

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт

з дисципліни «Програмування»

для студентів першого курсу всіх форм навчання
спеціальностей «Комп'ютерна інженерія» та «Кібербезпека»
(частина 2)

Затверджено
редакційно-видавничою
радою університету,
протокол № 3 від 10.10.2018 р.

Харків
НТУ «ХП»
2019

Методичні вказівки до виконання лабораторних робіт з дисципліни «Програмування» для студентів першого курсу всіх форм навчання спеціальностей «Комп’ютерна інженерія» та «Кібербезпека» (частина 2) / уклад.: Давидов В. В., Далека В. Д., Молчанов Г. І., Семенов С. Г. – Харків: НТУ «ХПІ», 2019. – 91 с.

Укладачі: В. В. Давидов
В. Д. Далека
Г. І. Молчанов
С. Г. Семенов

Рецензент М. Й. Заполовський

Кафедра обчислювальної техніки та програмування

ВСТУП

Програмування пройшло довгий шлях – від процедурного, модульного до об'єктно-орієнтованого (ООП). Суттєва відмінність ООП від традиційного проектування у тому, що акцент проектування переноситься з розробки алгоритмів функціонування системи на побудову абстракцій та їх взаємодію.

ООП дозволяє розкласти проблему на пов'язані між собою завдання. Кожне завдання стає самостійним об'єктом, що містить свої власні коди та дані, які мають відношення до цього об'єкта. У цьому випадку завдання у цілому спрощується, і програміст одержує можливість оперувати з більшими за обсягом програмами. Таке визначення ООП відображає відомий підхід до рішення складних завдань, кожне з яких розбивається на часткові завдання, що вирішуються окремо. З точки зору програмування це значно спрощує розробку, налагодження та тестування програмних продуктів. Саме до такого підходу у програмуванні необхідно заохочувати студента з самого початку його навчання.

Тематика цієї частини лабораторних робіт присвячена об'єктно-орієнтованому програмуванню: від розробки одного класу до ієрархічної структури класів з реалізацією поліморфізму та шаблонізації, ознайомлення з можливостями стандартної бібліотеки шаблонів (*STL*).

Специфіка лабораторних робіт полягає у тому, що завдання більшої кількості з них (близько 10) пов'язані між собою. Завдання попередньої лабораторної роботи доповнюється додатковими вимогами у наступних роботах. Як наслідок, не можна виконати, наприклад, 5-ту роботу, не виконавши всі попередні. Це вимагатиме від студента систематичної послідовної роботи впродовж семестру.

Лабораторна робота 1. КЛАСИ

Тема. Класи та специфікатори доступу. Інкапсуляція. Константи.

Мета – отримати базові знання про класи. Дослідити механізм інкапсуляції.

Теми для попереднього опрацювання:

- об'єктно-орієнтоване програмування;
- класи, інкапсуляція;
- методи;
- константні методи.

Загальні відомості

Термінологія. Атрибут класу – поле даних класу. Синоніми: властивості класу, дані класу.

Метод класу – функція, що використовується для реалізації поведінки класу.

Клас – синтаксична конструкція мови, комплексний тип даних, який створений програмістом.

Об'єкт – змінна типу «клас».

Завчасне оголошення (*Forward declaration*) – прототиповане оголошення класу.

Оголошення класу – синтаксична конструкція мови, що використовується для визначення методів та атрибутів класу. Зазвичай приводиться у *.h*-файлі.

Реалізація класу – набір синтаксичних конструкцій виконання певних дій методами класу; опис методів класу. Приводиться у *.cpp*-файлі.

Права доступу – режими обмеження, що діють для забезпечення цілісності даних.

Константний метод – це метод, в якому не дозволяється зміна атрибутів власного класу.

Константний параметр – це параметр функції, зміна якого під час виконання цієї функції не передбачається.

Завчасне оголошення використовується для декларування декількох класів, що мають бути наведені в одному файлі і використовують один одний. Приклад завчасного оголошення наведено нижче.

```
class CSmsMessage; // <- FORWARD DECLARATION
class CExperience
{
    public:
        CSmsMessage iMessage;
};
```

Для повного оголошення класу використовується такий фрагмент коду на C++:

```
class CSmsMessage
{
};
```

Створення об'єкта. Після оголошення класу можна використовувати цей клас у розроблюваній програмі. Клас потрібен, коли створюється об'єкт цього класу:

Нижче наведено приклад створення об'єкта *smsMessage* на стеку.

```
void main(void)
{
    CSmsMessage smsMessage;
}
```

Наступна програма використовує динамічне створення об'єкта *smsMessage*. Доречно зазначити, що необхідним є виділення та звільнення (наприкінці роботи програми) пам'яті, що була виділена.

```
void main(void)
{
    CSmsMessage* smsMessage = new CSmsMessage();
    // ... ваші виклики методів...
    delete smsMessage;
}
```

Виділення файлів для класу. Правильним стилем для збереження коду вважається виділення для кожної групи близьких класів двох файлів: *.h* та *.cpp*. Іменувати такі файли рекомендується на базі основного класу в наборі класів. Найчастіше використовують один клас для одного набору *h-cpp*.

Права доступу до членів класу. Взагалі-то права доступу потрібні для забезпечення обмеження доступу до певних членів класів. Зазвичай захищаються у першу чергу дані, а потім можуть обмежуватися методи роботи із ними.

Приклад оголошення класу із правами доступу наведено нижче.

```
// CLASS DECLARATION
class CSmsMessage
{
    public:
```

```

CSmsMessage(void);

...
public:
void setText(const char* aText);
const char* getText() const;
size_t getLength() const;
void setReceiver(const char* aReceiver);
const char* getReceiver() const;

protected:
// довжина тексту
size_t iLength;

private:
char* iText;
char* iReceiver;
};

```

У цьому випадку всі дані захищені від зовнішнього доступу. Самі дані – це *iText* та *iLength*. Поле *iLength* є калькулятивним полем довжини поточних даних (для економії часу на постійні обрахунки у процесі роботи). Якщо припустити, що кожен користувач цього класу буде мати доступ до вказівника *iText*, то це дозволить виконувати напряду такі дії як, наприклад, виділення та видалення пам'яті без урахування значення інших змінних. Такі дії можуть бути помилковими і це призведе до невловимих помилок (поле *iLength* не відповідатиме фактичній довжині поля даних). Саме від таких помилок і захищають програмісти один одного правами доступу.

Хочеться зазначити, що лише нащадки будуть мати можливість пришвидшеної калькуляції, а точніше, отримання даних про поточну довжину тексту. Це зроблено лише для прикладу розмежування доступу. Для реальної програми коректніше зробити цей метод публічним.

Зазвичай доступ типу *protected* використовують для механізму перевизначення поведінки разом із віртуальними функціями, а всі дані закривають специфікатором *private*. Лише для даних абстрактних класів може використовуватися доступ *protected*.

Важливо знати, як розподіляється доступ між членами класів. Для специфікатора доступу немає різниці – це метод чи атрибут. Усі режими доступу до членів класів C++ наведені у табл. 1.1.

Таблиця 1.1 – Типи розмежування доступу до членів класів у C++

	Власний клас	Клас нащадок	Інші класи
public	має доступ	має доступ	має доступ
protected	має доступ	має доступ	доступу немає
private	має доступ	доступу немає	доступу немає

За замовчуванням використовуються специфікатори права доступу private. При коді:

```
class CMmsMessage : CSmsMessage
{
}
```

Характерним повідомленням про помилку може бути таке:

```
error C2247: 'CSmsMessage::Receiver' not accessible because
'CMmsMessage' uses 'private' to inherit from 'CSmsMessage'
```

Константні методи. Після того, як у функцію передано константний об'єкт, програміст може використовувати лише читання публічних атрибутів, а також виконувати константні методи. Константний метод не може змінювати атрибутів власного класу і має специфікатор const після оголошення перед «;».

Синтаксис оголошення константного методу:

```
class CSmsMessage
{
public:
    size_t TextLength() const;
};
```

У разі присвоєння публічному атрибуту будь-якого значення, компілятор видає таке повідомлення про помилку:

```
...\lab3\outputconsole.cpp(26): error C2166: l-value specifies const
object
```

При спробі викликати неконстантний метод CSmsMessage::SetReceiver() у константного об'єкта, буде видано таке повідомлення:

```
...\lab3\outputconsole.cpp(40): error C2662:
'CSmsMessage::SetReceiver' : cannot convert 'this' pointer from
'const CSmsMessage' to 'CSmsMessage &'
```

Константні методи використовуються неявно в момент, коли оголошується функція, яка приймає константний параметр. Це може бути як метод класу, так і просто функція у програмі, наприклад:


```
void DemoConstFunc(const CSmsMessage& aMessage)
{
    // тут від aMessage можна викликати лише константні функції
}
```

При необхідності повертати дані полів об'єкта із константного методу рекомендується формувати лише константні об'єкти або константні вказівники. Немає сенсу повертати константні скалярні типи (int, char, bool тощо), оскільки робота з цими типами не передбачає відмінностей у константному режимі.

Приклади оголошення константних функцій:

```
const int GetSize() const;      // невірно, int - скалярний тип
const char GetFirstCharacter() const; //невірно, char-скалярний тип
const char* GetLine() const;    // вірно
```

Таким чином, специфікатор const визначає ще один тип доступу до даних класу – режим тільки для читання. Такий підхід убезпечує програміста, який використовує функцію, від можливих, неочікуваних змін даних всередині функції, а для програміста, що реалізує таку функцію, вказує на неможливість вносити зміни в отриманий об'єкт як аргумент.

Рекомендується усі методи, що виконують лише читання даних із полів, робити константними для подальшої можливості використання їх як константні параметри.

Загальне завдання

Для предметної галузі з табл. 1.2 розробити два класи:

- клас, що відображає сутність «базового класу». При цьому, в даному класі повинно бути мінімум три числових поля (бажано, щоб одне з цих полів було унікальним ідентифікатором об'єкта);

- клас, що має у собі динамічний масив об'єктів базового класу та має в собі методи додавання, видалення елемента, отримання елемента по індексу (або ідентифікатору), вивід усіх елементів на екран. Рекомендовані сигнатури методів:

- додавання:

```
void CList::addPhone (Phone& phone);
```

- видалення:

```
void CList::removePhone(int index);
```

- отримання по індексу:

```
CPhone& CList::getPhone(int index);
```

– виведення усіх елементів:

`void CList::showAll();` при цьому цей метод повинен викликати метод `getPhone(index)`, щоб не було дублювання коду.

Індивідуальне завдання. У табл. 1.2 обрати прикладну галузь за варіантом відповідно до номера у журналі групи.

Таблиця 1.2 – Варіанти завдань

№ з/п	Прикладна галузь	Базовий клас
1	2	3
1	Учнівська молодь	Студент
2	Навчальний корпус	Аудиторія
3	Кафедра	Співробітник
4	Директорія ПК	Файл
5	Залізничне депо	Поїзд
6	Самостійні роботи студентів	Розрахунково-графічне завдання
7	Компоненти комп'ютерів	Запам'ятовуючий пристрій
8	Світ	Країна
9	Навчання	Випускна кваліфікаційна робота
10	Роботи студентів	Аудиторні заняття
11	Навчальні заклади	Школа
12	Література	Підручник
13	Програмне забезпечення	Програма що виконується
14	Компоненти програм	Бібліотека, що підключається (модуль)
15	Електронні пристрої	Робот
16	Комп'ютерна техніка	Комп'ютер (PC)
17	Мобільні пристрої	Телефон (напр. Nokia 1100)
18	Монітори	LED монітор
19	Академічна група	Студент
20	Змагання	Олімпіада
21	Соціум	Людина
22	Література	Книга
23	Транспорт	Міський транспорт

Продовження табл. 1.2

1	2	3
24	Двигун внутрішнього згоряння	Двигун
25	Морський флот	Корабель
26	Навчання	Навчальні дисципліни
27	Об'єктно-орієнтоване програмування	Тип Клас
28	Двері	Дверна коробка

Додаткові умови виконання завдання:

- усі поля «базового класу» повинні бути приватними та мати публічні гетери та сетери (модифікатори доступу), використовувати механізм інкапсуляції;
- усі функції, що не повинні змінювати поля поточного об'єкта, повинні бути константними;
- усі аргументи функцій, що не змінюються, по можливості також повинні бути константними. Якщо їх не можна зробити константними, у такому разі повинно бути обґрунтування цього;
- у класі-списку метод додавання елемента не повинен вводити дані з клавіатури або файлу, а повинен приймати вже готовий об'єкт для додавання. Метод вводу даних має бути відокремленим;
- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів класу-списку за допомогою модульних тестів.

Контрольні запитання

1. Що таке клас? Чим він відрізняється від структури?
2. Що таке метод? Чим він відрізняється від функції?
3. Що таке інкапсуляція?
4. Що таке константні методи? Наведіть приклади.
5. Як визначити розмір об'єкта у пам'яті?
6. Для чого потрібні права доступу?
7. Які методи потрібні для доступу до *private*-атрибутів?
8. Для чого потрібні *const*-методи?
9. Коли є смисл у константних методах?
10. Чи може бути атрибут константним?

Лабораторна робота 2. ПЕРЕВАНТАЖЕННЯ МЕТОДІВ

Тема. Класи. Конструктори та деструктори. Перевантаження методів.

Мета – отримати базові знання про класи, конструктори та деструктори. Дослідити механізм створення та видалення об'єктів.

Теми для попереднього опрацювання:

- класи;
- конструктори та деструктори;
- функції та методи.

Загальні відомості

Конструктор – це метод, який викликається під час створення об'єкта.

При створенні об'єкта класу часто буває корисним виконати певні дії. Для цих цілей використовується спеціальний метод – конструктор. Конструктор має однакове ім'я із класом. Конструктор може мати вхідні аргументи, але не повертає вихідного значення, як звичайна функція. Конструктор може бути перевантажений.

Нижче наведено оголошення класу із трьома конструкторами:

- конструктор без параметрів;
- визначається параметр для конструктора;
- конструктор копіювання.

Кожен із конструкторів має коментарі у Doxygen-стилі.

```
class CSmsMessage
{
    public:
        /**
         * Конструктор без параметра
         */
        void CSmsMessage(void);

        /**
         * Конструктор з параметром
         * @param aTextLength Задає довжину тексту
         */
        void CSmsMessage(int aTextLength);

        /**
         * Конструктор копіювання
         * @param aSmsMessage Вихідний об'єкт з даними для копіювання
```

```

*/
void CSmsMessage(const CSmsMessage& aSmsMessage);

private:
    // довжина тексту
    int iTextLength;
};

```

Деструктор. Для видалення всіх ресурсів, зайнятих об'єктом, використовується спеціальний метод – деструктор. Деструктор не може мати вхідні параметри, повинен мати таку ж назву, як і клас, але з першим символом – «тільда», не може бути перевантажений.

Для оголошення класу із деструктором використовується така конструкція:

```

class CSmsMessage
{
    public:
        ~CSmsMessage();    // деструктор
};

```

Перевантажені методи. Часто буває, що програмісту потрібно реалізувати операції, однакові за змістом, але з різними вхідними параметрами. Наприклад, потрібно вивести на екран дані різних об'єктів. У такому разі, як варіант, доводиться іменувати методи, використовуючи нумерацію у назві, як показано нижче. Та така методика не додає пояснень, а лише ускладнює роботу програміста.

```

CTestConsole::OutputSmsObject1( smsMessage );
CTestConsole::OutputSmsObject2( smsMessage );
CTestConsole::OutputSmsObject3( &smsMessage );

```

Перевантажені методи – це такі, що мають однакову назву, але різні за типом параметри або їх кількість і, як правило, виконують подібні дії. При збиранні програми із перевантаженими методами компілятор вибирає найбільш підходящий метод для поточного виклику і використовує саме цю реалізацію. У разі, коли аргументи не відповідають жодному із наявних оголошень, компілятор видає помилку про те, що відповідний метод не знайдено.

