

Міністерство освіти і науки України  
Національний університет «Львівська політехніка»  
кафедра інформаційних систем та мереж

Григорович Віктор

**Об'єктно-орієнтоване програмування**  
**Контейнери та шаблони**

Навчальний посібник

2021

Григорович Віктор Геннадійович

**Об'єктно-орієнтоване програмування. Контейнери та шаблони.** Навчальний посібник.

Дисципліна «Об'єктно-орієнтоване програмування» вивчається після курсу «Алгоритмізація та програмування», цією дисципліною продовжується цикл предметів, що стосуються програмування та розробки програмного забезпечення.

В посібнику містяться теоретичні відомості, приклади, методичні вказівки з їх розв'язування, варіанти лабораторних завдань та питання і завдання з контролю знань з теми «Контейнери та шаблони».

Розглядаються наступні роботи лабораторного практикуму:

Лабораторна робота № 6.1.

Контейнери як параметри

Лабораторна робота № 6.2.

Контейнери-масиви

Лабораторна робота № 6.3.

Контейнери-списки

Лабораторна робота № 6.4.

Шаблони класів та шаблони функцій

Лабораторна робота № 6.5.

Шаблони класів та опрацювання виняткових ситуацій

Лабораторна робота № 6.6.

Шаблони класів

Лабораторна робота № 6.7.

Шаблони функцій, алгоритми та функтори

Відповідальний за випуск – Григорович В.Г.

## Стислий зміст

Перелік запозичених лістингів програм .....	15
Вступ.....	16
Тема 6. Контейнери та шаблони .....	17
Стисло та головне про контейнери та шаблони .....	17
Контейнери .....	17
Шаблони класів .....	20
Шаблони функцій.....	24
Теоретичні відомості.....	28
Контейнери .....	28
Шаблони класів .....	53
Шаблони функцій.....	81
Лабораторний практикум .....	104
Оформлення звіту про виконання лабораторних робіт .....	104
Лабораторна робота № 6.1. Контейнери як параметри .....	106
Лабораторна робота № 6.2. Контейнери-масиви .....	115
Лабораторна робота № 6.3. Контейнери-списки.....	129
Лабораторна робота № 6.4. Шаблони класів та шаблони функцій .....	135
Лабораторна робота № 6.5. Шаблони класів та опрацювання виняткових ситуацій .....	147
Лабораторна робота № 6.6. Шаблони класів .....	150
Лабораторна робота № 6.7. Шаблони функцій, алгоритми та функтори .....	159
Питання та завдання для контролю знань .....	168
Контейнери .....	168
Шаблони класів .....	169
Шаблони функцій.....	169
Шаблони.....	170
Предметний покажчик .....	173
Література .....	174

## Зміст

Перелік запозичених лістингів програм .....	15
Вступ.....	16
Тема 6. Контейнери та шаблони .....	17
Стисло та головне про контейнери та шаблони .....	17
Контейнери .....	17
Характеристики контейнерів .....	17
Доступ до елементів контейнера .....	18
Поняття ітератора.....	19
Операції контейнера .....	19
Підсумок.....	19
Шаблони класів .....	20
Створення шаблонів класів .....	20
Параметри шаблону .....	21
Використання шаблонів класів .....	21
Спеціалізація шаблонів класів .....	21
Підсумок.....	22
Шаблони функцій.....	24
Створення шаблону функції.....	24
Параметри шаблону за умовчанням .....	24
Перевантаження шаблонів функцій .....	25
Спеціалізація шаблонів функцій.....	25
Узагальнені алгоритми та функтори .....	25
Поняття функтора .....	25
Підсумок.....	25
Теоретичні відомості.....	28
Контейнери .....	28
Характеристики контейнерів .....	28
Доступ до елементів контейнера .....	29
Поняття ітератора.....	30
Операції контейнера .....	31
Доступ до елементів; зміна значень .....	31
Додавання та вилучення елементів та груп елементів .....	31
Пошук елементів та груп елементів .....	32
Об'єднання контейнерів .....	32

Спеціальні операції .....	33
Реалізація контейнерів .....	33
Числовий контейнер фіксованої довжини .....	35
Числовий контейнер змінної довжини .....	40
Послідовний контейнер – список .....	45
Послідовний контейнер – стек .....	50
Підсумок .....	52
Шаблони класів .....	53
Створення шаблонів класів .....	53
Параметри шаблону .....	53
Оголошення шаблону класу .....	55
Використання шаблонів класів .....	55
Зовнішнє визначення методів .....	55
Приклад шаблону класу .....	56
Лістинг 5. Простий масив – шаблон .....	56
Файл «Array.h» .....	56
Файл «L_6_6.cpp» .....	57
Результат виконання: .....	58
Пояснення .....	58
Ініціалізація нулем .....	59
Параметри шаблону, задані за умовчанням .....	60
Спеціалізація шаблонів класів .....	61
Статичні елементи в шаблонах класів .....	64
Лістинг 6. Шаблон класу із статичними елементами .....	64
Переваги та недоліки шаблонів .....	65
Шаблони класів з шаблонами .....	65
Поле-шаблон .....	66
Параметр-шаблон .....	67
Метод-шаблон .....	68
Лістинг 7. Метод-шаблон в шаблоні класу .....	68
Файл «Array.h» .....	68
Файл «Source.cpp» .....	69
Пояснення .....	69
Шаблони та успадкування .....	71
Клас → шаблон .....	71

Шаблон → шаблон .....	71
Шаблон → клас .....	71
Лістинг 8. Обчислення кількості об'єктів класу .....	72
Шаблони та дружні функції чи дружні класи .....	73
Позначення шаблонів на UML-діаграмах класів .....	77
Підсумок.....	78
Шаблони функцій.....	81
Створення шаблону функції.....	81
Параметри шаблону за умовчанням .....	83
Параметри шаблону – не типи .....	83
Перевантаження шаблонів функцій .....	84
Спеціалізація шаблонів функцій.....	84
Узагальнені алгоритми та функтори .....	84
Реалізація узагальнених алгоритмів .....	85
Вказівники на функції та вказівники на методи .....	87
Поняття функтора .....	90
Функтори-предикати. Унарні та бінарні функтори .....	92
Адаптери функторів.....	93
Патерн Strategy (стратегія) .....	98
Реалізація за допомогою інтерфейсів (абстрактних класів).....	99
Реалізація за допомогою шаблонів.....	100
Підсумок.....	102
Лабораторний практикум .....	104
Оформлення звіту про виконання лабораторних робіт .....	104
Вимоги до оформлення звіту про виконання лабораторних робіт №№ 6.1–6.7 .....	104
Зразок оформлення звіту про виконання лабораторних робіт №№ 6.1–6.7 .....	105
Лабораторна робота № 6.1. Контейнери як параметри .....	106
Мета роботи .....	106
Питання, які необхідно вивчити та пояснити на захисті.....	106
Приклад .....	106
Лістинг 2. Числовий контейнер змінної довжини.....	106
Варіанти завдань .....	111
Варіант 1.....	111
Варіант 2.....	111
Варіант 3.....	111

Варіант 4.....	111
Варіант 5.....	111
Варіант 6.....	111
Варіант 7.....	111
Варіант 8.....	111
Варіант 9.....	111
Варіант 10.....	112
Варіант 11.....	112
Варіант 12.....	112
Варіант 13.....	112
Варіант 14.....	112
Варіант 15.....	112
Варіант 16.....	112
Варіант 17.....	112
Варіант 18.....	112
Варіант 19.....	112
Варіант 20.....	112
Варіант 21.....	112
Варіант 22.....	113
Варіант 23.....	113
Варіант 24.....	113
Варіант 25.....	113
Варіант 26.....	113
Варіант 27.....	113
Варіант 28.....	113
Варіант 29.....	113
Варіант 30.....	113
Варіант 31.....	113
Варіант 32.....	113
Варіант 33.....	114
Варіант 34.....	114
Варіант 35.....	114
Варіант 36.....	114
Варіант 37.....	114
Варіант 38.....	114

Варіант 39.....	114
Варіант 40.....	114
Лабораторна робота № 6.2. Контейнери-масиви .....	115
Мета роботи .....	115
Питання, які необхідно вивчити та пояснити на захисті.....	115
Приклад .....	115
Варіант 0.....	115
Лістинг 1. Числовий контейнер фіксованої довжини .....	115
Варіанти завдань .....	121
Варіант 1.....	121
Варіант 2.....	122
Варіант 3.....	122
Варіант 4.....	122
Варіант 5.....	122
Варіант 6.....	122
Варіант 7.....	122
Варіант 8.....	123
Варіант 9.....	123
Варіант 10.....	123
Варіант 11.....	123
Варіант 12.....	123
Варіант 13.....	123
Варіант 14.....	124
Варіант 15.....	124
Варіант 16.....	124
Варіант 17.....	124
Варіант 18.....	124
Варіант 19.....	124
Варіант 20.....	125
Варіант 21.....	125
Варіант 22.....	125
Варіант 23.....	126
Варіант 24.....	126
Варіант 25.....	126
Варіант 26.....	126



Варіант 27.....	126
Варіант 28.....	126
Варіант 29.....	127
Варіант 30.....	127
Варіант 31.....	127
Варіант 32.....	127
Варіант 33.....	127
Варіант 34.....	127
Варіант 35.....	128
Варіант 36.....	128
Варіант 37.....	128
Варіант 38.....	128
Варіант 39.....	128
Варіант 40.....	128
Лабораторна робота № 6.3. Контейнери-списки.....	129
Мета роботи .....	129
Питання, які необхідно вивчити та пояснити на захисті.....	129
Приклад .....	129
Лістинг 3. Послідовний контейнер – список .....	129
Варіанти завдань .....	134
Лабораторна робота № 6.4. Шаблони класів та шаблони функцій .....	135
Мета роботи .....	135
Питання, які необхідно вивчити та пояснити на захисті.....	135
Приклад виконання завдання .....	135
Варіанти завдань .....	139
Варіант 1.....	139
Варіант 2.....	139
Варіант 3.....	139
Варіант 3*.....	139
Варіант 4.....	139
Варіант 5.....	139
Варіант 6.....	140
Варіант 6* .....	140
Варіант 7.....	140
Варіант 7*.....	140

Варіант 8.....	140
Варіант 8*.....	140
Варіант 9.....	140
Варіант 10.....	140
Варіант 10*.....	140
Варіант 11.....	141
Варіант 12.....	141
Варіант 12*.....	141
Варіант 13.....	141
Варіант 13*.....	141
Варіант 14.....	141
Варіант 14*.....	141
Варіант 15.....	141
Варіант 15*.....	141
Варіант 16.....	142
Варіант 17.....	142
Варіант 18.....	142
Варіант 18*.....	142
Варіант 19.....	142
Варіант 20.....	142
Варіант 21.....	142
Варіант 21*.....	142
Варіант 22.....	142
Варіант 22*.....	143
Варіант 23.....	143
Варіант 23*.....	143
Варіант 24.....	143
Варіант 25.....	143
Варіант 25*.....	143
Варіант 26.....	143
Варіант 27.....	143
Варіант 28.....	143
Варіант 28*.....	144
Варіант 29.....	144
Варіант 30.....	144

Варіант 31.....	144
Варіант 31*.....	144
Варіант 32.....	144
Варіант 32*.....	144
Варіант 33.....	144
Варіант 33*.....	144
Варіант 34.....	145
Варіант 35.....	145
Варіант 35*.....	145
Варіант 36.....	145
Варіант 37.....	145
Варіант 37*.....	145
Варіант 38.....	145
Варіант 38*.....	145
Варіант 39.....	145
Варіант 39*.....	146
Варіант 40.....	146
Варіант 40*.....	146
Лабораторна робота № 6.5. Шаблони класів та опрацювання виняткових ситуацій .....	147
Мета роботи .....	147
Питання, які необхідно вивчити та пояснити на захисті.....	147
Варіанти завдань .....	147
Загальні вказівки для всіх варіантів .....	147
Лабораторна робота № 6.6. Шаблони класів .....	150
Мета роботи .....	150
Питання, які необхідно вивчити та пояснити на захисті.....	150
Приклад .....	150
Лістинг 5. Простий масив – шаблон.....	150
Файл «Array.h».....	150
Файл «L_6_6.cpp» .....	151
Результат виконання: .....	152
Варіанти завдань .....	152
Варіант 1.....	152
Варіант 2.....	152
Варіант 3.....	152

Варіант 4.....	152
Варіант 5.....	153
Варіант 6.....	153
Варіант 7.....	153
Варіант 8.....	153
Варіант 9.....	153
Варіант 10.....	153
Варіант 11.....	153
Варіант 12.....	153
Варіант 13.....	154
Варіант 14.....	154
Варіант 15.....	154
Варіант 16.....	154
Варіант 17.....	154
Варіант 18.....	154
Варіант 19.....	154
Варіант 20.....	154
Варіант 21.....	155
Варіант 22.....	155
Варіант 23.....	155
Варіант 24.....	155
Варіант 25.....	155
Варіант 26.....	155
Варіант 27.....	155
Варіант 28.....	155
Варіант 29.....	156
Варіант 30.....	156
Варіант 31.....	156
Варіант 32.....	156
Варіант 33.....	156
Варіант 34.....	156
Варіант 35.....	156
Варіант 36.....	156
Варіант 37.....	156
Варіант 38.....	157

Варіант 39.....	157
Варіант 40.....	157
Бонуси.....	158
Лабораторна робота № 6.7. Шаблони функцій, алгоритми та функтори .....	159
Мета роботи .....	159
Питання, які необхідно вивчити та пояснити на захисті.....	159
Приклад виконання завдання .....	159
Варіанти завдань .....	162
Варіант 1.....	162
Варіант 2.....	162
Варіант 3.....	162
Варіант 4.....	162
Варіант 5.*.....	162
Варіант 6.*.....	162
Варіант 7.*.....	162
Варіант 8.....	163
Варіант 9.....	163
Варіант 10.*.....	163
Варіант 11.*.....	163
Варіант 12.*.....	163
Варіант 13.*.....	163
Варіант 14.*.....	163
Варіант 15.....	163
Варіант 16.....	164
Варіант 17.....	164
Варіант 18.....	164
Варіант 19.....	164
Варіант 20.....	164
Варіант 21.....	164
Варіант 22.....	164
Варіант 23.....	164
Варіант 24.....	165
Варіант 25.....	165
Варіант 26.....	165
Варіант 27.....	165

Варіант 28.....	165
Варіант 29.....	165
Варіант 30.*.....	165
Варіант 31.*.....	165
Варіант 32.*.....	166
Варіант 33.....	166
Варіант 34.....	166
Варіант 35.*.....	166
Варіант 36.*.....	166
Варіант 37.*.....	166
Варіант 38.*.....	166
Варіант 39.*.....	166
Варіант 40.....	166
Питання та завдання для контролю знань .....	168
Контейнери .....	168
Шаблони класів .....	169
Шаблони функцій.....	169
Шаблони.....	170
Предметний покажчик .....	173
Література .....	174

## Перелік запозичених лістингів програм

Назва	Стор.
Лістинг 1. Числовий контейнер фіксованої довжини	35, 115
Лістинг 2. Числовий контейнер змінної довжини	40, 106
Лістинг 3. Послідовний контейнер – список	45, 129
Лістинг 4. Послідовний контейнер – стек	50
Лістинг 5. Простий масив – шаблон	56, 150
Лістинг 6. Шаблон класу із статичними елементами	64
Лістинг 7. Метод-шаблон в шаблоні класу	68
Лістинг 8. Обчислення кількості об'єктів класу	72

# Вступ

Дисципліна «Об'єктно-орієнтоване програмування» вивчається після курсу «Алгоритмізація та програмування», цією дисципліною продовжується цикл предметів, що стосуються програмування та розробки програмного забезпечення.

В посібнику містяться теоретичні відомості, приклади, методичні вказівки з їх розв'язування, варіанти лабораторних завдань та питання і завдання з контролю знань з теми «Контейнери та шаблони».



## Тема 6. Контейнери та шаблони

### *Стисло та головне про контейнери та шаблони*

#### Контейнери

В кожній мові програмування є засоби для об'єднання однотипних даних в групи, зазвичай це – масиви. В мові C / C++ масиви – занадто прості, а тому дуже ненадійні конструкції. Тому масивів як засобів об'єднання однотипних даних явно не достатньо. В процесі розвитку мов програмування розроблена більш загальна конструкція для об'єднання однорідних даних в групу – *контейнер*, в деяких мовах (C#, java) вона називається *колекція*.

#### Характеристики контейнерів

*Контейнер* – це набір однотипних елементів. Множина, масив, каталог файлів на диску – це приклади контейнерів.

Кожний контейнер характеризується *іменем* та *типом елементів*, які входять до нього. Ім'я контейнера – це ім'я змінної, яке підпорядковується правилам стосовно імен змінних в мові C++. Як об'єкт, змінна-контейнер характеризується *областю видимості* та *часом існування* в залежності від місця та часу створення, причому час існування контейнера в загальному випадку не залежить від часу існування його елементів.

*Тип контейнера* складається із *виду контейнера* та *типу елементів*, які входять до нього. *Вид контейнера* визначається способом *доступу* до його елементів. Тип елементів може бути *вбудованим* або *реалізованим користувачем*, елементами контейнера в тому числі можуть бути інші контейнери.

*Розмір контейнера* може бути або визначеним при його оголошенні (тоді такий контейнер називається *контейнером фіксованої довжини*), або наперед не визначеним. В загальному випадку кількість елементів контейнера із заданою фіксованою довжиною може змінюватися від нуля до оголошеної максимальної кількості. Наприклад, літерні рядки `string` в мові Pascal можуть містити до 255 символів, функція `length(s)` повертає фактичну довжину (кількість символів) літерного рядка `s`, при цьому його загальний розмір – 256 байт (255 – максимальна довжина + 1 байт – нульовий байт містить службову інформацію: фактичну довжину). Тобто, розмір контейнера (255 елементів) та кількість елементів в ньому (яка визначається функцією `length()`) – різні речі.

Якщо ж розмір елементів не задано при його оголошенні, то кількість елементів змінюється під час виконання програми: елементи додаються в контейнер та вилучаються з нього. Такий контейнер називається *контейнером змінної довжини*.

## Доступ до елементів контейнера

Однією із найважливіших характеристик контейнера є спосіб *доступу* до його елементів. Зазвичай розрізняють *прямий*, *послідовний* та *асоціативний* доступ.

*Прямий доступ* до елемента – це доступ за номером (або ще кажуть, за індексом) елемента. Саме так організовано доступ до елементів масиву:

$$v[i]$$

Цей вираз означає, що ми хочемо оперувати елементом контейнера  $v$ , який має номер (індекс)  $i$ . Нумерація може починатися з будь-якого значення, проте загально прийнято починати нумерацію з нуля (як для вбудованих масивів).

*Послідовний доступ* характеризується тим, що немає ніяких «індексів» елементів, можна лише переміщуватися послідовно від одного елемента до іншого. Можна вважати, що існує певний «вказівник-індикатор», який переміщують по елементах контейнера за допомогою певних операцій. Той елемент, на який налаштований «вказівник-індикатор», називається поточним.

Зазвичай набір операцій для послідовного доступу містить такі операції:

- перехід до першого елемента;
- перехід до останнього елемента;
- перехід до наступного елемента;
- перехід до попереднього елемента;
- перехід на  $n$  елементів вперед (в напрямку від початку до кінця контейнера);
- перехід на  $n$  елементів назад (в напрямку від кінця до початку контейнера);
- отримання (зміна) значення поточного елемента.

*Асоціативний доступ* дещо подібний до прямого, проте оснований не на номерах (індексах) елементів, а на вмісті елементів контейнера – тобто, їх значеннях.

Поле, з вмістом якого *асоціюється* елемент контейнера, називається *ключем*. Елемент контейнера, який відповідає певному значенню ключа, так і називається: *значення*. Таким чином, асоціативний контейнер складається із набору пар «ключ-значення». Зазвичай,

асоціативний контейнер впорядкований за ключем. В наведеному прикладі ключем є поле Name.

Методи доступу до елементів контейнера – настільки важлива його характеристика, що в стандартній бібліотеці (STL – Standard Template Library, англ. – Стандартна бібліотека шаблонів) розрізняють контейнери *послідовні* та *асоціативні*. Послідовними контейнерами, які забезпечують і прямий і послідовний способи доступу, є контейнери **vector** та **deque**, асоціативним контейнером є контейнер **map**. Контейнер, в якому доступ лише послідовний – це контейнер **list**.

## Поняття ітератора

*Ітератор* – це об’єкт, який забезпечує послідовний доступ до елементів контейнера. Так само, як контейнер – це узагальнення поняття масив, так і ітератор – це узагальнення поняття вказівник.

## Операції контейнера

Всі операції з контейнером можна поділити на наступні категорії:

- Операції з окремими елементами чи з групами елементів контейнера:
  - операції доступу до елементів, в т.ч. операцію зміни значення елемента;
  - операції додавання та видалення окремих елементів чи груп елементів;
  - операції пошуку елементів чи груп елементів.
- Операції з контейнером як з цілим об’єктом, зокрема:
  - операції об’єднання контейнерів.
- Спеціальні операції, які залежать від виду контейнера.

## Підсумок

*Контейнер* – більш загальна конструкція для об’єднання однорідних елементів у групу, ніж масив.

Однією із найважливіших характеристик контейнера є *спосіб доступу до елементів*: послідовний, прямий чи асоціативний.

*Прямий* та *асоціативний* варіанти доступу зазвичай реалізують за допомогою перевантаження операції індексування **operator[]**, а *послідовний* – за допомогою реалізації класу-ітератора. Використання ітератора для організації доступу має ще ту перевагу, що інтерфейс доступу відокремлюється від інтерфейсу контейнера. Оскільки ітератор тісно

пов'язаний з внутрішньою структурою контейнера, його зазвичай створюють як вкладений клас.

Для контейнерів реалізують багато операцій, одна з найчастіше використовуваних – операція об'єднання контейнерів, вона реалізується різними способами.

Динамічні структури даних, такі як стек, черга, дек (двонапрямлена черга) – універсальні структури, робота з якими не залежить від типу елементів. З точки зору користувача, ці структури відрізняються лише способами додавання та вилучення елементів. Зазвичай ці структури реалізуються за допомогою зв'язаних списків. Проте написати універсальний клас-контейнер без використання механізму успадковування або шаблонів дуже складно – як правило, приходиться використовувати не типізовані вказівники, які потенційно небезпечні.

## Шаблони класів

Такі контейнери, як стеки та черги, характеризуються способом доступу до елементів і фактично не залежать від типу елементів. Аналогічно, контейнер-масив надає незалежний від типу доступ до елементів за допомогою операції індексування. Створити універсальний контейнер, не залежний від типу елементів, можна двома способами:

- використовуючи в якості елементів контейнера вказівник `void*`;
- на основі шаблону класу, в якому тип елементів визначається параметром шаблону.

Перший спосіб – успадкований від C, оскільки в цій мові не було інших засобів. В C++ зазвичай використовують інший спосіб – *шаблони*.

## Створення шаблонів класів

*Шаблон класу* – це заготовка, із якої створюються конкретні класи в процесі *інстанціювання* (тобто, створення сутності). Воно виконується шляхом підстановки конкретного типу в якості аргументу. Шаблон класу ще називають *параметризованим типом*. Терміни «параметризований тип», «шаблон класу», «шаблон-клас», «шаблон» – еквівалентні.

Синтаксис шаблону класу має наступний вигляд:

```
template <параметри> // визначення шаблону
class ім'я_класу      // визначення класу-шаблону
{
    ...               // елементи шаблону класу
};
```

Префікс `template <параметри>` показує компілятору, що наступне визначення класу є шаблоном, а не звичайним класом. Шаблоном може бути і структура. Як і ім'я звичайного

класу, ім'я класу-шаблону не має збігатися з іншими іменами в одній і тій самій області видимості.

## Параметри шаблону

Параметри шаблону можуть бути наступних видів:

- параметр-тип;
- параметр цілочисельного або перелічуваного типу;
- параметр-вказівник на об'єкт або вказівник на функцію;
- параметр-посилання на об'єкт або посилання на функцію;
- параметр-вказівник на елемент класу (поле чи метод).

## Використання шаблонів класів

Визначення об'єктів для деякого класу-шаблону в загальному випадку виглядає так:

```
ім'я_шаблону_класу <аргументи> ім'я_об'єкта           // одиночний об'єкт
ім'я_шаблону_класу <аргументи> ім'я_об'єкта[кількість] // масив
ім'я_шаблону_класу <аргументи> *ім'я_об'єкта          // вказівник
```

Або для випадку константного посилання в якості параметру:

```
const ім'я_шаблону_класу <аргументи> &ім'я_об'єкта    // константне посилання
```

В кутових дужках на місці параметрів при оголошенні об'єктів вказують конкретні аргументи. Параметри шаблону – позиційні (як і аргументи функції), тому фактичне значення, що підставляється, має відповідати виду параметра шаблону: якщо на певному місці визначення шаблону класу вказано параметр-тип, то і аргумент має бути типом чи класом.

## Спеціалізація шаблонів класів

Часто буває потрібно одночасно разом з загальним шаблоном використовувати деяку спеціалізовану його версію. Для цього в шаблонах реалізовано механізм *спеціалізації*. Спеціалізація полягає в тому, що на основі початкового *первинного* шаблону реалізується його спеціалізована версія для деяких конкретних значень параметрів. Спеціалізація шаблону називається *повною*, якщо конкретизовані всі параметри первинного шаблону. Якщо конкретизовано лише частину параметрів, то спеціалізація називається *частковою*.

Спеціалізація шаблону – це не присвоєння параметрам значення за умовчанням. Шаблон з параметрами за умовчанням – це первинний шаблон, який також можна спеціалізувати.

Спеціалізація шаблону – це «перевантаження» для класів.

Спеціалізація шаблону – це не інстанціювання. Інстанціювання – це створення екземпляру. Інстанціювання виконує компілятор при трансляції програми, коли зустрічає оголошення об'єкту з конкретними значеннями параметрів шаблону.

Спеціалізацію первинного шаблону реалізує програміст як окремий шаблон класу, в якому деякі (або всі) параметри первинного шаблону мають конкретні значення.

Спеціалізація найчастіше застосовується для параметрів-типів. Первинний шаблон визначає загальний варіант, а спеціалізована версія – частковий випадок для конкретних типів. Повна спеціалізація – це конкретний клас, реалізований для конкретних значень параметрів первинного шаблону.

Зазвичай в спеціалізованих версіях перевизначаються окремі (або навіть усі) методи, які для певного конкретного типу мають працювати не так, як в загальному випадку. Наприклад, нехай в первинному шаблоні визначено операцію присвоєння `operator=`. Для символьних масивів ця операція має працювати зовсім не так, як для інших типів. Тоді потрібно визначити спеціалізовану версію первинного шаблону для `const char[]` і перевизначити в ній операцію присвоєння.

При визначенні шаблону та його спеціалізованих версій слід дотримуватися порядку слідування: спочатку потрібно визначити (або оголосити) первинний шаблон, а лише потім можна визначати спеціалізовані версії.

## **Підсумок**

В C++ включено два види шаблонів: шаблони функцій та шаблони класів. Основне призначення шаблонів класів – реалізація узагальнених контейнерів, які не залежать від типу елементів.

*Шаблонний клас* – це клас, утворений в результаті *інстанціювання* шаблону класу.

Шаблони дають можливість визначити за допомогою одного фрагменту коду цілий набір взаємопов'язаних класів, які називаються *шаблонними класами*.

Шаблон класу являє собою деякий опис родового класу, на основі якого створюються специфічні версії для конкретних типів даних.

Шаблони класів часто називають параметризованими типами, оскільки вони мають один або кілька параметрів типу, які визначають налаштування родового шаблону класу на специфічний тип даних при створенні об'єкта класу.

Для того, щоб використовувати шаблонні класи, програмісту достатньо один раз описати шаблон класу. Кожного разу, коли потрібна реалізація класу для нового типу даних, програміст, використовуючи простий короткий запис, повідомляє про це компілятору, який і створює текст (початковий код) необхідного шаблонного класу.

Основний вид параметрів шаблону – ім'я типу. Проте шаблони класів можуть мати і параметри інших видів: в шаблонах є можливість використовувати так звані *нетипові параметри*. Одним із найбільш потужних засобів програмування є параметр-шаблон. Крім того, параметри шаблону класу можна задавати за умовчанням.

Опис шаблону класу виглядає як традиційне визначення класу, перед яким записують заголовок `template <class T>` або `template <typename T>` (ідентифікатор `T` вибрано в якості прикладу – можна вибирати інші імена). Цей заголовок вказує на те, що такий опис є шаблоном класу з параметром типу `T`, який позначає тип класів, що будуть створюватися на основі цього шаблону. Ідентифікатор `T` визначає тип даних, який може використовуватися як тип полів класу, в заголовку класу та в методах класу.

Кожне визначення методу поза шаблоном класу починається із заголовка `template <class T>` або `template <typename T>`, за яким записують визначення методу, подібне до стандартного визначення, але в якості типу елемента класу завжди вказують параметр типу `T`. Щоб пов'язати кожне визначення методу з областю дії шаблону класу, використовується бінарна операція доступу до області дії `::` з іменем шаблону класу `ім'яКласу< T >`.

Процес підстановки аргументів на місце параметрів шаблону називається *інстанціюванням* шаблону. Інстанціювання дозволяє створити із однієї «заготовки» ціле сімейство конкретних реалізацій.

На основі первинного шаблону можна отримати спеціалізовані версії, в яких задані частина або навіть всі параметри. Останній варіант називається повною спеціалізацією.

Класи-шаблони можуть бути вкладеними, можуть бути друзями, можуть успадковувати від шаблону або від класу.

Шаблон класу може бути похідним від шаблонного класу. Шаблон класу може бути похідним від нешаблонного класу. Шаблонний клас може бути похідним від шаблону класу. Нешаблонний клас може бути похідним від шаблону класу.

Локальні класи не можуть містити шаблони в якості своїх елементів.

Шаблони методів не можуть бути віртуальними.

Шаблони класів можуть містити статичні елементи, дружні функції та класи.

Всередині шаблону не можна визначати `friend`-шаблони.

Зазвичай визначають повну спеціалізацію шаблону для вказівників та використовують її в якості базового класу для частково спеціалізованого шаблону за вказівниками. Успадковування класу від повної спеціалізації шаблону дозволяє організувати лічильник об'єктів для конкретного класу, а не для всієї ієрархії успадковування.

Для спеціалізованого типу можна визначити клас, який скасовує дію шаблону класу для цього типу.

Функції та цілі класи можуть бути оголошені *дружніми* для нешаблонних класів. Для шаблонів класів також можна встановити відношення дружності. Дружність можна встановити між шаблоном класу та глобальною функцією, методом іншого класу (можливо, шаблонного класу), а також цілим класом (можливо, шаблонним класом).

Дружні функції можуть бути визначені як зовнішні функції-шаблони або безпосередньо всередині класу. В останньому випадку дружня функція є звичайною функцією, і її параметри мають залежати від параметрів шаблону, бо інакше при не однократному інстанціюванні виникне помилка повторного визначення.

Кожний шаблонний клас, отриманий на основі шаблону класу, має власний екземпляр кожного статичного поля шаблону; всі об'єкти цього шаблонного класу використовують цей свій власний екземпляр статичного поля. Як і статичні поля нешаблонних класів, статичні поля шаблонних класів мають бути ініціалізовані в області дії файлу.

Кожний шаблонний клас отримує власний екземпляр статичного методу шаблону класу.

## Шаблони функцій

Крім шаблонів класів, C++ дозволяє створювати та використовувати шаблони функцій.

### Створення шаблону функції

Синтаксис шаблону функції наступний:

```
template< параметри >  
заголовок  
{ тіло }
```

Параметри шаблону функції можуть бути такими ж самими, як і параметри шаблону класу: в шаблонах функцій можна використовувати нетипові параметри (тобто, параметри, які не є типами); параметр-тип можна вказувати в якості типу аргументів, так і в якості типу результату функції.

### Параметри шаблону за умовчанням

В шаблонах функцій не можна вказувати параметри за умовчанням.

Проте це обмеження легко обійти, якщо використати *клас-обгортку*.



## **Перевантаження шаблонів функцій**

Допускається перевантаження шаблонів функцій шаблонами та звичайними функціями. Як і при перевантаженні звичайних функцій, шаблони (і функції) мають відрізнятися списками параметрів. Компілятор при опрацюванні конкретного виклику намагається підібрати той варіант, який найбільше підходить.

## **Спеціалізація шаблонів функцій**

Для шаблонів функцій реалізовано повну спеціалізацію. Часткова спеціалізація для функцій-шаблонів не допускається.

## **Узагальнені алгоритми та функтори**

*Узагальнений алгоритм* – це шаблон функції, який може виконувати задану операцію з послідовним контейнером будь-якого виду. Зазвичай параметрами узагальненого алгоритму є ітератори вхідного (та вихідного) контейнеру.

## **Поняття функтора**

Одним із аргументом узагальненого алгоритму часто є деяка операція, яку необхідно виконати з кожним елементом контейнера. В якості параметра-операції можна задавати вказівник на функцію. Проте в узагальненому алгоритмі аргумент-операція має задаватися універсальним способом, тому параметр не може бути заданий у вигляді вказівника, бо вказівники на методи відрізняються від вказівників на функції. Аргументи-операції в узагальнених алгоритмах зазвичай задаються як об'єкти-функтори.

