP, Bjarne; Wrapping c++ member function calls. *C++ Report*. 2000/6, 12(6).

[4] Catch Org; Catch2 In: *Github* [online]. 2019-05-09 [cit. 2019-05-15]. Dostupné z: `https://github.com/catchorg/Catch2/tree/f1e14a1168fc31718362e34d075cb0e3298dc671`.

Příloha A

Seznam použitých zkratek

STL Standard template library

ADL Argument-dependent lookup

Příloha B

Obsah přiloženého CD

```
 readme.txt ............................ stručný popis obsahu CD
 impl ...................................... zdrojové kódy implementace
   headers ................................. zdrojové kódy knihovny
   tests ...................................... zdrojové kódy testů knihovny
 thesis
   src ...................................... zdrojová forma práce ve formátu LaTeX
   BP_Drabina_Michal_2019.pdf ................ text práce ve formátu PDF
```