Наприклад, оголошення двох перевантажених методів може мати такий вигляд:

```

class CTestConsole
{
    public:
        CTestConsole(CSmsMessage& aSmsMessage);
        ~CTestConsole(void);
};

```

```

public:
    static void OutputSmsObject(const CSmsMessage aSmsMessage);
    static void OutputSmsObject (const CSmsMessage* aSmsMessage);
    //!!! наступне перевантаження некоректне
    static void OutputSmsObject(const CSmsMessage& aSmsMessage);
};

```

Останній метод *OutputSmsObject* не може бути перевантаженим, оскільки компілятор не зможе знайти різницю між ним і першим оголошенням. Це найбільш неприємна помилка, так як на момент реалізації саме класів вона не обробляється компілятором, а діагностується лише при виклику даного методу.

Наприклад, при спробі використання методу *OutputSmsObject* у наступному фрагменті програмного коду

```

CSmsMessage data;
CTestConsole console( data );
console.OutputSmsObject( data );

```

буде видано повідомлення про помилку такого характеру:

```

error C2668: 'CTestConsole::OutputSmsObject' : ambiguous call to
overloaded function

```

Такі помилки мають знаходити та виправляти набори коду із самотестування для кожного класу.

Загальне завдання. Поширити попередню лабораторну роботу таким чином:

- 1) в базовому класі необхідно додати:
 - мінімум одне поле типу `char*`;
 - конструктор за замовчуванням, копіювання та конструктор з аргументами;
 - деструктор;
- 2) у клас-список потрібно додати метод обходу масиву для виконання індивідуального завдання.

Приклад сигнатури такого методу:

```

CPhone& findCheapestPhone(float diagonal);

```

У наведеному прикладі реалізоване завдання пошуку самого дешевого телефону з заданою діагоналлю (повертається один телефон).

Індивідуальне завдання. У табл. 2.1 оберіть завдання для обходу колекції за варіантом відповідно до номера у журналі групи.

Таблиця 2.1 – Варіанти завдань

№ з/п	Прикладна галузь	Завдання для обходу колекції
1	2	3
1	Студент	Обчислити витрати держави на видачу стипендії групі за один навчальний рік
2	Аудиторія	Обчислити середню площу аудиторій у навчальному корпусі
3	Співробітник	Обчислити середню заробітну плату співробітників за місяць
4	Файл	Знайти кількість файлів на диску, які мають атрибут «Прихований» і є системними
5	Поїзд	Визначити, у скільки разів максимальна швидкість електропоїздів відрізняється від максимальної швидкості всіх поїздів залізничного депо
6	Розрахунково-графічне завдання	Визначити кількість РГЗ, що виконує студент за весь період навчання в інституті відповідно до навчального плану
7	Запам'ятовуючий пристрій	Знайти запам'ятовуючі пристрої, які мають найменшу ціну за 1Гб
8	Країна	Визначити, яка країна має найменшу щільність населення
9	Випускна кваліфікаційна робота	Визначити відсоток магістерських робіт у порівнянні з бакалаврськими роботами
10	Самостійна робота	Визначити, яку кількість домашніх завдань виконує студент за семестр
11	Школа	Визначити кількість навчальних закладів з кількістю працівників більше за середній показник
12	Підручник	Обчислити середній обсяг (у сторінках) електронних ресурсів
13	Програма, що виконується	Отримати список програм, розмір яких більше заданого розміру (наприклад, 100 К байт). Зі списку виключити «трояни»
14	Бібліотека, що підключається (модуль)	Визначити, у скільки разів обсяг бібліотек, які динамічно підключаються, менше ніж загальний обсяг бібліотек

Продовження табл. 2.1

1	2	3
15	Робот	Визначити кількість роботів зазначеного виробника з найкращим WiFi модулем
16	Комп'ютер (PC)	Знайти ноутбук з максимальною діагоналлю
17	Телефон (наприклад, Nokia 1100)	Визначити телефон з найменшою щільністю пікселів
18	LED монітор	Визначити кількість моніторів у колекції з сенсорними датчиками
19	Студент	Визначити, хто із студентів за результатами сесії має заборгованості і який відсоток становлять заборгованості з програмування
20	Олімпіада	Визначити, на якій олімпіаді була найбільша кількість учасників
21	Людина	Визначити, який відсоток студентів технічних ВНЗ займаються за дистанційною формою навчання
22	Книга	Визначити кількість книг з програмування Шилдта і скільки в них запитань для самоперевірки
23	Міський транспорт	Визначити кількість пасажирів, що перевозить міський електричний транспорт
24	Двигун	Обчислити, у скільки разів обсяг споживаного палива роторних двигунів менше обсягу споживаного палива поршневих двигунів
25	Корабель	Визначити співвідношення пасажирських та вантажних кораблів
26	Навчальні дисципліни	Визначити, який відсоток у навчальному плані складають фахові дисципліни
27	Клас	Визначити, який клас має найбільшу кількість приватних методів
28	Дверна коробка	Знайти найдорожчу профільну коробку з алюмінію

Додаткові умови виконання завдання:

- реалізація конструкторів повинна бути продемонстрована за допомогою списків ініціалізацій;

- конструктори та деструктори повинні мати логіруючі повідомлення.

Студент повинен продемонструвати виклик деструктора та кожного типу конструктора, а також пояснити, коли вони викликаються;

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів класу-списку за допомогою модульних тестів.

Контрольні запитання

1. Для чого потрібні конструктори?
2. Які ви знаєте відмінності конструкторів?
3. Чи можна перевантажувати конструктори?
4. Для чого потрібні деструктори?
5. Які відмінності деструктора?
6. Що таке конструктор копіювання? Для чого він потрібен?
7. Коли викликаються конструктори та деструктор?
8. Що таке списки ініціалізації?
9. Що таке перевантаження методів?
10. Чим визначається виклик перевантажених методів?

Лабораторна робота 3. ПОТОКИ

Тема. Робота з потоками: потокове введення / виведення на консоль та у файл, рядки типу *string*, *stringstream*.

Мета – отримати знання про основи роботи з потоковим введенням / виведенням на мові C++, роботу з файлами та рядками типу *string*.

Теми для попереднього опрацювання:

- класи;
- потоки
- *string*;
- *stringstream*;
- *fstream*.

Загальні відомості

Потокове введення / виведення. При введенні / виведенні потоку всі дані розглядаються як потік окремих байтів. Для користувача потік – це файл на диску або фізичний пристрій, наприклад, дисплей, клавіатура, або пристрій для друку, з якого або на який направляється потік даних. Операції введення / виведення для потоку дозволяють обробляти дані різних розмірів і форматів від одиночного символу до великих структур даних. Найчастіше застосовують потокове введення / виведення даних, операції якого включені до складу класів *istream* або *iostream*. Доступ до бібліотеки цих класів здійснюється за допомогою використання у програмі такої директиви компілятора:

```
#include <iostream>
```

Для потокового введення даних вказується операція «>>» («читати з»). Це перевантажена операція, яка визначена для всіх простих типів і вказівника на тип *char*. Стандартним потоком введення є *cin*.

Формат запису операції введення має такий вигляд:

```
cin [>> values];
```

де *values* – змінна.

Кожна операція «>>» передбачає введення одного значення.

При введенні рядків, що мають у своєму складі пропускання, цей оператор не використовується. У такому випадку треба застосовувати функції *getline()* або *get()*.

Для потокового виведення даних необхідна операція «<<» («записати у»), що використовується разом з ім'ям вихідного потоку *cout*. Наприклад, вираз `cout << x;` означає виведення значення змінної *x* (або запис у потік). Ця операція вибирає необхідну функцію для перетворення даних у потік байтів.

В операціях виведення можна використати символи табуляції. Наприклад, «\t» поміщає кожне наступне ім'я або число у наступну позицію табуляції (через вісім символів). Для додаткового керування даними, що виводяться, використовують маніпулятори *setw(w)* та *setprecision(d)*.

Маніпулятор *setw(w)* призначений для зазначення довжини поля, що виділяється для виведення даних (*w* – кількість позицій).

Маніпулятор *setprecision(d)* визначає кількість позицій у дробовій частині дійсних чисел.

Для роботи з маніпуляторами треба підключити бібліотеку *iomanip.h*.

- ***string*** – стандартний клас, який представляє собою текстовий рядок. Клас реалізує типові операції з рядками, такі як порівняння, конкатенація, пошук і заміна. Клас має функцію для отримання підрядків.

Для роботи з рядками типу *string* треба написати директиву компілятора

```
#include <string> (без розширення .h)
```

і підключити простір імен бібліотеки шаблонів у вигляді

```
using namespace std;
```

Після цього можна оголошувати змінні типу *string*. Ініціювання рядків при оголошенні виконується одним із двох способів:

```
string st1 = "Це рядок класу string";  
string st2 ("Це другий рядок"); ...
```

- ***stringstream*** – це такий клас у бібліотеці *iostream*, який дозволяє зв'язати потік введення / виведення з рядком у пам'яті. Усе, що виводиться в такий потік, додається у кінець рядка; усе, що зчитується з потоку, береться з початку рядка.

Цей клас дозволяє:

- формувати складні рядки, наприклад,

```
int id;  
std::string data1, data2;  
std::stringstream ss;
```

```
ss << "Operation with id = " << id << " failed, because data1 (" <<
    data1 << ") is incompatible with data2 (" << data2 << ")";
```

– виконувати перетворення типів, наприклад,

```
std::stringstream ss;
ss << "22";
int k = 0;
ss >> k;
```

Потокове введення / виведення можна використати і для файлів як у текстовому, так і в бінарних режимах.

Для реалізації файлового введення / виведення необхідно включити у програму заголовок `<fstream>`. У ньому визначено кілька поточкових класів: *ifstream* (для створення потоку введення), *ofstream* (для потоку виведення) і *fstream* (для потоків, що одночасно реалізують введення та виведення). Перед початком роботи з файлом необхідно створити потік за допомогою, наприклад, однієї з таких інструкцій:

```
ifstream in; //введення
ofstream out; //виведення
fstream io; //введення й виведення
```

Потім із створеним потоком зв'язується файл. Для цього використається функція *open()* із прототипом:

```
void stream1 :: open("file1", ios::in);
```

Вона зв'язує файл за ім'ям *file1* з потоком введення у файл.

Режими роботи з файлами задаються за допомогою значень:

ios::in – задає режим відкриття файлу для введення;

ios::out – відкриття файлу для виведення;

ios::app – режим додавання в кінець файлу;

ios::binary – двійковий режим.

Використовувати функцію *open()* для відкриття файлу не обов'язково, тому що у класах *ifstream*, *ofstream* і *fstream* є конструктори, які відкривають файл автоматично. Відкрити файл можна таким чином:

```
ifstream in ("file1");
```

У цій функції другий параметр відсутній, оскільки він за умовчанням буде мати те значення (*ios::in*), що відповідає типу потоку, що відкривається.

Для закриття файлу використається функція *close()*. Наприклад, щоб закрити файл, пов'язаний з потоком *in*, потрібно записати:

```
in.close();
```

Якщо файл був відкритий за допомогою змінної типу *ifstream*, він

автоматично закривається по закінченні роботи функції. Будь-який файл закривається, коли пов'язаний з ним об'єкт виходить із області видимості.

Текстові файли. Форматоване введення / виведення. Текстовий файл – це послідовність ASCII-символів, які розділені на рядки. Для запису в текстовий файл і зчитування з файлу можна використати оператори << та >> так само, як це робиться для консольного введення / виведення. Необхідно тільки замінити потік *cin* або *cout* тим потоком, що пов'язаний з цим файлом. Уся інформація у файлі зберігається у тому ж форматі, якби вона перебувала на екрані. Тому файл, створений за допомогою <<, являє собою файл із відформатованим текстом.

Наприклад, записати рядок та число у текстовий файл.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream fout("test.txt"); //Створення файлу для виведення
    if(!fout)
    { //Якщо не вийшло створення файлу
        cout << " Can't open file\n";
        return 1;
    }
    fout << "Record to file\n"<<100<< endl; //Запис у файл
    fout.close();//Закриття файлу
    return 0;
}
```

У результаті роботи цього коду в поточній директорії програми буде створений файл *test.txt*.

Неформатоване двійкове введення / виведення. Текстові файли зручні для сприйняття, однак більш гнучку структуру мають двійкові (бінарні) файли. Такі файли призначені для неформатованого введення / виведення, що здійснюється за допомогою функцій: *put()*, *get()*, *getline()*, а також *write()* і *read()*.

1. Побайтове введення / виведення. Функція *get()* зчитує один байт з потоку і передає його значення аргументу *s*; функція *put()* записує символ *s* у потік і повертає посилання на потік:

```
istream &get (char &s);
ostream &put (char s);
```

2. Блокове введення / виведення. Функція *read()* зчитує з потоку стільки байтів, скільки задано в аргументі *num*, і передає їх у буфер, який визначений вказівником *buf*. Функція *write()* вилучає *num* байтів з буфера

buf і записує їх у потік.

```
istream &read (char *buf, num);  
ostream &write (const char *buf, num);
```

3. Рядкове введення. Функція *getline()* зчитує з потоку символи і передає їх у буфер *buf* доти, поки не буде зчитано (*num-1*) символів, або не зустрінеться символ нового рядка:

```
istream &getline (char *buf, num);
```

Відкриття й закриття бінарних файлів здійснюється так само, як і текстових файлів.

Загальне завдання. Поширити попередню лабораторну роботу таким чином:

- використання функцій *printf/scanf* замінити на використання *cin/cout*;
- усі конкатенації рядків замінити на використання *stringstream*;
- замінити метод виводу інформації про об'єкт на метод, що повертає рядок-інформацію про об'єкт, який далі можна виводити на екран;
- замінити метод вводу інформації про об'єкт на метод, що приймає рядок з інформацією про об'єкт, обробляє його та створює об'єкт на базі цієї інформації;
- поширити клас-список, шляхом реалізації методів роботи з файлами за допомогою файлових потоків (*fstream*) (якщо використовувалися функції *fprintf/fscanf* – замінити їх на класи *ifstream/ofstream*), при цьому сигнатури методів повинні виглядати таким чином:

- читання: `void CList::readFromFile(string fileName);`
де *CList* – клас-список об'єктів, при цьому слід пам'ятати, що при повторному читанні з файлу, попередні дані списку повинні бути очищені;
- запис: `void CList::writeToFile(string fileName);`

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;

- не використовувати конструкцію «using namespace std;», замість цього слід робити «using» кожного необхідного класу: using std::string, using std::cout;
- у проєкті не повинні використовуватися бібліотеки введення / виведення мови C, а також не повинні використовуватися рядки типу *char**.

Контрольні запитання

1. Як здійснювати виведення даних на екран за допомогою потоків?
2. Як здійснювати читання даних з клавіатури за допомогою потоків?
3. Для чого потрібен клас *stringstream*?
4. Для чого потрібен клас *string*? Наведіть аналогію роботи з типом *char**.
5. Що таке простір імен?
6. Як здійснювати виведення даних у текстовий файл за допомогою потоків?
7. Як здійснювати читання даних з файлу за допомогою потоків?
8. Як здійснювати виведення даних у бінарний файл за допомогою потоків?
9. Яке призначення маніпуляторів *setw(w)* та *setprecision(d)*? Що треба зробити, щоб можна було їх використовувати?
10. Порівняйте текстові та бінарні файли. Яка у них відмінність?

Лабораторна робота 4. РЕГУЛЯРНІ ВИРАЗИ

Тема. Регулярні вирази.

Мета – отримати знання про базові регулярні вирази та досвід роботи із застосування їх на практиці.

Теми для попереднього опрацювання:

- методи;
- рядки;
- регулярні вирази.

Загальні відомості

У програмуванні регулярний вираз (від англ. *regular expression*, скорочено *regex* або *regexp*, а іноді ще називають *rational expression*) – це рядок-шаблон, що описує або збігається з множиною рядків, створюється відповідно до набору спеціальних синтаксичних правил. Вони використовуються у багатьох текстових редакторах та допоміжних інструментах для пошуку та зміни тексту на основі заданих шаблонів. Багато мов програмування підтримують регулярні вирази для роботи з рядками. Завдяки набору утиліт (включаючи редактор *sed* та фільтр *grep*), що входили до складу дистрибутивів *Unix* регулярні вирази стали відомими та поширеними.

Регулярні вирази базуються на теорії автоматів та теорії формальних мов. Ці розділи теоретичної кібернетики займаються дослідженням моделей обчислення (автомати) та способами опису та класифікації формальних мов.

Регулярний вираз (часто називається шаблон) є послідовністю, що описує множину рядків. Ці послідовності використовують для точного опису множини без перелічення всіх її елементів. Наприклад, множина, що складається із слів «грати» та «іграти» може бути описана регулярним виразом «[гг]рати». У більшості формалізмів, якщо існує регулярний вираз, що описує задану множину, тоді існує нескінченна кількість варіантів, які описують цю множину.

Регулярні вирази дозволяють визначити відповідність символів у тексті (при необхідності із зазначенням кількості їх повторювань),

Для визначення відповідності символів використовуються метасимволи. **Метасимволи** вказують, що має бути знайдена деяка

незвичайна річ, або впливають на інші частини регулярного виразу, повторюючи або змінюючи їх значення.

Повний список метасимволів : . ^ \$ * + ? { [] | ()

Метасимволи «[» та «]» (квадратні дужки) використовуються для визначення класу символів, що є набором символів, з якими шукають збіг. Символи можуть бути перераховані, або дані у вигляді діапазону символів, позначеного першим і останнім символом, розділених знаком «-».

Наприклад, [abc] відповідатиме будь-якому з символів a, b або c. Вираз [a - c] задає діапазон для тієї ж множини символів.

Для того щоб знаходити відповідність символам поза цим класом, на початку класу додається **символ «^»**. Наприклад, вираз [^5] відповідає будь-якому символу, окрім «5».

Обернена коса риска «\». За нею можуть йти різні символи, що означають різні спеціальні послідовності. Вона також використовується для екранування метасимволів, щоб їх можна було використати у шаблонах.

Наприклад, якщо треба знайти відповідність «[» чи, для того, щоб позбавити її своєї особливої ролі метасимвола, перед нею треба поставити обернену косу риску: «\[».

Перевірка заданої кількості повторювань

Метасимвол «*» вказує, що попередній символ може бути зіставлений нуль або більше разів. Наприклад, sa*t відповідатиме st (0 символів a), sat (1 символ a), saaat (3 символи a), і так далі.

Метасимвол «+» повторює послідовність порівняння один або більше разів.

Для аналогічного прикладу sa+t зіставлятиметься sat або, наприклад, saaat, але ніяк не st.

Знак питання «?» перевіряє наявність збігу нуль або один раз. Наприклад, home-?brew відповідає як homebrew, так і home-brew.

Специфікатор {m, n}, де m і n – цілі числа, означає, що тут має бути не менше m і не більше за n повторень.

Наприклад, a/{1,3}b відповідає a/b, a//b і a///b. Але це не може бути ab або a///b.

Пошук співпадаючих підрядків виконується за допомогою статичного методу *IsMatch*, який як вхідні значення приймає два рядки: перший – в якому виконується пошук, другий – шаблон пошуку.

Для використання регулярних виразів у C++ необхідно підключити стандартну бібліотеку *regex*. Основні функції для роботи містяться у

просторі імен *std*. Функціями, які використовуються для застосування регулярних виразів є:

- *regex_match* – перевіряє відповідність усієї послідовності символів регулярному виразу;
- *regex_search* – перевіряє регулярний вираз у будь-якій частині послідовності;
- *regex_replace* – замінює знайдені регулярним виразом символи на форматований рядок.

Приклад 1. Перевірка, чи є рядок числом.

```
regex regex_integer ("(\\+|-)?[0-9]+");

cout << regex_match("1123", regex_integer) << endl; // true
cout << regex_match("-123", regex_integer) << endl; // true
cout << regex_match("123 3", regex_integer) << endl; //false
cout << regex_match("+a", regex_integer) << endl;    //false
```

Приклад 2. Перевірка часткового збігу у рядку: виконується пошук слів, в яких є послідовність символів «llo».

```
regex searchText("llo", regex_constants::icase);
cout << regex_search("Hello", searchText) << endl;    // true
cout << regex_search("My World", searchText) << endl; //false
```

Приклад 3. Пошук і заміна: знайти в рядку слово «world» і замінити на слово «planet».

```
regex replaceRegex("w(o|oooo)r[lkj]d");
string replacementBy = "planet";
string inputText = "my world is beautiful world (and wooorjd)";
cout << regex_replace(inputText, replaceRegex, replacementBy);
```

Функція *regex_replace* здійснює пошук у рядку за заданим регулярним виразом і робить заміну знайдених значень на заданий рядок, має три аргументи:

- перший аргумент – рядок типу *string*, в якому здійснюється пошук;
- другий аргумент – об'єкт класу *regex*, в якому записаний регулярний вираз;
- третій аргумент – рядок типу *string*, на який заміщаються знайдені значення.

Загальне завдання

Поширити попередню лабораторну роботу таким чином:

- при введенні інформації про базовий клас (нема різниці, чи з клавіатури, чи з файлу), організувати перевірку відповідності таким

критеріям з використанням регулярних виразів:

- можна вводити тільки кириличні символи, латинські символи, цифри, пропуски, розділові знаки;
- не повинно бути пропусків та розділових знаків, які повторюються;
- перше слово не повинно починатися з маленького символу;
- у клас-список додати метод, що виводить на екран список усіх об'єктів, які мають одне або більше полів з щонайменше двома словами (перевірку організувати за допомогою регулярних виразів).

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «using namespace std;» , замість цього слід робити «using» кожного необхідного класу:using std::string, using std::cout;
- у проекті не повинні використовуватися бібліотеки введення / виведення мови C, а також не повинні використовуватися рядки типу *char**.

Контрольні запитання

1. Що таке регулярні вирази? Для чого вони потрібні?
2. Наведіть основні класи, що використовуються для роботи з регулярними виразами.
3. Що таке квантифікація?
4. Як здійснювати перерахування при роботі з регулярними виразами?
5. Як здійснюється пошук за допомогою регулярних виразів?
6. Як здійснюється заміна за допомогою регулярних виразів?

Лабораторна робота 5. АГРЕГАЦІЯ ТА КОМПОЗИЦІЯ

Тема. Класи. Агрегація. Композиція. Ключові слова *typedef* та *auto*.

Мета – отримати поняття агрегація та композиція; отримати знання про призначення ключових слів *typedef* та *auto*.

Теми для попереднього опрацювання:

- класи;
- агрегація;
- композиція.

Загальні відомості

Агрегація та композиція – це відносини між класами.

Нехай є два класи і один з них вкладений в інший. Об'єкт зовнішнього класу називається контейнером, об'єкт внутрішнього класу – вміст контейнера. Основна відмінність агрегації від композиції концептуальна.

При композиції внутрішній об'єкт не може існувати без свого контейнера і він не може бути включеним в інші контейнери. При композиції при видаленні контейнера видаляється і його вміст.

При агрегації внутрішній об'єкт може існувати окремо від зовнішнього, може входити до складу інших контейнерів. При видаленні контейнера внутрішній об'єкт не видаляється і продовжує існувати. Властивість зовнішнього класу, що описує внутрішній клас, обов'язково є посиланням на об'єкт-вміст.

Оператор *typedef* дозволяє перейменувати один із існуючих типів даних. Як правило, це роблять для того, щоб полегшити подальшу модифікацію програми.

Наприклад, програма виконує обчислення з числами типу *float*. У ситуації можуть знадобитися більш точні обчислення. Якщо не використовувати перейменування типу, то прийдеться «пройтися» по всій програмі і замінити *float* на *double*. А якщо у програмі тип *float* був перейменований, то його псевдонім, наприклад *real*, стає деяким параметром, який можна знову перевизначити. Для того щоб перейти до обчислень з підвищеною точністю, досить змінити лише один оператор: замість

```
typedef float real;
```

варто написати

```
typedef double real;
```

Тоді програма набуде такого вигляду:

```
#include <iostream.h>
int main()
{
    typedef double real;
    real a=1.0e300, b=1.0e200, c=a*b;
    cout << c << endl;
    return 0;
}
```

Та така програма дуже підступна. Так, значення змінних a і b лежать у припустимому діапазоні типу *double*, але їхній добуток виходить за його межі. Однак у мові C++ такі ситуації за замовчуванням не відслідковуються. За таких значень a і b програма виведе на екран помилкову відповідь 1.127201e-231.

Очевидно, що у такому випадку необхідно розширити діапазон, задавши тип *long double*. У наведеному вище прикладі для цього досить замінити оператор

```
typedef double real;
```

оператором

```
typedef long double real;
```

Тепер відповідь буде правильною, а саме: 1.0e+500.

Звичайно, у такій короткій програмі застосування оператора *typedef* не є переконливим, однак у великій програмі, що складається з багатьох файлів, він дозволяє уникнути стомлюючої перевірки і заміни оголошення змінних типу *double* на змінні типу *long double* (як у прикладі).

Оператор *typedef* не створює новий тип, він просто створює псевдонім існуючого типу. Причому це перейменування діє у межах усієї програми. Інакше кажучи, один тип можна перейменувати тільки один раз.

Ключове слово *auto*, починаючи з C++ v11, сповіщає компілятор про те, що на етапі компіляції необхідно визначити тип змінної на основі даних, якими виконана її ініціалізація. Тобто ключове слово *auto* може використовуватися замість типу змінної. Це називається виведенням типу (також іноді називається виведення типу).

Наприклад, замість

```
double d = 5.0;
```

можна написати таке:

```
auto d = 5.0; // 5.0 є числом з рухомою комою типу double,  
// тому d буде типу double
```

або таке:

```
auto i = 1 + 2; // сума 1 + 2 дорівнює цілому числу,  
// тому i буде типу int
```

Це навіть працює з повернутими значеннями з функцій:

```
int add(int x, int y)  
{  
    return x + y;  
}  
  
int main()  
{  
    auto sum = add(5, 6); // add() повертає int, як наслідок,  
    // буде тип int  
    return 0;  
}
```

Але це працює тільки при ініціалізації змінної при її створенні. Змінні, створені без значень ініціалізації, не можуть використовувати цю функцію (у C++ немає контексту, з якого виводити тип).

Використання автоматичного типу дозволяє програмістам зберегти час на ініціалізацію змінних таких типів, які мають довгу назву.

Загальне завдання

Дослідити заздалегідь визначені типи даних з бібліотеки `<cstdint>` / `<stdint.h>`. Модернізувати розроблені у попередній роботі класи таким чином:

- замінити типи даних, що використовуються при індексуванні на типи з указаної бібліотеки;
- створити власний синонім типу, визначивши його необхідність;
- створити / оновити функцію сортування масиву, де крім поля, по якому виконується сортування, передається і вказівник на функцію, яка визначає напрям сортування;
- у базовий клас додати два поля, що мають кастомний тип даних (тип даних користувача) та які будуть відображати відношення «агрегація» та «композиція», при цьому оновити методи читання та запису об'єкта;
- ввести використання ключового слова *auto* як специфікатор зберігання типу змінної. Визначити плюси та мінуси цього використання.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу: *using std::string, using std::cout;*
- у проєкті не повинні використовуватися бібліотеки введення / виведення мови C, а також не повинні використовуватися рядки типу *char**.

Контрольні запитання

1. Що таке агрегація?
2. Що таке композиція?
3. Чим відрізняється композиція від агрегації?
4. Яке призначення ключового слова *typedef*?
5. Що таке автоматичні змінні (типу *auto*)?
6. Наведіть приклади заздалегідь визначених типів даних з бібліотеки *<cstdint>*. Для чого вони потрібні?

Лабораторна робота 6. СПАДКУВАННЯ

Тема. Класи. Спадкування.

Мета – отримати знання про парадигму ООП – спадкування. Навчитися застосовувати отримані знання на практиці.

Теми для попереднього опрацювання:

- класи;
- парадигми ООП.

Загальні відомості

Спадкування – процес створення нових класів на базі вже існуючих класів.

Ієрархія – структура класів, пов’язаних батьківськими відносинами.

Спадкування – це один із методів взаємодії між класами при побудові архітектури програми. Для використання цього механізму необхідно зазначити, що клас спадкоємець «успадковує» усі властивості базового класу, проте має змогу визначити власні додаткові члени, а також перевизначити поведінку вже існуючих методів. Схематично модель спадкування приводять так, як показано на рис. 6.1.



Рисунок 6.1 – Схематичне позначення спадкування

Основною метою спадкування є повторне використання коду у спадкоємців від батьківських класів. При визначенні класів ієрархії завжди потрібно чітко визначати сутність розроблюваних класів, щоб запобігти недовизначенню членів класів, а з іншого боку, не перенавантажити поточний клас непотрібними методами та даними. У класі повинно бути лише те, що необхідно.

Наприклад, потрібно реалізувати текстове повідомлення (*SMS*) та мультимедійне повідомлення (*MMS*).

Із предметної області необхідно виділити таке: *MMS* – це розширені можливості для повідомлення *SMS*, котре додатково може містити ще й файл із картинкою, аудіо, відео та ін.

У цьому прикладі, без спадкування кожен клас (для *SMS* та *MMS* повідомлень) треба було б реалізовувати окремо і всі операції із текстом повторити двічі. При великих ієрархіях ці «дублі» потенційно множаться із кожним об'єктом-нащадком і впливають на кінцевий розмір об'єктів, що створюються.

Розробляючи ієрархії класів, потрібно знайти подібності та відмінності для всіх класів ієрархії. Саме під час цього процесу можна зрозуміти, чи будуть класи належати до однієї ієрархії.

Для простоти вважатимемо, що в одному повідомленні можна додавати лише один файл. Усі можливості класів *SMS* та *MMS* наведені у табл. 6.1.

Таблиця 6.1 – Можливості класів *SMS* та *MMS*

SMS	MMS
Атрибути	
Текст повідомлення	Текст повідомлення
	Вміст файлу мультимедіа
Методи	
Встановити текст повідомлення	Встановити текст повідомлення
Взяти текст повідомлення	Взяти текст повідомлення
	Додати файл мультимедіа
	Видалити файл мультимедіа
	Отримати вміст файлу мультимедіа

З даних табл. 6.1 видно, що у клас *MMS*-мультимедіа повідомлення «вміщаються» всі члени та методи класу *SMS*-текстове повідомлення і немає зайвих методів у класі текстового повідомлення. Саме тому можна запропонувати спадкування мультимедіа від батьківського текстового повідомлення.

При спадкуванні потрібно явно задавати модифікатор прав доступу при спадкуванні для запобігання можливим проблемам із використанням даних за замовчуванням. Зміни прав доступу через модифікатор при спадкуванні відносно базових наведено у табл. 6.2.

Таблиця 6.2 – Модифікація доступу до членів класу при спадкуванні

Права доступу до членів класу	Спадкування за public	Спадкування за protected	Спадкування за private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Модифікатор доступу *private* – спадкування за замовчуванням. Тому при використанні такого оголошення використовується специфікатор права доступу *private*:

```
class CMmsMessage : CSmsMessage
{
}
```

У цьому випадку типова помилка буде такою:

```
error C2247: 'CSmsMessage::Receiver' not accessible because
'CMmsMessage' uses 'private' to inherit from 'CSmsMessage'
```

Це вказує на неможливість доступу до успадкованих методів класу *CMmsMessage* від класу *CSmsMessage*.

Загальне завдання

Модернізувати попередню лабораторну роботу шляхом:

- додавання класу-спадкоємця, котрий буде поширювати функціонал «базового класу» відповідно до індивідуального завдання;
- додавання ще одного класу-списку, що буде керувати лише елементами класу-спадкоємця;
- у функціях базового класу та класу-спадкоємця обов'язкове використання ключових слів *final* та *override*.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу: *using std::string*, *using std::cout*;
- у проекті не повинні використовуватися бібліотеки введення / виведення мови C, а також не повинні використовуватися рядки типу *char**.

Індивідуальне завдання. У табл. 6.3 оберіть завдання для створення класу-спадкоємця відповідно до номера у журналі групи.

Таблиця 6.3 – Варіанти завдань

№ з/п	Прикладна галузь	Додаткові поля у класі-спадкоємці
1	2	3
1	Учнівська молодь	набір додаткових обов'язків, розмір надбавки до стипендії
2	Навчальний корпус	кількість ЕОМ в аудиторії
3	Кафедра	основне місце роботи
4	Директорія ПК	перелік встановлених системних атрибутів
5	Залізничне депо	споживана електроенергія
6	Самостійні роботи студентів	розрахунково-графічне завдання за науковою тематикою керівника
7	Компоненти комп'ютерів	кількість циклів читання / запису
8	Світ	тип монархії (перерахування)
9	Навчання	наукова новизна, об'єм розділу з охорони праці
10	Роботи студентів	кількість завдань для домашньої роботи
11	Навчальні заклади	спеціалізація
12	Література	URL (посилання на джерело у мережі Інтернет)
13	Програмне забезпечення	тип зловмисного ПО (перерахування)
14	Компоненти програм	розширення (перерахування): – .dll – в Windows, – .so – у Linux , – .dylib – у MacOS
15	Електронні пристрої	тип wifi модуля (a/b/g/n)
16	Комп'ютерна техніка	розмір діагоналі монітора
17	Мобільні пристрої	тип операційної системи
18	Монітори	принцип роботи: резистивний, матричний, ємкісний, проекційно-ємкісний, оптичний
19	Академічна група	прізвище старости та куратора групи

Продовження таблиці 6.3

20	Змагання	найменування дисципліни, кількість завдань
21	Соціум	вищий навчальний заклад: технічний, гуманітарний (перерахування); форма навчання: аудиторна, дистанційна
22	Література	назва бібліотеки, кількість персоналу
23	Транспорт	тип електричного двигуна, напруга живлення
24	Двигун внутрішнього згоряння	тип конструкції
25	Морський флот	тип вантажного корабля, кількість членів екіпажу
26	Навчання	кількість аудиторних годин та годин самостійної роботи з дисципліни
27	Об'єктно-орієнтоване програмування	модифікація доступу до членів класу при спадкуванні
28	Двері	тип матеріалу (перерахування)

Контрольні запитання

1. Для чого потрібне спадкування?
2. Як впливають права доступу атрибутів на спадкування?
3. Які бувають атрибути при спадкуванні і на що вони впливають?
4. Коли працює спадкування для об'єктів-нащадків?
5. Що таке «ієрархія» класів?
6. Які ієрархії у спадкуванні можуть бути? Наведіть приклади.
7. Чим відрізняється спадкування від агрегації?
8. Яке призначення ключових слів *final* та *override*?

Лабораторна робота 7. ПОЛІМОРФІЗМ

Тема. Класи. Поліморфізм. Абстрактні класи.

Мета – отримати знання про парадигму ООП – поліморфізм; навчитися застосовувати отримані знання на практиці.

Теми для попереднього опрацювання:

- класи;
- парадигми ООП.

Загальні відомості

Поліморфізм (*poly* – багато, *morphos* – форма) – це властивість системи використовувати об’єкти з однаковим інтерфейсом без інформації про тип та внутрішню структуру об’єкта.

Віртуальний метод – це метод (функція) класу, який може бути перевизначений у класах-спадкоємцях так, що конкретна реалізація методу для виклику буде визначатися під час виконання. Користувачеві достатньо знати, що об’єкт належить класу або класу-спадкоємцю, в якому метод оголошений.

Віртуальні методи дозволяють створювати загальний код, який може працювати як з об’єктами базового класу, так і з об’єктами будь-якого його спадкоємця. При цьому базовий клас визначає спосіб роботи з об’єктами і будь-які його спадкоємці можуть надавати конкретну реалізацію цього способу. Базовий клас може і не надавати реалізації віртуального методу, а лише декларувати його існування. Такі методи без реалізації називаються «чистими віртуальними» (*pure virtual*) або абстрактними.

Клас, який містить хоча б один чисто віртуальний метод, називається абстрактним. Об’єкт такого класу створити не можна. Спадкоємці абстрактного класу повинні надати реалізацію для всіх його абстрактних методів, інакше вони, у свою чергу, будуть абстрактними класами. Для кожного класу, що має хоча б один віртуальний метод, створюється таблиця віртуальних методів. Кожен об’єкт зберігає вказівник на таблицю свого класу. Для виклику віртуального методу з об’єкта береться вказівник на відповідну таблицю віртуальних методів, а з неї, за фіксованим зміщенням, вказівник на реалізацію методу, використовуваного для цього класу. Механізм віртуальних методів дозволяє виконувати багато маніпуляцій з

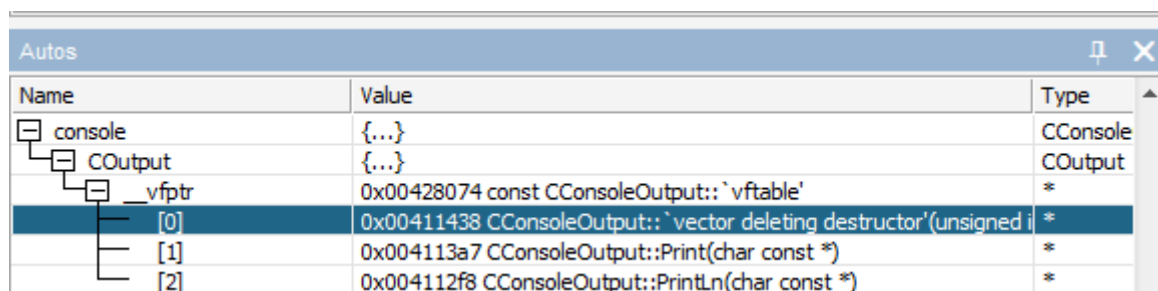
уточнення функцій у роботі з атрибутами, а також викликати методи відповідних класів-нащадків.

При створенні об'єкта, клас якого містить хоча б одну віртуальну функцію, створюється таблиця віртуальних методів. У цій таблиці зберігаються вказівники на всі віртуальні функції всіх класів ієрархії.

Нижче наведено оголошення класу із віртуальними методами:

```
class COutput
{
public:
    virtual ~COutput(void);
public:
    virtual void Print(const char* aText);
    virtual void PrintLn(const char* aText);
};
```

Для об'єкта *console* такого класу буде створена таблиця віртуальних методів, як показано на рис. 7.1.



Name	Value	Type
console	{...}	CConsole
COutput	{...}	COutput
_vfptr	0x00428074 const CConsoleOutput::`vftable'	*
[0]	0x00411438 CConsoleOutput::~`vector deleting destructor'(unsigned i	*
[1]	0x004113a7 CConsoleOutput::Print(char const *)	*
[2]	0x004112f8 CConsoleOutput::PrintLn(char const *)	*

Рисунок 7.1 – Таблиця віртуальних функцій об'єкта *console*

Важливо пам'ятати, що механізм віртуальності проявляється лише для вказівників на об'єкти, а не для статичних об'єктів, для яких пам'ять виділяється на стеку. Для останніх буде викликатись лише метод із реалізації класу.

Необхідно також розуміти, що перевизначенням віртуальної функції може вважатись лише така функція, що повністю співпадає за параметрами, а також за типами, що повертаються із методу. Неспівпадіння типів при оголошенні віртуальної функції у нащадках є класичною помилкою. У такому разі створюється новий метод, а не перевизначається базовий.

Зазвичай, віртуальні функції допомагають писати універсальні програми, абстрагуючись від конкретних реалізацій класів. Таким прикладом може бути наступний фрагмент програми. Спадкування реалізоване таким чином: *COutput* – базовий клас, *CConsoleOutput* – клас – його спадкоємець.


```

COutput* CreateOutput()
{
    COutput* console = new CConsoleOutput();
    return console;
}

void main()
{
    COutput* output = CreateOutput();
    output->PrintLn( "my text message" );
    DestroyOutput( output );
}

```

Важливо розуміти, що у цьому фрагменті створюється об'єкт від класу-нащадка *COutput*, і для нього можна викликати функцію виводу *PrintLn*, що оголошена у батьківському класі. Основна задача класу *COutput* у такому разі – задати правильні вихідні інтерфейси для нащадків.

Як видно із функції *CreateOutput()*, створюється об'єкт від класу нащадка *CConsoleOutput()*. Для програміста цей код створення може бути закритим, але він може викликати функції, що відповідають базовим інтерфейсам *COutput()*.

Поняття абстрактності невід'ємно пов'язано із поняттями віртуальних функцій, а також розумінням їх реалізації та виклику у програмах.

Як конструкція мови, абстрактний клас описується за допомогою віртуальної функції таким синтаксисом:

```

class COutput
{
    public:

    /**
     * Абстрактний метод. Виводить дані на пристрій виведення.
     * @param aText Текст для виведення
     */
    virtual void Print(const char* aText) = 0;
};

```

Семантика «= 0» показує компілятору, що цей клас містить абстрактний метод, у свою чергу, стаючи абстрактним.

За своєю сутністю абстрактні методи забов'язують програміста виконати реалізацію абстрактного методу, і у такий спосіб обов'язково задати поведінку цього методу для свого класу.

Як продовження розгляду прикладів із попередніх робіт, можна розглянути новий клас *CMessage*, що використовується як базовий абстрактний клас для ієрархії, як на рис. 7.2.

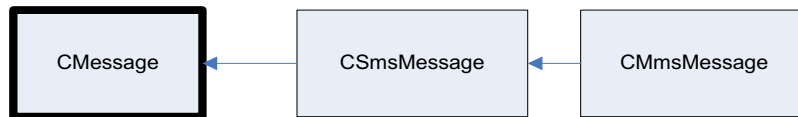


Рисунок 7.2 – Ієрархія класів

Цей абстрактний клас може бути базовим для можливого розширення ієрархії до вигляду як на рис. 7.3.

При наявності задачі із чітко визначеними базовими класами, а також для всього однієї гілки ієрархії зазвичай додавання абстрактного класу на прикінцевому етапі веде до ускладнення програми.

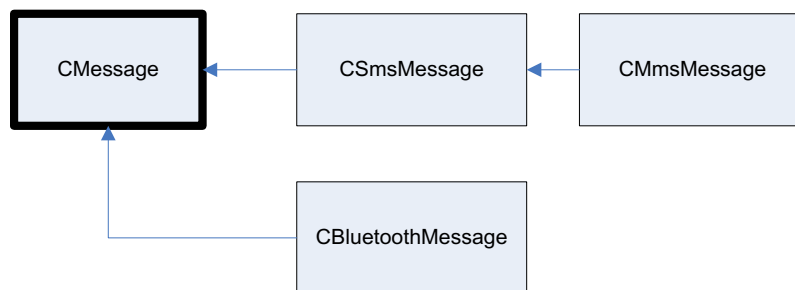


Рисунок 7.3 – Розгалужена ієрархія класів

Інтерфейси. Використання множинного спадкування із двох рівноцінних ієрархій дає багато проблем як компілятору, так і програмісту в реалізації та розв'язанні двояких назв функцій. Саме тому слід використовувати включення до більш узагальнюючого класу допоміжних атрибутів через механізм агрегації, а не використовувати об'єднання логіки за допомогою множинного спадкування.

Абстрактні методи ще називають інтерфейсними методами, оскільки вони надають програмісту розуміння про необхідні параметри для методу, не заглиблюючись у конкретну реалізацію.

Слід зазначити про дуже велику відповідальність програміста, що некоректно написав базові інтерфейси для ієрархії. Зазвичай найменша недосконалість може призвести до виявлення в одному із нащадків неприпустимості використання спадкування у цілому. Саме тому до розробки архітектури та написання оголошень базових класів запрошують спеціалістів високого рівня.

Зазвичай, на момент розробки потужного універсального класу неможливо передбачити всі набори даних та методів, задавши спеціальний

очікуваний базовий клас для здійснення всіх операцій над ним. Таку проблему вирішують за допомогою спеціального трюку.

Створюється один або більше абстрактних класів, що задають лише можливі бажані операції, але власне не виконують ніяких конкретних дій. Такі класи називають інтерфейсами, оскільки вони тільки вказують на можливий синтаксис та правила виконання операцій. Таким чином, потрібно тільки визначити певний набір функцій, а їх власну реалізацію програміст зможе виконати відносно своїх даних, використавши спадкування.

Як наслідок, при множинному спадкуванні після інтеграції у свій клас інтерфейсів, у програміста з'являється можливість використати потужні, заздалегідь розроблені класи, виконавши лише перевизначення декількох необхідних інтерфейсних методів.

У наступному прикладі використовується клас *CFileStorage*, котрий виконує збереження та завантаження даних користувацьких об'єктів. Інтерфейс, що пропонується використовувати для застосування класу *CFileStorage*, називається *MStorageInterface* і описаний він таким чином:

```
class CStorageInterface
{
public:
    virtual void OnStore(ostream& aStream) = 0;
    virtual void OnLoad(istream& aStream) = 0;
};
```

Найпростішим прикладом може бути реалізація інтерфейсу у класі *CData* так, як показано нижче:

```
class CData : public CStorageInterface
{
public:
    void OnStore(ostream& aStream)
    {
        int value = 0xF0;
        aStream.write( (const char*)&value, sizeof(value) );
    }
    void OnLoad(istream& aStream)
    {
        int value = 0;
        aStream.read( (char*)&value, sizeof(value) );
    }
};
```

Клас, що виконує основний обсяг роботи та використовує інтерфейс, оголошено таким чином:

```
class CFileStorage
```

```

{
    public:
        static CFileStorage* Create(CStorageInterface& aInterface,
        const char* aFileName);
        bool Store();
        bool Load();
        ...
};

```

Використати такий набір класів можна таким чином:

```

CData data;
CFileStorage* storage = CFileStorage::Create( data, "datafile.tst" );
bool ok = storage->Store(); // тут буде викликано неявно
data.OnStore();

```

Забути вказати специфікатор доступу *public* до абстрактних методів у інтерфейсі – це типова помилка у визначенні на початковому етапі розробки.

Загальне завдання

Модернізувати попередню лабораторну роботу шляхом:

- додавання ще одного класу-спадкоємця до базового класу. Поля обрати самостійно;
- базовий клас зробити абстрактним. Додати абстрактні поля;
- розроблені класи-списки поєднуються до одного класу таким чином, щоб він міг працювати як з базовим класом, так і з його спадкоємцями. При цьому серед полів класу-списку повинен бути лише один масив, що містить усі типи класів ієрархії. Оновити методи, що працюють з цим масивом.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу: *using std::string, using std::cout;*
- у проєкті не повинні використовуватися бібліотеки введення / виведення мови C, а також не повинні використовуватися рядки типу *char**.

Контрольні запитання

1. Що таке поліморфізм?
2. Для чого потрібні віртуальні методи?
3. Які методи називають «чисто віртуальними»?
4. Що таке таблиця віртуальних методів?
5. Що таке абстрактний клас?
6. Чи можна створити об'єкт абстрактного класу?
7. Що таке інтерфейс?
8. Які умови необхідно виконати для реалізації поліморфізму?
9. Чим відрізняється абстрактний клас від інтерфейсу?
10. Чим відрізняється абстрактний клас від звичайного класу?
11. Яких правил треба дотримуватись при перевизначенні віртуальних методів?

Лабораторна робота 8. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

Тема. Перевантаження операторів. Серіалізація.

Мета – отримати знання про призначення операторів, визначити їх ролі у житті об'єкта та можливість перевизначення.

Теми для попереднього опрацювання:

- функції;
- оператори;
- серіалізація та десеріалізація;
- класи.

Загальні відомості

Перевантажені оператори. Перевантажені оператори використовуються для збереження семантики мови C++. Однією із характерних помилок є некоректне використання операції присвоєння для об'єктів. Оголосимо клас *CMessage*:

```
class CMessage
{
public:
    CMessage(const char* aText)
    {
        iText = new char[strlen(aText) + 1];
        strcpy( iText, aText );
        iText[strlen(aText)] = 0;
    }
    ~CMessage()
    {
        delete[] iText;
    }
    char* getText()
    {
        return iText;
    }
    void setText(const char* aText)
    {
        delete[] iText;
        iText = new char[strlen(aText) + 1];
        strcpy( iText, aText );
        iText[strlen(aText)] = 0;
    }
private:
    char* iText;
};
```

Наступна операція призводила до помилки при виконанні програми:

```
CMessage smsMessage("message text");  
CMessage anotherMessage("another text");  
anotherMessage = smsMessage; //логічна помилка присвоєння  
// вказівників! в результаті first.iText == second.iText
```

Причиною помилки було звичайне побайтове копіювання даних у структурі, тобто після цієї операції вказівники показували на одну і ту ж область даних, а при виклику деструктора двічі (для *anotherMessage* та *smsMessage*) була спроба вивільнити пам'ять з-під однієї області даних.

Цей приклад демонструє один із випадків, коли необхідно обов'язково перевантажувати оператор, у даному випадку – присвоєння.

Щоб перевантажити оператор, необхідно використовувати зарезервоване слово *operator*, за яким іде позначення відповідного оператора. Код функції-оператора реалізує необхідні дії, які повинні виконуватися при виклику перевантаженого оператора. Перевантажений оператор буде виконувати зазначені для нього дії тільки у межах того класу, у якому він визначений.

Перевантажити можна практично будь-який оператор, за винятком таких:

- 1) `.` – (крапка) вибір елемента класу, доступ до полів;
- 2) `*` – (зірочка) визначення або розіменування вказівника;
- 3) `::` – (подвійна двокрапка) область видимості методу;
- 4) `?:` – (знак питання із двокрапкою) тернарний оператор порівняння;
- 5) `#` – (дієз) символ препроцесора;
- 6) `##` – (подвійний дієз) символ препроцесора;
- 7) `sizeof` – оператор знаходження розміру об'єкта у байтах.

За допомогою перевантаження неможливо створювати нові символи для операцій. Перевантаження операторів не змінює порядок виконання операцій і їх пріоритет. Кількість операндів, порядок виконання визначається стандартною версією оператора.

Перевантажені оператори повинні діяти так само, як і їхні базові версії. Перевантажувати можна тільки операції, для яких хоча б один аргумент має тип даних, який створений користувачем (*typedef* не враховується).

Перевантажувати оператори можна або функціями-членами класу, або зовнішніми функціями, але дружніми класу.

Наступні оператори можна перевантажити тільки методами класу:

- 1) `=` – присвоєння;

- 2) `->` – доступ до аргументів за вказівником;
- 3) `()` – виклик функції;
- 4) `[]` – доступ за індексом;
- 5) `->*` – доступ до вказівника на аргумент по вказівником;
- 6) оператори конверсії та керування пам'яттю.

Перевантажувати оператори треба лише у тих випадках, коли це виглядає природним і необхідним. Але якщо перевантажений один оператор, то необхідно перевантажувати та інші.

Якщо бінарна арифметична операція перевантажується з використанням функції-члена, то як свій перший аргумент вона одержує неявно передану змінну класу (вказівник *this* на об'єкт), а як другий – аргумент зі списку параметрів. Тобто, фактично, бінарна операція, що перевантажується функцією-членом класу, має один аргумент (правий операнд), а лівий передається неявно через вказівник *this*.

Якщо бінарна операція перевантажується дружньою функцією, то в списку параметрів вона повинна мати обидва аргументи.

Для оголошеного раніше класу *CMessage* оператор присвоєння може бути визначений таким чином:

```
public:
CSmsMessage& operator=(const CSmsMessage& aSmsMessage);
//Реалізація такого оператору може мати наступний вигляд:
CSmsMessage& CSmsMessage::operator=(const CSmsMessage& aSmsMessage)
{
    setText( aSmsMessage.getText() );
    return *this;
}
```

Таким чином, видно, що у самому операторі виконується поелементне копіювання даних. Треба звернути увагу на те, що в цій реалізації оператор повертає посилання на свій об'єкт *this* – **this*. Це потрібно для використання семантики мови C++ для ланцюжкового використання оператора присвоєння, наприклад, *A=B=V=D*.

У цьому фрагменті всі атрибути об'єкта *B* присвоюються відповідно атрибутам об'єкта *A* через оператор присвоєння за замовчуванням. Отже, відбувається копія «поле в поле». Однак у разі, якщо атрибут є класом, то буде викликана перевантажена операція присвоєння для цього класу. Така поведінка корисна, коли забезпечується безпечне розмноження об'єктів за доступом до одного ресурсу.

Серіалізація об'єктів. Процес серіалізації та десеріалізації загалом корисний для спрощеного формування бінарного набору із даних об'єкта з

можливістю збереження та подальшого відновлення, наприклад, між запусками програми. Найпростішим випадком серіалізації без подальшого відновлення можна розглядати оператори перенаправлення виведення. У цьому випадку дані об'єкта перетворюються у рядковий формат, для можливості їх виведення на екран.

Більш корисним прикладом є читання даних із стандартного потоку вводу, а наприкінці програми – запис даних у потік виводу за умови, що ці два потоки перенаправлені на роботу із файлом і зайві текстові повідомлення у процесі роботи програми не виводяться.

Продемонструвати такий приклад можна запуском програми `serialize.exe` із таким синтаксисом:

```
serialize.exe < in.txt > out.txt
```

та запропонованою реалізацією:

```
void main(void)
{
    int value = 0;
    cin >> value;
    value++;
    cout << value;
}
```

У разі, якщо в файлі *in.txt* міститься рядок «123», то після роботи програми на виході буде – 124. Отже, можна сказати, що тут використовується серіалізація значення *value* типу *int*. Аналогічно відбувається збереження даних більш складних типів – класів.

Важливо запам'ятати, що потрібно зберігати власне дані і вкрай помилково зберігати вказівники на об'єкти, відкриті дескриптори поточного процесу та інші тимчасові сутності на час роботи програми.

Загальне завдання. Поширити попередню лабораторну роботу таким чином:

- у базовому класі, та класі/класах-спадкоємцях перевантажити:
 - оператор присвоювання;
 - оператор порівняння (на вибір: `==` , `<` , `>` , `>=` , `<=` , `!=`);
 - оператор введення / виведення;
- у класі-списку перевантажити:
 - оператор індексування (`[]`);

- оператор введення / виведення з акцентом роботи, у тому числі і з файлами. При цьому продовжувати використовувати регулярні вирази для валідації введених даних.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу: *using std::string, using std::cout.*

Контрольні запитання

1. Для чого потрібні оператори?
2. Які оператори можна перевантажувати?
3. Які оператори не можна перевантажувати?
4. Що таке серіалізація та десеріалізація?
5. Як можна перевантажити оператори?
6. Чим відрізняється перевантаження операторів за допомогою функцій-членів класу та дружніх функцій?
7. Що таке «дружні функції», у чому їх особливості, коли вони необхідні?

Лабораторна робота 9. ВИКЛЮЧЕННЯ

Тема. Виключення.

Мета – навчитись розробляти програми з реалізацією виключень.

Теми для попереднього опрацювання:

- класи;
- виключення;
- спадкування.

Загальні відомості

Виключна ситуація або виключення – помилка часу виконання, для обробки якої передбачено спеціальні механізми.

Блок захоплення виключення – код, що розміщений всередині блоку *try{}*.

Код перехоплення виключення – код, що виконується всередині блоку *catch{}*.

Розмотування стеку – коректне видалення усіх створених на поточному стеку об'єктів та звільнення відповідної стекової пам'яті під об'єктами.

Цільова компіляція – набір параметрів, на базі яких виконується збірка проекту. Розрізняють компіляцію із кодом для налагодження (*Debug*) та фінальна збірка (*Release*).

Паніка (*Panic*) – неочікуваний терміновий вихід із додатка.

Виконання програми відбувається чітко згідно з інструкціями, які описав програміст під час розробки як на рівні простих атомарних операцій, так і використовуючи бібліотечні виклики.

У момент виконання застосунку, коли виникає помилка, її можна віднести до одного із таких типів:

- 1) виникає через неуважність програміста;
- 2) передбачувана ситуація програмістом;
- 3) логічна помилка програміста;
- 4) помилка у роботі зовнішнього пристрою;
- 5) помилка у роботі операційної системи;
- 6) критичні відмови у роботі обчислювального обладнання (процесор, ОЗП).

Такий поділ є досить умовним, оскільки програміст може передбачити обробку помилки зовнішнього пристрою і таким чином можна було б об'єднувати 2) та 4), але такий поділ дозволяє виділити логічні рівні для подальшого аналізу.

Вважається, що помилки, які належать до категорії 5) та нижче можна не обробляти, оскільки невідомо, чи зможе до кінця виконатись обробник помилкової ситуації при таких збоях.

Від помилок категорії 1) програмісти частково захищаються різноманітними перехресними перевірками аргументів функцій та проміжних умов виконання програм. Такого роду перевірки використовуються для режиму збирання *Debug* та контролюються, використовуючи макроси *ASSERT*. Також хорошим способом захисту від такого роду помилок є методи екстремального програмування через написання модулів самотестування коду.

Перевірка при налагодженні. Для режиму налагодження коду є декілька засобів роботи щодо відлову некоректно написаного коду, і всі вони базуються на макровизначеннях. Механізм макровизначень дозволяє регуляцію включення / виключення на момент компіляції. При компіляції фінальної збірки код всередині таких макровизначень пропускається, і, таким чином, код перевірок не відображається ні на швидкодії, ні на розмірі такої збірки.

Такими перевірками програмісти захищають себе від своїх же помилок на час первинної розробки програмного забезпечення та етапів подальшої модифікації та рефакторингу.

Усі макроси перевірки як основний елемент своєї роботи використовують завершення роботи панікою. Перевіряється умова і, в разі невідповідності, запускається визначений обробник, що в кінцевому випадку призводить до умовно некоректного виходу із застосунку.

Макрос *assert()* входить до базового набору мови C, тому його використання рекомендовано при програмуванні крос-платформового програмного забезпечення. Цей макрос оголошено у заголовку *assert.h*.

Приклад фрагмента програми, що виконує перевірку параметрів функції, наведено нижче:

```
void AssertValidationExample(const char* aItem, int aInt)
{
    assert( aItem );
    assert( aInt != INVALID_ARG_VALUE );
    // основний код роботи функції
}
int main(void)
```

```

{
    char* hello = NULL;
    int intValue = 123;
    AssertValidationExample( hello, intValue );
    // основний код програми
}

```

У результаті буде видано повідомлення про невиконану перевірку вхідних даних, наприклад таке:

```
Assertion failed: aItem != NULL, file c:\temp\hello.cpp, line 14
```

Після двокрапки міститься умова, яку перевіряли, потім ім'я файлу, де виникла проблема, далі номер рядка.

Якщо бажано відключити функціонал *assert()*, то перед першим включенням файлу заголовку потрібно оголосити макрос *NDEBUG* таким чином:

```

#define NDEBUG
#include <assert.h>

```

Механізм виключних ситуацій відпрацьовує на момент виконання коду програми. Для обробки виключних ситуацій у мові C++ передбачено конструкцію *try-catch*. Схематично вона наведена у цьому фрагменті:

```

try
{
    // фрагмент для перевірки
}
catch(ClassType aObject)
{
    // обробляємо виключну ситуацію відносно ClassType
}
catch(ClassType1 aObject)
{
}
}

```

У блок *try* включають весь код, який може породжувати виключні ситуації. Під цим блоком розміщують один або декілька блоків коду перехоплення для обробки виключення. Блок починається зі слова *catch*, а у дужках указують один очікуваний тип даних для відлову та параметр.

Виключна ситуація генерується оператором *throw* із бажаним генерованим об'єктом виключення. Наведемо фрагмент із генерацією та перехопленням виключення цілочислового типу *int*.

```

try
{
    throw 123;
}
catch (int aValue)
{
    printf( " Перехоплене значення: %d.", aValue );
}

```

У результаті на екран буде видано таке:

Перехоплене значення: 123.

Якщо треба перехопити виключення всіх типів, то необхідно скористатись спеціальним синтаксисом із трьома крапками конструкції *catch*. Фрагмент програмного коду, який оброблює «неочікуване» виключення, наведено нижче:

```
try
{
    throw "Hello World!";
    throw 123;
}
catch (int aValue)
{
    printf( "Значення: %d.", aValue );
}
catch(...)
{
    printf( "Неочікувана виключна ситуація!" );
}
```

У результаті буде виведено такий текст:

Неочікувана виключна ситуація!

Ця неочікувана ситуація виникає через генерацію виключення типу *char**, тип якого не вказано у списку очікуваних виключень. Оскільки сюди попадає виключення із невідомим типом, то і додаткові дані отримати з нього неможливо. Такий спосіб перехоплення залишає виключну ситуацію на цьому ж рівні та захищає програму від подальшого поширення виключної ситуації.

Логіка роботи виключних ситуацій. Виключні ситуації є одним із механізмів обробки помилок із передачею управління, а тому вимагають досить глибокого розуміння самого підходу та алгоритму роботи.

Для демонстрації підходу буде вирішена двома способами така задача: потрібно отримати результат, порахувавши дані у двох функціях. Кожна із функцій може повернути помилку, коли вважатиме вхідні дані некоректними. Програма повинна коректно обробити та вивести результати.

Перший спосіб – стандартний підхід:

```
const char* KFunc1Error = "Func1 error";
const char* KFunc2Error = "Func2 error";

int Func1(int aValue, float& aResult)
{
    // ...
    return 0;
}
```



```

int Func2(float& aResult)
{
    // ...
    return 0;
}
int main(void)
{
    float value = 0;
    char* errMsg = NULL;
    int retCode = Func1( 10, value);
    if ( retCode >= 0 )
    {
        retCode = Func2( value + 2 );
        if ( retCode >= 0 )
        {
            value = value * 25 - value/37;
        }
        else
        {
            errMsg = KFunc2Error;
        }
    }
    else
    {
        errMsg = KFunc1Error;
    }
    if ( retCode > 0 )
    {
        printf( "Calculated: %f", value );
    }
    else
    {
        printf( "Error happened: %s", errMsg );
    }
}

```

У результаті успішного виконання виведеться повідомлення із значенням, а у разі помилок при виконанні – відобразиться назва функції, що призвела до зупинки обчислень. Як бачите, навіть з двома рівнями вкладеності функцій код виходить заплутаним та громіздким.

Другий спосіб – із застосуванням виключних ситуацій як механізм обробки помилок, програмний код буде таким:

```

int main(void)
{
    float value = 0;
    try
    {
        Func1( 10, value );
        Func2( value );
        value = value * 25 - value/37;
        printf( "Calculated: %f", value );
    }
    catch (const char* aErrMsg)
    {
        printf( "Error happened: %s", aErrMsg );
    }
}

```

Для коректної роботи цього алгоритму потрібно змінити процес отримання повідомлення про помилки від кожної функції.

У разі виявлення помилки потрібно згенерувати виключну ситуацію із описом проблеми всередині реалізації кожної із функцій. Функція значення тепер не повертає, тому в сигнатурі записано *void*, а всередину функції додано код: у разі помилки генерувати виключення типу *const char**. Умовно реалізація виглядатиме таким чином:

```
void Func1(int aValue, float& aResult)
{
    //    ...
    if ( errorInData )
        throw KFunc1Error;
    float rithualCmd = aResult;
    //    ...
}
```

Без сумніву, такий код набагато легше читається та більш лаконічний, а функціонал залишився таким же.

Специфікатори у функціях. Аналогічно до специфікатору *const*, мова C++ допускає ще один тип специфікатору – *throw()* для уточнення використання виключних ситуацій у функції.

Хорошим тоном вважається використовувати специфікатор можливих виключень. Таким чином, програміст осмислює на своєму рівні, які виключення потрібно перехопити власними силами, а які необхідно пропустити, передавши далі, на наступний рівень обробки.

У вищенаведеному завданні функція *Func1()* мала б бути оголошена зі специфікатором, що показує генерацію *const char**:

```
void Func1(int aValue, float& aResult) throw(const char*);
```

Різні приклади використання специфікаторів наведено у оголошеннях, що наводяться нижче:

```
void Func(int aValue) throw(char*);    // генерує виключення char*
void AnotherFunc(float aValue) throw();// не генерує виключень
void AnotherFunc(float aValue);        // може генерувати будь-що
void CChild
{
    F(int aValue);
    const throw(int, CMyException);
}; // константна функція, що генерує int або CMyException
```

Порядок оголошення специфікаторів має значення, секція *throw()* повинна йти у прототипах функцій останньою, лише після службового слова *const*, інакше буде помилка компіляції. Опис призначення кожного типу специфікатора вказано у табл. 9.1.

Таблиця 9.1 – Опис призначення типів специфікатора *throw*

Тип специфікатора	Приклад заголовка	Опис
Пустий список	<code>void Func() throw();</code>	Функція не генерує виключення
Список із одним або декількома значеннями	<code>void Func() throw(int, CMyObject*);</code>	Функція генерує виключення лише одного із перелічених типів у дужках
Без специфікатора	<code>void Func();</code>	Функція може генерувати будь-які виключення або не генерувати їх

Помилка: наступні функції відрізняються лише специфікатором, тому компілятор не робить різниці між ними:

```
void f1(int) throw();
void f1(int) throw(Exception);
```

Компілятор, що поставляється із *Visual Studio*, ігнорує специфікатори виключень, але інші компілятори можуть їх враховувати.

Порядок вибору відповідного обробника. Вище зазначалось, що виключні ситуації можна відловлювати за типом даних. Отримувати цей параметр можна за значенням, за посиланням та через вказівник.

Відповідний блок вибирається відповідно до типу даних, який був використаний при генерації виключної ситуації. Необхідно підкреслити, що коли у випадку передачі об'єкту за значенням та передачі через посилання тип даних при пошуку буде вважатись одним і тим же, то у разі вказівника – це абсолютно інший – це вказівник на тип.

Порядок обробників має важливе значення. Програма шукає відповідний тип зверху вниз після блоку *try* і, якщо тип даних співпадає, то виконується саме цей код перехоплення. Це правило ще більш цікаво працює для ієрархій типів. Коли поставити перед класом-нащадком батьківський клас, то спадкоємці ніколи не отримають управління, оскільки перевірка на сумісність типу буде завжди видавати батьківський клас за збіг. Для ілюстрації роботи повторений фрагмент програмного коду із обчисленням функцій, але додано обробник для типу *std::exception* перед *std::logic_error*:

```
try
{
    Func1( 10, value );
    Func2( value );
    value = value * 25 - value/37;
```

```

        printf( "Calculated: %f", value );
    }
    catch (std::exception& aErr)
    {
        printf( "Base class caught exception: %s", aErr.what() );
    }
    catch (std::logic_error& aErr)
    {
        printf( "Error happened: %s", aErr.what() );
    }
}

```

У такому разі на екран буде видано такий текст:

```
Base class caught exception: Funcl error
```

У випадку, коли виключення отримано у коді перехоплення, але його обробку потрібно здійснити вище, то можна скористатися операцією *throw* без параметрів. Таким чином, виконається породження із тими ж даними на рівень вище. Приклад, що ілюструє таку поведінку, наведено нижче.

```

try
{
    try
    {
        throw "Hello";
    }
    catch(...)
    {
        throw;          // передаємо обробку на рівень вище
    }
}
catch (char* aMsg)
{
    printf( "Перехоплено char*: %s", aMsg );
}

```

Загальні рекомендації:

- виключними ситуаціями розділяють функціонал та обробку помилок;
- швидкодія системи не знижується у порівнянні із стандартним підходом, коли виключні ситуації не відбуваються;
- система у цілому програє на використанні блоків захоплення для поодиноких функцій, тому рекомендовано використовувати комплексну обробку;
- для стекових змінних у рамках блоку захоплення *try* будуть викликатися всі деструктори перед входом у *catch*, тому турбуватися треба лише про динамічно виділену пам'ять;
- не рекомендується створювати велику кількість власних класів-виключень, краще використовувати вже існуючі;

- виключні ситуації краще генерувати як об'єкт класу, а не як простий тип даних;
- не рекомендується викликати із функцій внутрішнього рівня вихід із додатку (*exit()* тощо). Краще передавати управління виключенням на рівень вище;
- рекомендується спадкувати власні класи-виключення від стандартних базових;
- не дивлячись на можливість у стандарті C++ v11 використовувати виключення між потоками, рекомендується не використовувати виключні ситуації в асинхронних операціях.

Загальне завдання. (Задача не пов'язана з попередніми роботами).

У файлі розміщена інформація про N масивів.

У першому рядку міститься інформація про кількість масивів, у кожній наступній – інформація про кількість елементів у кожному масиві та власне дані масиву.

Необхідно реалізувати програму, що виконує перераховані нижче дії, причому кожна з них в окремій функції, поки користувач не введе замість назви файлу рядок *exit*

Дії, що має виконувати програма, такі:

- введення з клавіатури назви вхідного файлу з даними;
- читання даних з файлу;
- виконання індивідуального завдання;
- введення з клавіатури імені вихідного файлу;
- запис результату операції у файл;
- доступ до елемента за індексом слід винести в окрему функцію, що виконує перевірку на можливість виходу за межі масиву.

Слід окремо звернути увагу, що при обробці виключення цикл не повинен перериватись.

Індивідуальні завдання

1. Визначити суму двох масивів. Результат операції – масив.
2. Перемножити два масиви. Результат операції – масив.
3. Підрахувати середнє значення елементів масиву. Результат операції – масив з середніх значень кожного із вхідних масивів.

4. Знайти у масиві елемент з максимальним значенням. Результат операції – масив з максимальних елементів кожного із вхідних масивів.
5. Знайти у масиві елемент з мінімальним значенням. Результат операції – масив з мінімальних елементів кожного із вхідних масивів.
6. Визначити кількість додатних елементів у масиві. Результат операції – масив з кількості додатних елементів у кожному із вхідних масивів.
7. Визначити кількість елементів, які належать діапазону $[a, b]$ (значення a та b ввести з клавіатури). Результат операції – масив з кількості знайдених елементів у кожному із вхідних масивів.
8. Визначити кількість від’ємних елементів у масиві. Результат операції – масив з кількості від’ємних елементів у кожному із вхідних масивів.
9. Знайти у масиві номер першого елемента з максимальним значенням. Результат операції – масив з номерів максимальних елементів кожного із вхідних масивів.
10. Знайти у масиві кількість елементів з максимальним значенням. Результат операції – масив з кількостей максимальних елементів кожного із вхідних масивів.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам’яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів (проємулювати роботу користувача з декількома файлами, командою `\exit`);
- не використовувати конструкцію «`using namespace std;`», замість цього слід робити «`using`» кожного необхідного класу: `using std::string, using std::cout`

Контрольні запитання

1. Від чого захищають макроси `assert`?
2. Що не варто поміщати всередину макросів `assert`?
3. Для чого потрібні різні класи для породження виключень?
4. Для чого потрібні специфікатори щодо виключних ситуацій?
5. У яких методах не рекомендується породжувати виключення?
6. У чому полягає «розмотування стека»?
7. Яке призначення конструкції `try-catch`?

Лабораторна робота 10. ШАБЛОННІ ФУНКЦІЇ

Тема. Шаблонні функції.

Мета – отримати базові знання про шаблонізацію (узагальнення) на основі шаблонних функцій.

Теми для попереднього опрацювання:

- функції;
- шаблони;
- масиви;
- типи даних.

Загальні відомості

Шаблон – механізм створення такого коду, для якого вхідними параметрами виступають типи даних.

Інстанціювання шаблону (instantiation) – створення компілятором об'єктного коду для кожного випадку використання комбінацій типів.

Спеціалізація шаблону (specialisation) – код, реалізований для специфічних вхідних типів даних.

За словами проектувальника мови C++ Страуструпа концепцію шаблонів було додано через бажання використовувати параметризовані контейнери, що могли б працювати з усіма можливими типами даних. Під контейнерами тут розуміються динамічний масив (вектор), список, стек, дек, ґешовані таблиці і т. д.

Як відомо, Сі-програміст має створювати окремі функції для кожного типу даних. Наприклад, знаходження максимального елемента у масиві *int* чисел і *double* чисел має виконуватися в окремих функціях, але ж алгоритм однаковий. Такий підхід не є зручним. Задача відокремлення алгоритмів від даних і їх типів є дуже важливою у проектуванні бібліотек. Контейнери на основі шаблонів розв'язують цю задачу.

Для декларування шаблонних конструкцій використовується префікс

```
template<typename T1, typename T2, ... >.
```

У дужках вказуються вхідні типи даних, тут *T1*, *T2* є ідентифікаторами типу даних.

Прийняті такі умови оформлення шаблонів:

– ідентифікатор типу даних має бути коротким, у найпростіших випадках складається із однієї, двох літер (часто *T*);

– для оголошення типу шаблонного аргументу ключове слово *typename* є еквівалентним слову *class*.

Виходячи із цих правил, можна писати *template <class T>*. У літературі і проектах зустрічаються обидва варіанти, але *typename* більше відповідає суті переданого параметра. Приклад шаблону функції *GetMax()*, яка повертає більше значення з двох:

```
template <typename T>
T& GetMax(T& aLeft, T& aRight)
{
    return aLeft < aRight ? aRight : aLeft;
}
```

Така функція може обробляти дані різних типів, наприклад, для типу *int* чи *double*:

```
int a = 1, b = 2;
int max = GetMax( a, b );
double d1 = 1, double d2 = 2.1;
double max = GetMax( d1, d2 );
```

Слід зазначити, що реалізація даної функції можлива, оскільки над стандартними типами реалізована операція «<». Для класів також можна використати *GetMax()*, але заздалегідь необхідно перевизначити *operator<()*. Нижче наводиться приклад такого класу, підготовленого для вказаного шаблону функції.

```
class TData
{
public:
    ...
    bool operator<(const TData& aData)
    {
        return iValue < aData.iValue;
    }
private:
    int iValue;
};

void main(void)
{
    TData data1( 10 ), data2( 20 );
    TData& res = GetMax( data1, data2 );
    // res містить посилання на data2
}
```

Слід звернути увагу, що функції узагальнені, тому для коректності їх роботи з власними типами даних повинні бути перевантажені необхідні оператори.

Загальне завдання. (Задача не пов'язана з попередніми роботами).

Створити клас, який не має полів, а всі необхідні дані передаються безпосередньо у функції.

Клас має виконувати такі дії:

- виводити вміст масиву на екран;
- визначати індекс переданого елемента в заданому масиві;
- сортувати елементи масиву;
- визначати значення мінімального елемента масиву.

При цьому необхідно продемонструвати роботу програми як з використанням стандартних типів даних, так і типів, створених користувачем.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу:
using std::string, using std::cout.

Контрольні запитання

1. Для чого потрібні шаблони?
2. Як описати шаблонну функцію?
3. Як використовувати шаблонну функцію?
4. Чим відрізняються шаблонні функції від звичайних функцій?
5. Які дії виконує компілятор при виклику шаблонної функції?
6. Що треба зробити, щоб як аргумент шаблонної функції можна було вказувати змінну класу?
7. Які використовуються ключові слова для оголошення типу шаблонного аргументу?

Лабораторна робота 11. ШАБЛОННІ КЛАСИ

Тема. Шаблонні класи.

Мета – поширити знання у шаблонізації (узагальненні) на основі вивчення шаблонних класів та створення власних шаблонних типів.

Теми для попереднього опрацювання:

- функції;
- шаблони;
- масиви;
- класи.

Загальні відомості

Класичними прикладами у літературі для шаблонів є контейнери, оскільки базовий набір операцій займає досить багато часу в реалізації, але алгоритми однотипні майже для всіх типів даних. Прикладом шаблонного класу може виступати стек – *CStack*.

```
template <typename T, int STACK_SIZE>
class CStack
{
    public:
        ...

    private:
        // Масив елементів стеку
        T iElements[STACK_SIZE];

        // Кількість елементів в стеку
        int iElementsCount;
}; // class Stack
```

Як уже згадувалось, шаблони можуть використовувати як базові типи, так і нові типи користувача. У поданому прикладі використовується можливість визначити масив змінної довжини, відомої на етапі компіляції, для зберігання даних стека. Оголосити змінні такого типу можна так:

```
CStack <int, 2> stack;
int value = 2;
CStack <int, value> stack;////! помилка, змінну вказувати не можна,
// компілятор не зможе згенерувати масив
```

У поданому вище фрагменті також цікавим є демонстрація можливості використання нешаблонних параметрів. Клас *CStack* використовує тип *int*. Ще може бути використано будь-який цілий числовий

тип (у тому числі і перерахування – *enum*) або посилання чи вказівник. Зумовлена така поведінка необхідністю компілятора абстрагуватись від реалізації специфічних типів, адже по суті вказівники, посилання, *enum* – це також тип *int*. Якщо буде спроба використати нешаблонний тип для *STACK_SIZE*, наприклад, тип *double*, то буде видано повідомлення:

```
error C2993: 'double' : illegal type for non-type template parameter
```

Статичний поліморфізм – це поліморфізм часу компіляції.

Наприклад, є два записи:

```
CStack <int, 10> intStack;  
CStack <double, 20> doubleStack;
```

Для кожного рядка коду компілятор згенерує окремі типи даних (у даному випадку – класи), але з єдиним синтаксисом. Це буде виглядати так, начебто були написані два класи, хоча насправді написаний один – шаблонний. У цьому випадку умова поліморфізму: робота з множиною форм з єдиним синтаксисом – виконується. Але об'єктний код методів для цих двох класів буде різний.

Інстанціювання шаблону – це процес генерації коду для кожного окремого шаблону.

Можливі випадки, коли узагальнений шаблонний клас, котрий відмінно зарекомендував себе у роботі, працює неправильно із певними вхідними типами даних. У таких випадках треба визначити спеціальну поведінку для цього (цих) типу, щоб запобігти можливим проблемам.

Для прикладу можна скористатись реалізацією функції *GetMax()*, що розглянута вище, яка добре себе зарекомендувала у роботі із типом *int* та *double*. Припустимо, що за новими вимогами у проекті необхідно також порівнювати *C*-рядки, базуючись на їх довжині. Є два можливі варіанти – «забути» про наявну функцію *GetMax()* і виконати власну реалізацію із новим іменем або використати механізм спеціалізації.

Спеціалізація «допомагає» компілятору використати готову реалізацію для конкретних типів даних замість універсальної.