Для забезпечення універсальності узагальненого алгоритму його параметр-операцію задають за допомогою *функтору*. *Функтор* – це об'єкт, що веде себе наче функція. Клас, в якому перевантажена операція `operator()` виклику функції, називається класом-функтором. Об'єкт такого класу називають *функтором*.

Операцію `operator()` можна визначати лише як метод класу. Операція `operator()` не може бути статичним методом. Операція `operator()` може бути віртуальним методом.

## **Підсумок**

В мові C++ крім *шаблонів класів* реалізовані *шаблони функцій*, – це подальший розвиток механізму перевантаження функцій. Порівняно з шаблонами класів, шаблони функцій мають багато обмежень. Наприклад, не можна задавати параметри шаблону функції за умовчанням, проте працює автоматичне виведення параметрів (хоч і з деякими обмеженнями). Шаблон функції можна спеціалізувати, проте часткова спеціалізація не

допускається. Шаблон функції можна перевантажити як іншим шаблоном, так і функцією. Всі такі обмеження можна «обійти», якщо «обгорнути» функцію в клас-шаблон.

*Шаблонна функція* – це функція, утворена в результаті *інстанціювання* шаблону функції.

Шаблони дають можливість визначити за допомогою одного фрагменту коду цілий набір взаємопов'язаних функцій (перевантажених), які називаються *шаблонними функціями*.

Щоб використати шаблони функцій, програміст має написати лише один опис шаблону функції. На основі типів аргументів, які використовуються при виклику цієї функції, компілятор буде автоматично генерувати об'єктні коди функцій, які опрацьовують кожний тип даних. Вони компілюються разом з іншими частинами тексту програми.

Всі описи шаблонів функцій починаються з ключового слова `template`, за яким йде список формальних параметрів шаблону, записаний в кутових дужках (< та >); кожному формальному параметру шаблону має передувати ключове слово `class` або `typename`. Ключове слово `class` або `typename`, яке використовується при визначенні типів параметрів шаблону функції, означає «будь-який тип, або тип, що визначається користувачем».

Формальні параметри в описі шаблону використовуються для визначення типів параметрів функції, типу результату функції та типів локальних змінних, визначених в тілі функції.

Сам шаблон функції можна перевантажити кількома способами. Можна визначити інші шаблони, що матимуть те саме ім'я функції, але з різними наборами параметрів. Шаблон функції також можна перевантажити, якщо ввести іншу нешаблонну функцію з тим же самим іменем, але іншим набором параметрів.

Шаблони функцій дозволяють реалізувати *узагальнені алгоритми*, які можуть працювати з *контейнерами* будь-якого виду. В якості параметрів в узагальненому алгоритмі виступають *ітератори*. Одним із аргументом узагальненого алгоритму часто є деяка операція, яку необхідно виконати з кожним елементом контейнера. Зазвичай в якості параметра-операції задається вказівник на функцію. Проте в узагальненому алгоритмі аргумент-операція має задаватися універсальним способом, тому параметр не може бути заданий у вигляді вказівника, бо вказівники на методи відрізняються від вказівників на функції. Аргументи-операції в узагальнених алгоритмах зазвичай задаються як об'єкти-функтори. *Функтор* – це клас, в якому перевантажена операція виклику функції `operator()`. Використання функторів дозволяє зробити алгоритм насправді універсальним. Класи-оболонки для вказівників на функції та вказівників на методи, які називають *адаптерами*,

збільшують ступінь універсальності узагальненого алгоритму. Один із важливих адаптерів – *фіксатор*, який дозволяє перетворити бінарний функтор в унарний.

## Теоретичні відомості

В цьому розділі розглядаються дві важливі теми: контейнери та шаблони, необхідні для розуміння принципів організації STL – Standard Template Library (стандартної бібліотеки шаблонів).

### Контейнери

В кожній мові програмування є засоби для об'єднання однотипних даних в групи, зазвичай це – масиви. В мові C / C++ масиви – занадто прості, а тому дуже ненадійні конструкції. Тому масивів як засобів об'єднання однотипних даних явно не достатньо. В процесі розвитку мов програмування розроблена більш загальна конструкція для об'єднання однорідних даних в групу – *контейнер*, в деяких мовах (C#, java) вона називається *колекція*.

### Характеристики контейнерів

*Контейнер* – це набір однотипних елементів. Множина, масив, каталог файлів на диску – це приклади контейнерів.

Кожний контейнер характеризується *іменем* та *типом елементів*, які входять до нього. Ім'я контейнера – це ім'я змінної, яке підпорядковується правилам стосовно імен змінних в мові C++. Як об'єкт, змінна-контейнер характеризується *областю видимості* та *часом існування* в залежності від місця та часу створення, причому час існування контейнера в загальному випадку не залежить від часу існування його елементів.

*Тип контейнера* складається із *виду контейнера* та *типу елементів*, які входять до нього. *Вид контейнера* визначається способом *доступу* до його елементів. Тип елементів може бути *вбудованим* або *реалізованим користувачем*, елементами контейнера в тому числі можуть бути інші контейнери.

*Розмір контейнера* може бути або визначеним при його оголошенні (тоді такий контейнер називається *контейнером фіксованої довжини*), або наперед не визначеним. В загальному випадку кількість елементів контейнера із заданою фіксованою довжиною може змінюватися від нуля до оголошеної максимальної кількості. Наприклад, літерні рядки `string` в мові Pascal можуть містити до 255 символів, функція `length(s)` повертає фактичну довжину (кількість символів) літерного рядка `s`, при цьому його загальний розмір – 256 байт (255 – максимальна довжина + 1 байт – нульовий байт містить службову інформацію: фактичну довжину). Тобто, розмір контейнера (255 елементів) та кількість елементів в ньому (яка визначається функцією `length()`) – різні речі.

Якщо ж розмір елементів не задано при його оголошенні, то кількість елементів змінюється під час виконання програми: елементи додаються в контейнер та вилучаються з нього. Такий контейнер називається *контейнером змінної довжини*.

## Доступ до елементів контейнера

Однією із найважливіших характеристик контейнера є спосіб *доступу* до його елементів. Зазвичай розрізняють *прямий*, *послідовний* та *асоціативний* доступ.

*Прямий доступ* до елемента – це доступ за номером (або ще кажуть, за індексом) елемента. Саме так організовано доступ до елементів масиву:

$$v[i]$$

Цей вираз означає, що ми хочемо оперувати елементом контейнера  $v$ , який має номер (індекс)  $i$ . Нумерація може починатися з будь-якого значення, проте загально прийнято починати нумерацію з нуля (як для вбудованих масивів).

*Послідовний доступ* характеризується тим, що немає ніяких «індексів» елементів, можна лише переміщуватися послідовно від одного елемента до іншого. Можна вважати, що існує певний «вказівник-індикатор», який переміщують по елементах контейнера за допомогою певних операцій. Той елемент, на який налаштований «вказівник-індикатор», називається поточним.

Зазвичай набір операцій для послідовного доступу містить такі операції:

- перехід до першого елемента;
- перехід до останнього елемента;
- перехід до наступного елемента;
- перехід до попереднього елемента;
- перехід на  $n$  елементів вперед (в напрямку від початку до кінця контейнера);
- перехід на  $n$  елементів назад (в напрямку від кінця до початку контейнера);
- отримання (зміна) значення поточного елемента.

*Асоціативний доступ* дещо подібний до прямого, проте оснований не на номерах (індексах) елементів, а на вмісті елементів контейнера – тобто, їх значеннях.

Припустимо, деякий асоціативний контейнер банківської системи містить записи про банківські рахунки клієнтів. Кожний запис обов'язково має містити поле, в якому зберігається прізвище клієнта, наприклад:

```

class Account
{
    string Name;    // прізвище
    unsigned Id;    // номер рахунку
    double Summa;   // сума на рахунку

public:
    ...
};

```

Якщо асоціативний контейнер `v` містить об'єкти класу `Account`, то наступний вираз

```
v["Іваненко"]
```

є рахунком, відкритим для клієнта з прізвищем Іваненко.

Такий вираз в мові C++ цілком коректний, оскільки операцію індексування можна перевантажити для аргументу будь-якого типу.

Поле, з вмістом якого *асоціюється* елемент контейнера, називається *ключем*. Елемент контейнера, який відповідає певному значенню ключа, так і називається: *значення*. Таким чином, асоціативний контейнер складається із набору пар «ключ-значення». Зазвичай, асоціативний контейнер впорядкований за ключем. В наведеному прикладі ключем є поле `Name`.

Методи доступу до елементів контейнера – настільки важлива його характеристика, що в стандартній бібліотеці (STL – Standard Template Library, англ. – Стандартна бібліотека шаблонів) розрізняють контейнери *послідовні* та *асоціативні*. Послідовними контейнерами, які забезпечують і прямий і послідовний способи доступу, є контейнери `vector` та `deque`, асоціативним контейнером є контейнер `map`. Контейнер, в якому доступ лише послідовний – це контейнер `list`.

## Поняття ітератора

*Ітератор* – це об'єкт, який забезпечує послідовний доступ до елементів контейнера. Так само, як контейнер – це узагальнення поняття масив, так і ітератор – це узагальнення поняття вказівник.

*Ітератор* описують як один із *патернів* програмування – `Iterator`.

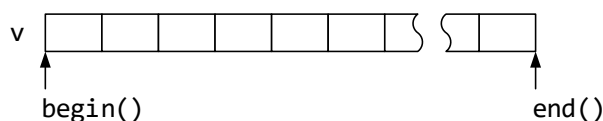
Якщо `v` – це контейнер послідовного доступу та `iv` – об'єкт-ітератор, пов'язаний з цим контейнером, то вище вказані операції з контейнером можна описати наступним чином:

```

iv = v.begin(); // перейти до першого елемента
iv = v.last();  // перейти до останнього елемента
++iv;          // перейти до наступного елемента
--iv;          // перейти до попереднього елемента
iv += n;        // перейти на n елементів вперед
iv -= n;        // перейти на n елементів назад
*iv             // доступ до поточного елемента

```

Зазвичай реалізують методи `begin()` та `end()`, які повертають значення ітератора для класу-контейнера. Якщо `v` – це контейнер послідовного доступу, то метод `begin()` повертає ітератор на перший елемент контейнера, а метод `end()` – повертає ітератор позиції після останнього елемента:



## Операції контейнера

Всі операції з контейнером можна поділити на наступні категорії:

- Операції з окремими елементами чи з групами елементів контейнера:
  - операції доступу до елементів, в т.ч. операцію зміни значення елемента;
  - операції додавання та видалення окремих елементів чи груп елементів;
  - операції пошуку елементів чи груп елементів.
- Операції з контейнером як з цілим об'єктом, зокрема:
  - операції об'єднання контейнерів.
- Спеціальні операції, які залежать від виду контейнера.

## Доступ до елементів; зміна значень

Операції доступу до елементів, в т.ч. операцію зміни значення елемента контейнера – розглядали в попередніх параграфах.

## Додавання та видалення елементів та груп елементів

Операції додавання та видалення елементів визначені лише для контейнерів змінної довжини. Очевидно, що такі операції можна виконувати різними способами:

- додавати та вилучати елементи на «початку» контейнера;
- додавати та вилучати елементи в «кінці» контейнера;
- комбінація перших двох способів: додавати елементи в «кінець (хвіст)» контейнера, вилучати з «початку (голови)»;
- вставляти елементи перед поточним елементом або після нього, вилучати поточний елемент;
- вставляти елементи відповідно до деякого порядку сортування елементів контейнера; в цьому випадку обов'язково відбувається операція пошуку;
- вилучати елементи, вміст яких відповідає заданому критерію; в цьому випадку теж обов'язково відбувається операція пошуку.

Перші чотири операції зазвичай застосовують до послідовних контейнерів. Спосіб вставки та видалення визначає вид послідовного контейнера.

Якщо вставка і видалення виконуються лише на одному кінці контейнера (на його «вершині»), то такий контейнер називають *стеком* (англ. *stack*), а принцип опрацювання елементів називається LIFO (Last In First Out – останнім зайшов, першим вийшов). Поточний елемент міститься на вершині стеку (єдина точка доступу).

Якщо елементи добавляються на одному кінці (в «хвіст»), а вилучаються з іншого (з «голови»), то контейнер називається *чергою* (англ. *queue*). Черга опрацьовується за принципом FIFO (First In First Out – першим зайшов, першим вийшов).

Якщо вставка і видалення виконується на обох кінцях контейнера, – такий контейнер називається *деком* (англ. *deque* – аббревіатура від «double ended queue», тобто «черга з двома кінцями»).

## Пошук елементів та груп елементів

Операції пошуку можна поділити на

- Пошук елемента (для впорядкованих контейнерів):
  - пошук заданого елемента;
  - пошук першого входження;
  - пошук останнього входження.
- Пошук групи елементів:
  - пошук входження всіх елементів групи;
  - пошук входження хоч якогось елемента групи;
  - пошук входження одного контейнера в інший (для впорядкованих контейнерів – на зразок пошуку входження підрядка у рядок).

## Об'єднання контейнерів

Найбільш часто виконується операція об'єднання двох контейнерів, яка повертає результат – новий контейнер. Ця операція може бути реалізована в наступних варіантах:

- звичайне *зчеплення* двох контейнерів: в новий контейнер потрапляють всі елементи і першого і другого контейнерів; операція не комутативна;
- об'єднання впорядкованих контейнерів, яке називають *злиттям*: в новий контейнер потрапляють всі елементи першого і другого контейнерів; об'єднаний контейнер впорядкований, операція комутативна;
- об'єднання двох контейнерів як *об'єднання множин*: в контейнер-результат потрапляють ті елементи, які є хоча б в одному контейнері; операція комутативна;



- об'єднання двох контейнерів як *перетин множин*: в контейнер-результат потрапляють лише ті елементи, які є хоча б в обох контейнерах; операція комутативна.

Окремо слід виділити контейнери-множини. *Множина* – це контейнер, в якому в якому кожний елемент унікальний. Для контейнерів-множин часто реалізується операція *різниці множин*: в контейнер-результат потрапляють лише ті елементи першого контейнера, яких немає у другому; операція не комутативна. Також для множин визначена операція перевірки входження, яка фактично є операцією пошуку (як окремого елементу, так і підмножини елементів).

## Спеціальні операції

Часто виконується операція отримання частини елементів деякого контейнера та формування на їх основі нового контейнера. Зазвичай цю операцію виконує конструктор, а частина контейнера визначається двома ітераторами.

Багато операцій залежать від типу елементів контейнера. Числові контейнери (динамічні масиви) як правило, мають забезпечувати арифметичні операції, пошук максимального та мінімального елементів, обчислення сум та добутків, впорядкування. Для літерних рядків операція впорядкування зазвичай не потрібна, а для числових контейнерів чи для переліку банківських рахунків – вона необхідна.

## Реалізація контейнерів

Контейнери зазвичай реалізуються за допомогою вказівників та динамічної пам'яті.

При реалізації контейнерів слід насамперед дати відповідь на важливі питання:

- 1) Як виділити пам'ять?
- 2) Як звільнити пам'ять?
- 3) Як реалізувати копіювання та присвоєння?

### 1. Виділення пам'яті

Є два способи виділення пам'яті.

За допомогою операції `new[]` виділяється неперервна область пам'яті. Кількість елементів зазвичай вказується виразом, який обчислюється під час роботи програми. Ця форма використовується для реалізації динамічних масивів. В цьому випадку клас обов'язково має містити поле-вказівник, яке отримує адресу динамічного масиву. Також визначається поле для зберігання розміру виділеної пам'яті. Якщо кількість елементів – фіксована, то це поле визначає кількість елементів контейнера-масиву. Якщо ж кількість

елементів може змінюватися під час виконання програми, то клас містить ще і поле для зберігання поточної кількості елементів.

Другий спосіб – виділення пам'яті для одиночного елемента операцією `new`. Такий спосіб потрібний для реалізації контейнерів із змінною кількістю елементів. В цьому випадку не лише пам'ять для елемента виділяється динамічно, а і до складу самого елемента входять один або два вказівники для зв'язку елементів один з одним. Зазвичай такі контейнери або послідовні, або асоціативні.

Виділення пам'яті виконує конструктор контейнера.

## 2. Звільнення пам'яті

Звільнення пам'яті зазвичай виконує деструктор. Операції звільнення пам'яті – «парні» до операцій виділення пам'яті. Якщо пам'ять виділялася операцією `new`, то звільнити її слід операцією `delete`. Якщо ж пам'ять виділялася масивом операцією `new[]`, то і звільнити її потрібно операцією `delete[]`.

## 3. Копіювання та присвоєння

Для динамічних класів, які використовують вказівники, операції копіювання та присвоєння, що надаються системою за умовчанням, зовсім не підходять.

Конструктор копіювання, який створюється за умовчанням, виконує по-елементне копіювання полів класу. Це приводить до помилок виду «підвішені вказівники» при знищенні об'єкта.

Стандартна операція присвоєння виконує по-елементне копіювання полів правого операнду в поля лівого (поточний об'єкт). Для динамічних контейнерів це приводить як до «підвішених», так і до «втрачених» вказівників («втрачена» пам'ять).

Потрібна явна реалізація цих операцій.

Конструктор копіювання має виділити пам'ять для нового контейнера такого ж розміру та скопіювати в новий контейнер елементи контейнера-ініціалізатора. Операція присвоєння має створити новий динамічний контейнер, скопіювати туди елементи вхідного контейнера і знищити попередній контейнер.

Для всіх методів, які виділяють пам'ять, зазвичай вказується специфікація винятку `throw(bad_alloc)`.

Крім того, часто в контейнері визначаються і власні типи винятків, характерні для конкретної реалізації. Наприклад, спроба вилучити елемент із порожнього стеку має генерувати виняток типу «порожній стек».

## Числовий контейнер фіксованої довжини

### Лістинг 1. Числовий контейнер фіксованої довжини

[9 – с. 84-87]

#### файл «Array.h»

```
#pragma once
#include <iostream>

using namespace std;

class Array
{
public:
    // типи
    typedef unsigned int UINT;
    typedef int* iterator;
    typedef const int* const_iterator;
    typedef int& reference;
    typedef const int& const_reference;
    typedef int value_type;
    typedef size_t size_type;
    typedef value_type* TArray;

private:
    static const size_type minsize = 10;
    UINT size_array;
    value_type* data;
    UINT Count;
    size_type First;

public:
    // конструктори
    Array(const size_type n = minsize);
    Array(const Array& throw(bad_alloc);
    Array(const iterator first, const iterator last)
        throw(bad_alloc, invalid_argument);
    Array(const size_type first, const size_type last)
        throw(bad_alloc, invalid_argument);
    ~Array();
    Array& operator =(const Array& rhs);

    // операції індексування
    reference operator[](UINT);
    const_reference operator[](UINT) const;

    // дружні функції вводу/виводу
    friend ostream& operator <<(ostream& out, const Array& a);
    friend istream& operator >>(istream& in, Array& a);

    // розмір
    UINT size() const;

    // ітератори
    iterator begin() { return data; }
    const_iterator begin() const { return data; }
    iterator end() { return data + Count; }
    const_iterator end() const { return data + Count; }
```

```

// Лабораторна робота 6.2
double Sum();
double Perymetr();
};

```

### файл «Array.cpp»

```

#include "Array.h"
#include <iostream>
#include <stdexcept>
#include <exception>
#include <math.h>
#include <time.h>

using namespace std;

Array::Array(const Array::size_type n)
{
    First = 0;
    Count = size_array = n;
    data = new value_type[size_array];
    for (UINT i = 0; i < size_array; i++)
        data[i] = 0;
}

Array::Array(const iterator first, const iterator last)
    throw(bad_alloc, invalid_argument)
{
    First = 0;
    if (first <= last)
    {
        Count = size_array = (last - first) + 1;
        data = new value_type[size_array];
        for (UINT i = 0; i < size_array; ++i)
            data[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const size_type first, const size_type last)
    throw(bad_alloc, invalid_argument)
{
    if (first <= last)
    {
        First = first;
        Count = size_array = (last - first) + 1;
        data = new value_type[size_array];
        for (UINT i = 0; i < size_array; ++i)
            data[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const Array& t) throw(bad_alloc)
    : size_array(t.size_array), Count(t.Count), First(t.First),
    data(new value_type[size_array])
{
    for (UINT i = 0; i < size_array; ++i)
        data[i] = t.data[i];
}

```

```

Array::~~Array()
{
    delete[] data;
    data = 0;
}

Array& Array::operator =(const Array& rhs)
{
    if (this != &rhs)
    {
        size_array = rhs.size_array;
        Count = rhs.Count;
        First = rhs.First;
        value_type* new_data = new value_type[size_array];

        for (UINT i = 0; i < size_array; ++i)
            new_data[i] = rhs.data[i];
        delete[] data;
        data = new_data;
    }
    return *this;
}

Array::reference Array::operator[](UINT index) throw(out_of_range)
{
    if (index < size_array)
        return data[index];
    else
        throw out_of_range("Index out of range!");
}

Array::const_reference Array::operator[](UINT index) const throw(out_of_range)
{
    if (index < size_array)
        return data[index];
    else
        throw out_of_range("Index out of range!");
}

Array::UINT Array::size() const
{
    return size_array;
}

ostream& operator <<(ostream& out, const Array& tmp)
{
    for (size_t j = 0; j < tmp.size_array; j++)
        cout << tmp[j] << " ";
    cout << endl;
    return out;
}

istream& operator >>(istream& in, const Array& tmp)
{
    // тут має бути введення елементів масиву!
    return in;
}

double Array::Sum()
{
    value_type x_i1, x_i2, y_i1, y_i2;

```

```

Array::iterator l = const_cast<Array::iterator>(this->begin());

double S = 0;
x_i1 = *l;
l++;
y_i1 = *l;
l++;
while (l < this->end())
{
    x_i2 = *l;
    l++;
    y_i2 = *l;
    l++;
    S += abs(((1.0 * x_i1 - 1.0 * x_i2) * (1.0 * y_i1 + 1.0 * y_i2)) / 2.0);
    x_i1 = x_i2;
    y_i1 = y_i2;
}
l = const_cast<Array::iterator>(this->begin());
x_i2 = *l;
l++;
y_i2 = *l;
S += abs(((1.0 * x_i1 - 1.0 * x_i2) * (1.0 * y_i1 + 1.0 * y_i2)) / 2.0);

return S;
}

double Array::Perymetr()
{
    value_type x_i1, x_i2, y_i1, y_i2;
    Array::iterator l = const_cast<Array::iterator>(this->begin());

    double P = 0;
    x_i1 = *l;
    l++;
    y_i1 = *l;
    l++;
    while (l < this->end())
    {
        x_i2 = *l;
        l++;
        y_i2 = *l;
        l++;

        P += sqrt((1.0 * x_i2 - 1.0 * x_i1) * (1.0 * x_i2 - 1.0 * x_i1) +
            (1.0 * y_i2 - 1.0 * y_i1) * (1.0 * y_i2 - 1.0 * y_i1));
        x_i1 = x_i2;
        y_i1 = y_i2;
    }
    l = const_cast<Array::iterator>(this->begin());
    x_i2 = *l;
    l++;
    y_i2 = *l;

    P += sqrt((1.0 * x_i2 - 1.0 * x_i1) * (1.0 * x_i2 - 1.0 * x_i1) +
        (1.0 * y_i2 - 1.0 * y_i1) * (1.0 * y_i2 - 1.0 * y_i1));

    return P;
}

```

файл «Source.cpp»

// Lab\_6\_2.cpp : Defines the entry point for the console application.

```
//
#include "Array.h"
#include <iostream>
#include <time.h>
#include <stdexcept>
#include <exception>
#include <cmath>

using namespace std;

typedef Array::value_type* TArray;

int main()
{
    int n;
    cout << "n= "; cin >> n;
    Array c = Array(2 * n);

    srand((unsigned)time(NULL));
    Array::value_type A = -10;
    Array::value_type B = 10;

    Array::TArray a = new Array::value_type[n];

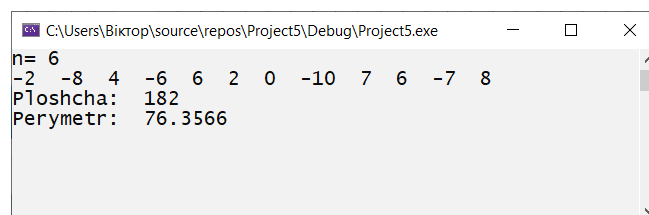
    for (int i = 0; i < 2 * n; i++)
        a[i] = A + rand() % (B - A + 1);

    Array::iterator l = const_cast<Array::iterator>(c.begin());
    for (int j = 0; j < 2 * n; j++, l++)
        *l = a[j];
    cout << c;

    cout << "Ploshcha: " << c.Sum() << endl;
    cout << "Perymetr: " << c.Perymetr() << endl;

    cin.get();
    cin.get();
    return 0;
}
```

### Результат виконання:



```
C:\Users\Bikrop\source\repos\Project5\Debug\Project5.exe
n= 6
-2 -8 4 -6 6 2 0 -10 7 6 -7 8
Ploshcha: 182
Perymetr: 76.3566
```

## Числовий контейнер змінної довжини

### Лістинг 2. Числовий контейнер змінної довжини

[9 – с. 88-90]

#### файл «Array.h»

```
#pragma once
#include <iostream>

using namespace std;

class Array
{
public:
    // типи
    typedef unsigned int    UINT;
    typedef double          value_type;
    typedef double*         iterator;
    typedef const double*   const_iterator;
    typedef double&         reference;
    typedef const double&   const_reference;
    typedef std::size_t     size_type;

private:
    static const size_type minsize = 10; // мінімальний розмір масиву
    size_type Size;                     // виділено пам'яті для елементів
    size_type Count;                     // кількість елементів в масиві
    size_type First;                     // значення індексу першого елемента в масиві
    value_type* elems;                   // вказівник на дані

public:
    // конструктори/копіювання/деструктор
    Array(const size_type& n = minsize)
        throw(bad_alloc, invalid_argument);
    Array(const Array&) throw(bad_alloc);
    Array(const iterator first, const iterator last)
        throw(bad_alloc, invalid_argument);
    Array(const size_type first, const size_type last)
        throw(bad_alloc, invalid_argument);
    ~Array();
    Array& operator=(const Array&);

    // ітератори
    iterator begin() { return elems; }
    const_iterator begin() const { return elems; }
    iterator end() { return elems + Count; }
    const_iterator end() const { return elems + Count; }

    // розміри
    size_type size() const;           // поточний розмір
    bool empty() const;              // якщо є елементи
    size_type capacity() const;      // потенційний розмір
    void resize(size_type newsize)   // змінити розмір
        throw(bad_alloc);

    // доступ до елементів
    reference operator [] (size_type) throw(out_of_range);
    const_reference operator [] (size_type) const throw(out_of_range);
    reference front() { return elems[0]; }
```



```

const_reference front() const { return elems[0]; }
reference back() { return elems[size() - 1]; }
const_reference back() const { return elems[size() - 1]; }

// методи-модифікатори
void push_back(const value_type& v); // додати елемент в кінець
void pop_back(); // видалити останній елемент – реалізувати самостійно
void clear() { Count = 0; } // очистити масив
void swap(Array& other); // поміняти з другим масивом
void assign(const value_type& v); // заповнити масив – реалізувати самостійно

// дружні функції вводу/виводу
friend ostream& operator <<(ostream& out, const Array& a);
friend istream& operator >>(istream& in, Array& a);
};

```

### файл «Array.cpp»

```

#include "Array.h"
#include <iostream>
#include <stdexcept>
#include <exception>

using namespace std;

Array::Array(const Array::size_type& n)
    throw(bad_alloc, invalid_argument)
{
    First = 0;
    Count = Size = n;
    elems = new value_type[Size];
    for (UINT i = 0; i < Size; i++)
        elems[i] = 0;
}

Array::Array(const iterator first, const iterator last)
    throw(bad_alloc, invalid_argument)
{
    First = 0;
    if (first <= last) {
        Count = Size = (last - first) + 1;
        elems = new value_type[Size];
        for (UINT i = 0; i < Size; ++i)
            elems[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const size_type first, const size_type last)
    throw(bad_alloc, invalid_argument)
{
    if (first <= last) {
        First = first;
        Count = Size = (last - first) + 1;
        elems = new value_type[Size];
        for (UINT i = 0; i < Size; ++i)
            elems[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

```

```

Array::Array(const Array& t) throw(bad_alloc)
    : Size(t.Size), Count(t.Count), First(t.First), elems(new value_type[Size])
{
    for (UINT i = 0; i < Size; ++i)
        elems[i] = t.elems[i];
}

Array& Array::operator =(const Array& t)
{
    Array tmp(t);
    swap(tmp);
    return *this;
}

Array::~Array()
{
    delete[]elems;
    elems = 0;
}

void Array::push_back(const value_type& v)
{
    if (Count == Size)           // місця нема
        resize(Size * 2);       // збільшили "місткість"
    elems[Count++] = v;         // присвоїли
}

Array::reference Array::operator [](size_type index) throw(out_of_range)
{
    if ((First <= index) && (index < First + Size))
        return elems[index - First];
    else
        throw out_of_range("Index out of range!");
}

Array::const_reference Array::operator [](size_type index) const
    throw(out_of_range)
{
    if ((First <= index) && (index < First + Size))
        return elems[index - First];
    else
        throw out_of_range("Index out of range!");
}

void Array::resize(size_type newsize) throw(bad_alloc)
{
    if (newsize > capacity())
    {
        value_type* data = new value_type[newsize];
        for (size_type i = 0; i < Count; ++i)
            data[i] = elems[i];

        delete[] elems;
        elems = data;
        Size = newsize;
    }
}

void Array::swap(Array& other)
{
    std::swap(elems, other.elems);           // стандартна функція обміну
}

```

```

        std::swap(Size, other.Size);
    }

Array::size_type Array::capacity() const
{
    return Size;
}

Array::size_type Array::size() const
{
    return Count;
}

bool Array::empty() const
{
    return Count == 0;
}

ostream& operator <<(ostream& out, const Array& tmp)
{
    for (size_t j = 0; j < tmp.Count; j++)
        out << tmp[j] << " ";
    out << endl;
    return out;
}

istream& operator >>(istream& in, Array& tmp)
{
    // тут має бути введення елементів масиву!
    return in;
}

```

### файл «Source.cpp»

```

// Lab_6_2.cpp : Defines the entry point for the console application.
//

#include "Array.h"
#include <iostream>
#include <time.h>
#include <stdexcept>
#include <exception>
#include <cmath>

using namespace std;

typedef Array::value_type* TArray;

int main()
{
    int n;
    cout << "n= "; cin >> n;
    Array c = Array(2 * n);

    srand((unsigned)time(NULL));
    Array::value_type A = -10;
    Array::value_type B = 10;

    TArray a = new Array::value_type[n];

    for (int i = 0; i < 2 * n; i++)
        a[i] = A + rand() % int(B - A + 1);
}

```

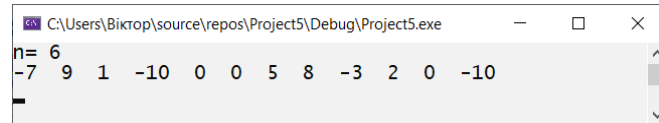
```

Array::iterator l = const_cast<Array::iterator>(c.begin());
for (int j = 0; j < 2 * n; j++, l++)
    *l = a[j];
cout << c;

cin.get();
cin.get();
return 0;
}

```

### Результат виконання:



```

C:\Users\Bikrop\source\repos\Project5\Debug\Project5.exe
n= 6
-7 9 1 -10 0 0 5 8 -3 2 0 -10
_

```

## Послідовний контейнер – список

### Лістинг 3. Послідовний контейнер-список

[9 – с. 91-94]

#### файл «List.h»

```
#pragma once

class NullIterator // клас винятків
{
};

class List
{
public:
    typedef double value_type;
    typedef size_t size_type;

    class iterator;

    // конструктори, копіювання, присвоєння
    List();
    List(const value_type& a, size_type n = 1);
    List(iterator, iterator);
    List(const List& r);
    ~List();
    List& operator=(const List& r);

    // ітератори
    iterator begin() { return head; }
    iterator end() { return tail; }
    iterator begin() const { return head; }
    iterator end() const { return tail; }

    // розміри
    bool empty() const { return (Head == Tail); }
    size_type length() const { return count; }

    // доступ до елементів
    value_type& front() { return *begin(); }
    value_type& back() { iterator it; --it; return *it; }
    iterator find(const value_type& a); // пошук

    // модифікатори контейнера
    void push_front(const value_type&); // додати в початок
    value_type pop_front(); // видалити перший
    void push_back(const value_type&); // додати в кінець
    value_type pop_back(); // видалити останній
    void insert(iterator, const value_type&); // вставити після
    void erase(iterator); // видалити вказаний (в позиції)
    void erase(iterator, iterator); // видалити групу вказаних
    void remove(const value_type&); // видалити заданий (значенням)
    void swap(List&); // обміняти із вказаним списком
    void clear(); // видалити всі елементи
    void splice(List&); // додати список в кінець
    void splice(iterator, List&); // додати список після
    // вказаного елемента
    void sort(); // сортування
    void merge(List&); // злиття відсортованих
```

```

private:
    // елемент
    struct Node
    {
        Node(const value_type& a) : item(a), next(), prev() {}
        Node() : item(), next(), prev() {}
        value_type item;           // інформаційна частина
        Node* next;                // наступний елемент
        Node* prev;                // попередній елемент
    };

    long count;                   // кількість елементів
    Node* Head;                   // "голова" списку
    Node* Tail;                   // "хвіст" списку

public:
    // вкладений клас-ітератор
    class iterator
    {
        friend class List;        // клас-контейнер
        iterator(Node* el) : elem(el) {}

    public:
        // конструктори
        iterator() : elem(0) {}
        iterator(const iterator& it) : elem(it.elem) {}

        // порівняння ітераторів
        bool operator ==(const iterator& it) const
        {
            return elem == it.elem;
        }
        bool operator !=(const iterator& it) const
        {
            return elem != it.elem;
        }

        // переміщення ітератора
        iterator& operator++()      // вперед
        {                          // префіксна форма
            if (elem != 0)
                elem = elem->next;
            return *this;
        }

        iterator operator++(int)    // вперед
        {                          // постфіксна форма
            return operator++();    // виклик префіксної форми
        }

        iterator& operator--()      // назад
        {                          // префіксна форма
            if (elem != 0)
                elem = elem->prev;
            return *this;
        }

        iterator operator--(int)    // назад
        {                          // постфіксна форма
            return operator--();    // виклик префіксної форми
        }
    };

```

```

// *****
// вперед на n позицій - реалізувати самостійно
iterator& operator +=(int n) { return *this; }

// назад на n позицій - реалізувати самостійно
iterator& operator -=(int n) { return *this; }

value_type& operator *() // роз'іменування
{
    if (elem != 0)
        return elem->item;
    else
        throw NullIterator();
}
private:
    Node* elem; // вказівник на елемент
};

private:
    iterator head, tail; // для ітератора - вказівники на голову та хвіст
};

```

## файл «List.cpp»

```

#include "List.h"

// *****
// конструктори, деструктор, присвоєння

// конструктор за умовчанням -
// порожній список з невидимим елементом
List::List()
    : Head(new Node()), Tail(Head), count(0)
{
    Tail->next = Tail->prev = 0;
    head = iterator(Head); // ініціалізація для ітератора
    tail = iterator(Tail);
}

// створюємо список із n елементів та
// заповнюємо його значенням
List::List(const value_type& a, size_type n)
{
    List tmp; // порожній список з фіктивним елементом
    for (size_type i = 0; i < n; i++) // приєднуємо елементи
        tmp.push_front(a);
    *this = tmp; // зробили поточним
}

// конструктор, який створює новий список на основі
// ітераторів іншого списку - відрізняється лише
// іншим способом організації циклу
List::List(iterator first, iterator last)
{
    List tmp;
    for (iterator ip = first; ip != last; ip++)
        tmp.push_front(*ip);
    *this = tmp;
}

```

```

// конструктор копіювання
List::List(const List& r)
{
    List tmp(r.begin(), r.end()); // виконується конструктор з ітераторами
    *this = tmp;
}

// деструктор
List::~List()
{
    Node* deleting_Node = Head; // елемент, який видаляється
    for (Node* p = Head; p != Tail; ) // поки не дійшли до невидимого
    {
        p = p->next; // підготували наступний
        delete deleting_Node; // видалили елемент
        --count;
        deleting_Node = p; // підготували для видалення
    }
    delete deleting_Node; // видалили останній
}

// присвоєння - реалізовано як обмін з тимчасовим об'єктом-контейнером,
// щоб не писати явного повернення пам'яті поточного об'єкту-списку
// - для цього слід реалізувати метод обміну swap()
List& List::operator=(const List& t)
{
    List tmp(t); // тимчасовий локальний об'єкт-список tmp=t
    swap(tmp); // обмін полями з поточним об'єктом
    return *this; // повернення поточного об'єкта
} // tmp знищується

// *****
// методи вставки та видалення

// вставити після
void List::insert(iterator it, const value_type& r)
{
    Node* el = it.elem; // поточний елемент
    if (it == end()) // якщо останній
        push_back(r);
    else
    {
        Node* next_el = it.elem->next; // наступний елемент
        Node* p = new Node(r); // створили новий елемент
        p->next = next_el; // зв'язки в новому
        p->prev = el; // елементі
        next_el->prev = p; // прив'язали новий
        el->next = p;
    }
    ++count;
}

// видалити вказаний (в заданій позиції)
void List::erase(iterator it)
{
    Node* el = it.elem; // поточний елемент
    if (it == begin()) // якщо перший
        pop_front();
    else if (it == end()) // якщо останній
        pop_back();
    else
    {

```



```

        Node* prev_el = it.elem->prev; // попередній елемент
        Node* next_el = it.elem->next; // наступний елемент
        prev_el->next = next_el;      // переналаштовуємо вказівники
        next_el->prev = prev_el;
        delete el;                    // вилучаємо - повертаємо пам'ять
    }
    --count;
}

// *****
// інші методи, які слід реалізувати самостійно

// модифікатори контейнера
void List::push_front(const value_type& v) {} // додати в початок
List::value_type List::pop_front() { return *begin(); } // видалити перший
void List::push_back(const value_type& v) {} // додати в кінець
List::value_type List::pop_back() { return *begin(); } // видалити останній
void List::erase(iterator first, iterator last) {} // видалити групу вказаних
void List::remove(const value_type& v) {} // видалити заданий (значенням)
void List::swap(List& L) {} // обміняти із вказаним списком
void List::clear() {} // видалити всі елементи
void List::splice(List& L) {} // додати список в кінець
void List::splice(iterator it, List& L) {} // додати список після
// вказаного елемента
void List::sort() {} // сортування
void List::merge(List& L) {} // злиття відсортованих

List::iterator List::find(const value_type& a) { return nullptr; } // пошук

```

## файл «Source.cpp»

```

// Lab_6_3.cpp : Defines the entry point for the console application.
//

#include "List.h"
#include <iostream>
#include <stdexcept>
#include <exception>

using namespace std;

int main()
{
    List l;
    for (int i = 1; i <= 10; i++)
    {
        l.insert(l.end(), i); // слід дописати методи класу List
    }

    while (!l.empty()) // слід дописати методи класу List
    {
        cout << l.pop_front() << endl; // слід дописати методи класу List
    }

    cin.get();
    cin.get();
    return 0;
}

```

## Послідовний контейнер – стек

### Лістинг 4. Послідовний контейнер – стек

[9 – с. 95]

#### файл «Stack.h»

```
#pragma once

class Stack
{
public:
    typedef long value_type;

    class EmptyStack {};           // клас винятків

    // конструктор, деструктор
    Stack() : Head(0) {}           // порожній стек

    ~Stack()
    {
        while (!empty())
            pop();
    }

    // додати елемент
    void push(const value_type& d)
    {
        Head = new Node(d, Head);
    }

    // отримати елемент з вершини стеку
    value_type top() const throw(EmptyStack)
    {
        return !empty() ? Head->data : throw EmptyStack();
    }

    // видалити елемент із стеку
    void pop() throw(EmptyStack)
    {
        if (empty())
            throw EmptyStack();    // якщо стек порожній

        Node* old_head = Head;     // запам'ятали вказівник на вершину
        Head = Head->next;           // пересунули вершину
        delete old_head;             // звільнили пам'ять
    }

    // перевірка, чи стек - порожній
    bool empty() const
    {
        return Head == 0;           // стек порожній, якщо вказівник = 0
    }

private:
    // елемент
    struct Node
    {
        value_type data;
```

```

    Node* next;

    Node(const value_type& d, Node* p)
        : data(d), next(p)
    {}
};

Node* Head; // вершина стеку
Stack(const Stack&); // закрили копіювання
Stack& operator =(const Stack&); // закрили присвоєння
};

```

### файл «Stack.cpp»

```

#include "Stack.h"

Stack& Stack::operator =(const Stack& r)
{
    return *this;
}

```

### файл «Source.cpp»

```

#include "Stack.h"
#include <iostream>
#include <stdexcept>
#include <exception>

using namespace std;

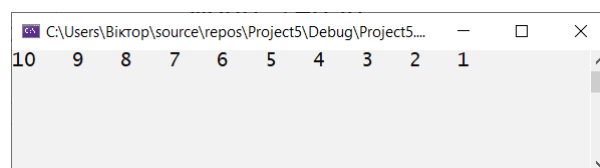
int main()
{
    Stack s;
    for (int i = 1; i <= 10; i++)
    {
        s.push(i);
    }

    while (!s.empty())
    {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;

    cin.get();
    cin.get();
    return 0;
}

```

### Результати виконання:



## Підсумок

*Контейнер* – більш загальна конструкція для об'єднання однорідних елементів у групу, ніж масив.

Однією із найважливіших характеристик контейнера є *спосіб доступу до елементів*: послідовний, прямий чи асоціативний.

*Прямий* та *асоціативний* варіанти доступу зазвичай реалізують за допомогою перевантаження операції індексування `operator[]`, а *послідовний* – за допомогою реалізації класу-ітератора. Використання ітератора для організації доступу має ще ту перевагу, що інтерфейс доступу відокремлюється від інтерфейсу контейнера. Оскільки ітератор тісно пов'язаний з внутрішньою структурою контейнера, його зазвичай створюють як вкладений клас.

Для контейнерів реалізують багато операцій, одна з найчастіше використовуваних – операція об'єднання контейнерів, вона реалізується різними способами.

Динамічні структури даних, такі як стек, черга, дек (двонапрямлена черга) – універсальні структури, робота з якими не залежить від типу елементів. З точки зору користувача, ці структури відрізняються лише способами додавання та вилучення елементів. Зазвичай ці структури реалізуються за допомогою зв'язаних списків. Проте написати універсальний клас-контейнер без використання механізму успадковування або шаблонів дуже складно – як правило, приходится використовувати не типізовані вказівники, які потенційно небезпечні.

## Шаблони класів

Такі контейнери, як стеки та черги, характеризуються способом доступу до елементів і фактично не залежать від типу елементів. Аналогічно, контейнер-масив надає незалежний від типу доступ до елементів за допомогою операції індексування. Створити універсальний контейнер, не залежний від типу елементів, можна двома способами:

- використовуючи в якості елементів контейнера вказівник `void*`;
- на основі шаблону класу, в якому тип елементів визначається параметром шаблону.

Перший спосіб – успадкований від C, оскільки в цій мові не було інших засобів. В C++ зазвичай використовують інший спосіб – *шаблони*.

## Створення шаблонів класів

*Шаблон класу* – це заготовка, із якої створюються конкретні класи в процесі *інстанціювання* (тобто, створення сутності). Воно виконується шляхом підстановки конкретного типу в якості аргументу. Шаблон класу ще називають *параметризованим типом*. Терміни «*параметризований тип*», «*шаблон класу*», «*шаблон-клас*», «*шаблон*» – еквівалентні.

Синтаксис шаблону класу має наступний вигляд:

```
template <параметри> // визначення шаблону
class ім'я_класу      // визначення класу-шаблону
{
    ...                // елементи шаблону класу
};
```

Префікс `template <параметри>` показує компілятору, що наступне визначення класу є шаблоном, а не звичайним класом. Шаблоном може бути і структура. Як і ім'я звичайного класу, ім'я класу-шаблону не має збігатися з іншими іменами в одній і тій самій області видимості.

## Параметри шаблону

Параметри шаблону можуть бути наступних видів:

- параметр-тип;
- параметр цілочисельного або перелічуваного типу;
- параметр-вказівник на об'єкт або вказівник на функцію;
- параметр-посилання на об'єкт або посилання на функцію;
- параметр-вказівник на елемент класу (поле чи метод).

До цілочисельних типів відносяться всі цілі, символьні та булевий (`bool`) типи. Не можна використовувати в якості параметру жодного із вбудованих дробових типів – ні `float`, ні `double`, ні `long double`.

Параметри шаблону, якщо їх кілька, записуються через кому. Всі види параметрів, крім параметрів-типів, записуються звичним способом (як у функціях), наприклад:

```
long p, int (*f)(double), int *t, const char *s
```

Імена параметрів видимі в межах усього шаблону, тому внутрішні для шаблону імена мають бути різними.

Найчастіше використовується *параметр-тип*, він має бути оголошений як

```
class ім'я
```

або

```
typename ім'я
```

В якості імені параметру можна використовувати будь-який допустимий ідентифікатор. Всередині класу такий параметр можна записувати скрізь, де допускається вказувати конкретний тип.

*Приклад.* Шаблон класу з двома полями можна визначити різними способами:

```
template <class T1, class T2>
class Pair
{
    T1 first;
    T2 second;
};
```

або

```
template <typename T1, typename T2>
class Pair
{
    T1 first;
    T2 second;
};
```

або

```
template <class T1, typename T2>
class Pair
{
    T1 first;
    T2 second;
};
```

або

```
template <typename T1, class T2>
class Pair
{
    T1 first;
    T2 second;
};
```

## Оголошення шаблону класу

Шаблон, як і звичайний клас, можна оголосити. Оголошення складається із заголовку шаблону і не містить тіла класу, наприклад:

```
template <typename T> class Stack;
template <typename T1, typename T2> struct Pair;
template <class T> class Array;
```

## Використання шаблонів класів

Визначення об'єктів для деякого класу-шаблону в загальному випадку виглядає так:

```
ім'я_шаблону_класу <аргументи> ім'я_об'єкта           // одиничний об'єкт
ім'я_шаблону_класу <аргументи> ім'я_об'єкта[кількість] // масив
ім'я_шаблону_класу <аргументи> *ім'я_об'єкта          // вказівник
```

Або для випадку константного посилання в якості параметру:

```
const ім'я_шаблону_класу <аргументи> &ім'я_об'єкта    // константне посилання
```

В кутових дужках на місці параметрів при оголошенні об'єктів вказують конкретні аргументи. Параметри шаблону – позиційні (як і аргументи функції), тому фактичне значення, що підставляється, має відповідати виду параметра шаблону: якщо на певному місці визначення шаблону класу вказано параметр-тип, то і аргумент має бути типом чи класом.

Наприклад, визначення конкретних об'єктів-пар може бути таким:

```
Pair<int, int> x;
Pair<int, double> y;
Pair<string, date> z;
```

В якості аргументу-типу можна використовувати і шаблон класу, наприклад:

```
Pair<Pair<int, long>, float> pair;
```

## Зовнішнє визначення методів

Якщо метод шаблону визначається ззовні (поза визначенням шаблону класу), то визначення методу має починатися заголовком `template <параметри>`, в якому мають бути вказані описи всіх параметрів шаблону.

В якості префіксу, який уточнює область дії метода, вказується тип шаблону класу:

```
ім'я_класу < імена_параметрів >
```

де `ім'я_класу` – ім'я шаблону класу, а `імена_параметрів` – список імен параметрів шаблону, відокремлених комами (як в списку `параметри`, лише без ключових слів `class` або `typename` для параметрів-типів та без типів параметрів нетипових параметрів):

```
template <class T, unsigned N> // визначення шаблону
class A                       // A - шаблон класу
{
    void f();                 // оголошення методу f()
};
```

```

template <class T, unsigned N> // зовнішнє визначення методу
void A<T, N>::f()              // f() належить до області дії
{                              // шаблону A<T,N>
    ...
}

```

В тілі методу для всіх імен шаблону префікс може бути відсутній. Для методів, визначених всередині шаблону, префікси не вказуються.

## Приклад шаблону класу

Найпростіше застосування шаблону класу – реалізація «розумного» масиву. В шаблоні із лістингу 5 реалізована операція індексування з перевіркою допустимості індексу. Тип елементів масиву та розмір масиву – параметри шаблону.

### Лістинг 5. Простий масив – шаблон

[9, с.109-110]

#### Файл «Array.h»

```

#pragma once
#include <stdexcept>

template <class T, std::size_t N>
class Array
{
public:
    // типи
    typedef T          value_type;
    typedef T&         reference;
    typedef const T&    const_reference;
    typedef std::size_t size_type;

    // розмір масиву
    static const size_type static_size = N;

    Array(const T& t = T());           // конструктор

    size_type size() const              // отримання розміру
    {
        return static_size;
    }

    // доступ до елементів
    reference operator [] (const size_type& i)
    {
        rangecheck(i);
        return elem[i];
    }

    const_reference operator [] (const size_type& i) const
    {
        rangecheck(i);
        return elem[i];
    }
}

```



```

private:
    // перевірка індексу
    void rangecheck(const size_type& i) const
    {
        if (i >= size())
            throw std::range_error("Array: index out of range!");
    }

    // поле-масив
    T elem[N];
};

// реалізація конструктора
// - має бути у тому ж файлі,
// що і визначення шаблону
template <class T, std::size_t N>
Array<T, N>::Array(const T& t)
{
    for (int i = 0; i < N; i++)
        elem[i] = t;
}

```

#### Файл «L\_6\_6.cpp»

```

#include <iostream>
#include "Array.h"
#include <Windows.h> // забезпечення відображення кирилиці

using namespace std;

int main()
{
    SetConsoleCP(1251); // забезпечення відображення кирилиці
    SetConsoleOutputCP(1251); // забезпечення відображення кирилиці

    const int N = 10;

    Array<int, N> a;
    Array<int, N> b(0);
    Array<int, N> c(b);

    for (int i = 0; i < N; i++)
        cout << a[i] << " ";
    cout << endl;

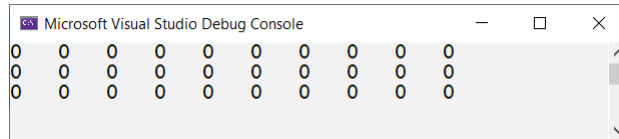
    for (int i = 0; i < N; i++)
        cout << b[i] << " ";
    cout << endl;

    for (int i = 0; i < N; i++)
        cout << c[i] << " ";
    cout << endl;

    return 0;
}

```

## Результат виконання:



```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

## Пояснення

На початку в тілі шаблону класу визначено кілька синонімів типів.

Перевірка індексу в операціях індексування `[]` здійснюється приватним методом `rangecheck()`, який генерує стандартний виняток при «виході» за межі масиву.

Конструктор ініціалізації водночас є конструктором за умовчанням. Конструктор шаблону визначений ззовні, тому визначення методу має починатися заголовком `template <class T, std::size_t N>`, в якому вказані всі параметри шаблону. Тип шаблону класу `Array<T, N>` вказується в якості префіксу. В тілі методу для всіх імен шаблону префікс може бути відсутній. Для методів, визначених всередині шаблону, префікси не вказуються.

## Ініціалізація нулем

У вище наведеному прикладі параметру конструктора присвоюється значення за умовчанням:

```
const T &t = T()
```

При інстанціюванні шаблону (тобто, підстановці конкретного типу при оголошенні об'єктів) така конструкція означає ініціалізацію нулем для елементарних вбудованих типів, для класів приводить до виклику конструктора за умовчанням (тому клас, який використовується в якості значення аргументу-типу шаблону мусить мати конструктор за умовчанням).

Таку конструкцію можна використовувати не лише в списку параметрів, а скрізь, де допустимо оголосити змінну та присвоїти їй значення. Зокрема, в циклі ініціалізації масиву можна було б написати:

```
elem[i] = T();
```

– коректна конструкція при підстановці будь-якого типу, як вбудованого, так і реалізованого. Для будь-якого класу T така конструкція означає явний виклик конструктора за умовчанням (тому клас, який підставляється замість T мусить мати цей конструктор). Але для вбудованих типів конструкторів не існує, і тому для цих типів стандарт C++ визначає зміст такої конструкції

```
T t = T();
```

як ініціалізацію нулем, наприклад:

```
int t = int();
```

– означає

```
int t = 0;
```

В шаблоні можна використовувати ініціалізацію нулем і в списку ініціалізації конструктора. У вищенаведеному прикладі можна замість присвоєння значення за умовчанням використовувати явну ініціалізацію нулем:

```
template <class T, std::size_t N>  
Array<T, N>::Array(): elem() {}    // реалізація конструктора
```

## Параметри шаблону, задані за умовчанням

Параметрам шаблону можна присвоїти значення за умовчанням, в тому числі і параметрам-типам. Для цілочисельних параметрів можна вказувати в якості значення константний вираз, який може обчислити компілятор. Для параметрів-типів можна вказувати або вбудований тип, або будь-яке видиме ім'я типу, видиме в точці визначення шаблону.

Для попереднього прикладу «розумного» масиву (лістинг 5) можна присвоїти за умовчанням тип `double` та визначити розмір масиву за умовчанням – 10 елементів.

Тоді заголовок шаблону набуде вигляду:

```
template <class T = double, std::size_t N = 10>
...
```

Решта визначення шаблону – залишиться без змін, в реалізації методів теж нічого змінювати не треба буде. Тоді допускаються наступні оголошення масивів:

```
Array<int, 20> t;    // повне оголошення
Array<date> d;      // розмір - за умовчанням
Array<> p;          // тип і розмір - за умовчанням
```

У третьому випадку порожні кутові дужки слід вказувати обов'язково – інакше виникне помилка компіляції, бо компілятор буде шукати звичайний клас `Array`. У всіх цих оголошеннях ініціалізація виконується неявно, тому клас `date` мусить мати конструктор без аргументів або конструктор ініціалізації з аргументом за умовчанням.

Як і для функцій з аргументами за умовчанням, присвоювати значення потрібно правим параметром шаблону, тобто можна написати заголовок шаблону `Array<>` так:

```
template <class T, std::size_t N = 10>
...
```

і не можна – так:

```
template <class T = double, std::size_t N>
...
```

Як і для функцій з аргументами за умовчанням, при визначенні об'єкта не можна пропускати ліві параметри, наприклад:

```
Array<20> t;    // помилка трансляції
```

Помилка виникне із-за того, що компілятор не знайде визначення шаблону з одним цілим параметром.

## Спеціалізація шаблонів класів

Часто буває потрібно одночасно разом з загальним шаблоном використовувати деяку спеціалізовану його версію. Для цього в шаблонах реалізовано механізм *спеціалізації*. Спеціалізація полягає в тому, що на основі початкового *первинного* шаблону реалізується його спеціалізована версія для деяких конкретних значень параметрів. Спеціалізація шаблону називається *повною*, якщо конкретизовані всі параметри первинного шаблону. Якщо конкретизовано лише частину параметрів, то спеціалізація називається *частковою*.

Спеціалізація шаблону – це не присвоєння параметрам значення за умовчанням. Шаблон з параметрами за умовчанням – це первинний шаблон, який також можна спеціалізувати.

Спеціалізація шаблону – це «перевантаження» для класів.

Спеціалізація шаблону – це не інстанціювання. Інстанціювання – це створення екземпляру. Інстанціювання виконує компілятор при трансляції програми, коли зустрічає оголошення об'єкту з конкретними значеннями параметрів шаблону.

Спеціалізацію первинного шаблону реалізує програміст як окремий шаблон класу, в якому деякі (або всі) параметри первинного шаблону мають конкретні значення.

Спеціалізація найчастіше застосовується для параметрів-типів. Первинний шаблон визначає загальний варіант, а спеціалізована версія – частковий випадок для конкретних типів. Повна спеціалізація – це конкретний клас, реалізований для конкретних значень параметрів первинного шаблону.

Зазвичай в спеціалізованих версіях перевизначаються окремі (або навіть усі) методи, які для певного конкретного типу мають працювати не так, як в загальному випадку. Наприклад, нехай в первинному шаблоні визначено операцію присвоєння `operator =`. Для символьних масивів ця операція має працювати зовсім не так, як для інших типів. Тоді потрібно визначити спеціалізовану версію первинного шаблону для `const char[]` і перевизначити в ній операцію присвоєння.

При визначенні шаблону та його спеціалізованих версій слід дотримуватися порядку слідування: спочатку потрібно визначити (або оголосити) первинний шаблон, а лише потім можна визначати спеціалізовані версії.

*Приклад.*

```
template <class T>    // первинний шаблон
class Class
{
    ...                // поля та методи шаблону
};
```

```

template <>          // повна спеціалізація – порожні <>
class Class<void *>  // - спеціалізація для безтипових вказівників
{
    ...              // поля та методи спеціалізованої версії
};

```

Аргументи шаблону, які спеціалізуються, вказують в кутових дужках після імені класу. У цьому випадку шаблон Class спеціалізовано для без типових вказівників. Об'єкт такого спеціалізованого шаблонного класу слід оголошувати як об'єкт класу-шаблону з аргументом `void *`, наприклад:

```
Class<void *> d;
```

При частковій спеціалізації конкретизується лише частина параметрів первинного шаблону.

Для вказівників можлива часткова спеціалізація, навіть якщо шаблон має лише один параметр, наприклад:

```

template <class T>    // часткова спеціалізація (не порожні <>)
class Class<T*>       // - спеціалізація для вказівників
{
    ...              // поля та методи спеціалізованої версії
};

```

Позначення `<T*>` після імені класу означає, що така спеціалізація буде використовуватися завжди, коли аргументом шаблону є вказівник будь-якого типу, крім вказівника `void *`, для якого «реалізовано більш спеціалізовану повну спеціалізацію».

Визначення об'єктів виглядає так:

```

Class<date*> pd;      // <T*> - це <date*>,    тому T - це date
Class<int*>  pi;       // <T*> - це <int*>,      тому T - це int
Class<double**> ppd;  // <T*> - це <double**>,  тому T - це double*

```

Спеціалізації первинного шаблону визначаються наявністю аргументів в кутових дужках після імені класу. Повну спеціалізацію компілятор визначає за порожніми кутовими дужками `<>` після слова `template`.

Для того, щоб спеціалізувати шаблон, не обов'язково мати повне визначення первинного шаблону – достатньо оголошення, наприклад:

```

template <class T> class Class;  // оголошення первинного шаблону

template <class T>              // часткова спеціалізація (не порожні <>)
class Class<T*>                 // - спеціалізація для вказівників
{
    ...                         // поля та методи спеціалізованої версії
};

template <>                     // повна спеціалізація (порожні <>)
class Class<void *>             // - спеціалізація для безтипових вказівників
{
    ...                         // поля та методи спеціалізованої версії
};

```

– все буде коректно працювати, якщо не намагатися інстанціювати первинний шаблон.

Спеціалізувати шаблон можна і за нетиповими параметрами, наприклад:

```
template <int n, int k> class A {}; // первинний шаблон
template <int n> class A<n, n> {}; // спеціалізація n = k
```

Спеціалізація – це не успадковування. Первинний шаблон та його спеціалізована версія – це два *різні шаблони*, які утворюють *різні шаблонні класи*. Повна спеціалізація шаблону – це реалізований на основі первинного шаблону незалежний клас. Тому покладатися на те, що в спеціалізованій версії будуть методи, реалізовані в первинному шаблоні, не можна: припустимо, що в первинному шаблоні визначено метод

```
void f(void);
```

Якщо в спеціалізованій версії цей метод не визначено, то спроби викликати метод первинного шаблону для шаблонного класу спеціалізованої версії приведуть до помилок трансляції.

Приклади спеціалізації шаблону класу:

```
template<class A, class B> // первинний шаблон
class C
{
public:
    void f(); // метод, який буде перевизначено
}; // в спеціалізованих версіях

// часткові спеціалізації шаблону

template<class A> // B = int
class C<A, int>
{
public:
    void f();
};

template<class B> // A = int
class C<int, B>
{
public:
    void f();
};

template<class A> // A = B
class C<A, A>
{
public:
    void f();
};
```

```

template<class A, class B> // часткова спеціалізація для вказівників
class C<A*, B*>
{
public:
    void f();
};

template<class A> // A = A*, B = void*
class C<A*, void*>
{
public:
    void f();
};

template<> // повна спеціалізація
class C<int*, double>
{
public:
    void f();
};

```

## Статичні елементи в шаблонах класів

В шаблоні класу можна оголошувати і *статичні методи* і *статичні поля*. В цьому випадку при різних варіантах аргументів шаблону інстанціюються фактично різні шаблонні класи, і кожний з них буде мати свій власний екземпляр статичних елементів, наприклад:

### Лістинг 6. Шаблон класу із статичними елементами

[9, с.113-114]

```

template <typename T>
class StaticClass
{
    static T t; // поле, залежне від типу
    static int count; // поле, не залежне від типу

public:
    StaticClass() { ++count; } // конструктор збільшує лічильник
    ~StaticClass() { --count; } // деструктор зменшує лічильник

    static void print(); // статичний метод
};

template <typename T>
int StaticClass<T>::count = 0; // ініціалізація лічильника

template <typename T>
T StaticClass<T>::t = T(); // визначення та ініціалізація t

template <typename T> // реалізація методу print()
void StaticClass<T>::print()
{
    std::cout << t << " " << count << std::endl;
}

```



В класі визначено лічильник кількості об'єктів – статичне поле `count`. Це поле не залежить від параметру шаблону, проте при визначенні та ініціалізації цього поля слід дотримуватися синтаксису для елементів шаблону класу.

Реалізація статичного методу, як і звичайного, має супроводжуватися заголовком шаблону `template <typename T>`. Слід вказати область дії методу `print()` – шаблон `StaticClass<T>`.

Статичні поля можна спеціалізувати:

```
template<> int StaticClass<int>::t = 25;           // спеціалізація t
template<> double StaticClass<double>::t = 2.5;    // спеціалізація t
```

Спеціалізація статичного поля викликає інстанціювання шаблону з параметром заданого типу. У цьому випадку створюються класи `StaticClass<int>` та `StaticClass<double>`. Для кожного з них будуть створені статичні поля-лічильники `count`; статичні поля `t`, яким присвоюються значення; та створюються статичні методи `print()`.

## Переваги та недоліки шаблонів

Основна перевага – можливість створювати універсальні контейнери: шаблони дають можливість визначати за допомогою одного фрагменту коду цілий набір взаємопов'язаних (перевантажених) функцій, які називаються *шаблонними функціями*, або набір пов'язаних між собою класів, які називаються *шаблонними класами*. Шаблони – це одна із найбільш потужних можливостей C++ стосовно створення універсального програмного забезпечення, яке можна повторно використовувати.

Недолік – використання шаблонів приводить до збільшення обсягу програми – для кожного інстанційованого екземпляру компілятор створює свій код шаблонного класу (шаблонної функції).

## Шаблони класів з шаблонами

Шаблони можуть бути вкладені в інші шаблони. Не існує обмежень на вкладеність класів-шаблонів: клас може бути оголошений всередині шаблону, і шаблон – як всередині класу, так і шаблону. Єдине обмеження – шаблон класу не можна оголосити (чи визначити) в блоці функції (а звичайний клас – можна).

Шаблон може успадковувати від звичайного класу, клас може успадковувати від шаблону, шаблон може успадковувати від шаблону.

В шаблоні класу допускаються вкладені конструкції:

- інстанційований шаблонний клас в якості поля в класі і в шаблоні;
- шаблон класу в якості параметра-типу за умовчанням;

- метод-шаблон, в тому числі конструктор.

## Поле-шаблон

В класі можна оголосити поле-об'єкт шаблонного класу, утвореного в результаті інстанціювання деякого шаблону. Оголошення такого поля нічим не відрізняється від оголошення звичайного поля.

Наприклад, на основі шаблону

```
template <class T, std::size_t N>
class Array
{
    ...
    T elem[N];    // поле-масив
};
```

Можна реалізувати стек фіксованого розміру:

```
class Stack
{
    Array<double, 100> t;    // поле - інстанційований шаблон

public:
    ...                    // методи
};
```

Створити об'єкт-стек можна звичним способом:

```
Stack s;
```

Поле може бути не лише об'єктом звичайного чи шаблонного класу, а і шаблоном.

Поле-шаблон в шаблоні виглядає так:

```
template <class T, std::size_t N = 100>    // параметри стека
class Stack
{
    Array<T, N> t;                        // залежне ім'я

public:
    ...                                    // методи
};
```

Оголошувати змінні такого класу-шаблону можна таким чином:

```
Stack<double> s;
```

або так – з явним визначенням кількості елементів:

```
Stack<double, 500> s;
```

## Параметр-шаблон

В якості параметра шаблону можна використовувати інший шаблон. Параметру-шаблону можна присвоїти значення за умовчанням, наприклад:

```
template <typename T,                // тип елементів
          std::size_t N = 100,       // кількість
          template <class, std::size_t> // параметр-шаблон
          class Container = Array    // Array - значення за умовчанням
        >                             // кінець списку параметрів
class Stack
{
    Container<T, N> s;                // поле-контейнер

public:
    ...                               // методи
};
```

В цьому прикладі параметр-шаблон оголошений останнім:

```
template <class, std::size_t> class Container = Array
```

Список параметрів параметра-шаблону не містить імен параметрів.

Оголошувати змінні-стеки – об'єкти шаблонних класів на основі інстанціювання шаблону `Stack` можна так:

```
Stack <double> s1;           // розмір і контейнер за умовчанням
Stack <int, 500> s2;         // контейнер за умовчанням
Stack <long, 1000, Array> s3; // всі параметри задані явно
```

В третьому випадку вказується лише ім'я контейнера без аргументів. Замість шаблону `Array` можна вказувати будь-який контейнер, що має два параметри: тип елементів та без-знакове ціле.

## Метод-шаблон

В звичайному класі і в шаблоні класу можна визначити метод-шаблон.

Вдосконалимо приклад, наведений в лістингу 5: додам можливість присвоювати масиви різної довжини і / або масиви з різними типами елементів. Для цього додам до шаблону класу Array метод-шаблон операції присвоєння:

### Лістинг 7. Метод-шаблон в шаблоні класу

[9, с.116-117]

#### Файл «Array.h»

```
#pragma once
#include <cstdlib> // для size_t
#include <stdexcept>

template <class T = double, std::size_t N = 10>
class Array
{
public:
    // типи
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    typedef std::size_t size_type;

    // розмір масиву
    static const size_type static_size = N;