```
template <typename T>  
T& GetMax(T& aLeft, T& aRight)  
{  
    return aLeft < aRight ? aRight : aLeft;  
}  
template <>  
const char*& GetMax(const char*& aLeft, const char*& aRight)  
{  
    return ( strlen(aLeft) < strlen(aRight)  ? aRight : aLeft );  
}  
void main()
```

```

{
    const char* name1 = "Hello";
    const char* name2 = "Worlds";
    const char* resRef = GetMax( name1, name2 );
    // resRef посилається на name2;
}

```

Така спеціалізація вирішує одну проблему, але залишає неявною іншу. При використанні неконстантних аргументів функція використовує стандартний шаблон, а отже, порівнює значення вказівників. Зазвичай цю проблему важко відслідкувати, оскільки на ймовірність бінарного варіанту накладається ймовірність бінарного випадіння значень вказівників.

Треба зазначити, що при використанні типу даних при спеціалізації *char** він не буде застосований до типу *const char**, оскільки для компілятора – це два різні типи. Правильним варіантом буде використання ще додаткової спеціалізації.

```

char*& GetMax(char*& aLeft, char*& aRight)
{
    return ( strlen(aLeft) < strlen(aRight)  ? aRight : aLeft );
}

```

Ще одним прикладом використання шаблонів та спеціалізації можна вважати класичний приклад метапрограмування [1]:

```

// Загальний шаблон класу
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};
// Спеціалізація для випадку з нулем
template <>
struct Factorial<0>
{
    enum { value = 1 };
};
// Підрахунок виконується під час компіляції проекту!
int x = Factorial<4>::value; // == 24

```

Загальне завдання

Модернізувати клас, що був розроблений у попередній роботі таким шляхом:

- зробити його шаблонним;
- додати поле – шаблонний масив;
- видалити з аргументів існуючих методів масив, а замість цього використовувати масив-поле класу.

Необхідно продемонструвати роботу програми як з використанням стандартних типів даних, так і типів, які створені користувачем.

Додаткове завдання на оцінку «відмінно»:

- продемонструвати роботу шаблонного класу, в масиві якого знаходиться ієрархія класів (тобто не тільки базовий клас, а ще й клас-спадкоємець).

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію *«using namespace std;»*, замість цього слід робити *«using»* кожного необхідного класу: *using std::string, using std::cout.*

Контрольні запитання

1. Для чого потрібні шаблонні класи?
2. Як декларувати шаблонні класи?
3. Що таке статичний поліморфізм?
4. Що таке спеціалізація?
5. Чи є шаблонними методи у шаблонному класі?

Лабораторна робота 12. STL

Тема. STL. Ітератори. Послідовні контейнери. Цикл *range-for*. Асоціативні контейнери.

Мета – отримати базові знання про *STL*-контейнери. Освоїти основні механізми роботи з *STL* контейнерами.

Теми для попереднього опрацювання:

- класи;
- стандартна бібліотека шаблонів;
- контейнери;
- ітератори.

Загальні відомості

Стандартна бібліотека шаблонів (*Standard Template Library; STL*) – бібліотека для C++, що містить набір узагальнених алгоритмів, контейнерів, засобів доступу до їхнього вмісту і різних допоміжних функцій.

У бібліотеці виділяють п'ять основних компонентів:

- **контейнер** (*container*) – зберігання набору об'єктів у пам'яті; розділяють на чотири категорії: послідовні, асоціативні, контейнери-адаптери і псевдоконтейнери; використовується семантика передачі об'єктів за значенням;
- **ітератор** (*iterator*) – забезпечення засобів послідовного доступу до вмісту контейнера; кожен контейнер підтримує свій вид ітератора, який є інтелектуальним вказівником, що «знає» як отримати доступ до елементів конкретного контейнера;
- **алгоритм** (*algorithm*) – визначення обчислювальної процедури;
- **адаптер** (*adaptor*) – адаптація компонентів для забезпечення різного інтерфейсу;
- **функціональний об'єкт** (*functor*) – заховання функції в об'єкті для використання іншими компонентами.

Розділення дозволяє зменшити кількість компонентів. Наприклад, замість написання окремої функції пошуку елемента для кожного типу контейнера забезпечується єдина версія, яка працює з кожним з них, поки дотримуються основні вимоги.

Контейнери (інакше колекції) – це об’єкти, що зберігають інші елементи і реалізують механізми доступу до них. Кожний контейнер описується шаблонним класом, у якому реалізуються механізми доступу і функції для обробки елементів, що містяться у контейнері. Для використання контейнера у програмі треба додати наступну директиву:

```
#include <T>
```

де *T* – назва контейнера.

Найбільш використовувані такі контейнери:

- ***vector*** – колекція елементів, що збережені у масиві, розмір якого змінюється в міру необхідності (звичайно, що збільшується);
- ***list*** – контейнер, що зберігає елементи у вигляді двонаправленого зв’язаного списку;
- ***map*** – контейнер, що зберігає пари виду *<const Key, T>*, тобто кожен елемент – це пари виду *<ключ, значення>*, при цьому однозначна (кожному ключу відповідає єдине значення). Ключ – деяка величина, що характеризує значення, для якого може бути застосована операція порівняння. Пари зберігаються у відсортованому вигляді, що дозволяє здійснювати швидкий пошук по ключу. Вставка нової пари у контейнер виконується так, щоб умова відсортованості не порушилася;
- ***set*** – це відсортована колекція одних тільки ключів, тобто значень, для яких застосовується операція порівняння. Ключі унікальні – кожен ключ може зустрітись у контейнері (*set* – множина) тільки один раз;
- ***multimap*** – це *map*, в якому відсутня умова унікальності ключа, тобто якщо виконується пошук по ключу, то видається не єдине значення, а набір елементів з однаковим значенням ключа. Для використання контейнера в коді використовується директива: *#include <map>*;
- ***multiset*** – це *set*, для якої відсутня умова унікальності ключа. Для роботи треба підключити *#include <set>*.

Класи для подання рядків. У *STL* рядки подаються як у форматі *ASCII*, так і *Unicode*:

- ***string*** – колекція однобайтових символів у форматі *ASCII*;
 - ***wstring*** – колекція двобайтових символів у форматі *Unicode*;
- включається командою

```
#include <xstring>.
```

Кожний контейнер має свої ітератори для перебору елементів і функції для їх обробки. Наприклад, клас, що описує роботу із вектором, має

відповідний ітератор для прямого доступу до елементів вектора, а також функції для вставки елемента, видалення тощо.

Ітератори – це об’єкти, що реалізують концепцію інтелектуального вказівника і дозволяють одержати доступ до елементів контейнера. Оскільки ітератор є аналогом вказівника, до нього можна застосовувати ті ж самі операції, що і до звичайного вказівника: розіменування, інкремент, декремент, порівняння.

Існують три типи ітераторів:

- *(forward) iterator* – для обходу елементів контейнера від меншого індексу до більшого;
- *reverse iterator* – для обходу елементів контейнера від більшого індексу до меншого;
- *random access iterator* – для обходу елементів контейнера у будь-якому напрямку.

Методи контейнерів. Основними методами, присутніми майже у всіх контейнерах є такі:

- *empty* – визначає, чи порожній контейнер;
- *size* – повертає розмір контейнера;
- *begin* – повертає прямий ітератор, що вказує на початок контейнера;
- *end* – повертає прямий ітератор, що вказує на кінець контейнера, тобто на неіснуючий елемент, що йде після останнього;
- *rbegin* – повертає зворотний ітератор на початок контейнера;
- *rend* – повертає зворотний ітератор на кінець контейнера;
- *clear* – очищає контейнер, тобто видаляє всі його елементи;
- *erase* – видаляє певні елементи з контейнера;
- *capacity* – повертає місткість контейнера, тобто кількість елементів, яку може вмістити цей контейнер (фактично, скільки пам’яті під контейнер виділено);

Місткість контейнера, як було сказано на початку, міняється в міру потреби, тобто якщо вся виділена під контейнер пам’ять уже заповнена, то при додаванні нового елемента місткість контейнера буде збільшена, а всі значення, що були в ньому до збільшення, будуть скопійовані в нову область пам’яті. Це досить «дорога» операція.

range-based for – це цикл по контейнеру, прийнятий у стандарті C++11. Наприклад, оголошено вектор *vect*:

```
std::vector<int> vect;
```

```
// спочатку заповнюється вектор
for (int x : vect)
    std::cout << x << std::endl;
```

Також працює модель посилань:

```
for (int& x : vect)
    x *= 2;
for (const int& x : vect)
    std::cout << x << std::endl;
```

range-based for неявно викликає методи контейнера *begin()* та *end()*, які повертають, у свою чергу, звичні ітератори.

range-based for працює і на звичайних статичних масивах, наприклад:

```
int vect [] = {1, 4, 6, 7, 8};
for (int x : vect)
    std::cout << x << std::endl;
```

Загальне завдання

Маючи класи з прикладної області РГЗ (тільки базовий клас та клас / класи спадкоємці), створити діалогове меню, що дозволяє продемонструвати роботу *STL*-контейнерів (додавання / видалення / отримання даних, показ усіх елементів) та показати їх принципову різницю:

- *vector*;
- *set*;
- *list*;
- *map*.

При цьому врахувати, що контейнери містять елементи одного типу, наприклад, базового.

Прохід по всьому контейнеру повинен виконуватися за допомогою циклу мови C++11 – *range-for*.

Додаткове завдання на оцінку «відмінно»:

- контейнери повинні оперувати даними не тільки базового класу, а ще даними класів-спадкоємців.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std*;», замість цього слід робити «*using*» кожного необхідного класу: *using std::string*,

using std::cout.

Контрольні запитання

1. Що таке стандартна бібліотека шаблонів?
2. Які складові входять до *STL*?
3. Наведіть приклад використання циклу *range-for*.
4. Яка різниця між послідовними контейнерами *vector* та *list*?
5. Яка різниця між послідовними контейнерами *vector* та *set*?
6. Що таке «ітератор», яке його призначення?
7. Які бувають ітератори?
8. Які операції можуть застосовуватися до ітераторів?

Лабораторна робота 13. АЛГОРИТМИ ПЕРЕМІЩЕННЯ ТА ПОШУКУ

Тема. STL. Алгоритми переміщення та пошуку.

Мета – на практиці порівняти *STL*-алгоритми, що не модифікують послідовність.

Теми для попереднього опрацювання:

- класи;
- стандартна бібліотека шаблонів;
- контейнери;
- алгоритми;
- ітератори.

Загальні відомості

Алгоритми виконують операції над елементами, що містяться у контейнері. Алгоритми визначені у заголовному файлі *<algorithm.h>*.

STL має у своєму складі величезний набір оптимальних реалізацій популярних алгоритмів, які забезпечують роботу з STL-колекціями. При цьому операції виконуються над діапазонами елементів. Діапазон визначається як *[first, last)*, де *last* – елемент, що є наступним за останнім, над яким виконуються операції. Іншими словами, операції виконуються над елементами з номерами від *first* до *last-1*, включно.

Усі реалізовані функції можна поділити на три групи:

- методи перебору усіх елементів колекції та їх обробку:

count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search, copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition

- методи сортування колекції:

sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, make_heap, pop_heap, set_symmetric_difference, push_heap, sort_heap, min, max, min_element, max_element, lexicographical_compare, next_permutation, prev_permutation

- методи виконання арифметичних операцій над членами колекцій:

accumulate, inner_product, partial_sum, adjacent_difference.

Існує ще така класифікація перерахованих функцій:

- функції, які не змінюють послідовність елементів у колекції: перевірка умови, пошук, порівняння;
- функції, які змінюють послідовність елементів у колекції: копіювання, видалення, переміщення, зсув елементів, присвоювання елементу вказаного значення;
- функції розподілу елементів у колекції на діапазони;
- функції сортування;
- функції пошуку заданого елемента;
- функції пошуку мінімального / максимального елемента.

Основними алгоритмами з указаних є такі: заповнення контейнера за відповідним правилом (*fill*), сортування за вказаним критерієм (*sort*), пошук за заданим критерієм (*lower_bound*, *max_element*, *min_element*), заміна елементів контейнера (*replace*).

for_each – функція обходу колекції. Для роботи вимагає три параметри: перші два задають діапазон, третій – вказівник на функцію, яку алгоритм має виконати. Наприклад,

```
/* шаблонуна функція, яка виводить переданий їй елемент */
template <typename T> void print_element (T el)
{
    cout << el << "; " ;
}

int iarray [] = {5,6,1,56,7,234,7,8,45,65,87} ;
int ilen = sizeof (iarray)/sizeof(iarray[0]) ;
/* виводяться значення усіх елементів масиву на екран */
for_each (iarray, iarray+ilen, print_element<int>);
```

Загальне завдання.

Поширити попередню лабораторну роботу, додаючи такі можливості діалогового меню:

- виведення всіх елементів масиву за допомогою *STL*-функції *for_each*;
- визначення кількості елементів за заданим критерієм;
- пошук елемента за заданим критерієм.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу: *using std::string, using std::cout.*

Контрольні запитання

1. Що таке алгоритми?
2. Що таке біндери? Які бувають типи біндерів?
3. Що робить функція *for_each* ?
4. Які функції дозволяють виконувати пошук у контейнері?
5. Які функції дозволяють виконувати визначення кількості елементів у контейнері за заданими характеристиками?

Лабораторна робота 14. СОРТУВАННЯ

Тема. STL. Алгоритми зміни послідовності. Сорткування. Функтори.

Мета – на практиці порівняти STL-алгоритми, що модифікують послідовність; отримати навички роботи з STL-функторами.

Теми для попереднього опрацювання:

- класи;
- стандартна бібліотека шаблонів;
- контейнери;
- алгоритми;
- ітератори.

Загальні відомості

Припустимо, поставлена така задача: вивести на екран усі елементи контейнера, що містять дані типу *int*, а потім окремо визначити їх суму.

Поставлену задачу можна реалізувати «в лоб» написавши окремі функції виводу та калькуляції суми, або більш вишукано – написавши функцію перебору всіх елементів, назвавши її, наприклад, *ListAll()*, передавши у неї першим параметром контейнер, а другим власне об'єкт, який виконуватиме потрібну операцію. Функція *ListAll()* повинна для кожного елемента викликати певний метод об'єкта. Якщо допустити, що узгодженим методом буде перевизначена операція *()*, то буде отриманий такий код функції:

```
template<typename F>
void ListAll(const CLimitSizeContainer& aContainer, F& aFunctor)
{
    for ( int i = 0; i < aContainer.Count(); i++ )
    {
        aFunctor( aContainer.At(i) );
    }
}
```

Функцію, що виконує універсальні дії, прийнято називати **алгоритмом**.

Класи, в яких перевантажений оператор *()*, а їх об'єкти схожі на функцію, називаються **функторами** та **предикатами**.

Вимога для предиката – оператор *()* повинен повертати значення типу *bool*. Вимога для функтора – у оператора *()* тип значення, що вертається, повинен бути відмінним від *bool*.

Оскільки круглі дужки – це сигнатура функції, то об’єкт класу, у якого перевизначена операція `()`, є функтором.

Наведемо приклад функтора, що відображає один елемент:

```
class CDisplayer
{
public:
void operator() (int aItem)
{
    printf( "%d\n", aItem);
}
};
```

Використання алгоритму та функтора виглядатиме таким чином:

```
CDisplayer displayer;
ListAll( container, displayer );
```

У результаті на екран будуть видані значення всіх елементів контейнера.

Для реалізації другої частини програми, де потрібно порахувати суму, необхідно написати вже більш складний клас.