    Array(const T& t = T()); // конструктор

    size_type size() const // отримання розміру
    {
        return static_size;
    }

    // доступ до елементів
    reference operator[] (const size_type& i)
    {
        rangecheck(i);
        return elem[i];
    }

    const_reference operator[] (const size_type& i) const
    {
        rangecheck(i);
        return elem[i];
    }

    // присвоєння ***** <= нова операція!
    template <typename TI, std::size_t NI>
    Array<T, N>& operator = (const Array<TI, NI>& r);

private:
    void rangecheck(const size_type& i) const // перевірка індексу
    {
```

```

        if (i >= size())
            throw std::range_error("Array: index out of range!");
    }

    // поле-масив
    T elem[N];
};

template <class T, std::size_t N>           // реалізація конструктора
Array<T, N>::Array(const T& t)
{
    for (int i = 0; i < N; i++)
        elem[i] = t;
}

// визначення метода-шаблону
template <class T, std::size_t N>           // зовнішній шаблон
template <typename TI, std::size_t NI>      // внутрішній шаблон
Array<T, N>& Array<T, N>::operator = (const Array<TI, NI>& r)
{
    if ((void*)this != (void*)&r)           // перевірка самоприсвоєння
    {
        if (N >= NI)                       // перевірка розміру
        {
            for (int i = 0; i < NI; i++)     // перетворення типу
                elem[i] = static_cast<T>(r[i]);
        }
    }
    return *this;
}

```

### Файл «Source.cpp»

```

#include <iostream>
#include "Array.h"
#include <Windows.h>           // забезпечення відображення кирилиці

using namespace std;

int main()
{
    SetConsoleCP(1251);        // забезпечення відображення кирилиці
    SetConsoleOutputCP(1251);  // забезпечення відображення кирилиці

    Array<int, 10> mn;
    Array<int, 10> mm;          // тип і розмір mn = mm
    mm = mn;                   // вбудована операція

    Array<int, 5> x;            // розмір x < розмір mn
    mn = x;                     // операція-шаблон

    Array<double, 15> y;        // інший тип та розмір
    y = x;                      // операція-шаблон

    return 0;
}

```

### Пояснення

Прототип метода-шаблону має вигляд:

```

template <typename TI, std::size_t NI>
Array<T, N> operator =(const Array<TI, NI> &r);

```

Його визначення починається із заголовка `template <class T, std::size_t N>` шаблону класу, після якого вказується власний заголовок метода-шаблону `template <typename TI, std::size_t NI>`.

Методи-шаблони не можуть бути віртуальними, для звичайних методів шаблону класу такого обмеження немає.

Шаблон операції присвоєння, наведений у цьому прикладі, не заміщує операцію присвоєння, яка генерується автоматично за умовчанням: для присвоєння масивів одного типу буде викликатися стандартна операція присвоєння. Аналогічно, шаблон конструктора копіювання ніколи не заміщує конструктор, який генерується автоматично.

За допомогою шаблону-присвоєння виконується присвоєння «меншого» масиву «більшому» – як за розміром, так і за типом:

```
Array<int, 10> mn;  
Array<int, 10> mm;           // тип і розмір mn = mm  
mm = mn;                    // вбудована операція  
  
Array<int, 5> x;             // розмір x < розмір mn  
mn = x;                     // операція-шаблон  
  
Array<double, 15> y;         // інші тип та розмір  
y = x;                      // операція-шаблон
```

## Шаблони та успадкування

### Клас → шаблон

Шаблон може успадковувати від звичайного класу – тоді такий базовий клас називається *незалежним базовим класом*:

```
class Base {};  
  
template <class T>  
class Template: public Base  
{  
    ...  
};
```

Це буває потрібно, наприклад, для того, щоб зробити статичні поля спільними для всіх шаблонних класів, які інстанціюються на основі шаблону-нащадка:

```
class CommonData  
{  
public:  
    static const double Exp;  
    static const double Pi;  
};  
  
const double CommonData::Exp = 2.7182818284590452353;  
const double CommonData::Pi  = 3.1415926535897932384;  
  
template <typename T>  
class DeriveTemplate: public CommonData  
{  
    ...  
};
```

В цьому випадку статичні поля `Exp` та `Pi` будуть спільними для всіх класів, що інстанціюються на основі шаблону `DeriveTemplate`.

### Шаблон → шаблон

Шаблон може успадковувати від шаблону:

```
template <typename T1, typename T2>           // базовий шаблон  
class BasePair  
{  
    T1 first;  
    T2 second;  
};  
  
template <class T>  
class DeriveTemplate: public BasePair<T, T>    // шаблон-нащадок  
{};
```

### Шаблон → клас

Звичайний клас також може успадковувати від шаблону. Звичайний клас не допускає параметрів (які потрібно шаблону класу), тому фактично успадкування відбувається не від

шаблону, а від конкретного шаблонного класу, інстанційованого на основі базового шаблону:

```
template <class T>                                // базовий шаблон
class BaseTemplate
{
};

class DeriveClass: public BaseTemplate<int> // клас-нащадок
{
};
```

Таке успадковування використовується для того, щоб відстежувати кількість об'єктів класу:

## Лістинг 8. Обчислення кількості об'єктів класу

[9, с.118]

```
template <class T>                                // шаблон класу з лічильником
class Count
{
    static unsigned int counter; // лічильник

public:
    Count()
    {
        ++counter;
    }

    Count(const Count<T>& t)
    {
        ++counter;
    }

    ~Count()
    {
        --counter;
    }

    static unsigned int getCount()
    {
        return counter;
    }
};

template <class T>                                // ініціалізація лічильника
unsigned int Count<T>::counter = 0;

// успадковування від шаблону
class Class01 : public Count<Class01>
{
};

class Class02 : public Count<Class02>
{
};
```

Тепер кожен із похідних класів буде мати власний екземпляр лічильника об'єктів.



## Шаблони та дружні функції чи дружні класи

Шаблони можуть мати друзів, і шаблони можуть бути друзями.

Наприклад, дружня функція виводу для шаблону класу має бути такою:

```
// файл "TemplateClass.h"

#include <iostream>

using namespace std;

template <typename T>
class TClass
{
    T x;

public:
    TClass(const T& t = T())           // конструктор ініціалізації
        : x(t) {}

    friend ostream& operator << <>(ostream& os, const TClass<T>& t);
};

// зовнішнє визначення дружньої функції
template <typename T>
ostream& operator << (ostream& os, const TClass<T>& t)
{
    return os << "(" << t.x << ")";
}
```

Порожні кутові дужки <> після імені функції, вказані в її прототипі, означають, що дружня функція є шаблоном.

Визначення дружньої функції-шаблону подібне до визначення методу-шаблону, але простіше – немає другого заголовка.

Використовувати такий шаблон – дуже просто:

```
// файл "Source.cpp"

#include <iostream>
#include <Windows.h>           // забезпечення відображення кирилиці
#include "TemplateClass.h"

using namespace std;

int main()
{
    SetConsoleCP(1251);        // забезпечення відображення кирилиці
    SetConsoleOutputCP(1251);  // забезпечення відображення кирилиці

    TClass<int> a(1);           // створення об'єкту шаблонного класу
                                // - ініціалізуємо поле значенням 1
    cout << a << endl;         // використання дружньої операції

    return 0;
}
```

Для дружніх функцій-шаблонів (тих, які визначені поза тілом шаблону класу) є обмеження: не виконуються за умовчанням перетворення типів аргументів.

Добавимо до шаблону класу дружню операцію додавання:

```
// файл "TemplateClass.h"

#include <iostream>

using namespace std;

template <typename T>
class TClass
{
    T x;

public:
    TClass(const T& t = T())           // конструктор ініціалізації
        : x(t) {}

    friend ostream& operator << <>(ostream& os, const TClass<T>& t);
    friend TClass<T> operator + <>(const TClass<T>& a, const TClass<T>& b);
};

// зовнішні визначення дружніх функцій
template <typename T>
ostream& operator << (ostream& os, const TClass<T>& t)
{
    return os << "(" << t.x << ")";
}

template <typename T>
TClass<T> operator + (const TClass<T>& a, const TClass<T>& b)
{
    return TClass<T>(a.x + b.x);
}
```

Попробуємо використати цей шаблон:

```
// файл "Source.cpp"

#include <iostream>
#include <Windows.h>           // забезпечення відображення кирилиці
#include "TemplateClass.h"

using namespace std;

int main()
{
    SetConsoleCP(1251);        // забезпечення відображення кирилиці
    SetConsoleOutputCP(1251);  // забезпечення відображення кирилиці

    TClass<int> a(1), d1(2), d2(3); // створення об'єктів шаблонного класу

    cout << a << endl;          // використання дружньої операції <<
    cout << d1 + d2 << endl;      // використання дружньої операції +
    cout << d1 + 1 << endl;      // помилка компіляції

    return 0;
}
```

Вираз  $d1 + 1$  приводить до помилки компіляції, бо для шаблонів конструктор автоматично не викликається (на відміну від звичайних класів).

Розв'язок буде такий: слід визначити дружні функції безпосередньо в тілі шаблону класу:

```
// файл "TemplateClass.h"

#include <iostream>

using namespace std;

template <typename T>
class TClass
{
    T x;

public:
    TClass(const T& t = T())           // конструктор ініціалізації
        : x(t) {}

    friend ostream& operator << <>(ostream& os, const TClass<T>& t)
    {
        return os << "(" << t.x << ")";
    }

    friend TClass<T> operator + <>(const TClass<T>& a, const TClass<T>& b)
    {
        return TClass<T>(a.x + b.x);
    }
};

// файл "Source.cpp"

#include <iostream>
#include <Windows.h>           // забезпечення відображення кирилиці
#include "TemplateClass.h"

using namespace std;

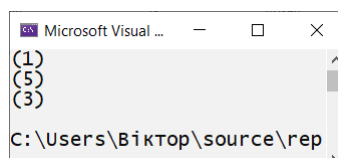
int main()
{
    SetConsoleCP(1251);        // забезпечення відображення кирилиці
    SetConsoleOutputCP(1251);  // забезпечення відображення кирилиці

    TClass<int> a(1), d1(2), d2(3); // створення об'єктів шаблонного класу

    cout << a << endl;          // використання дружньої операції <<
    cout << d1 + d2 << endl;      // використання дружньої операції +
    cout << d1 + 1 << endl;      // все вірно

    return 0;
}
```

Результат виконання:



Якщо дружня функція визначається в тілі шаблону класу, то вона вважається звичайною функцією, а не функцією-шаблоном, не дивлячись на те, що аргументи цієї функції залежать від параметрів шаблону.

При визначенні дружньої функції в тілі шаблону класу її аргументи мають залежати від параметрів шаблону.

Таким чином, відношення дружності може бути встановлене між шаблоном класу і глобальною функцією, функцією-методом іншого класу (можливо, шаблонного класу) або навіть цілим класом (можливо, шаблонним класом).

Якщо всередині шаблону класу `X`, оголошеного як

```
template<class T> class X
{
};
```

міститься оголошення дружньої функції

```
friend void f1();
```

то функція `f1()` є дружньою для кожного шаблонного класу, отриманого з цього шаблону.

Якщо всередині шаблону класу `X`, оголошеного як

```
template<class T> class X
{
};
```

міститься оголошення дружньої функції

```
friend void f2(X<T> &);
```

то для конкретного типу `T`, наприклад, `float`, дружньою для класу `X<float>` буде тільки функція `f2(X<float> &)`.

Усередині шаблону класу можна оголосити функцію-метод іншого класу дружньою для будь-якого шаблонного класу, отриманого з цього шаблону. Для цього потрібно використовувати ім'я функції-методу іншого класу, ім'я цього класу і бінарну операцію доступу до області дії. Наприклад, якщо усередині шаблону класу `X`, який був оголошений як

```
template<class T> class X
{
};
```

оголошується дружня функція у формі

```
friend void A::f4();
```

то функція-елемент `f4` класу `A` буде дружньою для кожного шаблонного класу, отриманого з цього шаблону.

Усередині шаблону класу `X`, який був оголошений наступним чином

```
template<class T> class X
{
};
```

оголошення дружньої функції у вигляді

```
friend void C<T>::f5(X<T> &);
```

для конкретного типу `T`, наприклад, `float`, зробіть функцію-метод шаблонного класу `C<float>`

```
C<float>::f5 (X<float> &)
```

другом тільки шаблонного класу `X<float>`.

Усередині шаблону класу `X`, оголошеного як

```
template<class T> class X
{};
```

можна оголосити інший, дружній клас `Y`

```
friend class Y;
```

в результаті чого, кожна з функцій-елементів класу `Y` буде дружньою для кожного шаблонного класу, виробленого з шаблону класу `X`.

Якщо всередині шаблону класу `X`, який був оголошений як

```
template<class T> class X
{};
```

оголошений другий клас `Z` у вигляді

```
friend class Z<T>;
```

то при створенні шаблонного класу з конкретним типом `T`, наприклад, типом `float`, всі елементи класу `Z<float>` будуть друзями шаблонного класу `X<float>`.

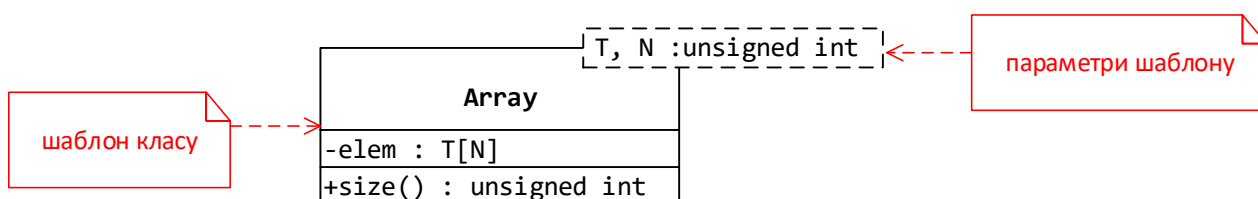
## Позначення шаблонів на UML-діаграмах класів

Розглянемо приклад шаблону класу, оголошеного як

```
template <class T, std::size_t N>
class Array
{
public:
    ...
    std::size_t size() const;

private:
    T elem[N];
};
```

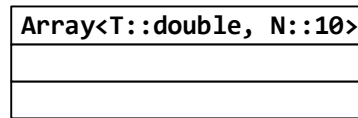
На UML-діаграмах класів шаблони позначаються подібно до звичайних класів, лише верхній правий кут прямокутника, що позначає шаблон класу, містить перелік параметрів шаблону, записаних в пунктирному прямокутнику:



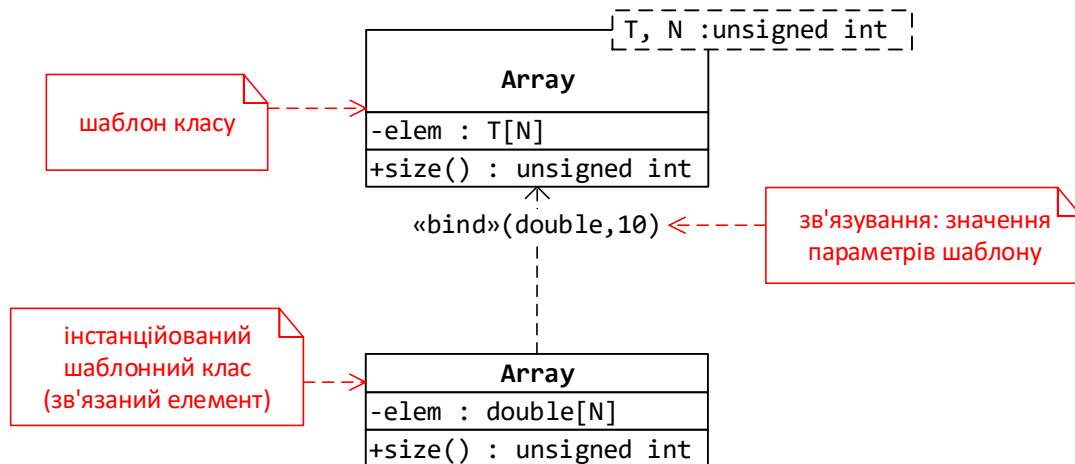
Припустимо, що шаблон класу `Array<T, N>` інстанціюється наступним чином:

```
Array<double, 10> a;
```

Тоді шаблонний клас, утворений в результаті такого інстанціювання, на UML-діаграмах класів позначається так:



Інший спосіб позначити шаблонний клас, інстанційований від шаблону `Array<T,N>`:



## Підсумок

В C++ включено два види шаблонів: шаблони функцій та шаблони класів. Основне призначення шаблонів класів – реалізація узагальнених контейнерів, які не залежать від типу елементів.

*Шаблонний клас* – це клас, утворений в результаті *інстанціювання* шаблону класу.

Шаблони дають можливість визначити за допомогою одного фрагменту коду цілий набір взаємопов'язаних класів, які називаються *шаблонними класами*.

Шаблон класу являє собою деякий опис родового класу, на основі якого створюються специфічні версії для конкретних типів даних.

Шаблони класів часто називають параметризованими типами, оскільки вони мають один або кілька параметрів типу, які визначають налаштування родового шаблону класу на специфічний тип даних при створенні об'єкта класу.

Для того, щоб використовувати шаблонні класи, програмісту достатньо один раз описати шаблон класу. Кожного разу, коли потрібна реалізація класу для нового типу даних, програміст, використовуючи простий короткий запис, повідомляє про це компілятору, який і створює текст (початковий код) необхідного шаблонного класу.

Основний вид параметрів шаблону – ім'я типу. Проте шаблони класів можуть мати і параметри інших видів: в шаблонах є можливість використовувати так звані *нетипові*

*параметри*. Одним із найбільш потужних засобів програмування є параметр-шаблон. Крім того, параметри шаблону класу можна задавати за умовчанням.

Опис шаблону класу виглядає як традиційне визначення класу, перед яким записують заголовок `template <class T>` або `template <typename T>` (ідентифікатор `T` вибрано в якості прикладу – можна вибирати інші імена). Цей заголовок вказує на те, що такий опис є шаблоном класу з параметром типу `T`, який позначає тип класів, що будуть створюватися на основі цього шаблону. Ідентифікатор `T` визначає тип даних, який може використовуватися як тип полів класу, в заголовку класу та в методах класу.

Кожне визначення методу поза шаблоном класу починається із заголовка `template <class T>` або `template <typename T>`, за яким записують визначення методу, подібне до стандартного визначення, але в якості типу елемента класу завжди вказують параметр типу `T`. Щоб пов'язати кожне визначення методу з областю дії шаблону класу, використовується бінарна операція доступу до області дії `::` з іменем шаблону класу `ім'яКласу< T >`.

Процес підстановки аргументів на місце параметрів шаблону називається *інстанціюванням* шаблону. Інстанціювання дозволяє створити із однієї «заготовки» ціле сімейство конкретних реалізацій.

На основі первинного шаблону можна отримати спеціалізовані версії, в яких задані частина або навіть всі параметри. Останній варіант називається повною спеціалізацією.

Класи-шаблони можуть бути вкладеними, можуть бути друзями, можуть успадковувати від шаблону або від класу.

Шаблон класу може бути похідним від шаблонного класу. Шаблон класу може бути похідним від нешаблонного класу. Шаблонний клас може бути похідним від шаблону класу. Нешаблонний клас може бути похідним від шаблону класу.

Локальні класи не можуть містити шаблони в якості своїх елементів.

Шаблони методів не можуть бути віртуальними.

Шаблони класів можуть містити статичні елементи, дружні функції та класи.

Всередині шаблону не можна визначати `friend`-шаблони.

Зазвичай визначають повну спеціалізацію шаблону для вказівників та використовують її в якості базового класу для частково спеціалізованого шаблону за вказівниками. Успадковування класу від повної спеціалізації шаблону дозволяє організувати лічильник об'єктів для конкретного класу, а не для всієї ієрархії успадковування.

Для спеціалізованого типу можна визначити клас, який скасовує дію шаблону класу для певного типу.

Функції та цілі класи можуть бути оголошені *дружніми* для нешаблонних класів. Для шаблонів класів також можна встановити відношення дружності. Дружність можна встановити між шаблоном класу та глобальною функцією, методом іншого класу (можливо, шаблонного класу), а також цілим класом (можливо, шаблонним класом).

Дружні функції можуть бути визначені як зовнішні функції-шаблони або безпосередньо всередині класу. В останньому випадку дружня функція є звичайною функцією, і її параметри мають залежати від параметрів шаблону, бо інакше при не однократному інстанціюванні виникне помилка повторного визначення.

Кожний шаблонний клас, отриманий на основі шаблону класу, має власний екземпляр кожного статичного поля шаблону; всі об'єкти цього шаблонного класу використовують цей свій власний екземпляр статичного поля. Як і статичні поля нешаблонних класів, статичні поля шаблонних класів мають бути ініціалізовані в області дії файлу.

Кожний шаблонний клас отримує власний екземпляр статичного методу шаблону класу.



## Шаблони функцій

Крім шаблонів класів, C++ дозволяє створювати та використовувати шаблони функцій.

### Створення шаблону функції

Синтаксис шаблону функції наступний:

```
template< параметри >
заголовок
{ тіло }
```

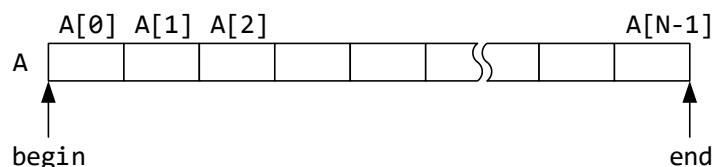
Параметри шаблону функції можуть бути такими ж самими, як і параметри шаблону класу: в шаблонах функцій можна використовувати нетипові параметри (тобто, параметри, які не є типами); параметр-тип можна вказувати в якості типу аргументів, так і в якості типу результату функції.

Наприклад, шаблон функції пошуку мінімального значення в масиві може бути таким:

```
template <typename T>
T Min(T const *begin, T const *end)
{
    T min = *begin;           // перший елемент масиву
    while (begin != end)      // поки не розглянули всі елементи
        if (min > *++begin)    // перейшли до наступного елемента
            min = *begin;      // і порівняли його з min
    return min;
}
```

Цей шаблон функції характеризується одним параметром-типом `T`. Пошук мінімального значення відбувається наступним чином:

Функція-шаблон приймає два параметри-ітератори `begin` – вказівник на перший елемент масиву (тобто, на початок області, відведеної для масиву) та `end` – вказівник на елемент, що слідує за останнім елементом масиву (тобто, на кінець області, відведеної для масиву):



Спочатку змінна `min` отримує значення першого елемента масиву:

```
T min = *begin;
```

Потім в циклі *поки не розглянули всі елементи масиву*

```
while (begin != end)
```

переходимо до наступного елемента `++begin` та порівнюємо його з `min`.

Якщо умова *поточний елемент менший за min* – справджується, то min присвоюємо значення поточного елемента:

```
if (min > *(++begin))
    min = *begin;
```

Після завершення циклу змінна min отримає значення мінімального елемента масиву.

Виклик такої функції-шаблону виглядає як виклик звичайної функції:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout << Min(a, a+10) << endl;

double b[10] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.01};
cout << Min(b, b+10) << endl;
```

При виклику шаблону-функції аргументи шаблону в кутових дужках можна не вказувати, як і самі кутові дужки <>. Компілятор сам підставляє тип в якості значення параметру шаблону на основі інформації про тип фактичних аргументів функції при її виклику. Цей механізм самостійної підстановки типу-параметру шаблону називається *виведенням типу*.

Для шаблонів класів аргументи шаблону завжди слід вказувати явно. При виклику функції-шаблону теж можна явно вказувати аргументи шаблону:

```
cout << Min<int>(a, a+10) << endl;
cout << Min<double>(b, b+10) << endl;
```

Виведення типу не спрацьовує для типу результату функції – такий аргумент шаблону при виклику функції-шаблону завжди слід вказувати явно.

Для функцій з аргументами за умовчанням, параметри із значенням за умовчанням вказуються останніми в списку параметрів. Аналогічно і для параметрів шаблонів функцій – параметри, які можна не вказувати, мають бути останніми в списку параметрів шаблону. Тому, якщо необхідно, щоб тип результату функції також був параметром шаблону, його слід записувати першим в списку параметрів:

```
template <class RT, class T>
RT Mul(T const *begin, T const *end)
{
    RT result = 1;
    while (begin != end)
    {
        result *= *begin;
        ++begin;
    }
    return result;
}
```

Виклики шаблонних функцій, інстанційованих від такого шаблону, слід записувати з явною вказівкою аргументу для типу результату функції, наприклад:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout << Mul<long>(a, a+10) << endl;
```

```
double b[10] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.01};
cout << Mul<double>(b, b+10) << endl;
```

Типи параметрів функції компілятор виводить самостійно, але спроба викликати таку шаблонну функцію без аргументу-типу результату, наприклад так:

```
cout << Mul(b, b+10) << endl;
```

– приводить до помилки трансляції.

## Параметри шаблону за умовчанням

В шаблонах функцій не можна вказувати параметри за умовчанням.

Проте це обмеження легко обійти, якщо використати *клас-обгортку*. В наступному прикладі така функція реалізована як статичний метод класу (тоді її можна використовувати, не створюючи об'єктів цього класу):

```
template <class RT = double, class T = float>
class Function
{
public:
    static RT Mul(T const *begin, T const *end)
    {
        RT result = 1;
        while (begin != end)
        {
            result *= *begin;
            ++begin;
        }
        return result;
    }
};
```

Виклик такої функції слід записувати з префіксом шаблону класу:

```
float f[10] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 10.01};
cout << Function<>::Mul(f, f+10) << endl;

int L[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout << Function<float, int>::Mul(L, L+10) << endl;
```

## Параметри шаблону – не типи

Нетипові параметри компілятор зазвичай не може вивести, тому їх слід вказувати явно, наприклад шаблон функції ініціалізації числового масиву може бути наступним:

```
template <std::size_t N, class T, T t>
void Init(T array[N]) // масив
{
    for (std::size_t i=0; i < N; i++)
        array[i] = t;
}
```

Виклик такої функції вимагає вказівки всіх аргументів шаблону:

```
int a[100];
Init<100, int, 3>(a);
```

## Перевантаження шаблонів функцій

Допускається перевантаження шаблонів функцій шаблонами та звичайними функціями. Як і при перевантаженні звичайних функцій, шаблони (і функції) мають відрізнятися списками параметрів. Компілятор при опрацюванні конкретного виклику намагається підібрати той варіант, який найбільше підходить.

## Спеціалізація шаблонів функцій

Для шаблонів функцій реалізовано повну спеціалізацію. Часткова спеціалізація для функцій-шаблонів не допускається.

Наприклад, функція `max()` з параметрами-вказівниками на символи, відповідний шаблон та спеціалізація шаблону з параметрами-вказівниками виглядають таким чином:

```
char* max(const char *x, const char *y)           // функція
{
    return ( strcmp(x,y) > 0 ) ? x : y;
}

template <typename T>                             // шаблон
T const& max(T const& x, T const& y)
{
    return (x > y) ? x : y;
}

template <>                                         // спеціалізація шаблону
char* const& max(char* const& x, char* const& y)
{
    return ( strcmp(x,y) > 0 ) ? x : y;
}
```

Спеціалізація починається ключовим словом `template`, за яким записують порожні кутові дужки `<>`.

В загальному випадку, для повної спеціалізації необхідно вказувати аргументи спеціалізації після імені шаблону:

```
template <>                                         // спеціалізація шаблону
char* const& max<char*>(char* const& x, char* const& y)
{
    return ( strcmp(x,y) > 0 ) ? x : y;
}
```

Проте для шаблонів функцій це робити – не обов’язково, бо компілятор може вивести тип-аргумент спеціалізації самостійно.

## Узагальнені алгоритми та функтори

Узагальнений алгоритм – це шаблон функції, який може виконувати задану операцію з послідовним контейнером будь-якого виду. Зазвичай параметрами узагальненого алгоритму є ітератори вхідного (та вихідного) контейнеру.

Прикладом узагальненого алгоритму, який опрацьовує та модифікує вхідний контейнер, може бути шаблон функції сортування, який може мати наступний прототип:

```
template <class Iterator>
void sort(Iterator first, Iterator last);
```

Прикладом узагальненого алгоритму, який послідовно перебирає вхідний контейнер і записує результати у вихідний, може бути шаблон функції копіювання, прототип якого може бути таким:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first,
                   InputIterator last,
                   OutputIterator result);
```

Ітератори типу `InputIterator` визначають діапазон елементів вхідного контейнера, які будуть опрацьовуватися; ітератор типу `OutputIterator` – початковий ітератор вихідного контейнера. Зазвичай, вихідний контейнер має існувати на момент виклику функції-узагальненого алгоритму і він мусить мати достатній розмір, щоб вмістити всі елементи вихідного контейнеру-результату роботи узагальненого алгоритму.

## Реалізація узагальнених алгоритмів

Розглянемо, як можна розв’язати наступну задачу: написати функцію-фільтр, яка буде вибирати із послідовного контейнера додатні елементи та копіювати їх в інший контейнер.

Перший крок узагальнення – використання шаблонів: потрібно написати шаблон функції-фільтру, щоб не залежати від типу елементів контейнеру. Проте цього – недостатньо: якщо ми будемо передавати контейнер в якості параметру і повертати новий контейнер як результат, то наша функція-фільтр буде залежати від типу контейнера, – така функція не зможе опрацьовувати контейнери іншого типу. Крім того, за допомогою такої функції не можна буде опрацьовувати масиви.

Розв’язок нам відомий – ітератори. Вхідний контейнер, в якому здійснюється пошук, слід представити за допомогою двох ітераторів, які визначають напіввідкритий інтервал `[first, last)`. Для забезпечення більшої універсальності вихідний контейнер теж слід подати за допомогою одного ітератора – який визначає початок області, в яку будуть записуватися результати роботи узагальненого алгоритму.

Такий підхід одразу накладає певні обмеження на використання цієї функції-фільтру:

1. Для того, щоб задати ітератор вихідного контейнера при виклику функції, вихідний контейнер має існувати на момент виклику.
2. В контейнері мають існувати елементи та їх має бути достатньо для того, щоб вмістити всі елементи вхідного контейнера, які задовольняють критерію відбору.
3. Вихідний контейнер має бути сумісним за типом елементів із вхідним.

Таким чином, по відношенню до вихідного контейнера, ми повністю покладаємося на програміста, який використовує нашу функцію. Це приводить до того, що ми отримуємо універсальний алгоритм, який справді не залежить від вхідного та вихідного контейнерів:

```
template < class InputIterator,      // ітератор вхідного контейнера
          class OutputIterator,     // ітератор вихідного контейнера
          class T                    // тип елементів
        >
void copy_if( InputIterator first,
              InputIterator last,
              OutputIterator result,
              const T &v
            )
{
    for ( ; first != last; ++first)
        if ( *first > v )
        {
            *result = *first;
            ++result;
        }
    return;
}
```

– алгоритм вірно працює і з масивами і з контейнерами, які забезпечують послідовний доступ за допомогою ітераторів (в т.ч. і з контейнерами стандартної бібліотеки шаблонів).

Проте цей алгоритм все ще не універсальний – критерій відбору елементів жорстко заданий в тілі функції у вигляді умови команди `if`. Алгоритм став би по-справжньому універсальним, якби критерій відбору можна було би задавати у вигляді параметру.

Це можна зробити, якщо визначити ще один параметр – вказівник на функцію. Така функція буде в нашому алгоритмі викликатися в умові команди `if`, тому вона має повертати результат типу `bool`.

Функція, яка повертає результат типу `bool`, називається *предикатом*. Функція-предикат з одним параметром називається *унарним* предикатом; функція-предикат з двома параметрами – *бінарним* предикатом.

Визначимо тип вказівників на функції-унарні предикати:

```
typedef bool (*Predicate) (const T &a);
```

Тоді цей тип можна вказати в якості параметру шаблону для нашої функції-фільтру (оскільки параметром шаблону може бути тип вказівників на функції):

```
template < class InputIterator,      // ітератор вхідного контейнера
          class OutputIterator,     // ітератор вихідного контейнера
          class Predicate            // предикат
        >
void copy_if( InputIterator first,
              InputIterator last,
              OutputIterator result,
              Predicate function
            )
{
```

```

        for ( ; first != last; ++first)
            if ( function(*first) )
            {
                *result = *first;
                ++result;
            }
        return;
    }
}

```

Предикат має один параметр – елемент контейнера. Всі дії та інші необхідні величини предикат інкапсулює в собі, наприклад:

```

int odd(const int &a)
{
    return (a%2);
}

```

– перевірка непарності аргументу.

Ще приклад:

```

bool negative(const int &a)
{
    return (a<0);
}

```

– цей предикат дозволяє відібрати від’ємні елементи контейнера.

Якщо потрібно відібрати лише ті елементи контейнера, значення яких більше 5, то предикат має бути наступним:

```

bool greater5(const int &a)
{
    return (a>5);
}

```

Такий код функції-фільтру `copy_if()` достатньо універсальний, проте за цю універсальність приходится платити ефективністю: виклик предикату виконується для кожного елемента контейнера – кожного разу формується стек параметрів, викликається функція-предикат, потім виконується повернення з неї. Причому функції-предикати не можна зробити `inline`-функціями, оскільки їх виклик – не прямий, а опосередкований (через вказівник на функцію). Код `inline`-функції підставляється в місце виклику, при цьому не витрачається час на заповнення стеку, виклик функції та повернення з неї. Компілятор не може підставити функцію, виклик якої є опосередкований – тобто, здійснюється за допомогою вказівника на функцію.

## Вказівники на функції та вказівники на методи

Існує ще один недолік вищенаведеної функції-фільтру `copy_if()`, який не дозволяє вважати її достатньо універсальною: ми не можемо передати такій функції-фільтру метод-предикат. Причина полягає в тому, що вказівник на метод суттєво відрізняється за типом від

звичайного вказівника на функцію, навіть якщо прототипи функції та метода виглядають однаково.

Вказівник на функцію визначається так:

```
тип_результату (*ім'я_вказівника) (список_параметрів);
```

Для скорочення запису зазвичай вводять нове ім'я типу:

```
typedef тип_результату (*тип_вказівника) (список_параметрів);
```

Після цієї команди вказівник на функцію можна оголошувати так:

```
тип_вказівника ім'я_вказівника;
```

Вказівнику присвоюється значення адреси функції:

```
ім'я_вказівника = &ім'я_функції;
```

Виклик функції за допомогою вказівника виконується так:

```
(*ім'я_вказівника) (аргументи);
```

або – в простішій формі – так:

```
ім'я_вказівника(аргументи);
```

Розглянемо приклади. Припустимо, є оголошення вказівника на функцію:

```
int (*pf) (void);
```

Цьому вказівнику можна присвоїти адресу будь-якої функції з таким ж самим прототипом, в т.ч. і адресу бібліотечної функції (наприклад, адресу функції `rand()`, яка генерує випадкові числа. Ця функція оголошена в файлі заголовку `<cstdlib>`):

```
int f1(void) { return 1; }
int f2(void) { return 2; }

int main()
{
    pf = &f1;    cout << pf() << endl;    // виклик f1()
    pf = &f2;    cout << pf() << endl;    // виклик f2()
    pf = &rand;  cout << pf() << endl;    // виклик rand()

    return 0;
}
```

Тепер розглянемо вказівники на методи.

Припустимо, є наступне визначення класу, який містить поле `x` цілого типу, статичне поле `s` цілого типу, конструктор ініціалізації, метод `getX()` – повертає значення поля `x` та статичний метод `getS()` – повертає значення статичного поля `s`:

```
class C
{
    int x;
    static int s;

public:
```



```

        C(const int &a): x(a) {}
        int getX(void) { return x; }
        static int getS(void) { return s; }
};

```

Метод `getX()` має такий ж самий прототип, що і функції `f1()`, `f2()` та `rand()`. Проте спроби присвоїти адресу метода `getX()` вказівнику `pf` відхиляються компілятором. Таким чином, тип вказівника на метод класу кардинально відрізняється від типу вказівника на функцію: адресу метода не можна присвоїти вказівнику на функцію, навіть якщо їх прототипи однакові. Це стає зрозумілим, якщо пригадати, що нестатичні методи отримують додатковий параметр – вказівник `this`.

Так само не можна присвоїти звичайному вказівнику на функцію адресу віртуального метода – в цьому випадку справа ускладнюється ще і наявністю в складі об'єкта вказівника на таблицю віртуальних методів класу.

А ось із статичними методами – все інакше! Статичний метод не отримує додаткових параметрів, тому його адресу можна присвоїти звичайному вказівнику на функцію без всяких перетворень:

```

#include <iostream>
#include <cstdlib>

using namespace std;

int odd(const int &a)
{
    return (a % 2);
}

int (*pf) (void);

int f1(void) { return 1; }
int f2(void) { return 2; }

class C
{
    int x;
    static int s;

public:
    C(const int &a): x(a) {}
    int getX(void) { return x; }
    static int getS(void) { return s; }
};

int C::s = 1;

int main()
{
    pf = &f1;      cout << pf() << endl;  // виклик f1()
    pf = &f2;      cout << pf() << endl;  // виклик f2()
    pf = &rand;    cout << pf() << endl;  // виклик rand()

    pf = &C::getS; cout << pf() << endl;  // виклик C::getS()
}

```

```
        return 0;
    }
```

Зрозуміло, що в команді

```
pf = &C::getS;
```

необхідно вказувати префікс.

Вказівник на метод оголошується інакше – слід вказати ім'я класу в якості префіксу:

```
int (C::*pm) (void);
```

Такому вказівнику можна присвоювати адреси звичайних та віртуальних методів, наприклад:

```
pm = &C::getX;
```

Адресу статичного метода та адресу звичайної функції такому вказівнику присвоїти не можна, бо виникне помилка компіляції.

Опосередкований виклик метода (через вказівник) виконується не так, як опосередкований виклик звичайної функції – за допомогою операції вибору елемента класу `.*` або `->*`. Такий виклик метода через вказівник на нього можливий лише за наявності об'єкта, наприклад:

```
C a(5); cout << (a.*pm)() << endl; // виклик C::getX()
```

Вираз `(a.*pm)()` означає: для об'єкта `a` викликати метод, адреса якого записана в вказівнику `pm`. Ліворуч від операції `.*` – об'єкт, праворуч – вказівник на метод. Дужки довкола виразу `a.*pm` записувати обов'язково, бо пріоритет операції виклику функції `()` вищий, ніж пріоритет операції вибору елемента класу `.*`.

У виразі `(об'єкт.*вказівник_на_метод)` можна замінити частину `об'єкт.` на `вказівник_на_об'єкт->`, наприклад:

```
C *pc = new C(7); cout << (pc->*pm)() << endl;
```

У виразі `(pc->*pm)` ліворуч – звичайний вказівник на динамічний об'єкт, а праворуч – вказівник на метод цього об'єкту. Це – різні типи вказівників!

## Поняття функтора

Одним із аргументом узагальненого алгоритму часто є деяка операція, яку необхідно виконати з кожним елементом контейнера. В якості параметра-операції можна задавати вказівник на функцію. Проте в узагальненому алгоритмі аргумент-операція має задаватися універсальним способом, тому параметр не може бути заданий у вигляді вказівника, бо

вказівники на методи відрізняються від вказівників на функції. Аргументи-операції в узагальнених алгоритмах зазвичай задаються як об'єкти-функтори.

Для забезпечення універсальності узагальненого алгоритму його параметр-операцію задають за допомогою *функтору*. *Функтор* – це об'єкт, що веде себе наче функція. Клас, в якому перевантажена операція `operator()` виклику функції, називається класом-функтором. Об'єкт такого класу називають *функтором*.

Операцію `operator` () можна визначати лише як метод класу. Операція `operator` () не може бути статичним методом. Операція `operator` () може бути віртуальним методом.

Наприклад, функтор, що перевіряє непарність свого аргументу, має такий вигляд:

```
class Odd
{
public:
    int operator() (const int& d)
    {
        return (d % 2);
    }
};
```

Порівняно із функцією зміни виглядають лише косметичними: ім'я функції замінили на `operator()`, а сама функція «обгорнута» в клас. Проте наслідки таких невеликих змін – колосальні – ми тепер можемо використовувати всі можливості об'єктно-орієнтованого підходу, зокрема:

- 1) реалізувати в класі будь-які необхідні методи і операції, в т.ч. перевантажити операцію `operator()` довільну кількість разів;
- 2) оголосити в класі будь-які поля для збереження стану функтора – в різні моменти часу функтор може мати різні стани, функтор можна ініціалізувати;
- 3) можна побудувати ієрархію функторів, в т.ч. і з віртуальними функціями – операція перевантаження виклику функції `operator()` теж може бути віртуальною.

На відміну від вказівників на функції, функтори можуть бути різного типу навіть при однакових прототипах операції `operator()` – для цього достатньо назвати класи по-різному. Це дозволяє уніфікувати програмування з використанням шаблонів, бо стає можливим передавати поведінку як параметр шаблону. Такий підхід використовується в реалізації патерну **Strategy** (*стратегія*) на етапі компіляції (цей патерн розглядається далі).

Виклик такого функтора виглядає як виклик звичайної функції:

```
Odd f;
... f(5) ...
```

## Функтори-предикати. Унарні та бінарні функтори

Функтор, який повертає булевий результат, називається *функтором-предикатом*. Функтор з одним аргументом називається *унарним функтором*; функтор з двома аргументами – *бінарним функтором*.

*Клас-функтор* – це звичайний клас, тому в ньому можуть бути визначені будь-які поля, методи, конструктори. Будь-які методи, в тому числі і перевантажені операції виклику функції (їх може бути кілька), можна оголосити віртуальними. Клас-функтор може бути учасником ієрархії успадковування, може вимагати динамічної пам'яті, може бути абстрактним. Приклад класу-функтора з полем і конструктором:

```
class GreaterEqual
{
    int value;

public:
    GreaterEqual(const int &v)
        : value(v) {}

    bool operator() (const int &d)
    {
        return (d >= value);
    }
};
```

За наявності конструктора ініціалізації виклик функтора можна реалізувати за допомогою виклику конструктора, наприклад:

```
GreaterEqual(3)
```

Зазвичай функтор оформлюється у вигляді шаблону класу, наприклад:

```
template <class T>
class GreaterEqual
{
    T value;

public:
    GreaterEqual(const T &v)
        : value(v) {}

    bool operator() (const T &d)
    {
        return (d >= value);
    }
};
```

Шаблон класу-функтора – це ще один спосіб реалізації шаблону функції, до якого можна застосовувати всі переваги шаблонів класів.

Реалізація узагальненого алгоритму з функтором показана в наступному прикладі – функція-фільтр, яка вибирає із послідовного контейнера певні елементи та копіює їх в інший контейнер:

```
template < class InputIterator,      // ітератор вхідного контейнера
           class OutputIterator,    // ітератор вихідного контейнера
           class Predicate          // клас функтора-предиката
         >
void copy_if( InputIterator first,
              InputIterator last,
              OutputIterator result,
              Predicate functor
            )
{
    for ( ; first != last; ++first)
        if ( functor(*first) )
        {
            *result = *first;
            ++result;
        }
    return;
}
```

Цей алгоритм копіює елементи, які задовольняють предикату functor, із заданого інтервалу вхідного контейнера у вихідний.

Вказаний алгоритм можна використовувати з вищенаведеними функторами:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
int b[10] = {0};
copy_if(a, a+10, b, Odd());
Odd f;
copy_if(a, a+10, b, f);
GreaterEqual five(5);
copy_if(a, a+10, b, five);
copy_if(a, a+10, b, GreaterEqual(6));
copy_if(a, a+10, b, GreaterEqual<int>(4));
```

## Адаптери функторів

Запропонована реалізація функтору GreaterEqual не є універсальною – вона розрахована (іншим словом, «адаптована») для використання функцією-фільтром copy\_if():

```
template <class T>
class GreaterEqual
{
    T value;

public:
    GreaterEqual(const T &v)
        : value(v) {}

    bool operator() (const T &d)
    {
        return (d >= value);
    }
};
```

«Адаптація» полягає в тому, що ми бінарний предикат «більше або дорівнює» записали у вигляді унарного, визначивши перший операнд як аргумент операції `operator()`, а другий – як поле класу, причому фіксуємо його значення при створенні об’єкту. Завдяки цьому при виклику функтора в тілі `copy_if()` йому передається лише один аргумент – елемент контейнера, другий – інкапсульований у функторі і його значення зафіксовано при конструюванні функтора.

Можна зробити інакше: написати універсальний бінарний функтор-предикат `GreaterEqual` і постаратися використати його в якості аргументу функції `copy_if()`:

```
template <class T> // універсальний функтор "більше або дорівнює"
class GreaterEqual
{
public:
    bool operator() (const T &left, const T &right) const
    {
        return (left >= right);
    }
};
```

Цей предикат ми можемо використовувати в інших узагальнених алгоритмах, наприклад, в алгоритмі сортування. Проте, його не можна безпосередньо використати у функції `copy_if()` – бо цей предикат має два аргументи.

Щоб позбутися одного з аргументів, слід написати клас-адаптер, в якому один з аргументів предикату буде задаватися як аргумент конструктора – тим самим ми перетворимо бінарний предикат в унарний. Адаптерів має бути два – різні для першого та другого аргументів:

```
template <class Predicate, class T> // фіксатор першого аргументу
class FirstArg
{
    Predicate op; // предикат
    T value; // фіксований аргумент

public:
    FirstArg(const Predicate &P, const T &left)
        : op(P), value(left) {}

    bool operator() (const T &right) const
    {
        return ( op(value, right) ); // виклик предикату
    }
};

template <class Predicate, class T> // фіксатор другого аргументу
class SecondArg
{
    Predicate op; // предикат
    T value; // фіксований аргумент

public:
    SecondArg(const Predicate &P, const T &right)
        : op(P), value(right) {}
};
```

```

        bool operator() (const T &left) const
        {
            return ( op(left, value) ); // виклик предикату
        }
};

```

Використовувати наведені шаблони можна таким чином:

```

#include <iostream>

using namespace std;

template <class T> // універсальний функтор "більше або дорівнює"
class GreaterEqual
{
public:
    bool operator() (const T &left, const T &right) const
    {
        return (left >= right);
    }
};

template <class Predicate, class T> // фіксатор першого аргументу
class FirstArg
{
    Predicate op; // предикат
    T value; // фіксований аргумент

public:
    FirstArg(const Predicate &P, const T &left)
        : op(P), value(left) {}

    bool operator() (const T &right) const
    {
        return ( op(value, right) ); // виклик предикату
    }
};

template <class Predicate, class T> // фіксатор другого аргументу
class SecondArg
{
    Predicate op; // предикат
    T value; // фіксований аргумент

public:
    SecondArg(const Predicate &P, const T &right)
        : op(P), value(right) {}

    bool operator() (const T &left) const
    {
        return ( op(left, value) ); // виклик предикату
    }
};

template < class InputIterator, // ітератор вхідного контейнера
          class OutputIterator, // ітератор вихідного контейнера
          class Predicate // клас функтора-предиката
        >
void copy_if( InputIterator first,
             InputIterator last,
             OutputIterator result,
             Predicate functor
            )

```

```

{
    for ( ; first != last; ++first)
        if ( functor(*first) )
        {
            *result = *first;
            ++result;
        }
    return;
}

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[10] = {0};

    copy_if(a, a+10, b,
            SecondArg<GreaterEqual<int>, int>( GreaterEqual<int>(), 4 ) );

    for (int i=0; i<10; ++i)
        cout << b[i] << " ";
    cout << endl;

    return 0;
}

```

Програма виводить на екран наступне:

```
4 5 6 7 8 9 10 0 0 0
```

Пояснення:

`SecondArg<GreaterEqual<int>, int>(...)` – виклик конструктора шаблону адаптера-фіксатора `SecondArg<>`, при якому шаблон інстанціюється наступними значеннями параметрів-типів (вказаних в кутових дужках):

`GreaterEqual<int>` – значення параметру `Predicate`;

`int` – значення параметру `T`.

Аргументи конструктора шаблону – вказані в круглих дужках:

`GreaterEqual<int>()` – аргумент, що відповідає параметру `const Predicate &P`, це – виклик конструктора функтора-предиката `GreaterEqual<int>`, при якому параметр-тип `T` отримує значення `int`;

`4` – значення параметру `const T &right`, – другого операнда функтора-предиката, який фіксується адаптером.

Очевидно, що використовувати такі шаблони – не зовсім зручно, бо доводиться при виклику `copy_if()` двічі вказувати предикат, – що робить аргументи функції-фільтра занадто громіздкими. Для зручності використання наших шаблонів напишемо функції-оболонки класів-фіксаторів. Параметри функції-шаблону будуть виводитися компілятором, і запис аргументів суттєво спроститься:



```

template <class Predicate, class T>
inline
FirstArg<Predicate, T> bindLeft(const Predicate &P, const T &left)
{
    T value(left);
    return ( FirstArg<Predicate, T>(P, value) );
}

template <class Predicate, class T>
inline
SecondArg<Predicate, T> bindRight(const Predicate &P, const T &right)
{
    T value(right);
    return ( SecondArg<Predicate, T>(P, value) );
}

```

Використовувати наведені функції-шаблони можна так:

```

#include <iostream>
using namespace std;

template <class T> // універсальний функтор "більше або дорівнює"
class GreaterEqual
{
public:
    bool operator() (const T &left, const T &right) const
    {
        return (left >= right);
    }
};

template <class Predicate, class T> // фіксатор першого аргументу
class FirstArg
{
    Predicate op; // предикат
    T value; // фіксований аргумент

public:
    FirstArg(const Predicate &P, const T &left)
        : op(P), value(left) {}

    bool operator() (const T &right) const
    {
        return ( op(value, right) ); // виклик предикату
    }
};

template <class Predicate, class T> // фіксатор другого аргументу
class SecondArg
{
    Predicate op; // предикат
    T value; // фіксований аргумент

public:
    SecondArg(const Predicate &P, const T &right)
        : op(P), value(right) {}

    bool operator() (const T &left) const
    {
        return ( op(left, value) ); // виклик предикату
    }
};

```

```

template < class InputIterator,      // ітератор вхідного контейнера
          class OutputIterator,     // ітератор вихідного контейнера
          class Predicate           // клас функтора-предиката
        >
void copy_if( InputIterator first,
             InputIterator last,
             OutputIterator result,
             Predicate functor
            )
{
    for ( ; first != last; ++first)
        if ( functor(*first) )
        {
            *result = *first;
            ++result;
        }
    return;
}

template <class Predicate, class T>
inline
FirstArg<Predicate, T> bindLeft(const Predicate &P, const T &left)
{
    T value(left);
    return ( FirstArg<Predicate, T>(P, value) );
}

template <class Predicate, class T>
inline
SecondArg<Predicate, T> bindRight(const Predicate &P, const T &right)
{
    T value(right);
    return ( SecondArg<Predicate, T>(P, value) );
}

int main()
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[10] = {0};

    copy_if(a, a+10, b, bindRight( GreaterEqual<int>(), 4 ) );

    for (int i=0; i<10; ++i)
        cout << b[i] << " ";
    cout << endl;

    return 0;
}

```

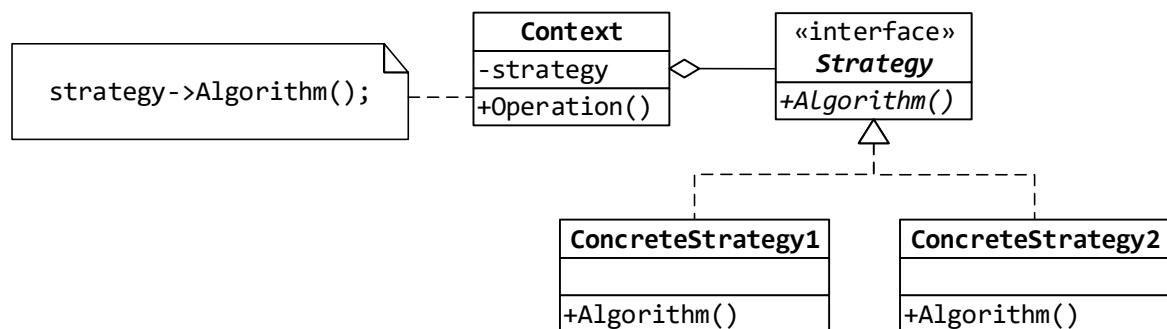
Питання: який саме патерн тут був використаний: *адаптер об'єктів* (Тема 04. Успадковування, с. 24-30) чи *адаптер класів* (Тема 04. Успадковування, с. 35) ?

## Патерн Strategy (стратегія)

Патерн **Strategy** (*стратегія*) визначає сімейство алгоритмів, інкапсулює кожний з них та робить їх взаємозамінними. Патерн *Стратегія* дозволяє модифікувати алгоритми незалежно від їх використання на стороні клієнтів.

## Реалізація за допомогою інтерфейсів (абстрактних класів)

Інтерфейси класів `Strategy` та `Context` можуть забезпечити об'єкту класу `ConcreteStrategy` ефективний доступ до будь-яких даних контексту і навпаки.



**Strategy** – стратегія:

- оголошує загальний інтерфейс для сімейства алгоритмів. Клас `Context` використовує цей інтерфейс для виклику конкретного алгоритму, визначеного в класі `ConcreteStrategy`;

**ConcreteStrategy** – конкретна стратегія:

- реалізує алгоритм, який використовує інтерфейс, оголошений в класі `Strategy`;

**Context** – контекст:

- конфігурується об'єктом класу `ConcreteStrategy`;
- містить вказівник на об'єкт класу `Strategy`;
- може визначати інтерфейс, який дозволяє об'єкту `Strategy` отримати доступ до даних контексту.

Класи `Strategy` та `Context` взаємодіють для реалізації вибраного алгоритму. Контекст може передавати стратегії всі необхідні алгоритму дані в момент його виклику. Також контекст може дозволити звертатися до своїх операцій, передавши посилання на самого себе операціям класу `Strategy`.

Контекст переадресовує (делегує) запити своїх клієнтів об'єкту-стратегії. Зазвичай клієнт створює об'єкт `ConcreteStrategy` та передає його контенту, після чого клієнт «спілкується» виключно з контентом. Часто клієнт в своєму розпорядженні має кілька класів `ConcreteStrategy`, з яких він може вибирати потрібний:

```
#include <iostream>

using namespace std;

class Strategy
{
public:
    virtual void Algorithm()=0;
};
```

```

class Context
{
    Strategy *strategy;

public:
    Context(Strategy *s): strategy(s) {}

    void setStrategy(Strategy *s)
    {
        strategy = s;
    }

    void Operation()
    {
        strategy->Algorithm();
    }
};

class ConcreteStrategy1
    : public Strategy
{
    virtual void Algorithm()
    {
        cout << "ConcreteStrategy1::Algorithm()" << endl;
    }
};

class ConcreteStrategy2
    : public Strategy
{
    virtual void Algorithm()
    {
        cout << "ConcreteStrategy2::Algorithm()" << endl;
    }
};

int main()
{
    ConcreteStrategy1 *cs1 = new ConcreteStrategy1();
    ConcreteStrategy2 *cs2 = new ConcreteStrategy2();

    Context c(cs1);
    c.Operation();

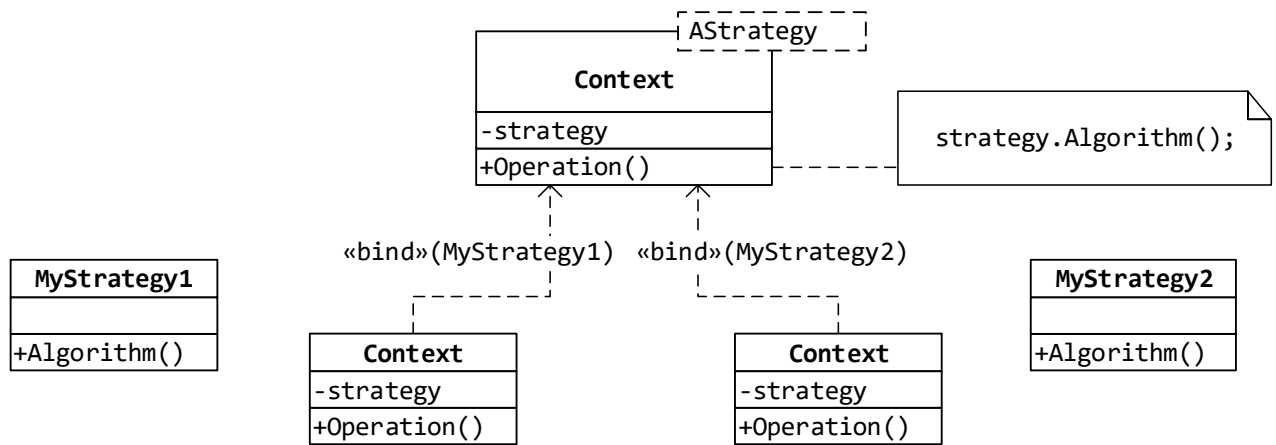
    c.setStrategy(cs2);
    c.Operation();

    return 0;
}

```

### Реалізація за допомогою шаблонів

В мові C++ для конфігурування класу Context стратегією можна використовувати шаблони. Цей спосіб використовується лише тоді, коли стратегія визначається ще на етапі компіляції і її не потрібно змінювати під час виконання. Тоді клас `Context`, який буде конфігуруватися, визначається у вигляді шаблону, для якого клас `Strategy` є параметром. Шаблон класу `Context` конфігурується класом `Strategy` в момент інстанціювання:



При використанні шаблонів відпадає необхідність в абстрактному класі для визначення інтерфейсу `Strategy`. Передавання стратегії у вигляді параметру шаблону дозволяє статично зв'язати стратегію з контекстом, що збільшує швидкодію програми.

```
#include <iostream>

using namespace std;

template <class AStrategy>
class Context
{
    AStrategy strategy;

public:
    void Operation()
    {
        strategy.Algorithm();
    }
};

class MyStrategy1
{
public:
    void Algorithm()
    {
        cout << "MyStrategy1::Algorithm()" << endl;
    }
};

class MyStrategy2
{
public:
    void Algorithm()
    {
        cout << "MyStrategy2::Algorithm()" << endl;
    }
};

int main()
{
    Context<MyStrategy1> c1;
    c1.Operation();

    Context<MyStrategy2> c2;
```

```
        c2.Operation();  
        return 0;  
    }
```

## Підсумок

В мові C++ крім *шаблонів класів* реалізовані *шаблони функцій*, – це подальший розвиток механізму перевантаження функцій. Порівняно з шаблонами класів, шаблони функцій мають багато обмежень. Наприклад, не можна задавати параметри шаблону функції за умовчанням, проте працює автоматичне виведення параметрів (хоч і з деякими обмеженнями). Шаблон функції можна спеціалізувати, проте часткова спеціалізація не допускається. Шаблон функції можна перевантажити як іншим шаблоном, так і функцією. Всі такі обмеження можна «обійти», якщо «обгорнути» функцію в клас-шаблон.

*Шаблонна функція* – це функція, утворена в результаті *інстанціювання* шаблону функції.

Шаблони дають можливість визначити за допомогою одного фрагменту коду цілий набір взаємопов'язаних функцій (перевантажених), які називаються *шаблонними функціями*.

Щоб використати шаблони функцій, програміст має написати лише один опис шаблону функції. На основі типів аргументів, які використовуються при виклику цієї функції, компілятор буде автоматично генерувати об'єктні коди функцій, які опрацьовують кожний тип даних. Вони компілюються разом з іншими частинами тексту програми.

Всі описи шаблонів функцій починаються з ключового слова `template`, за яким йде список формальних параметрів шаблону, записаний в кутових дужках (< та >); кожному формальному параметру шаблону має передувати ключове слово `class` або `typename`. Ключове слово `class` або `typename`, яке використовується при визначенні типів параметрів шаблону функції, означає «будь-який тип, або тип, що визначається користувачем».

Формальні параметри в описі шаблону використовуються для визначення типів параметрів функції, типу результату функції та типів локальних змінних, визначених в тілі функції.

Сам шаблон функції можна перевантажити кількома способами. Можна визначити інші шаблони, що матимуть те саме ім'я функції, але з різними наборами параметрів. Шаблон функції також можна перевантажити, якщо ввести іншу нешаблонну функцію з тим же самим іменем, але іншим набором параметрів.

Шаблони функцій дозволяють реалізувати *узагальнені алгоритми*, які можуть працювати з *контейнерами* будь-якого виду. В якості параметрів в узагальненому алгоритмі виступають *ітератори*. Одним із аргументом узагальненого алгоритму часто є деяка

операція, яку необхідно виконати з кожним елементом контейнера. Зазвичай в якості параметра-операції задається вказівник на функцію. Проте в узагальненому алгоритмі аргумент-операція має задаватися універсальним способом, тому параметр не може бути заданий у вигляді вказівника, бо вказівники на методи відрізняються від вказівників на функції. Аргументи-операції в узагальнених алгоритмах зазвичай задаються як об'єкти-функтори. *Функтор* – це клас, в якому перевантажена операція виклику функції `operator()`. Використання функторів дозволяє зробити алгоритм насправді універсальним. Класи-оболонки для вказівників на функції та вказівників на методи, які називають *адаптерами*, збільшують ступінь універсальності узагальненого алгоритму. Один із важливих адаптерів – *фіксатор*, який дозволяє перетворити бінарний функтор в унарний.

Патерн **Strategy** (*стратегія*) визначає сімейство алгоритмів, інкапсулює кожний з них та робить їх взаємозамінними. Патерн *Стратегія* дозволяє змінювати алгоритми незалежно від клієнтів, які використовують ці алгоритми.

# Лабораторний практикум

## Оформлення звіту про виконання лабораторних робіт

### Вимоги до оформлення звіту про виконання лабораторних робіт №№ 6.1–6.7

Звіт про виконання лабораторних робіт №№ 6.1–6.7 має містити наступні елементи:

- 1) заголовок;
- 2) мету роботи;
- 3) умову завдання;

*Умова завдання має бути вставлена у звіт як фрагмент зображення (скрін) сторінки посібника.*

- 4) UML-діаграму класів;
- 5) структурну схему програми;

*Структурна схема програми зображує взаємозв'язки програми та всіх її програмних одиниць: схему вкладеності та охоплення підпрограм, програми та модулів; а також схему звертання одних програмних одиниць до інших.*

- 6) текст програми;

*Текст програми має бути правильно відформатований: відступами і порожніми рядками слід відображати логічну структуру програми; програма має містити необхідні коментарі – про призначення підпрограм, змінних та параметрів – якщо їх імена не значущі, та про призначення окремих змістовних фрагментів програми. Текст програми слід подавати моноширинним шрифтом (Courier New розміром 10 пт. або Consolas розміром 9,5 пт.) з одинарним міжрядковим інтервалом;*

- 7) посилання на git-репозиторій з проектом (див. інструкції з Лабораторної роботи № 2.2 з предмету «Алгоритмізація та програмування»);
- 8) хоча б для одної функції, яка повертає результат (як результат функції чи як параметр-посилання) – результати unit-тесту: текст програми unit-тесту та скрін результатів її виконання (див. інструкції з Лабораторної роботи № 5.6 з предмету «Алгоритмізація та програмування»);
- 9) висновки.



## **Зразок оформлення звіту про виконання лабораторних робіт №№ 6.1–6.7**

### **ЗВІТ**

про виконання лабораторної роботи № < номер >

« назва теми лабораторної роботи »

з дисципліни

«Об'єктно-орієнтоване програмування»

студента(ки) групи КН-26

< Прізвище Ім'я По\_батькові >

#### **Мета роботи:**

...

#### **Умова завдання:**

...

#### **UML-діаграма класів:**

...

#### **Структурна схема програми:**

...

#### **Текст програми:**

...

#### **Посилання на git-репозиторій з проектом:**

...

#### **Результати unit-тесту:**

...

#### **Висновки:**

...

# Лабораторна робота № 6.1. Контейнери як параметри

## Мета роботи

Освоїти створення і використання контейнерів.

## Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення контейнера.
- 2) Поняття та призначення ітератора.
- 3) Види контейнерів.
- 4) Прямий доступ до елементів.
- 5) Послідовний доступ до елементів.
- 6) Асоціативний доступ до елементів.
- 7) Метод begin().
- 8) Метод end().
- 9) Операції контейнера.

## Приклад

### Лістинг 2. Числовий контейнер змінної довжини

[9 – с. 88-90]

#### файл «Array.h»

```
#pragma once
#include <iostream>
using namespace std;

class Array
{
public:
    // типи
    typedef unsigned int    UINT;
    typedef double          value_type;
    typedef double*         iterator;
    typedef const double*   const_iterator;
    typedef double&         reference;
    typedef const double&   const_reference;
    typedef std::size_t     size_type;

private:
    static const size_type minsize = 10;    // мінімальний розмір масиву
    size_type Size;                        // виділено пам'яті для елементів
    size_type Count;                       // кількість елементів в масиві
    size_type First;                       // значення індексу першого елемента в масиві
    value_type* elems;                     // вказівник на дані

public:
```

```

// конструктори/копіювання/деструктор
Array(const size_type& n = minsize)
    throw(bad_alloc, invalid_argument);
Array(const Array&) throw(bad_alloc);
Array(const iterator first, const iterator last)
    throw(bad_alloc, invalid_argument);
Array(const size_type first, const size_type last)
    throw(bad_alloc, invalid_argument);
~Array();
Array& operator=(const Array&);

// ітератори
iterator begin() { return elems; }
const_iterator begin() const { return elems; }
iterator end() { return elems + Count; }
const_iterator end() const { return elems + Count; }

// розміри
size_type size() const;           // поточний розмір
bool empty() const;              // якщо є елементи
size_type capacity() const;      // потенційний розмір
void resize(size_type newsize)    // змінити розмір
    throw(bad_alloc);

// доступ до елементів
reference operator[](size_type) throw(out_of_range);
const_reference operator[](size_type) const throw(out_of_range);
reference front() { return elems[0]; }
const_reference front() const { return elems[0]; }
reference back() { return elems[size() - 1]; }
const_reference back() const { return elems[size() - 1]; }

// методи-модифікатори
void push_back(const value_type& v); // додати елемент в кінець
void pop_back();                    // видалити останній елемент – реалізувати самостійно
void clear() { Count = 0; }         // очистити масив
void swap(Array& other);            // поміняти з другим масивом
void assign(const value_type& v);    // заповнити масив – реалізувати самостійно

// дружні функції вводу/виводу
friend ostream& operator <<(ostream& out, const Array& a);
friend istream& operator >>(istream& in, Array& a);
};

```

### файл «Array.cpp»

```

#include "Array.h"
#include <iostream>
#include <stdexcept>
#include <exception>
using namespace std;

Array::Array(const Array::size_type& n)
    throw(bad_alloc, invalid_argument)
{
    First = 0;
    Count = Size = n;
    elems = new value_type[Size];
    for (UINT i = 0; i < Size; i++)
        elems[i] = 0;
}

```

```

Array::Array(const iterator first, const iterator last)
    throw(bad_alloc, invalid_argument)
{
    First = 0;
    if (first <= last) {
        Count = Size = (last - first) + 1;
        elems = new value_type[Size];
        for (UINT i = 0; i < Size; ++i)
            elems[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const size_type first, const size_type last)
    throw(bad_alloc, invalid_argument)
{
    if (first <= last) {
        First = first;
        Count = Size = (last - first) + 1;
        elems = new value_type[Size];
        for (UINT i = 0; i < Size; ++i)
            elems[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const Array& t) throw(bad_alloc)
    : Size(t.Size), Count(t.Count), First(t.First), elems(new value_type[Size])
{
    for (UINT i = 0; i < Size; ++i)
        elems[i] = t.elems[i];
}

Array& Array::operator =(const Array& t)
{
    Array tmp(t);
    swap(tmp);
    return *this;
}

Array::~~Array()
{
    delete[]elems;
    elems = 0;
}

void Array::push_back(const value_type& v)
{
    if (Count == Size)           // місця нема
        resize(Size * 2);       // збільшили "місткість"
    elems[Count++] = v;         // присвоїли
}

Array::reference Array::operator [](size_type index) throw(out_of_range)
{
    if ((First <= index) && (index < First + Size))
        return elems[index - First];
    else
        throw out_of_range("Index out of range!");
}

```

```

Array::const_reference Array::operator [] (size_type index) const
    throw(out_of_range)
{
    if ((First <= index) && (index < First + Size))
        return elems[index - First];
    else
        throw out_of_range("Index out of range!");
}

void Array::resize(size_type newsize) throw(bad_alloc)
{
    if (newsize > capacity())
    {
        value_type* data = new value_type[newsize];
        for (size_type i = 0; i < Count; ++i)
            data[i] = elems[i];

        delete[] elems;
        elems = data;
        Size = newsize;
    }
}

void Array::swap(Array& other)
{
    std::swap(elems, other.elems);           // стандартна функція обміну
    std::swap(Size, other.Size);
}

Array::size_type Array::capacity() const
{
    return Size;
}

Array::size_type Array::size() const
{
    return Count;
}

bool Array::empty() const
{
    return Count == 0;
}

ostream& operator <<(ostream& out, const Array& tmp)
{
    for (size_t j = 0; j < tmp.Count; j++)
        out << tmp[j] << " ";
    out << endl;
    return out;
}

istream& operator >>(istream& in, Array& tmp)
{
    // тут має бути введення елементів масиву!
    return in;
}

```

файл «Source.cpp»

```

// Lab_6_2.cpp : Defines the entry point for the console application.
//

#include "Array.h"
#include <iostream>
#include <time.h>
#include <stdexcept>
#include <exception>
#include <cmath>

using namespace std;

typedef Array::value_type* TArray;

int main()
{
    int n;
    cout << "n= "; cin >> n;
    Array c = Array(2 * n);

    srand((unsigned)time(NULL));
    Array::value_type A = -10;
    Array::value_type B = 10;

    TArray a = new Array::value_type[n];

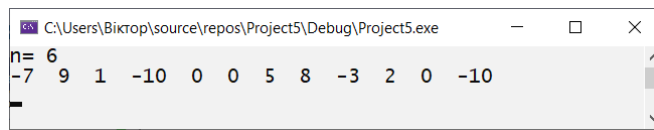
    for (int i = 0; i < 2 * n; i++)
        a[i] = A + rand() % int(B - A + 1);

    Array::iterator l = const_cast<Array::iterator>(c.begin());
    for (int j = 0; j < 2 * n; j++, l++)
        *l = a[j];
    cout << c;

    cin.get();
    cin.get();
    return 0;
}

```

**Результат виконання:**



```

C:\Users\Bikrop\source\repos\Project5\Debug\Project5.exe
n= 6
-7 9 1 -10 0 0 5 8 -3 2 0 -10

```

## **Варіанти завдань**

У всіх завданнях написати функцію, що приймає звичайний масив як аргумент і повертає динамічний масив зростаючої довжини (див. лістинг 2), як результат. Розмір звичайного масиву передається як аргумент. Початковий масив заповнити випадковими числами в діапазоні від  $-50$  до  $+50$ . Додати в масив-результат суму і середнє арифметичне за абсолютною величиною.

### **Варіант 1.**

Додати до кожного числа корінь квадратний з модуля добутку максимуму та останнього числа.

### **Варіант 2.**

Розділити кожне число на пів-суму першого від'ємного і 50-го числа.

### **Варіант 3.**

Відняти від кожного числа найбільше число (далі «максимум»).

### **Варіант 4.**

Помножити кожне число на мінімальне.

### **Варіант 5.**

Поділити всі непарні числа на середнє арифметичне за абсолютною величиною.

### **Варіант 6.**

Відняти від кожного числа суму чисел.

### **Варіант 7.**

Помножити кожне третє число на подвоєну суму першого і останнього від'ємних чисел.

### **Варіант 8.**

Додати до кожного числа абсолютну величину першого непарного числа.

### **Варіант 9.**

Помножити кожне парне число на перше від'ємне число.

### **Варіант 10.**

Додати до кожного числа половину останнього від'ємного числа.

### **Варіант 11.**

Поділити кожне число на половину найбільшого числа.

### **Варіант 12.**

Замінити всі нулі середнім арифметичним.

### **Варіант 13.**

Замінити всі додатні числа квадратом найменшого числа.

### **Варіант 14.**

Додати до кожного числа пів-суму всіх від'ємних чисел.

### **Варіант 15.**

Додати до кожного числа середнє найменшого і найбільшого за абсолютною величиною.

### **Варіант 16.**

Кожне число поділити на найменше число і додати найбільше число.

### **Варіант 17.**

Замінити всі додатні числа на найбільше число.

### **Варіант 18.**

Замінити кожне парне число на різницю найбільшого і найменшого за абсолютною величиною чисел.

### **Варіант 19.**

Замінити кожне друге від'ємне число половиною найбільшого числа.

### **Варіант 20.**

Замінити кожне третє додатне число середнім арифметичним.

### **Варіант 21.**

Додати до кожного числа суму модуля добутку максимуму і останнього числа.



**Варіант 22.**

Розділити кожне число на середнє арифметичне модулів першого від'ємного і 50-го числа.

**Варіант 23.**

Відняти від кожного числа квадрат найбільшого числа.

**Варіант 24.**

Помножити кожне число на модуль мінімального.

**Варіант 25.**

Поділити всі непарні числа на абсолютну величину середнього арифметичного модулів всіх чисел.

**Варіант 26.**

Додати до кожного числа корінь квадратний з модуля добутку максимуму та останнього числа.

**Варіант 27.**

Розділити кожне число на пів-суму першого від'ємного і 50-го числа.

**Варіант 28.**

Відняти від кожного числа найбільше число (далі «максимум»).

**Варіант 29.**

Помножити кожне число на мінімальне.

**Варіант 30.**

Поділити всі непарні числа на середнє арифметичне за абсолютною величиною.

**Варіант 31.**

Відняти від кожного числа суму чисел.

**Варіант 32.**

Помножити кожне третє число на подвоєну суму першого і останнього від'ємних чисел.

**Варіант 33.**

Додати до кожного числа абсолютну величину першого непарного числа.

**Варіант 34.**

Помножити кожне парне число на перше від'ємне число.

**Варіант 35.**

Додати до кожного числа половину останнього від'ємного числа.

**Варіант 36.**

Поділити кожне число на половину найбільшого числа.

**Варіант 37.**

Замінити всі нулі середнім арифметичним.

**Варіант 38.**

Замінити всі додатні числа квадратом найменшого числа.

**Варіант 39.**

Додати до кожного числа пів-суму всіх від'ємних чисел.

**Варіант 40.**

Додати до кожного числа середнє найменшого і найбільшого за абсолютною величиною.

## Лабораторна робота № 6.2. Контейнери-масиви

### Мета роботи

Освоїти створення і використання контейнерів-масивів.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення контейнера.
- 2) Поняття та призначення ітератора.
- 3) Види контейнерів.
- 4) Прямий доступ до елементів.
- 5) Послідовний доступ до елементів.
- 6) Асоціативний доступ до елементів.
- 7) Метод begin().
- 8) Метод end().
- 9) Операції контейнера.

### Приклад

#### Варіант 0.

Площа плоского многокутника з вершинами в точках  $(x_1, y_1), \dots, (x_n, y_n)$ , де

$$x_{n+1} = x_1, y_{n+1} = y_1,$$

обчислюється за формулою

$$S = \left| \sum_{i=1}^n \frac{(x_i - x_{i+1})(y_{i+1} + y_i)}{2} \right|$$

Реалізувати на базі контейнера фіксованої довжини клас Polygon з методами обчислення периметра і площі многокутника.

### Лістинг 1. Числовий контейнер фіксованої довжини

[9 – с. 84-87]

#### файл «Array.h»

```
#pragma once
#include <iostream>

using namespace std;

class Array
{
public:
```

```

// типи
typedef unsigned int UINT;
typedef int* iterator;
typedef const int* const_iterator;
typedef int& reference;
typedef const int& const_reference;
typedef int value_type;
typedef size_t size_type;
typedef value_type* TArray;

private:
    static const size_type minsize = 10;
    UINT size_array;
    value_type* data;
    UINT Count;
    size_type First;

public:
    // конструктори
    Array(const size_type n = minsize);
    Array(const Array&) throw(bad_alloc);
    Array(const iterator first, const iterator last)
        throw(bad_alloc, invalid_argument);
    Array(const size_type first, const size_type last)
        throw(bad_alloc, invalid_argument);
    ~Array();
    Array& operator =(const Array& rhs);

    // операції індексування
    reference operator[](UINT);
    const_reference operator[](UINT) const;

    // дружні функції вводу/виводу
    friend ostream& operator <<(ostream& out, const Array& a);
    friend istream& operator >>(istream& in, Array& a);

    // розмір
    UINT size() const;

    // ітератори
    iterator begin() { return data; }
    const_iterator begin() const { return data; }
    iterator end() { return data + Count; }
    const_iterator end() const { return data + Count; }

    // Лабораторна робота 6.2 – добавлено методи до контейнера
    double Sum();
    double Perymetr();
};

```

### файл «Array.cpp»

```

#include "Array.h"
#include <iostream>
#include <stdexcept>
#include <exception>
#include <math.h>
#include <time.h>

using namespace std;

```

```

Array::Array(const Array::size_type n)
{
    First = 0;
    Count = size_array = n;
    data = new value_type[size_array];
    for (UINT i = 0; i < size_array; i++)
        data[i] = 0;
}

Array::Array(const iterator first, const iterator last)
    throw(bad_alloc, invalid_argument)
{
    First = 0;
    if (first <= last)
    {
        Count = size_array = (last - first) + 1;
        data = new value_type[size_array];
        for (UINT i = 0; i < size_array; ++i)
            data[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const size_type first, const size_type last)
    throw(bad_alloc, invalid_argument)
{
    if (first <= last)
    {
        First = first;
        Count = size_array = (last - first) + 1;
        data = new value_type[size_array];
        for (UINT i = 0; i < size_array; ++i)
            data[i] = 0;
    }
    else
        throw invalid_argument("!!!");
}

Array::Array(const Array& t) throw(bad_alloc)
    : size_array(t.size_array),
      Count(t.Count),
      First(t.First),
      data(new value_type[size_array])
{
    for (UINT i = 0; i < size_array; ++i)
        data[i] = t.data[i];
}

Array::~~Array()
{
    delete[] data;
    data = 0;
}

Array& Array::operator =(const Array& rhs)
{
    if (this != &rhs)
    {
        size_array = rhs.size_array;
        Count = rhs.Count;
        First = rhs.First;
    }
}

```

```

        value_type* new_data = new value_type[size_array];

        for (UINT i = 0; i < size_array; ++i)
            new_data[i] = rhs.data[i];
        delete[] data;
        data = new_data;
    }
    return *this;
}

Array::reference Array::operator[](UINT index) throw(out_of_range)
{
    if (index < size_array)
        return data[index];
    else
        throw out_of_range("Index out of range!");
}

Array::const_reference Array::operator[](UINT index) const throw(out_of_range)
{
    if (index < size_array)
        return data[index];
    else
        throw out_of_range("Index out of range!");
}

Array::UINT Array::size() const
{
    return size_array;
}

ostream& operator <<(ostream& out, const Array& tmp)
{
    for (size_t j = 0; j < tmp.size_array; j++)
        cout << tmp[j] << " ";
    cout << endl;
    return out;
}

istream& operator >>(istream& in, const Array& tmp)
{
    // тут має бути введення елементів масиву!
    return in;
}

double Array::Sum()
{
    value_type x_i1, x_i2, y_i1, y_i2;
    Array::iterator l = const_cast<Array::iterator>(this->begin());

    double S = 0;
    x_i1 = *l;
    l++;
    y_i1 = *l;
    l++;
    while (l < this->end())
    {
        x_i2 = *l;
        l++;
        y_i2 = *l;
        l++;
        S += abs(((1.0 * x_i1 - 1.0 * x_i2) * (1.0 * y_i1 + 1.0 * y_i2)) / 2.0);
    }
}

```

```

        x_i1 = x_i2;
        y_i1 = y_i2;
    }
    l = const_cast<Array::iterator>(this->begin());
    x_i2 = *l;
    l++;
    y_i2 = *l;
    S += abs(((1.0 * x_i1 - 1.0 * x_i2) * (1.0 * y_i1 + 1.0 * y_i2)) / 2.0);

    return S;
}

double Array::Perymetr()
{
    value_type x_i1, x_i2, y_i1, y_i2;
    Array::iterator l = const_cast<Array::iterator>(this->begin());

    double P = 0;
    x_i1 = *l;
    l++;
    y_i1 = *l;
    l++;
    while (l < this->end())
    {
        x_i2 = *l;
        l++;
        y_i2 = *l;
        l++;

        P += sqrt((1.0 * x_i2 - 1.0 * x_i1) * (1.0 * x_i2 - 1.0 * x_i1) +
                   (1.0 * y_i2 - 1.0 * y_i1) * (1.0 * y_i2 - 1.0 * y_i1));
        x_i1 = x_i2;
        y_i1 = y_i2;
    }
    l = const_cast<Array::iterator>(this->begin());
    x_i2 = *l;
    l++;
    y_i2 = *l;

    P += sqrt((1.0 * x_i2 - 1.0 * x_i1) * (1.0 * x_i2 - 1.0 * x_i1) +
               (1.0 * y_i2 - 1.0 * y_i1) * (1.0 * y_i2 - 1.0 * y_i1));

    return P;
}

```

### файл «Source.cpp»

```

// Lab_6_2.cpp : Defines the entry point for the console application.
//

#include "Array.h"
#include <iostream>
#include <time.h>
#include <stdexcept>
#include <exception>
#include <cmath>

using namespace std;

typedef Array::value_type* TArray;

```

```

int main()
{
    int n;
    cout << "n= "; cin >> n;
    Array c = Array(2 * n);

    srand((unsigned)time(NULL));
    Array::value_type A = -10;
    Array::value_type B = 10;

    Array::TArray a = new Array::value_type[n];

    for (int i = 0; i < 2 * n; i++)
        a[i] = A + rand() % (B - A + 1);

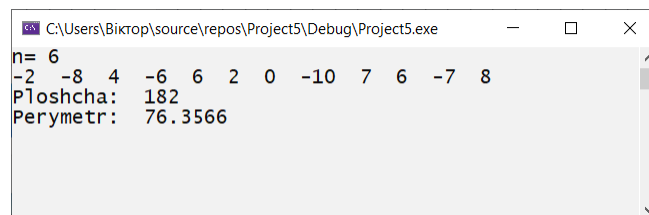
    Array::iterator l = const_cast<Array::iterator>(c.begin());
    for (int j = 0; j < 2 * n; j++, l++)
        *l = a[j];
    cout << c;

    cout << "Ploshcha: " << c.Sum() << endl;
    cout << "Perymetr: " << c.Perymetr() << endl;

    cin.get();
    cin.get();
    return 0;
}

```

### Результат виконання:



```

C:\Users\Bikrop\source\repos\Project5\Debug\Project5.exe
n= 6
-2 -8 4 -6 6 2 0 -10 7 6 -7 8
Ploshcha: 182
Perymetr: 76.3566

```



## Варіанти завдань

У всіх завданнях обов'язково мають бути реалізовані:

- конструктори без аргументів і
- конструктори ініціалізації, зокрема
- конструктор з двома аргументами – ітераторами,
- конструктор копіювання,
- деструктор,
- операція присвоєння,
- операції вводу-виводу.

Відповідні операції реалізуються як методи класу, а останні – як зовнішні дружні функції.

Має бути реалізоване **опрацювання виняткових ситуацій** із-за нестачі пам'яті: всі конструктори зобов'язані мати специфікацію виняткової ситуації `bad_alloc`.

У завданнях 1-10, 25-34 використовувати як зразок числовий масив фіксованого розміру (див. лістинг 1). Розміри масиву потрібно задавати в конструкторі. Обов'язково мають бути реалізовані:

- операція присвоєння і
- відповідні варіанту завдання операції з присвоєнням;
- операція індексування `[]` має перевіряти індекс на допустимість і генерувати **виняткову ситуацію** у випадку помилки;
- операції з двома масивами мають перевіряти, чи однакові їх розміри і генерувати **виняткову ситуацію** у випадку помилки.

У завданнях варіантів 11-24, 35-40 використовувати динамічне виділення пам'яті для масиву. Масив – змінного розміру (див. лістинг 2); початковий розмір задавати в конструкторі. З операцій, що змінюють розмір масиву, реалізувати тільки додавання і видалення елементу в кінці масиву. Реалізувати операцію індексування `operator[]`. Обов'язково мають бути підтримані операція присвоєння і відповідні завданню операції з присвоєнням. Відповідні операції реалізувати як дружні функції. Реалізувати конструктор ініціалізації літерним рядком, операцію перетворення в рядок і операції вводу/виводу. У тих завданнях, де це доцільно, ввести функцію перетворення в `double`.

### Варіант 1.

Створити клас `Vector` для роботи з векторами розмірності, що задається. Мають бути

реалізовані: операції додавання і віднімання векторів, скалярний добуток векторів, множення і ділення на скаляр, обчислення евклідової норми, порівняння норм векторів. (Евклідова норма обчислюється як квадратний корінь з суми квадратів координат.)

## Варіант 2.

Масив  $(y_0, y_1, \dots, y_n)$  є значеннями деякої функції на відрізку  $[a, b]$ , причому  $a = y_0$ ,  $b = y_n$ . Створити клас **Integral**, в якому реалізовано обчислення певного інтеграла методом прямокутників, методом трапецій і методом Сімпсона.

## Варіант 3.

Використовуючи Лістинг 1 як зразок, реалізувати клас **Agguy** – масив чисел з повним набором арифметичних операцій. Реалізувати операції перевірки на рівність і нерівність.

## Варіант 4.

Масив пар дійсних чисел  $(x_1, p_1), \dots, (x_n, p_n)$  є набором значень дискретної випадкової величини. Створити клас **Rand**, в якому реалізувати методи обчислення математичного очікування і дисперсії. Реалізувати метод обчислення інтегралу для заданої функції  $f(x)$  методом Монте-Карло.

## Варіант 5.

Створити клас **Agguy** – одновимірний масив дійсних чисел з межами індексів, що задаються, з можливістю задати від'ємні індекси. Обов'язково мають бути реалізовані: відстеження кількості елементів, множення і ділення на скаляр, пошук максимуму та мінімуму, обчислення суми та середнього арифметичного елементів, по-елементне додавання та віднімання масивів.

## Варіант 6.

Створити клас **Agguy** – одновимірний масив цілих чисел з межами індексів, що задаються, з можливістю визначення від'ємних індексів. Обов'язково мають бути реалізовані: відстеження кількості елементів, всі операції з масивом і цілим числом, пошук заданого елементу, всі по-елементні операції, реалізовані в C++ для цілих чисел.

## Варіант 7.

Створити клас **BitString** для роботи з бітовими рядками. Бітовий рядок повинен бути представлений масивом типу **unsigned char**, кожен елемент якого приймає значення 0 або 1. Молодший біт має молодший індекс. Реалізувати всі операції для роботи з бітовими рядками:

and, or, xor, not.

### **Варіант 8.**

Створити клас `Array` – одновимірний масив. Реалізувати: відстеження кількості елементів, сортування, пошук заданого елемента, випадкову перестановку елементів, пошук середнього значення елементів.

### **Варіант 9.**

Створити клас `Binary` для роботи з двійковими числами фіксованої довжини. Число повинне бути представлене масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Молодший біт має молодший індекс.

### **Варіант 10.**

Створити клас `Decimal` для роботи з беззнаковими цілими десятковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реалізувати арифметичні операції, аналогічні до тих, що є для цілих в C++, та операції порівняння.

### **Варіант 11.**

Створити клас `Polinom` для роботи з многочленами. Коефіцієнти мають бути представлені масивом, кожен елемент якого – коефіцієнт. Молодша степінь має менший індекс (нульовий степінь – нульовий індекс). Реалізувати арифметичні операції, обчислення для заданого  $x$ , інтегрування, отримання похідної.

### **Варіант 12.**

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс. Реалізувати всі арифметичні операції для цілих в C++ і операції порівняння.

### **Варіант 13.**

Створити клас `Decimal` для роботи із знаковими цілими десятковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Знак представити окремим полем `sign`. Реалізувати арифметичні операції, вбудовані для цілих в C++, і операції

порівняння.

#### **Варіант 14.**

Створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині повинна задаватися в окремому полі та ініціалізуватися конструктором. Знак представити окремим полем `sign`.

#### **Варіант 15.**

Створити клас `Long` для роботи з без-знаковими цілими двійковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є десятковою цифрою. Молодша цифра має менший індекс (розряд одиниць – в нульовому елементі масиву). Реалізувати арифметичні операції, вбудовані для цілих в C++, і операції порівняння.

#### **Варіант 16.**

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реалізувати арифметичні операції, підтримані для цілих в C++, і операції порівняння.

#### **Варіант 17.**

Створити клас `Money` для роботи з грошовими сумами. Сума повинна бути представлена масивом, кожен елемент якого – десяткова цифра. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

#### **Варіант 18.**

Створити клас `Long` для роботи із знаковими цілими двійковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого являється десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Знак представити окремим полем `sign`. Реалізувати арифметичні операції, визначені в C++ для цілих, та операції порівняння.

#### **Варіант 19.**

Створити клас `Hex` для роботи із знаковими цілими шістнадцятковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є шістнадцятковою цифрою. Молодша цифра має менший індекс. Знак представити окремим полем `sign`. Реалізувати арифметичні операції, підтримані для цілих в C++, і операції

порівняння.

## Варіант 20.

Створити клас `Octal` для роботи із знаковими цілими вісімковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є шістнадцятковою цифрою. Молодша цифра має менший індекс. Знак представити окремим полем `sign`. Реалізувати арифметичні операції, підтримані для цілих в C++, і операції порівняння.

## Варіант 21.

Створити клас `Fraction` для роботи з без-знаковими дробовими десятковими числами. Число повинне бути представлене двома масивами типу `unsigned char`: ціла і дробова частина, кожен елемент – десяткова цифра. Для цілої частини молодша цифра має менший індекс, для дробової частини – старша цифра має менший індекс (десяті – в нульовому елементі, соті – в першому, і т. д.). Реалізувати арифметичні операції додавання, віднімання і множення і операції порівняння.

## Варіант 22.

Реалізувати клас `Rational`, використовуючи два масиви типу `unsigned char` для представлення чисельника і знаменника. Кожен елемент є десятковою цифрою. Молодша цифра має менший індекс. Реалізувати арифметичні операції, підтримані для цілих в C++.

Рациональний (нескоротний) дріб представляється парою  $(a, b)$ , де поля:

- $a$  – чисельник,
- $b$  – знаменник

(у цьому завданні  $a$  та  $b$  – масиви).

Обов'язково мають бути реалізовані наступні операції:

Унарна операція (аргументом є цей об'єкт):

- обчислення значення `value()`,  $a / b$ ;

Бінарні операції (перший аргумент – цей об'єкт, другий аргумент – об'єкт-параметр):

- додавання `add()`,  $(a_1, b_1) + (a_2, b_2) = (a_1 \cdot b_2 + a_2 \cdot b_1, b_1 \cdot b_2)$ ;
- віднімання `sub()`,  $(a_1, b_1) - (a_2, b_2) = (a_1 \cdot b_2 - a_2 \cdot b_1, b_1 \cdot b_2)$ ;
- множення `mul()`,  $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2, b_1 \cdot b_2)$ ;
- ділення `div()`,  $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot b_2, a_2 \cdot b_1)$ ;
- порівняння «чи рівне» `equal()`;
- порівняння «чи більше» `great()`;

- порівняння «чи менше» `less()`.

Повинна бути реалізована приватна функція скорочення дробу `Reduce()`, яка обов'язково викликається при виконанні арифметичних операцій.

### Варіант 23.

Створити клас `Vector` для роботи з векторами, кількість елементів яких задає користувач. Мають бути реалізовані: операції додавання і віднімання векторів, скалярний добуток векторів, множення і ділення на скаляр, обчислення евклідової норми, порівняння векторів за їх нормами. (Евклідова норма обчислюється як квадратний корінь з суми квадратів координат.)

### Варіант 24.

Масив  $(y_0, y_1, \dots, y_n)$  є значеннями деякої функції на відрізку  $[a, b]$ , причому  $a = y_0$ ,  $b = y_n$ . Створити клас `Integral`, в якому реалізовано обчислення певного інтеграла методом лівих, правих та центральних прямокутників, а також методом трапецій.

### Варіант 25.

Створити клас `Vector` для роботи з векторами розмірності, що задається. Мають бути реалізовані: операції додавання і віднімання векторів, скалярний добуток векторів, множення і ділення на скаляр, обчислення евклідової норми, порівняння норм векторів. (Евклідова норма обчислюється як квадратний корінь з суми квадратів координат.)

### Варіант 26.

Масив  $(y_0, y_1, \dots, y_n)$  є значеннями деякої функції на відрізку  $[a, b]$ , причому  $a = y_0$ ,  $b = y_n$ . Створити клас `Integral`, в якому реалізовано обчислення певного інтеграла методом прямокутників, методом трапецій і методом Сімпсона.

### Варіант 27.

Використовуючи Лістинг 1 як зразок, реалізувати клас `Agguy` – масив чисел з повним набором арифметичних операцій. Реалізувати операції перевірки на рівність і нерівність.

### Варіант 28.

Масив пар дійсних чисел  $(x_1, p_1), \dots, (x_n, p_n)$  є набором значень дискретної випадкової величини. Створити клас `Rand`, в якому реалізувати методи обчислення математичного очікування і дисперсії. Реалізувати метод обчислення інтегралу для заданої функції  $f(x)$  методом Монте-Карло.

## Варіант 29.

Створити клас **Agau** – одновимірний масив дійсних чисел з межами індексів, що задаються, з можливістю задати від’ємні індекси. Обов’язково мають бути реалізовані: відстеження кількості елементів, множення і ділення на скаляр, пошук максимуму та мінімуму, обчислення суми та середнього арифметичного елементів, по-елементне додавання та віднімання масивів.

## Варіант 30.

Створити клас **Agau** – одновимірний масив цілих чисел з межами індексів, що задаються, з можливістю визначення від’ємних індексів. Обов’язково мають бути реалізовані: відстеження кількості елементів, всі операції з масивом і цілим числом, пошук заданого елементу, всі по-елементні операції, реалізовані в C++ для цілих чисел.

## Варіант 31.

Створити клас **BitString** для роботи з бітовими рядками. Бітовий рядок повинен бути представлений масивом типу **unsigned char**, кожен елемент якого приймає значення 0 або 1. Молодший біт має молодший індекс. Реалізувати всі операції для роботи з бітовими рядками: **and**, **or**, **xor**, **not**.

## Варіант 32.

Створити клас **Agau** – одновимірний масив. Реалізувати: відстеження кількості елементів, сортування, пошук заданого елемента, випадкову перестановку елементів, пошук середнього значення елементів.

## Варіант 33.

Створити клас **Binary** для роботи з двійковими числами фіксованої довжини. Число повинне бути представлене масивом типу **unsigned char**, кожен елемент якого приймає значення 0 або 1. Молодший біт має молодший індекс.

## Варіант 34.

Створити клас **Decimal** для роботи з беззнаковими цілими десятковими числами, використовуючи масив елементів типу **unsigned char**, кожен елемент якого є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реалізувати арифметичні операції, аналогічні до тих, що є для цілих в C++, та операції порівняння.

### **Варіант 35.**

Створити клас `Polinom` для роботи з многочленами. Коефіцієнти мають бути представлені масивом, кожен елемент якого – коефіцієнт. Молодша степінь має менший індекс (нульовий степінь – нульовий індекс). Реалізувати арифметичні операції, обчислення для заданого  $x$ , інтегрування, отримання похідної.

### **Варіант 36.**

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс. Реалізувати всі арифметичні операції для цілих в C++ і операції порівняння.

### **Варіант 37.**

Створити клас `Decimal` для роботи із знаковими цілими десятковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Знак представити окремим полем `sign`. Реалізувати арифметичні операції, вбудовані для цілих в C++, і операції порівняння.

### **Варіант 38.**

Створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині повинна задаватися в окремому полі та ініціалізуватися конструктором. Знак представити окремим полем `sign`.

### **Варіант 39.**

Створити клас `Long` для роботи з без-знаковими цілими двійковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є десятковою цифрою. Молодша цифра має менший індекс (розряд одиниць – в нульовому елементі масиву). Реалізувати арифметичні операції, вбудовані для цілих в C++, і операції порівняння.

### **Варіант 40.**

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи масив елементів типу `unsigned char`, кожен елемент якого є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реалізувати арифметичні операції, підтримані для цілих в C++, і операції порівняння.



## Лабораторна робота № 6.3. Контейнери-списки

### Мета роботи

Освоїти створення і використання контейнерів-списків.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення контейнера.
- 2) Поняття та призначення ітератора.
- 3) Види контейнерів.
- 4) Прямий доступ до елементів.
- 5) Послідовний доступ до елементів.
- 6) Асоціативний доступ до елементів.
- 7) Метод begin().
- 8) Метод end().
- 9) Операції контейнера.

### Приклад

#### Лістинг 3. Послідовний контейнер – список

[9 – с. 91-94]

##### файл «List.h»

```
#pragma once

class NullIterator // клас винятків
{};

class List
{
public:
    typedef double value_type;
    typedef size_t size_type;

    class iterator;

    // конструктори, копіювання, присвоєння
    List();
    List(const value_type& a, size_type n = 1);
    List(iterator, iterator);
    List(const List& r);
    ~List();
    List& operator=(const List& r);

    // ітератори
    iterator begin() { return head; }
    iterator end() { return tail; }
```

```

iterator begin() const { return head; }
iterator end()   const { return tail; }

// розміри
bool empty() const { return (Head == Tail); }
size_type length() const { return count; }

// доступ до елементів
value_type& front() { return *begin(); }
value_type& back() { iterator it; --it; return *it; }

iterator find(const value_type& a);    // пошук

// модифікатори контейнера
void push_front(const value_type&);    // додати в початок
value_type pop_front();                // видалити перший
void push_back(const value_type&);     // додати в кінець
value_type pop_back();                // видалити останній
void insert(iterator, const value_type&); // вставити після
void erase(iterator);                 // видалити вказаний (в позиції)
void erase(iterator, iterator);       // видалити групу вказаних
void remove(const value_type&);       // видалити заданий (значенням)
void swap(List&);                    // обміняти із вказаним списком
void clear();                        // видалити всі елементи
void splice(List&);                  // додати список в кінець
void splice(iterator, List&);        // додати список після
                                     // вказаного елемента
void sort();                         // сортування
void merge(List&);                   // злиття відсортованих

private:
// елемент
struct Node
{
    Node(const value_type& a) : item(a), next(), prev() {}
    Node() : item(), next(), prev() {}
    value_type item;                // інформаційна частина
    Node* next;                    // наступний елемент
    Node* prev;                    // попередній елемент
};

long count;                        // кількість елементів
Node* Head;                       // "голова" списку
Node* Tail;                       // "хвіст" списку

public:
// вкладений клас-ітератор
class iterator
{
    friend class List;             // клас-контейнер
    iterator(Node* el) : elem(el) {}

public:
// конструктори
    iterator() : elem(0) {}
    iterator(const iterator& it) : elem(it.elem) {}

// порівняння ітераторів
    bool operator ==(const iterator& it) const
    {
        return elem == it.elem;
    }
}

```

```

bool operator !=(const iterator& it) const
{
    return elem != it.elem;
}

// переміщення ітератора
iterator& operator++()    // вперед
{
    // префіксна форма
    if (elem != 0)
        elem = elem->next;
    return *this;
}

iterator operator++(int) // вперед
{
    // постфіксна форма
    return operator++(); // виклик префіксної форми
}

iterator& operator--()    // назад
{
    // префіксна форма
    if (elem != 0)
        elem = elem->prev;
    return *this;
}

iterator operator--(int) // назад
{
    // постфіксна форма
    return operator--(); // виклик префіксної форми
}

// *****
// вперед на n позицій - реалізувати самостійно
iterator& operator +=(int n) { return *this; }

// назад на n позицій - реалізувати самостійно
iterator& operator -=(int n) { return *this; }

value_type& operator *() // роз'іменування
{
    if (elem != 0)
        return elem->item;
    else
        throw NullIterator();
}

private:
    Node* elem; // вказівник на елемент
};

private:
    iterator head, tail; // для ітератора - вказівники на голову та хвіст
};

```

## файл «List.cpp»

```

#include "List.h"

// *****
// конструктори, деструктор, присвоєння

```

```

// конструктор за умовчанням -
// порожній список з невидимим елементом
List::List()
    : Head(new Node()), Tail(Head), count(0)
{
    Tail->next = Tail->prev = 0;
    head = iterator(Head); // ініціалізація для ітератора
    tail = iterator(Tail);
}

// створюємо список із n елементів та
// заповнюємо його значенням
List::List(const value_type& a, size_type n)
{
    List tmp; // порожній список з фіктивним елементом
    for (size_type i = 0; i < n; i++) // приєднуємо елементи
        tmp.push_front(a);
    *this = tmp; // зробили поточним
}

// конструктор, який створює новий список на основі
// ітераторів іншого списку - відрізняється лише
// іншим способом організації циклу
List::List(iterator first, iterator last)
{
    List tmp;
    for (iterator ip = first; ip != last; ip++)
        tmp.push_front(*ip);
    *this = tmp;
}

// конструктор копіювання
List::List(const List& r)
{
    List tmp(r.begin(), r.end()); // виконується конструктор з ітераторами
    *this = tmp;
}

// деструктор
List::~List()
{
    Node* deleting_Node = Head; // елемент, який видаляється
    for (Node* p = Head; p != Tail; ) // поки не дійшли до невидимого
    {
        p = p->next; // підготували наступний
        delete deleting_Node; // видалили елемент
        --count;
        deleting_Node = p; // підготували для видалення
    }
    delete deleting_Node; // видалили останній
}

// присвоєння - реалізовано як обмін з тимчасовим об'єктом-контейнером,
// щоб не писати явного повернення пам'яті поточного об'єкту-списку
// - для цього слід реалізувати метод обміну swap()
List& List::operator=(const List& t)
{
    List tmp(t); // тимчасовий локальний об'єкт-список tmp=t
    swap(tmp); // обмін полями з поточним об'єктом
    return *this; // повернення поточного об'єкта
}
// tmp знищується

```

```

// *****
// методи вставки та видалення

// вставити після
void List::insert(iterator it, const value_type& r)
{
    Node* el = it.elem; // поточний елемент
    if (it == end()) // якщо останній
        push_back(r);
    else
    {
        Node* next_el = it.elem->next; // наступний елемент
        Node* p = new Node(r); // створили новий елемент
        p->next = next_el; // зв'язки в новому
        p->prev = el; // елементі
        next_el->prev = p; // прив'язали новий
        el->next = p;
    }
    ++count;
}

// видалити вказаний (в заданій позиції)
void List::erase(iterator it)
{
    Node* el = it.elem; // поточний елемент
    if (it == begin()) // якщо перший
        pop_front();
    else if (it == end()) // якщо останній
        pop_back();
    else
    {
        Node* prev_el = it.elem->prev; // попередній елемент
        Node* next_el = it.elem->next; // наступний елемент
        prev_el->next = next_el; // переналаштуємо вказівники
        next_el->prev = prev_el;
        delete el; // вилучаємо - повертаємо пам'ять
    }
    --count;
}

// *****
// інші методи, які слід реалізувати самостійно

// модифікатори контейнера
void List::push_front(const value_type& v) {} // додати в початок
List::value_type List::pop_front() { return *begin(); } // видалити перший
void List::push_back(const value_type& v) {} // додати в кінець
List::value_type List::pop_back() { return *begin(); } // видалити останній
void List::erase(iterator first, iterator last) {} // видалити групу вказаних
void List::remove(const value_type& v) {} // видалити заданий (значенням)
void List::swap(List& L) {} // обміняти із вказаним списком
void List::clear() {} // видалити всі елементи
void List::splice(List& L) {} // додати список в кінець
void List::splice(iterator it, List& L) {} // додати список після
// вказаного елемента
void List::sort() {} // сортування
void List::merge(List& L) {} // злиття відсортованих

List::iterator List::find(const value_type& a) { return nullptr; } // пошук

```

### файл «Source.cpp»

```
// Lab_6_3.cpp : Defines the entry point for the console application.
//

#include "List.h"
#include <iostream>
#include <stdexcept>
#include <exception>

using namespace std;

int main()
{
    List l;
    for (int i = 1; i <= 10; i++)
    {
        l.insert(l.end(), i);           // слід дописати методи класу List
    }

    while (!l.empty())                 // слід дописати методи класу List
    {
        cout << l.pop_front() << endl; // слід дописати методи класу List
    }

    cin.get();
    cin.get();
    return 0;
}
```

### Варіанти завдань

Реалізувати завдання свого варіанту Лабораторної роботи № 6.2 «Контейнери-масиви», використовуючи замість масиву двонаправлений список (див. лістинг 3). Замість операції індексування реалізувати ітератор як набір методів списку.

## Лабораторна робота № 6.4. Шаблони класів та шаблони функцій

### **Мета роботи**

Освоїти шаблонів класів та шаблонів функцій.

### **Питання, які необхідно вивчити та пояснити на захисті**

- 1) Поняття та призначення шаблону класу.
- 2) Поняття та призначення шаблону функції.
- 3) Параметри шаблону класу.
- 4) Параметри шаблону функції.
- 5) Спеціалізація шаблону класу.
- 6) Спеціалізація шаблону функції.
- 7) Перевантаження шаблону функції.
- 8) Статичні елементи в шаблонах класів.
- 9) Механізм опрацювання винятків в шаблоні класу.
- 10) Шаблон класу з елементами-шаблонами.
- 11) Шаблони та успадкування.
- 12) Шаблони та дружні функції.

### **Приклад виконання завдання**

Визначити шаблон класу-одновимірного масиву (вектору) з методами та операціями:

- вводу-виводу;
- присвоєння;
- індексування;
- додавання векторів;
- пошуку мінімального / максимального елемента;
- обчислення норми;
- сортування в порядку зростання / спадання.

Продемонструвати інстанціювання створеного шаблону.

```
#include <iostream>
#include <math.h>
#include <string>

using namespace std;
```

```

template <class T>          // T - тип
class vector
{
    T *v;                  // внутрішній масив
    int size;              // кількість елементів

public:
    vector(int newsize);    // конструктор ініціалізації
    vector(vector&);        // конструктор копіювання
    ~vector();              // деструктор

    T      extr(const char *); // пошук мінімального чи максимального елемента
    vector& sort(const char *); // сортування
    double norma(void);      // обчислення норми

    friend vector<T> operator + <>(vector<T>& x, vector<T>& y); // додавання
    vector& operator += (vector&); // збільшення на
    T&      operator [] (int index); // індексування
    vector& operator = (const vector&); // присвоєння

    friend ostream& operator << <>(ostream&, vector&); // виведення
    friend istream& operator >> <>(istream&, vector&); // введення
};

template <class T>
vector<T>::vector(int newsize) // конструктор ініціалізації
{
    v = new T[size = newsize]; // новий розмір, виділення пам'яті для елементів

    for (int i = 0; i < size; i++) // присвоєння нульових значень елементам
        v[i] = T();
}

template <class T>
vector<T>::vector(vector<T>& x) // конструктор копіювання
{
    v = new T[size = x.size]; // новий розмір, виділення пам'яті для елементів

    for (int i = 0; i < size; i++) // присвоєння значень елементам
        v[i] = x.v[i];
}

template <class T>
vector<T>::~~vector()
{
    delete[] v; // звільнення пам'яті
}

template <class T>
T vector<T>::extr(const char* MinOrMax) // MinOrMax - визначає, що шукаємо
{
    T ExtrElem = v[0];
    for (int i = 0; i < size; i++) {
        if (!strcmp(MinOrMax, "min")) {
            if (v[i] < ExtrElem) ExtrElem = v[i]; // мінімальний елемент
        }
        else
            if (v[i] > ExtrElem) ExtrElem = v[i]; // максимальний елемент
    }
    return ExtrElem;
}

```



```

template <class T>
vector<T>& vector<T>::sort(const char* UpOrDown)
{
    T x;
    for (int i = 0; i < size - 1; i++)
        for (int j = i + 1; j < size; j++)
            if (!strcmp(UpOrDown, "up"))
            {
                // за зростанням
                if (v[i] > v[j])
                {
                    x = v[i];
                    v[i] = v[j];
                    v[j] = x;
                }
            }
            else
            {
                // за спаданням
                if (v[i] < v[j])
                {
                    x = v[i];
                    v[i] = v[j];
                    v[j] = x;
                }
            }
    return *this;
}

template <class T>
double vector<T>::norma(void) // норма = сума квадратів елементів (для числових типів)
{
    double s = 0;
    for (int i = 0; i < size; i++)
        s += v[i] * v[i];
    return sqrt(s);
}

template <class T>
vector<T> operator + (vector<T>& x, vector<T>& y) // додавання
{
    if (x.size != y.size)
    {
        throw exception("The sizes should be the same!");
    }

    vector<T> z(x.size);

    for (int i = 0; i < x.size; i++)
        z.v[i] = x.v[i] + y.v[i];

    return z;
}

template <class T>
vector<T>& vector<T>::operator += (vector<T>& y)
{
    for (int i = 0; i < size; i++)
        v[i] += y.v[i];
    return *this;
}

template <class T>
T& vector<T>::operator [] (int index)
{
    if (index < 0 || index >= size)
    {
        throw exception("Index out of the range");
    }
}

```

```

    return v[index];
}

template <class T>
vector<T>& vector<T>::operator = (const vector<T>& x)
{
    if (this != &x)
    {
        delete[] v;
        v = new T[size = x.size];
        for (int i = 0; i < size; i++)
            v[i] = x.v[i];
    }
    return *this;
}

template <class T>
istream& operator >> (istream& is, vector<T>& x)
{
    cout << "Input " << x.size << " elements of vector" << endl;
    for (int i = 0; i < x.size; i++) {
        cout << "element[" << i << "] = ? ";
        is >> x.v[i];
    }
    return is;
}

template <class T>
ostream& operator << (ostream& os, vector<T>& x)
{
    for (int i = 0; i < x.size; i++)
        os << x.v[i] << ' ';
    os << endl;
    return os;
}

int main()
{
    try
    {
        vector<int> V(5), U(5), Z(5), T(5);

        cin >> V;

        cin >> U;

        T = V + U;
        cout << "summa of vectors:" << endl;
        cout << T;

        Z += V;
        Z += U;
        cout << "summa of vectors:" << endl;
        cout << Z;

        cout << "min element = " << Z.extr("min") << endl;

        Z.sort("up");
        cout << "sorted:" << endl;
        cout << Z;

        cout << "norma = " << Z.norma() << endl;
    }
}

```

```

    catch (exception e)
    {
        cerr << e.what() << endl;
    }

    system("pause");
    return 0;
}

```

## ***Варіанти завдань***

Визначення шаблону класу та опис реалізації його методів слід розмістити в одному модулі (одному і тому самому файлі).

У всіх варіантах слід продемонструвати інстанціювання створених шаблонів.

Варіанти завдань наступні:

### **Варіант 1.**

Написати шаблон функції послідовного пошуку значення в масиві за заданим ключем. Функція повертає індекс першого знайденого елемента масиву, який дорівнює ключеві.

### **Варіант 2.**

Написати функцію-шаблон, яка обчислює максимальне значення елемента масиву.

### **Варіант 3.**

Створити шаблон класу – параметризований одновимірний масив-вектор з операціями додавання векторів та множення вектора на число.

### **Варіант 3\*.**

Створити шаблон класу – параметризований стек з операціями додавання та вилучення елементів.

### **Варіант 4.**

Створити шаблон класу – параметризований масив з конструкторами, деструктором і перевантаженими операціями [ ], =, виведення у потік та введення з потоку.

### **Варіант 5.**

Написати шаблон – параметризовану функцію сортування одновимірного масиву по зростанню елементів методом бульбашки. Суть методу полягає в порівнянні та перестановці сусідніх елементів.

### **Варіант 6.**

Створити шаблон класу – параметризований двовимірний масив-матрицю з операціями додавання та віднімання матриць.

### **Варіант 6\*.**

Створити шаблон класу – параметризовану чергу з операціями додавання та вилучення елементів.

### **Варіант 7.**

Створити шаблон класу – параметризований двовимірний масив-матрицю з операціями множення матриць та множення матриці на число.

### **Варіант 7\*.**

Створити шаблон класу – параметризовану циклічну чергу з операціями додавання та вилучення елементів.

### **Варіант 8.**

Створити шаблон класу – параметризований одновимірний масив-вектор з операціями додавання та скалярного добутку векторів.

### **Варіант 8\*.**

Створити шаблон класу – параметризований лінійний список з подвійними зв'язками з операціями додавання, вилучення та пошуку елементів.

### **Варіант 9.**

Написати функцію-шаблон, яка обчислює середнє значення в масиві.

### **Варіант 10.**

Написати функцію-шаблон, яка перевіряє, чи масив – впорядкований (розглянути всі види впорядкування).

### **Варіант 10\*.**

Створити шаблон класу – параметризований клас бінарного дерева з методами – додати елемент у дерево, проходження по дереву в спадному й у висхідному порядку. Здійснити пошук по дереву.

### **Варіант 11.**

Написати функцію-шаблон бінарного пошуку у впорядкованому масиві даних. При використанні бінарного методу на першому кроці перевіряється серединний елемент. Якщо він більше ключа пошуку, то перевіряється серединний елемент другої половини масиву. Ця процедура повторюється доти, поки не буде знайдений збіг, або доти, поки більше не залишиться елементів, які можна було б перевіряти.

### **Варіант 12.**

Написати функцію-шаблон, яка перевіряє, чи масив містить однакові сусідні елементи.

### **Варіант 12\*.**

Створити шаблон класу – параметризований клас однозв'язного списку з операціями додавання, вилучення та пошуку елементів.

### **Варіант 13.**

Написати функцію-шаблон, яка перевіряє, чи масив містить елементи-дублікати.

### **Варіант 13\*.**

Створити шаблон класу – параметризований лінійний кільцевий список з подвійними зв'язками з операціями додавання, вилучення та пошуку елементів.

### **Варіант 14.**

Написати функцію-шаблон, яка перевіряє, чи вірно, що масив не містить однакових сусідніх елементів.

### **Варіант 14\*.**

Створити шаблон класу – параметризований однонаправлений лінійний кільцевий список з операціями додавання, вилучення та пошуку елементів.

### **Варіант 15.**

Написати функцію-шаблон, яка перевіряє, чи масив містить лише унікальні елементи.

### **Варіант 15\*.**

Створити параметризований клас – множину, який призначений для збереження елементів і виконання операцій над ними. Реалізувати класичні операції над множинами –

об'єднання, перетину, різниці.

### **Варіант 16.**

Написати шаблон функції послідовного пошуку значення в масиві за заданим ключем. Функція повертає індекс першого знайденого елемента масиву, який дорівнює ключеві; також обчислюється кількість знайдених елементів.

### **Варіант 17.**

Написати функцію-шаблон, яка обчислює мінімальне значення елемента масиву.

### **Варіант 18.**

Створити шаблон класу – параметризований одновимірний масив-вектор з операціями скалярного добутку векторів та додавання вектора і числа.

### **Варіант 18\*.**

Створити шаблон класу – параметризований стек з операціями додавання, вилучення елементів та з операцією перевірки, чи стек – порожній.

### **Варіант 19.**

Створити шаблон класу – параметризований масив з конструкторами, деструктором і перевантаженими операціями індексування, присвоєння, порівняння.

### **Варіант 20.**

Написати шаблон – параметризовану функцію сортування одновимірного масиву по спаданню елементів методом бульбашки. Суть методу полягає в порівнянні та перестановці сусідніх елементів.

### **Варіант 21.**

Створити шаблон класу – параметризований двовимірний масив-матрицю з операціями додавання та множення матриць.

### **Варіант 21\*.**

Створити шаблон класу – параметризовану чергу з операціями додавання, вилучення елементів та перевірки, чи черга – не порожня.

### **Варіант 22.**

Створити шаблон класу – параметризований двовимірний масив-матрицю з

операціями додавання матриць та множення матриці на число.

### **Варіант 22\*.**

Створити шаблон класу – параметризовану циклічну чергу з операціями додавання, вилучення елементів та перевірки, чи циклічна черга –порожня.

### **Варіант 23.**

Створити шаблон класу – параметризований одновимірний масив-вектор з операціями додавання векторів та додавання вектора і числа.

### **Варіант 23\*.**

Створити шаблон класу – параметризований лінійний список з подвійними зв'язками з операціями додавання, вилучення та обчислення кількості заданих елементів.

### **Варіант 24.**

Написати функцію-шаблон, яка обчислює середнє значення модулів елементів масиву.

### **Варіант 25.**

Написати функцію-шаблон, яка перевіряє, чи масив – не впорядкований (розглянути всі види впорядкування).

### **Варіант 25\*.**

Створити шаблон класу – параметризований клас бінарного дерева з методами – додання елемента у дерево, обходу дерева в префіксному, інфіксному та постфіксному порядку. Реалізувати пошук заданого елемента в дереві.

### **Варіант 26.**

Написати шаблон функції послідовного пошуку значення в масиві за заданим ключем. Функція повертає індекс першого знайденого елемента масиву, який дорівнює ключеві.

### **Варіант 27.**

Написати функцію-шаблон, яка обчислює максимальне значення елемента масиву.

### **Варіант 28.**

Створити шаблон класу – параметризований одновимірний масив-вектор з операціями додавання векторів та множення вектора на число.

### **Варіант 28\*.**

Створити шаблон класу – параметризований стек з операціями додавання та вилучення елементів.

### **Варіант 29.**

Створити шаблон класу – параметризований масив з конструкторами, деструктором і перевантаженими операціями [ ], =, виведення у потік та введення з потоку.

### **Варіант 30.**

Написати шаблон – параметризовану функцію сортування одновимірного масиву по зростанню елементів методом бульбашки. Суть методу полягає в порівнянні та перестановці сусідніх елементів.

### **Варіант 31.**

Створити шаблон класу – параметризований двовимірний масив-матрицю з операціями додавання та віднімання матриць.

### **Варіант 31\*.**

Створити шаблон класу – параметризовану чергу з операціями додавання та вилучення елементів.

### **Варіант 32.**

Створити шаблон класу – параметризований двовимірний масив-матрицю з операціями множення матриць та множення матриці на число.

### **Варіант 32\*.**

Створити шаблон класу – параметризовану циклічну чергу з операціями додавання та вилучення елементів.

### **Варіант 33.**

Створити шаблон класу – параметризований одновимірний масив-вектор з операціями додавання та скалярного добутку векторів.

### **Варіант 33\*.**

Створити шаблон класу – параметризований лінійний список з подвійними зв'язками з операціями додавання, вилучення та пошуку елементів.



### **Варіант 34.**

Написати функцію-шаблон, яка обчислює середнє значення в масиві.

### **Варіант 35.**

Написати функцію-шаблон, яка перевіряє, чи масив – впорядкований (розглянути всі види впорядкування).

### **Варіант 35\*.**

Створити шаблон класу – параметризований клас бінарного дерева з методами – додати елемент у дерево, проходження по дереву в спадному й у висхідному порядку. Здійснити пошук по дереву.

### **Варіант 36.**

Написати функцію-шаблон бінарного пошуку у впорядкованому масиві даних. При використанні бінарного методу на першому кроці перевіряється серединний елемент. Якщо він більше ключа пошуку, то перевіряється серединний елемент другої половини масиву. Ця процедура повторюється доти, поки не буде знайдений збіг, або доти, поки більше не залишиться елементів, які можна було б перевіряти.

### **Варіант 37.**

Написати функцію-шаблон, яка перевіряє, чи масив містить однакові сусідні елементи.

### **Варіант 37\*.**

Створити шаблон класу – параметризований клас однозв'язного списку з операціями додавання, вилучення та пошуку елементів.

### **Варіант 38.**

Написати функцію-шаблон, яка перевіряє, чи масив містить елементи-дублікати.

### **Варіант 38\*.**

Створити шаблон класу – параметризований лінійний кільцевий список з подвійними зв'язками з операціями додавання, вилучення та пошуку елементів.

### **Варіант 39.**

Написати функцію-шаблон, яка перевіряє, чи вірно, що масив не містить однакових

сусідніх елементів.

### **Варіант 39\*.**

Створити шаблон класу – параметризований однонаправлений лінійний кільцевий список з операціями додавання, вилучення та пошуку елементів.

### **Варіант 40.**

Написати функцію-шаблон, яка перевіряє, чи масив містить лише унікальні елементи.

### **Варіант 40\*.**

Створити параметризований клас – множину, який призначений для збереження елементів і виконання операцій над ними. Реалізувати класичні операції над множинами – об'єднання, перетину, різниці.

## Лабораторна робота № 6.5. Шаблони класів та опрацювання виняткових ситуацій

### Мета роботи

Освоїти опрацювання виняткових ситуацій в шаблонах класів.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення шаблону класу.
- 2) Поняття та призначення шаблону функції.
- 3) Параметри шаблону класу.
- 4) Параметри шаблону функції.
- 5) Спеціалізація шаблону класу.
- 6) Спеціалізація шаблону функції.
- 7) Перевантаження шаблону функції.
- 8) Статичні елементи в шаблонах класів.
- 9) Механізм опрацювання винятків в шаблоні класу.
- 10) Шаблон класу з елементами-шаблонами.
- 11) Шаблони та успадкування.
- 12) Шаблони та дружні функції.

### Варіанти завдань

#### Загальні вказівки для всіх варіантів

У всіх варіантах вимагається створити шаблон деякого цільового класу **A**, можливо, реалізований використанням деякого серверного класу **B**. Це означає, що об'єкт класу **B** використовується як елемент класу **A**. В якості серверного класу може бути вказаний лише клас, створений програмістом (в рамках цього самого ж завдання), або клас із стандартної бібліотеки – наприклад, `std::vector`.

Варіанти цільових або серверних класів, створюваних програмістом, приведені в Таблиці 1.

Таблиця 1. Варіанти цільових або серверних класів

Ім'я класу	Призначення
Vect	одновимірний динамічний масив
List	дво-направлений список

Stack	стек
BinaryTree	бінарне дерево
Queue	одностороння черга
Deque	двостороння черга (допускає вставку і видалення з обох кінців черги)
Set	множина (елементи, які повторюються, в множині не заносяться; елементи в множині зберігаються відсортованими)
SparseArray	розріджений масив

Якщо замість серверного класу вказаний динамічний масив, то це означає, що для зберігання елементів контейнерного класу використовується масив, який створюється за допомогою операції `new`.

У всіх варіантах необхідно передбачити генерацію і опрацювання винятків для можливих помилкових ситуацій.

У всіх варіантах продемонструвати в клієнті `main()` використання створеного класу, включаючи випадки, які приводять до генерації винятків. Показати інстанціювання шаблону для типів `int`, `double`, `std::string`.

Варіанти завдань приведені в Таблиці 2.

**Таблиця 2.** Варіанти завдань

Варіант	Цільовий шаблонний клас	Реалізація з використанням
1	Vect	<code>std::list</code>
2	List	<code>std::list</code>
3	Stack	динамічний масив
4	Stack	Vect
5	Stack	List
6	Stack	<code>std::vector</code>
7	Stack	<code>std::list</code>
8	BinaryTree	–
9	Queue	Vect
10	Queue	List
11	Queue	<code>std::list</code>
12	Deque	Vect
13	Deque	List
14	Deque	<code>std::list</code>
15	Set	динамічний масив
16	Set	Vect

17	Vect	динамічний масив
18	List	динамічний масив
19	Stack	динамічний масив
20	Stack	std::list
21	Stack	Vect
22	Stack	List
23	Stack	std::vector
24	BinaryTree	std::list
25	Queue	Vect
26	Queue	List
27	Queue	std::list
28	Deque	Vect
29	Deque	List
30	Deque	std::list
31	Set	динамічний масив
32	Set	Vect
33	Vect	динамічний масив
34	List	динамічний масив
35	Stack	динамічний масив
36	Stack	std::list
37	Stack	Vect
38	Stack	List
39	Stack	std::vector
40	BinaryTree	std::list

## Лабораторна робота № 6.6. Шаблони класів

### Мета роботи

Освоїти опрацювання шаблонів класів.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення шаблону класу.
- 2) Поняття та призначення шаблону функції.
- 3) Параметри шаблону класу.
- 4) Параметри шаблону функції.
- 5) Спеціалізація шаблону класу.
- 6) Спеціалізація шаблону функції.
- 7) Перевантаження шаблону функції.
- 8) Статичні елементи в шаблонах класів.
- 9) Механізм опрацювання винятків в шаблоні класу.
- 10) Шаблон класу з елементами-шаблонами.
- 11) Шаблони та успадкування.
- 12) Шаблони та дружні функції.

### Приклад

#### Лістинг 5. Простий масив – шаблон

[9, с.109-110]

##### Файл «Array.h»

```
#pragma once
#include <stdexcept>

template <class T, std::size_t N>
class Array
{
public:
    // типи
    typedef T          value_type;
    typedef T&         reference;
    typedef const T&    const_reference;
    typedef std::size_t size_type;

    // розмір масиву
    static const size_type static_size = N;

    Array(const T& t = T());           // конструктор
```

```

size_type size() const           // отримання розміру
{
    return static_size;
}

// доступ до елементів
reference operator [] (const size_type& i)
{
    rangecheck(i);
    return elem[i];
}

const_reference operator [] (const size_type& i) const
{
    rangecheck(i);
    return elem[i];
}

private:
// перевірка індексу
void rangecheck(const size_type& i) const
{
    if (i >= size())
        throw std::range_error("Array: index out of range!");
}

// поле-масив
T elem[N];
};

// реалізація конструктора
// - має бути у тому ж файлі,
// що і визначення шаблону
template <class T, std::size_t N>
Array<T, N>::Array(const T& t)
{
    for (int i = 0; i < N; i++)
        elem[i] = t;
}

```

### Файл «L\_6\_6.cpp»

```

#include <iostream>
#include "Array.h"
#include <Windows.h>           // забезпечення відображення кирилиці

using namespace std;

int main()
{
    SetConsoleCP(1251);        // забезпечення відображення кирилиці
    SetConsoleOutputCP(1251);  // забезпечення відображення кирилиці

    const int N = 10;

    Array<int, N> a;
    Array<int, N> b(0);
    Array<int, N> c(b);

    for (int i = 0; i < N; i++)
        cout << a[i] << " ";
    cout << endl;
}

```

```

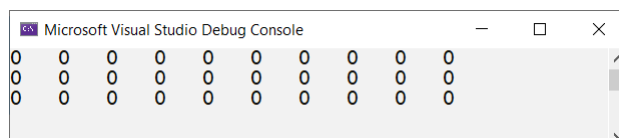
for (int i = 0; i < N; i++)
    cout << b[i] << " ";
cout << endl;

for (int i = 0; i < N; i++)
    cout << c[i] << " ";
cout << endl;

return 0;
}

```

### Результат виконання:



## Варіанти завдань

У наступних завданнях потрібно реалізувати дружні операції вводу/виводу двома способами: як зовнішні функції-шаблони і як «нешаблонні», визначені всередині класу. У всіх завданнях реалізувати ітератор як набір методів класу. У головній функції продемонструвати не менше двох екземплярів шаблонів з різними аргументами і виклик всіх методів.

Варіанти завдань наступні:

### Варіант 1.

Узявши за зразок шаблон `Array` (див. лістинг 5), реалізувати шаблон масиву з межами індексів, що задаються.

### Варіант 2.

Узявши за зразок шаблон `Array` (див. лістинг 5), реалізувати шаблон числового масиву фіксованої довжини:

Використовуючи лістинг 1, реалізувати масив дійсних чисел з повним набором арифметичних операцій та операціями перевірки на рівність та нерівність.

### Варіант 3.

Реалізувати шаблон `Array` як клас-шаблон числового масиву з межами індексів, що задаються.

### Варіант 4.

Реалізувати динамічний числовий масив фіксованої довжини як клас-шаблон:



Створити клас `Array` – одновимірний масив цілих чисел, з границями індексів, які задаються, та з можливістю від’ємних індексів. Реалізувати відстеження кількості елементів, всі арифметичні операції з масивом та цілим числом, пошук заданого елемента.

### **Варіант 5.**

Реалізувати масив зростаючої довжини (див. лістинг 2) у вигляді класу-шаблону.

### **Варіант 6.**

Розробити шаблон числового масиву зростаючої довжини (див. лістинг 2) додавши всі арифметичні операції та операції перевірки на рівність і нерівність.

### **Варіант 7.**

Реалізувати шаблон стеку у вигляді масиву зростаючої довжини (див. лістинг 2).

### **Варіант 8.**

Розробити шаблон «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Додавання і видалення елементів виконувати тільки поодиночі в довільному місці.

### **Варіант 9.**

Розробити шаблон числового «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Реалізувати всі арифметичні операції і операції перевірки на рівність і нерівність.

### **Варіант 10.**

Реалізувати шаблон черги у вигляді «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Додавання і видалення елементів виконувати тільки по-одному; додавання – в кінці масиву, видалення – на початку.

### **Варіант 11.**

Реалізувати шаблон двосторонньої черги у вигляді «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Додавання і видалення елементів виконувати тільки по-одному на початку та в кінці.

### **Варіант 12.**

Розробити шаблон «гнучкого» масиву, реалізувавши групові додавання, видалення і заміну елементів. Група представляється діапазоном ітераторів. Реалізувати методи пошуку

елементів і груп елементів.

### **Варіант 13.**

Реалізувати шаблон масиву зростаючої довжини (див. лістинг 2) як двонаправлений список. У шаблоні мають бути лише методи, представлені в лістингу 2. Операції доступу за індексом замінити ітератором.

### **Варіант 14.**

Реалізувати шаблон числового масиву зростаючої довжини (див. лістинг 2) як двонаправлений список. Реалізувати всі арифметичні операції та операції перевірки на рівність і нерівність. Операції доступу за індексом замінити ітератором.

### **Варіант 15.**

Розробити шаблон списку. Додавання і видалення елементів виконується тільки по одному в довільному місці списку – за допомогою ітератора.

### **Варіант 16.**

Розробити шаблон списку, що підтримує групові додавання, видалення і заміну елементів. Група представляється діапазоном ітераторів. Реалізувати методи пошуку елементів і груп елементів.

### **Варіант 17.**

Реалізувати стек у вигляді шаблону однонаправленого списку (див. лістинг 4 в параграфі Теоретичні відомості. Контейнери. Реалізація контейнерів. Стэк).

### **Варіант 18.**

Реалізувати чергу у вигляді шаблону двонаправленого списку (див. лістинг 3).

### **Варіант 19.**

Реалізувати двосторонню чергу у вигляді шаблону двонаправленого списку (див. лістинг 3).

### **Варіант 20.**

Реалізувати шаблон списку, що самоорганізується. Основною операцією є операція пошуку. При знаходженні елементу він відкріплюється зі свого місця і вставляється в початок списку.

## **Варіант 21.**

Розробити шаблон списку. Додавання і видалення елементів виконувати тільки по одному в довільному місці списку – за допомогою ітератора. Підтримати методи простого злиття і злиття впорядкованих списків.

## **Варіант 22.**

Розробити шаблон списку, реалізувавши групові додавання, видалення і заміну елементів. Група представляється діапазоном ітераторів. Реалізувати методи пошуку елементів і груп елементів. Додати методи комбінування списків як об'єднання і перетин множин.

## **Варіант 23.**

Реалізувати шаблон стеку, визначивши параметр-шаблон. Як аргумент використовувати шаблон масиву зростаючої довжини (завдання 5) і шаблон двосторонньої черги (завдання 11).

## **Варіант 24.**

Реалізувати шаблон черги, визначивши параметр-шаблон. Як аргумент використовувати шаблон двосторонньої черги (завдання 19) і шаблон списку (завдання 15).

## **Варіант 25.**

Реалізувати шаблон стеку, визначивши параметр-шаблон. Як аргумент використовувати шаблон черги (завдання 10) і шаблон двосторонньої черги (завдання 19).

## **Варіант 26.**

Реалізувати шаблон множини, визначивши параметр-шаблон. Як аргумент використовувати шаблон «гнучкого» масиву (завдання 12) і шаблон списку (завдання 16).

## **Варіант 27.**

Реалізувати шаблон черги, визначивши параметр-шаблон. Як аргумент використовувати шаблон «гнучкого» масиву (завдання 8) і шаблон списку (завдання 15).

## **Варіант 28.**

Реалізувати шаблон двосторонньої черги, визначивши параметр-шаблон. Як аргумент використовувати шаблон «гнучкого» масиву (завдання 8) і шаблон списку (завдання 15).

### **Варіант 29.**

Реалізувати шаблон черги, визначивши параметр-шаблон. Аргумент – шаблон «гнучкого» масиву (завдання 12) і шаблон списку (завдання 16).

### **Варіант 30.**

Узявши за зразок шаблон `Array` (див. лістинг 5), реалізувати шаблон масиву з межами індексів, що задаються.

### **Варіант 31.**

Узявши за зразок шаблон `Array` (див. лістинг 5), реалізувати шаблон числового масиву фіксованої довжини:

Використовуючи лістинг 1, реалізувати масив дійсних чисел з повним набором арифметичних операцій та операціями перевірки на рівність та нерівність.

### **Варіант 32.**

Реалізувати шаблон `Array` як клас-шаблон числового масиву з межами індексів, що задаються.

### **Варіант 33.**

Реалізувати динамічний числовий масив фіксованої довжини як клас-шаблон:

Створити клас `Array` – одновимірний масив цілих чисел, з границями індексів, які задаються, та з можливістю від'ємних індексів. Реалізувати відстеження кількості елементів, всі арифметичні операції з масивом та цілим числом, пошук заданого елемента.

### **Варіант 34.**

Реалізувати масив зростаючої довжини (див. лістинг 2) у вигляді класу-шаблону.

### **Варіант 35.**

Розробити шаблон числового масиву зростаючої довжини (див. лістинг 2) додавши всі арифметичні операції та операції перевірки на рівність і нерівність.

### **Варіант 36.**

Реалізувати шаблон стеку у вигляді масиву зростаючої довжини (див. лістинг 2).

### **Варіант 37.**

Розробити шаблон «гнучкого» масиву, узявши за зразок динамічний масив зростаючої

довжини (див. лістинг 2). Додавання і видалення елементів виконувати тільки поодиночі в довільному місці.

### **Варіант 38.**

Розробити шаблон числового «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Реалізувати всі арифметичні операції і операції перевірки на рівність і нерівність.

### **Варіант 39.**

Реалізувати шаблон черги у вигляді «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Додавання і видалення елементів виконувати тільки по-одному; додавання – в кінці масиву, видалення – на початку.

### **Варіант 40.**

Реалізувати шаблон двосторонньої черги у вигляді «гнучкого» масиву, узявши за зразок динамічний масив зростаючої довжини (див. лістинг 2). Додавання і видалення елементів виконувати тільки по-одному на початку та в кінці.

## Бонуси

- \* Виконати завдання 1-40, визначивши ітератор як вкладений клас.
- \* Виконати завдання 1-40, додавши конструктор копіювання і операцію присвоєння у вигляді шаблонів.
- \* Виконати завдання 1-10, 25-34 Лабораторної роботи № 6.2, оголосивши в класі, що реалізовується, масив як поле-шаблон `Array` (див. лістинг 5).
- \* Реалізувати класи в завданнях 1-10, 25-34 Лабораторної роботи № 6.2 як спеціалізації шаблону `Array` (див. лістинг 5).
- \* Виконати завдання 1-10, 25-34 Лабораторної роботи № 6.2, визначивши класи як спеціалізації шаблону числового масиву фіксованої довжини (див. завдання 4).
- \* Виконати завдання 1-10, 25-34 Лабораторної роботи № 6.2, оголосивши класи як нащадки шаблону `Array` (див. лістинг 5).
- \* Реалізувати класи в завданнях 1-10, 25-34 Лабораторної роботи № 6.2 як нащадки шаблону числового масиву фіксованої довжини (див. завдання 4).
- \* Оформити класи в завданнях 11-24, 35-40 Лабораторної роботи № 6.2 як спеціалізації шаблону масиву зростаючої довжини, реалізованого у вигляді списку (див. завдання 14).
- \* Оформити класи в завданнях 11-24, 35-40 Лабораторної роботи № 6.2 як шаблони-нащадки, спираючись на шаблон масиву зростаючої довжини, реалізованого у вигляді списку (див. завдання 14).
- \* Виконати завдання 11-24, 35-40 Лабораторної роботи № 6.2, представивши класи як шаблони-нащадки. Як базовий потрібно використовувати шаблон масиву зростаючої довжини (див. завдання 5).
- \* Реалізувати динамічний числовий масив фіксованої довжини (див. завдання 7 Лабораторної роботи № 6.2) як абстрактний клас-шаблон, оголосивши арифметичні операції чистими віртуальними функціями. Реалізувати завдання 11-24, 35-40 Лабораторної роботи № 6.2 як шаблони-нащадки.

# Лабораторна робота № 6.7. Шаблони функцій, алгоритми та функтори

## Мета роботи

Освоїти опрацювання шаблоні функцій, узагальнених алгоритмів та функторів.

## Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення шаблону класу.
- 2) Поняття та призначення шаблону функції.
- 3) Параметри шаблону класу.
- 4) Параметри шаблону функції.
- 5) Спеціалізація шаблону класу.
- 6) Спеціалізація шаблону функції.
- 7) Перевантаження шаблону функції.
- 8) Статичні елементи в шаблонах класів.
- 9) Механізм опрацювання винятків в шаблоні класу.
- 10) Шаблон класу з елементами-шаблонами.
- 11) Шаблиони та успадкування.
- 12) Шаблиони та дружні функції.
- 13) Поняття та призначення функтора.
- 14) Узагальнені алгоритми та функтори.

## Приклад виконання завдання

Для контейнерів із завдань Лабораторних робіт 6.1-6.3 або для інших контейнерів виконати завдання (використовувати шаблони функцій та функтори). Функція `***_if` має приймати умову відбору як параметр-функтор.

Реалізувати алгоритми дублювання елементів `duplicate()` та дублювання тих елементів, які задовольняють умові `duplicate_if()`.

```
#include <iostream>

using namespace std;

// інтерфейс, що описує функтори - унарні предикати
template<class T>
class Predicate
{
public:
    virtual bool operator () (T x) = 0;
};
```

```

// реалізуємо інтерфейс функтором - перевірка, чи значення дорівнює нулю
template<class T>
class Zero : public Predicate<T>
{
public:
    virtual bool operator () (T x)
    {
        T zero = T();
        return x == zero;
    }
};

// реалізуємо інтерфейс функтором - перевірка, чи значення додатне
// працює лише для числових типів
template<class T>
class Positive : public Predicate<T>
{
public:
    virtual bool operator () (T x)
    {
        return x > 0;
    }
};

// реалізуємо інтерфейс функтором - перевірка, чи значення від'ємне
// працює лише для числових типів
template<class T>
class Negative : public Predicate<T>
{
public:
    virtual bool operator () (T x)
    {
        return x < 0;
    }
};

// реалізуємо інтерфейс функтором - перевірка, чи значення парне
// працює лише для цілих типів
template<class T>
class Even : public Predicate<T>
{
public:
    virtual bool operator () (T x)
    {
        return x % 2 == 0;
    }
};

// реалізуємо інтерфейс функтором - перевірка, чи значення не парне
// працює лише для цілих типів
template<class T>
class Odd : public Predicate<T>
{
public:
    virtual bool operator () (T x)
    {
        return x % 2 != 0;
    }
};

// дублюємо всі елементи
// begin - ітератор початку вхідного контейнера (вказує на перший елемент)

```



```

// end - ітератор кінця вхідного контейнера (вказує на елемент після останнього)
// to - ітератор початку результуючого контейнера (вказує на перший елемент)
template<class T>
void duplicate(T *begin, T *end, T *to)
{
    for (T* from = begin; from < end; from++) // from - ітератор вхідного контейнера
    {                                           // вказує на поточний елемент
        *to = *from;
        to++;
        *to = *from;
        to++;                                // два рази скопіювали кожний елемент
    }
}

// дублюємо елементи, для яких предикат p набуває значення true
// begin - ітератор початку вхідного контейнера (вказує на перший елемент)
// end - ітератор кінця вхідного контейнера (вказує на елемент після останнього)
// to - ітератор початку результуючого контейнера (вказує на перший елемент)
template<class T>
int duplicate_if(T *begin, T *end, T *to, Predicate<T> &p)
{
    int n = 0; // n - кількість елементів у результуючому контейнері
    for (T* from = begin; from < end; from++) // from - ітератор вхідного контейнера
    {                                           // вказує на поточний елемент
        *to = *from;
        to++;                                // скопіювали елемент
        n++;
        if ( p(*from) )                       // якщо справджується умова предикату
        {
            *to = *from;
            to++;                            // то скопіювали цей елемент ще раз
            n++;
        }
    }
    return n;
}

int main()
{
    int a[5] = { 1, -2, 0, 4, -5 };           // вхідний контейнер
    int b[10], c[10], d[10];                 // результуючі контейнери
                                            // (потрібно виділити достатньо пам'яті)

    duplicate(&a[0], &a[5], &b[0]);           // продублювали всі елементи

    for (int i = 0; i < 10; i++)
        cout << b[i] << ' ';               // вивели результат
    cout << endl;

    Predicate<int> *zero = new Zero<int>();   // функтор: "нульові елементи"
    int n = duplicate_if(&a[0], &a[5], &c[0], *zero); // продублювали нульові елементи

    for (int i = 0; i < n; i++)
        cout << c[i] << ' ';
    cout << endl;

    Predicate<int> *pos = new Positive<int>(); // функтор: "додатні елементи"
    int k = duplicate_if(&a[0], &a[5], &d[0], *pos); // продублювали додатні елементи

    for (int i = 0; i < k; i++)
        cout << d[i] << ' ';
    cout << endl;
}

```

```
    system("pause");  
    return 0;  
}
```

## **Варіанти завдань**

Для контейнерів із завдань Лабораторних робіт 6.1-6.3 або для інших контейнерів виконати завдання (використовувати шаблони функцій та функтори). Функція `***_if` має приймати умову відбору як параметр-функтор.

### **Варіант 1.**

Реалізувати алгоритми пошуку мінімального значення `min_element()` та мінімального значення з тих, які задовольняють умові `min_element_if()`.

### **Варіант 2.**

Реалізувати алгоритми обчислення кількості елементів `count()` та кількості тих елементів, які задовольняють умові `count_if()`.

### **Варіант 3.**

Реалізувати алгоритм копіювання `copy()` вхідного контейнера в вихідний і алгоритм копіювання тих елементів, які задовольняють умові `copy_if()`.

### **Варіант 4.**

Реалізувати алгоритми видалення `erase()` та видалення тих елементів, які задовольняють умові `erase_if()`.

### **Варіант 5.\***

Реалізувати алгоритми пошуку `search()` (серед всіх елементів) та `search_if()` (серед тих елементів, які задовольняють умові) першого входження послідовності елементів в іншу послідовність.

### **Варіант 6.\***

Реалізувати алгоритми пошуку `search_end()` (серед всіх елементів) та `search_end_if()` (серед тих елементів, які задовольняють умові) останнього входження послідовності елементів в іншу.

### **Варіант 7.\***

Реалізувати алгоритми порівняння послідовностей `equal()` (серед всіх елементів) та

`equal_if( )` (серед тих елементів, які задовольняють умові).

### **Варіант 8.**

Реалізувати алгоритми видалення всіх елементів `erase_copy( )` та видалення тих елементів, які задовольняють умові `erase_copy_if( )`. Послідовності, що видаляються, копіюються у вихідний контейнер.

### **Варіант 9.**

Реалізувати алгоритми сортування всіх елементів `sort( )` та сортування тих елементів, які задовольняють умові `sort_if( )`.

### **Варіант 10.\***

Реалізувати алгоритми злиття сортованих послідовностей `merge( )` (всіх елементів) та `merge_if( )` (тих елементів, які задовольняють умові).

### **Варіант 11.\***

Реалізувати алгоритми об'єднання послідовностей як множин `set_or( )` (всіх елементів) та `set_or_if( )` (тих елементів, які задовольняють умові).

### **Варіант 12.\***

Реалізувати алгоритми перетину послідовностей як множин `set_and( )` (всіх елементів) та `set_and_if( )` (тих елементів, які задовольняють умові).

### **Варіант 13.\***

Реалізувати алгоритми симетричного віднімання послідовностей як множин `set_xor( )` (всіх елементів) та `set_xor_if( )` (тих елементів, які задовольняють умові).

### **Варіант 14.\***

Реалізувати алгоритми заміни однієї послідовності на іншу `replace_copy( )` (всіх елементів) та `replace_copy_if( )` (тих елементів, які задовольняють умові). Результат записується у вихідну послідовність.

### **Варіант 15.**

Розробити алгоритм `for_each( )`, що застосовує заданий функтор до кожного елемента послідовності. Реалізувати алгоритм `generate( )`, що застосовує заданий функтор для заповнення елементів контейнера.

## Варіант 16.

Реалізувати алгоритми `accumulate( )` та `accumulate_if( )`, що виконують накопичення всіх елементів послідовності чи тих елементів, які задовольняють умові. Спосіб накопичення визначається функтором, який передається в якості аргументу.

## Варіант 17.

Реалізувати алгоритми `binoperate( )` та `binoperate_if( )`, що виконують задану функтором операцію з двома інтервалами для всіх елементів чи тих елементів, які задовольняють умові, і записують результат у вихідну послідовність.

## Варіант 18.

Реалізувати алгоритми `unique_copy( )` та `unique_copy_if( )` видалення всіх елементів-дублікатів (що повторюються) чи тих елементів-дублікатів (що повторюються), які задовольняють умові, із послідовності. Результат записується у вихідну послідовність.

## Варіант 19.

Реалізувати алгоритми `reverse_copy( )` та `reverse_copy_if( )` «обернення» послідовності: всіх елементів чи тих елементів, які задовольняють умові. Результат записується у вихідну послідовність.

## Варіант 20.

Реалізувати алгоритми сортування всіх елементів чи тих елементів, які задовольняють умові: `sort_copy( )` та `sort_copy_if( )`. Результат записується у вихідну послідовність.

## Варіант 21.

Реалізувати алгоритми пошуку максимального значення `max_element( )` та максимального значення з тих елементів, які задовольняють умові `max_element_if( )`.

## Варіант 22.

Реалізувати алгоритми обчислення суми елементів `sum( )` та суми тих елементів, які задовольняють умові `sum_if( )`.

## Варіант 23.

Реалізувати алгоритм копіювання `copy( )` вхідного контейнера в вихідний і алгоритм копіювання тих елементів, які задовольняють умові `copy_if( )`.

### **Варіант 24.**

Реалізувати алгоритми інверсії елементів `inverse( )` та інверсії тих елементів, які задовольняють умові `inverse_if( )`.

### **Варіант 25.**

Реалізувати алгоритми пошуку серед всіх елементів чи серед тих елементів, які задовольняють умові `search( )` та `search_if( )` першого входження однієї послідовності елементів в іншу послідовність.

### **Варіант 26.**

Реалізувати алгоритми пошуку мінімального значення `min_element( )` та мінімального значення з тих, які задовольняють умові `min_element_if( )`.

### **Варіант 27.**

Реалізувати алгоритми обчислення кількості елементів `count( )` та кількості тих елементів, які задовольняють умові `count_if( )`.

### **Варіант 28.**

Реалізувати алгоритм копіювання `copy( )` вхідного контейнера в вихідний і алгоритм копіювання тих елементів, які задовольняють умові `copy_if( )`.

### **Варіант 29.**

Реалізувати алгоритми видалення `erase( )` та видалення тих елементів, які задовольняють умові `erase_if( )`.

### **Варіант 30.\***

Реалізувати алгоритми пошуку `search( )` (серед всіх елементів) та `search_if( )` (серед тих елементів, які задовольняють умові) першого входження послідовності елементів в іншу послідовність.

### **Варіант 31.\***

Реалізувати алгоритми пошуку `search_end( )` (серед всіх елементів) та `search_end_if( )` (серед тих елементів, які задовольняють умові) останнього входження послідовності елементів в іншу.

### **Варіант 32.\***

Реалізувати алгоритми порівняння послідовностей `equal( )` (серед всіх елементів) та `equal_if( )` (серед тих елементів, які задовольняють умові).

### **Варіант 33.**

Реалізувати алгоритми видалення всіх елементів `erase_copy( )` та видалення тих елементів, які задовольняють умові `erase_copy_if( )`. Послідовності, що видаляються, копіюються у вихідний контейнер.

### **Варіант 34.**

Реалізувати алгоритми сортування всіх елементів `sort( )` та сортування тих елементів, які задовольняють умові `sort_if( )`.

### **Варіант 35.\***

Реалізувати алгоритми злиття сортованих послідовностей `merge( )` (всіх елементів) та `merge_if( )` (тих елементів, які задовольняють умові).

### **Варіант 36.\***

Реалізувати алгоритми об'єднання послідовностей як множин `set_or( )` (всіх елементів) та `set_or_if( )` (тих елементів, які задовольняють умові).

### **Варіант 37.\***

Реалізувати алгоритми перетину послідовностей як множин `set_and( )` (всіх елементів) та `set_and_if( )` (тих елементів, які задовольняють умові).

### **Варіант 38.\***

Реалізувати алгоритми симетричного віднімання послідовностей як множин `set_xor( )` (всіх елементів) та `set_xor_if( )` (тих елементів, які задовольняють умові).

### **Варіант 39.\***

Реалізувати алгоритми заміни однієї послідовності на іншу `replace_copy( )` (всіх елементів) та `replace_copy_if( )` (тих елементів, які задовольняють умові). Результат записується у вихідну послідовність.

### **Варіант 40.**

Розробити алгоритм `for_each( )`, що застосовує заданий функтор до кожного

елементу послідовності. Реалізувати алгоритм `generate()`, що застосовує заданий функтор для заповнення елементів контейнера.

# ***Питання та завдання для контролю знань***

## **Контейнери**

1. Що таке контейнер?
2. Які види вбудованих контейнерів в C++ Ви знаєте?
3. Які види операцій над контейнерами в C++ Ви знаєте?
4. Які способи доступу до елементів контейнера Вам відомі?
5. Чим відрізняється прямий доступ від асоціативного ?
6. Перелічіть операції, які реалізуються зазвичай при послідовному доступі до елементів контейнера.
7. Перелічіть операції, які реалізуються зазвичай для об'єднання контейнерів.
8. Що таке ітератор? (поясніть поняття та дайте визначення)
9. Що служить ітератором для вбудованого масиву?
10. Що служить ітератором для зв'язаного списку?
11. Чи можна реалізувати послідовний доступ без ітератора?
12. В чому перевага реалізації послідовного доступу за допомогою ітератора?
13. Дайте визначення вкладеного класу
14. Чи можна клас-ітератор реалізувати як зовнішній клас?
15. Чи можна клас-ітератор реалізувати як вкладений клас?
16. В чому полягає різниця між реалізацією класу-ітератора як вкладеного та як зовнішнього класу?
17. Чи може оточуючий клас мати повний доступ до елементів вкладеного класу?
18. Чи може вкладений клас мати повний доступ до елементів оточуючого класу?
19. Чи обмежена глибина вкладеності класів?
20. Яким чином оточуючий клас може використовувати методи вкладеного класу?
21. Яким чином вкладений клас може використовувати методи оточуючого класу?
22. Що таке «елемент за межею» та яку роль він відіграє в контейнерах?
23. Що таке стек?
24. Що таке черга?
25. Що таке дек?
26. Призначення методів `begin()` та `end()`
27. Чи обов'язково при реалізації контейнерів програмувати операцію присвоєння?
28. Чи обов'язково при реалізації контейнерів програмувати конструктор копіювання?
29. Наведіть мінімальний інтерфейс, який необхідно реалізувати при розробці класу-контейнера масиву (напишіть фрагмент визначення класу).
30. Наведіть мінімальний інтерфейс, який необхідно реалізувати при розробці класу-контейнера стеку (напишіть фрагмент визначення класу).



31. Поясніть, чому складно написати універсальний контейнер, елементи якого можуть мати довільний тип?

## Шаблони класів

1. Для чого призначені шаблони?
2. Які види шаблонів в мові C++ Ви знаєте?
3. Поясніть термін «інстанціювання шаблону».
4. Яка різниця між оголошенням та визначенням шаблону.
5. Поясніть призначення ключового слова `typename`.
6. Які види параметрів можна вказувати в шаблоні класу?
7. Чи можна параметрам шаблону присвоювати значення за умовчанням?
8. Чи може параметром шаблону бути інший шаблон? Які особливості оголошення параметра-шаблону?
9. Що таке спеціалізація шаблону? Поясніть різницю між повною і частковою спеціалізацією.
10. Чи потрібно вказувати визначення первинного шаблону для визначення спеціалізованих версій, чи достатньо оголошення?
11. Чи може шаблонний клас бути вкладений в інший шаблонний клас? А в звичайний клас?
12. Чи можна в класі оголосити шаблонний метод? А шаблонний конструктор?
13. Чи може шаблон класу бути нащадком звичайного класу? А звичайний клас – нащадком шаблонного?
14. Чи може вкладений клас бути шаблоном?
15. Чи можна звичайний клас зробити внутрішнім класом шаблону? Чи існують якісь обмеження в цьому випадку?
16. Яким чином можна використати можливість успадковування звичайного класу від шаблону?
17. Чи може шаблонний конструктор бути конструктором за умовчанням?
18. Чи може шаблонний клас мати «друзів»?
19. Які проблеми виникають при оголошенні дружньої функції для шаблонного класу?
20. Чи можна визначати в шаблонному класі статичні поля? А статичні методи?

## Шаблони функцій

1. Вкажіть, чим відрізняється шаблон функції від шаблону класу (вказати всі відмінності).
2. Чи можна перевантажувати функцію-шаблон?
3. Які параметри функції-шаблону виводяться автоматично?
4. Чи можна параметрам шаблону функції присвоювати значення за умовчанням?
5. Яким чином «обійти» обмеження часткової спеціалізації для шаблонів функцій?
6. Вкажіть прийоми програмування, за допомогою яких шаблон функції стає більш універсальним.
7. Дайте визначення функтора.
8. Чи можна операцію виклику функції реалізувати як віртуальну?

9. Чи можна операцію виклику функції реалізувати як статичну?
10. Дайте визначення функції-предикату.
11. Чим відрізняється функтор зі станом від звичайного функтора?
12. Чи можна вказівник на функцію присвоїти вказівнику на метод?
13. Чи можна вказівник на метод присвоїти вказівнику на функцію?
14. Чи відрізняється вказівник на звичайний метод від вказівника на статичний метод?
15. Чи може клас-функтор приймати участь в ієрархії успадковування?
16. Поясніть призначення адаптера-фіксатора.
17. Чи можна перевантажувати операцію виклику функції?

## Шаблони

1. Визначити, чи наступні твердження істинні чи хибні. Якщо твердження хибне, поясніть чому.
  - a) Функція, дружна по відношенню до шаблону класу, повинна бути шаблонною функцією.
  - b) Якщо кілька шаблонних класів інстанційовані від одного і того ж самого шаблону класу з єдиним статичним полем, то кожний із шаблонних класів спільно використовує єдиний екземпляр статичного поля.
  - c) Шаблонна функція може бути перевантажена іншою шаблонною функцією з тим ж самим іменем.
  - d) Ім'я формального параметру можна використовувати лише один раз в списку формальних параметрів шаблону: імена формальних параметрів шаблону мають бути унікальними серед всіх шаблонів.
  - e) Ключові слова `class` та `typename`, які використовуються для параметрів-типів шаблону, означають: «будь-який тип, що визначається користувачем».
2. Заповніть пропуски в кожному з наступних тверджень:
  - a) Шаблони дають можливість визначити за допомогою одного фрагменту коду групу пов'язаних функцій, які називаються \_\_\_\_\_, або групу пов'язаних класів, які називаються \_\_\_\_\_.
  - b) Всі описи шаблонів функцій починаються з ключового слова \_\_\_\_\_, за яким йде список формальних параметрів шаблону, записаний в \_\_\_\_\_.
  - c) Всі функції, інстанційовані від одного шаблону функції, мають одне і те ж саме ім'я, тому компілятор використовує механізм \_\_\_\_\_ для того, щоб забезпечити виклик відповідної функції.
  - d) Шаблони класів ще називаються \_\_\_\_\_ типами.
  - e) \_\_\_\_\_ операція \_\_\_\_\_ використовується з іменем шаблону класу для того, щоб пов'язати визначення методу з областю дії шаблону класу.
  - f) Як і статичні поля не шаблонних класів, статичні поля шаблонів класів мають бути ініціалізовані в області дії \_\_\_\_\_.

3. Написати простий шаблон предикатної функції `isEqualTo()`, яка порівнює два своїх параметри за допомогою операції перевірки на рівність (`==`) і повертає `true`, якщо вони однакові, та `false`, якщо відрізняються. Використати цей шаблон в програмі, яка викликає `isEqualTo()` з різними типами аргументів. Написати окрему версію програми, яка викликає `isEqualTo()` з визначеним користувачем типом та не перевантаженою операцією перевірки на рівність. Що відбудеться, якщо поспробувати виконати цю програму? Тепер перевантажте операцію перевірки на рівність (використовуйте функцію-операцію `operator==`). Що тепер відбудеться, якщо поспробувати виконати цю програму?
4. Поясніть різницю між шаблоном функції та шаблонною функцією.
5. Що можна порівняти з трафаретом: шаблон класу чи шаблонний клас? Відповідь пояснити.
6. Який зв'язок між шаблонами функцій та перевантаженням?
7. Чому краще використовувати шаблони функцій, а не макроси?
8. Як може вплинути на ефективність програми використання шаблонів функцій та шаблонів класів?
9. При виклику функції компілятор обирає шаблонну функцію, яка відповідає цьому виклику. За яких обставин цей процес закінчується помилкою компіляції?
10. Чому шаблон класу називають параметризованим типом?
11. Поясніть, чому шаблони класів для контейнерів типу масив або стек часто використовують нетипові параметри?
12. Опишіть, як потрібно визначити клас для специфікованого типу, щоб перевизначити шаблон класу саме для цього типу.
13. Опишіть зв'язок між шаблонами класу та успадковуванням.
14. Припустимо, що шаблон класу має заголовок
- ```
template <class T1> class C1
```
- Опишіть відношення дружності, які виникають, коли всередині шаблону класу помістити наведені нижче команди, що оголошують друзів. Ідентифікатори, які починаються з символу «f» – це функції; ідентифікатори, які починаються з символу «C» – це класи; ідентифікатори, які починаються з символу «T» – позначають будь-який тип (тобто, вбудований тип або тип класу)
- a) `friend void f1();`
  - b) `friend void f2( C1<T1> & );`
  - c) `friend void C2::f4();`
  - d) `friend void C3<T1>::f5( C1<T1> & );`
  - e) `friend class C5;`
  - e) `friend class C6<T1>;`
15. Припустимо, що шаблон класу `Employee` має статичне поле `count`. Припустимо, що із цього шаблону були інстанційовані три шаблонні класи. Скільки екземплярів статичного поля буде існувати в такому випадку? Як буде використовуватися кожний із них (якщо вони взагалі будуть існувати)?

16. Для шаблону

```
template <class A, class B> // шаблон класу
class C
{
    ...                      // елементи шаблону
};
```

Визначити шаблонні класи, які відповідають вказаним способам інстанціювання:

- a) `A = int*`, `B = void*`
- b) `A = double`, `B = void*`
- c) `A = int*`, `B = double`
- d) `A = int*`, `B = int`
- e) `A = double*`, `B = int`
- f) `A = double*`, `B = void*`
- g) `A = int*`, `B = double*`

17. Для первинного шаблону

```
template <class T, class U> // первинний шаблон
class S
{
public:
    void p();                // метод, який буде перевизначено
};                          // в спеціалізованих версіях
```

визначити вказані спеціалізовані версії:

- a) часткову спеціалізацію для `U = int`
- b) часткову спеціалізацію для `T = int`
- c) часткову спеціалізацію для `T = U`
- d) часткову спеціалізацію для вказівників `T*`, `U*`
- e) повну спеціалізацію для вказівників `T* = U* = void*`
- f) часткову спеціалізацію для вказівників `T*`, `U* = void*`
- g) повну спеціалізацію для `T = int*`, `U = double`

# Предметний покажчик

|                                        |          |                                      |                                       |
|----------------------------------------|----------|--------------------------------------|---------------------------------------|
|                                        | <b>В</b> |                                      | <b>П</b>                              |
| begin, 31                              |          | Параметри шаблону, 21                |                                       |
|                                        | <b>Е</b> | Параметризований тип, 20, 22, 53, 78 |                                       |
| end, 31                                |          | Параметр-шаблон, 67                  |                                       |
|                                        | <b>У</b> | Перевантаження шаблону функції, 84   |                                       |
| UML-діаграми, 77                       |          | Поле-шаблон, 66                      |                                       |
|                                        | <b>А</b> | Послідовний доступ, 18, 19, 29, 52   |                                       |
| Адаптер функтора, 26, 94, 103          |          | Предикат, 86                         |                                       |
| Асоціативний доступ, 18, 19, 29, 52    |          | бінарний, 86                         |                                       |
|                                        | <b>В</b> | унарний, 86                          |                                       |
| Вид контейнера, 17, 28                 |          | Прямий доступ, 18, 19, 29, 52        |                                       |
| Вказівник на метод, 88                 |          |                                      | <b>С</b>                              |
| Вказівник на функцію, 88               |          |                                      | Спеціалізація шаблону, 21, 23, 61, 79 |
|                                        | <b>І</b> |                                      | повна, 21, 61                         |
| Інстанціювання, 20, 22, 23, 61, 78, 79 |          |                                      | часткова, 21, 61                      |
| Ітератор, 19, 30, 52                   |          |                                      | Спеціалізація шаблону функції, 84     |
| поняття, 30                            |          |                                      |                                       |
|                                        | <b>К</b> |                                      | <b>У</b>                              |
| Клас-обгортка, 83                      |          |                                      | Узагальнений алгоритм, 25, 84, 102    |
| Клас-функтор, 92                       |          |                                      | реалізація з функтором, 93            |
| Контейнер, 17, 19, 28, 52              |          |                                      |                                       |
| доступ до елементів, 29                |          |                                      | <b>Ф</b>                              |
| поняття, 17, 28                        |          |                                      | Фіксатор функтора, 27, 96, 103        |
| реалізація, 33                         |          |                                      | Функтор, 25, 26, 91, 103              |
|                                        | <b>М</b> |                                      | бінарний, 92                          |
| Метод-шаблон, 68                       |          |                                      | унарний, 92                           |
|                                        | <b>О</b> |                                      | Функтор-предикат, 92                  |
| Операції контейнера, 31                |          |                                      |                                       |
|                                        |          |                                      | <b>Ш</b>                              |
|                                        |          |                                      | Шаблон класу, 20, 25, 78              |
|                                        |          |                                      | параметри, 53                         |
|                                        |          |                                      | параметри за умовчанням, 60           |
|                                        |          |                                      | поняття, 53                           |
|                                        |          |                                      | Шаблон функції, 24, 25, 81            |
|                                        |          |                                      | Шаблони та друзі, 73                  |
|                                        |          |                                      | Шаблони та успадкування, 71           |
|                                        |          |                                      | Шаблонна функція, 26, 102             |
|                                        |          |                                      | Шаблонний клас, 22, 63, 78            |

# Література

## Основна

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня СПб.: Питер, 2007. – 461 с.
2. Павловская Т.А., Щупак Ю.А. С/С++. Объектно-ориентированное программирование: Практикум СПб.: Питер, 2005. – 265 с.
3. Дейтел Х.М., Дейтел П.Дж. Как программировать на С++ М.: Бином-Пресс, 2005. – 1248 с.
4. Уэллин С. Как не надо программировать на С++ СПб.: Питер, 2004. – 240 с.
5. Хортон А. Visual C++ 2005: Базовый курс М.: Вильямс, 2007. – 1152 с.
6. Солтер Н.А., Клеппер С.Дж. С++ для профессионалов. М.: Вильямс, 2006. – 912 с.
7. Лафоре Р. Объектно-ориентированное программирование в С++ СПб.: Питер, 2006. – 928 с.
8. Лаптев В.В. С++. Объектно-ориентированное программирование СПб.: Питер, 2008. – 464 с.
9. Лаптев В.В., Морозов А.В., Бокова А.В. С++. Объектно-ориентированное программирование. Задачи и упражнения СПб.: Питер. – 2007. – 288 с.: ил.

## Додаткова

10. Прата С. Язык программирования С++. Лекции и упражнения СПб.: ДиаСофт, 2003. – 1104 с.
11. Мейн М., Савитч У. Структуры данных и другие объекты в С++ М.: Вильямс, 2002. – 832 с.
12. Саттер Г. Решение сложных задач на С++ М.: Вильямс, 2003. – 400 с.
13. Чепмен Д. Освой самостоятельно Visual C++ .NET за 21 день М: Вильямс, 2002. – 720 с.
14. Мартынов Н.Н. Программирование для Windows на С/С++ М.: Бином-Пресс, 2004. – 528 с.
15. Паппас К., Мюррей У. Эффективная работа: Visual C++ .NET СПб.: Питер, 2002. – 816 с. М.: Вильямс, 2001. – 832 с.
16. Грэхем И. Объектно-ориентированные методы. Принципы и практика М.: Вильямс, 2004. – 880 с.
17. Элиенс А. Принципы объектно-ориентированной разработки программ М.: Вильямс, 2002. – 496 с.

18. Ларман К. Применение UML и шаблонов проектирования М.: Вильямс, 2002. – 624 с.
19. Шилэт Г. Полный справочник по С. 4-е издание. М.-СПб.-К: Вильямс, 2002.
20. Прата С. Язык программирования С. Лекции и упражнения. М.: ДиаСофтЮП, 2002.
21. Александреску А. Современное проектирование на С++ М.: Вильямс, 2002.
22. Браунси К. Основные концепции структур данных и реализация в С++ М.: Вильямс, 2002.
23. Подбельский В.В. Язык СИ++. Учебное пособие. М.: Финансы и статистика, 2003.
24. Павловская Т.А., Щупак Ю.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2002.
25. Савитч У. Язык С++. Объектно-ориентированного программирования. М.-СПб.-К.: Вильямс, 2001.