```
class CSumOfAll
{
public:
    CSumOfAll(): iSum(0){};
    int Sum() { return iSum; }
    void operator() (int aItem){ iSum += aItem; }
private:
    int iSum;
};
```

Використання такого класу буде таким:

```
CSumOfAll summarizer;
ListAll( container, summarizer );
printf( "Sum = %d", summarizer.Sum() );
```

Реалізація за допомогою функтора має очевидні недоліки та переваги. До недоліків можна віднести надлишковий об’єм коду для простих операцій. Безумовними перевагами є реалізація відносно незалежних один від одного фрагментів програмного коду, що повністю відповідає концепції ООП.

Загальне завдання

Поширити попередню лабораторну роботу, додаючи такі можливості діалогового меню:

- об’єднання двох STL-контейнерів типу *vector*;
- сортувати заданий контейнер з використанням функтора.

Додаткове завдання на оцінку «відмінно»

Додати можливість об'єднання двох STL-контейнерів типу *map*. При цьому, якщо в обох контейнерах існують однакові ключі, то значення повинні конкатенуватися, наприклад, якщо є дві мапи для країн:

- Мапа1:
 - Україна : Харків, Київ;
 - Росія: Москва, Белгород;
 - Білорусь: Мінськ, Бобруйськ.

- Мапа2:
 - Польща: Варшава;
 - Росія: Санкт-Петербург;
 - Україна: Харків, Запоріжжя;

то об'єднана мапа повинна містити таке:

- Україна: Харків, Київ, Запоріжжя;
- Росія: Москва, Белгород, Санкт-Петербург;
- Білорусь: Мінськ, Бобруйськ;
- Польща: Варшава.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів;
- не використовувати конструкцію «*using namespace std;*», замість цього слід робити «*using*» кожного необхідного класу: *using std::string, using std::cout.*

Контрольні запитання

1. Що таке алгоритми у стандартній бібліотеці шаблонів?
2. Які методи дозволяють об'єднувати колекції?
3. Які методи дозволяють виконувати сортування у колекції за заданим критерієм для вказаного діапазону?
4. Що таке функтор?
5. Як працює контейнер *map*?

Лабораторна робота 15. РОЗУМНІ ВКАЗІВНИКИ

Тема. Розумні вказівники.

Мета – по результатах практичної роботи порівняти розумні вказівники бібліотеки STL.

Теми для попереднього опрацювання:

- класи;
- стандартна бібліотека шаблонів;
- контейнери;
- вказівники;
- розумні вказівники.

Загальні відомості

Припустимо, дано такий фрагмент програмного коду:

```
Object *a = new Object();  
list<Object*> list1, list2;    //об'ява двох контейнерів  
list1.push_back(a);          // додавання об'єкта у контейнер list1  
list1.push_back(a);          // додавання об'єкта у контейнер list1  
list2.push_back(a);          // додавання об'єкта у контейнер list2
```

При спробі очистити контейнери з видаленням з них усіх об'єктів може виникнути проблема подвійного очищення пам'яті. Використання розумних вказівників, приклад, *shared_ptr*, дозволяє уникнути такої помилки – вони самі знають, коли дійсно необхідно звільняти пам'ять. Що виділена для об'єкта.

Розумні вказівники (*Smart pointer*) – це об'єкт, працювати з яким можна як зі звичайним вказівником, але при цьому, на відміну від останнього, він надає деякий додатковий функціонал (наприклад, автоматичне звільнення закріпленої за вказівником області пам'яті).

Розумні вказівники призначені для боротьби з витоками пам'яті, які складно уникнути у великих проектах. Особливо вони зручні у місцях, де виникають виключення, тому що у цих випадках відбувається процес розкручування стека і знищуються локальні об'єкти. У випадку звичайного вказівника буде знищена змінна-вказівник, але при цьому ресурс пам'яті залишиться незвільненим. У випадку ж розумного вказівника буде викликаний деструктор, який і звільнить виділений раніше ресурс пам'яті.

У новому стандарті C++ введені такі розумні вказівники: *unique_ptr*, *shared_ptr* та *weak_ptr*. Усі вони оголошені у заголовному файлі *<memory>*.

unique_ptr – вказівник, який замінив *auto_ptr*. Основна проблема останнього полягає у правах володіння. Об'єкт цього класу втрачає права володіння ресурсом при копіюванні (присвоюванні, використанні у конструкторові копій, передачі у функцію за значенням). На відміну від *auto_ptr*, вказівник *unique_ptr* забороняє копіювання.

```
std::unique_ptr<int> x_ptr(new int(42));
std::unique_ptr<int> y_ptr;
// помилка при компіляції
y_ptr = x_ptr;
```

```
// помилка при компіляції
std::unique_ptr<int> z_ptr(x_ptr);
```

Зміна прав володіння ресурсом здійснюється за допомогою допоміжної функції *std::move* (яка є частиною механізму переміщення).

```
std::unique_ptr<int> x_ptr(new int(42));
std::unique_ptr<int> y_ptr;
// також, як і у випадку з ``auto_ptr``, права володіння переходять
// до y_ptr, а x_ptr починає вказувати на null pointer
y_ptr = std::move(x_ptr);
```

Вказівник *unique_ptr* має методи *reset()*, який скидає права володіння, і *get()*, який повертає класичний вказівник.

```
std::unique_ptr<Foo> ptr = std::unique_ptr<Foo>(new Foo);
// одержується класичний вказівник
Foo *foo = ptr.get();
foo->bar();
// скидаються права володіння
ptr.reset();
```

Як видно, *unique_ptr* не є зручим у використанні, але він убезпечив від неявних змін прав володіння ресурсом.

shared_ptr – самий популярний і самий широко використовуваний розумний вказівник. Він реалізує підрахунок посилань на ресурс. Ресурс звільниться тоді, коли лічильник посилань на нього буде дорівнювати 0. Як видно, система реалізує одне з основних правил збирання сміття.

```
std::shared_ptr<int> x_ptr(new int(42));
std::shared_ptr<int> y_ptr(new int(13));
// після виконання даного рядка, ресурс
// на який вказував раніше y_ptr (int(13)) звільниться,
// а на int(42) будуть посилатися обидва вказівника
y_ptr = x_ptr;
std::cout << *x_ptr << "\t" << *y_ptr << std::endl;
// int(42) звільниться лише при знищенні останнього, що посилається
// на нього вказівника
```

Також, як *unique_ptr* та *auto_ptr*, цей клас надає методи *get()* і *reset()*.

```
auto ptr = std::make_shared<Foo>();
Foo *foo = ptr.get();
foo->bar();
```

```
ptr.reset();
```

При роботі з розумним вказівником слід побоюватися їхнього створення на льоту. Наприклад, наступний код може привести до витоку пам'яті.

```
somefunction(std::shared_ptr<Foo>(new Foo), getrandomkey());
```

Тому що стандарт C++ не визначає порядок обчислення аргументів. Може трапитися так, що спочатку виконається *new Foo*, потім *getrandomkey()* і лише потім конструктор *shared_ptr*. Якщо ж функція *getrandomkey()* згенерує виключення, то конструктора *shared_ptr* не буде викликано, хоча ресурс (об'єкт *Foo*) був уже виділений.

У випадку з *shared_ptr* є вихід – використовувати функцію *std::make_shared<>*, яка створює об'єкт заданого типу і повертає вказівник *shared_ptr*, що вказує на нього.

```
somefunction(std::make_shared<Foo>(), getrandomkey());
```

Як вже було сказано вище, *make_shared* повертає *shared_ptr*. Цей результат є тимчасовим об'єктом, а стандарт C++ чітко декларує, що тимчасові об'єкти знищуються у випадку появи виключення.

new Foo теж повертає тимчасовий об'єкт. Однак тимчасовим є вказівник на виділений ресурс, і у випадку виключення – знищиться вказівник, при цьому ресурс залишиться виділеним.

weak_ptr – цей клас дозволяє зруйнувати циклічну залежність, яка, безсумнівно, може утворюватися при використанні *shared_ptr*. Припустимо, є така ситуація (змінні-члени не інкапсульовані для спрощення коду)

```
class Bar;
class Foo
{
public:
    Foo() { std::cout << "Foo()" << std::endl; }
    ~foo() { std::cout << "~foo()" << std::endl; }

    std::shared_ptr<Bar> bar;
};
```

```
class Bar
{
public:
    Bar() { std::cout << "Bar()" << std::endl; }
    ~bar() { std::cout << "~bar()" << std::endl; }

    std::shared_ptr<Foo> foo;
};
```

```
int main()
{
    auto foo = std::make_shared<Foo>();

    foo->bar = std::make_shared<Bar>();
    foo->bar->foo = foo;

    return 0;
}
```

Як видно, об'єкт *foo* посилається на *bar* і навпаки. Як наслідок, утворено цикл, через який не будуть викликатися деструктори об'єктів.

Для того щоб розірвати цей цикл, досить у класі *Bar* замінити вказівник *shared_ptr* на *weak_ptr*.

Чому ж утворено цикл? При виході із блоку (у цьому випадку функції *main()*) знищуються локальні об'єкти. Локальним об'єктом є *foo*. При знищенні *foo* лічильник посилань на його ресурс зменшиться на одиницю. Однак ресурс звільнений не буде, тому що на нього є посилання з боку ресурсу *bar*. А на *bar* є посилання з боку того ж ресурсу *foo*.

Вказівник *weak_ptr* не дозволяє працювати з ресурсом прямо, але має метод *lock()*, який генерує *shared_ptr*.

```
std::shared_ptr<Foo> ptr = std::make_shared<Foo>();
std::weak_ptr<Foo> w(ptr);
if (std::shared_ptr<Foo> foo = w.lock())
{
    foo->dosomething();
}
```

Варто відзначити, що розглянуті розумні вказівники (крім *unique_ptr*) не призначені для володіння масивами. Це пов'язано з тим, що деструктор викликає саме *delete*, а не *delete[]* (а саме це потрібно для масивів).

Для *unique_ptr* мають справу з визначеною спеціалізацією для масивів. Для її використання необхідно вказати *[]* біля параметра шаблону. Виглядає це так.

```
std::unique_ptr<Foo[]> arr(new Foo[2]);
arr[0].dosomething();
```

Наостанок слід зазначити таке: один розумний вказівник може використовуватися у декількох контейнерах, що дозволяє економити пам'ять.

Загальне завдання

Створити STL-контейнер, що містить у собі об'єкти ієрархії класів, використати розумні вказівники:

- *auto_ptr*;

- *unique_ptr*;
- *shared_ptr*;
- *weak_ptr*.

Додаткове завдання на оцінку «відмінно»

Створити власний розумний вказівник, поданий у вигляді шаблонного класу, який:

- має перевантажений оператор * та -> для отримання фактичного об'єкта та його вказівника;
- дозволяє підраховувати кількість вказівників на об'єкт. Продемонструвати дії, коли виникає інкремент та декремент кількості вказівників;
- контролювати виток пам'яті при виникненні виняткової ситуації.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті при відсутності явного видалення пам'яті за допомогою функцій *delete* / *free*.

Контрольні запитання

1. Що таке розумний вказівник? Чим він відрізняється від звичайного?
2. Порівняйте використані розумні вказівники.
3. Як можна використовувати розумні вказівники при роботі з контейнерами STL?
4. Яке призначення розумних вказівників?
5. Порівняйте вказівники *unique_ptr*, *shared_ptr* та *weak_ptr*.

Лабораторна робота 16. РОБОТА З ДИНАМІЧНОЮ ПАМ'ЯТТЮ

Тема. Системна робота з динамічною пам'яттю.

Мета – дослідити особливості мови C++ при роботі з динамічною пам'яттю.

Теми для попереднього опрацювання:

- класи;
- перевантаження операторів;
- функції виділення та видалення пам'яті.

Загальні відомості

Програміст може керувати виділенням пам'яті, перевантажуючи оператори *new* і *delete*. Перевантажена операторна функція має такий вигляд:

```
void* operator new(size_t size);
```

Вона виділяє *size* байт пам'яті і повертає адресу виділеної пам'яті. Конструктор і деструктор об'єктів викликаються автоматично. Тип *size_t* є цілочисловим.

Перевантажений оператор *delete* звільняє пам'ять, виділену перевантаженим оператором *new*.

Розглянемо програму, в якій оператор *new* реалізує виділення пам'яті за допомогою функції *malloc()*.

Таке перевантаження може виявитися корисним при сполученні декількох модулів, написаних мовами C і C++. Як відомо, одночасне використання механізмів розподілу пам'яті, передбачених у мовах C і C++, тобто використання пар *new* – *free()* та *malloc()* – *delete*, може привести до непередбачуваних результатів. Отже, перевантаження оператора *new*, може призвести різні модулі «до загального знаменника».

Оператор *new* має особливу форму перевантаження, яка називається синтаксисом розміщення. Вона дозволяє створювати об'єкт, розміщуючи його в осередку з заданою адресою. А саме у цьому випадку необхідно явно викликати деструктор. Синтаксис розміщення такий:

```
#include <iostream.h>
#include <new.h>
class TClass
{
    public:
```

```

    int a;
    TClass() {cout << "Ctor TClass " << endl;}
    ~TClass() {cout << "Dtor TClass " << endl;}
};
int main()
{
    char memory[sizeof(TClass)];
    void *p=memory;
    TClass* q = new(p) TClass;
    cout << q << endl;
    q->~TClass();
    return 0;
}

```

Результат роботи цієї програми виглядає так:

```

Ctor TClass
0x0065fdc4
Dtor TClass

```

При створенні власного механізму розподілу пам'яті для масивів, можна перевантажити оператори *new[]* і *delete[]*, керуючись тими ж правилами.

```

#include <iostream.h>
#include <malloc.h>
void* operator new[](size_t size)
{
    void *p;
    cout << "New " << endl;
    p = malloc(size);
    return p;
}
void operator delete[](void* p)
{
    cout << "Delete\n";
    free(p);
}
int main()
{
    char* q;
    q = new char[5];
    delete[] q;
    return 0;
}

```

У цій програмі виділяється пам'ять для масиву, що складається з 5 символів, а потім вона звільняється.

Як правило, перевантажені оператори *new* і *delete* генерують виняткові ситуації. Однак існують версії цих операторів, у яких генерація виняткової ситуації скасована.

Загальне завдання

Маючи класи з прикладної області РЗ (тільки базовий клас та клас / класи-спадкоємці), перевантажити оператори *new* / *new []* та *delete* / *delete []*. Продемонструвати їх роботу і роботу операторів розміщення *new* / *delete* при розробці власного менеджера пам'яті (сховища).

Детальна інформація про власне сховище: є статично виділений масив заданого обсягу. Організувати виділення і звільнення пам'яті елементів ієрархії класів тільки у рамках цього сховища.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті.

Контрольні запитання

1. Що таке «оператор»?
2. Чим відрізняється оператор *new* та *new []* / *delete* і *delete []*?
3. Що таке оператор «розміщення» *new* / *delete*?

Лабораторна робота 17. СЕРІАЛІЗАЦІЯ

Тема. Серіалізація у JSON та XML-форматах.

Мета – поглибити навички роботи з серіалізацією об'єктів. Дослідити механізм серіалізації та десеріалізації об'єктів у JSON та XML-форматах.

Теми для попереднього опрацювання:

- класи;
- робота з файлами;
- бібліотеки.

Загальні відомості

JSON (англ. *JavaScript Object Notation*, укр. об'єктний запис *JavaScript*, вимовляється джейсон) – це текстовий формат обміну даними між комп'ютерами. *JSON* базується на тексті, може бути прочитаним людиною. Формат дозволяє описувати об'єкти та інші структури даних. Цей формат головним чином використовується для передачі структурованої інформації через мережу (завдяки процесу, що називають серіалізацією).

JSON знайшов своє головне призначення у написанні веб-програм, а саме при використанні технології *AJAX*. *JSON*, що використовується в *AJAX*, виступає як заміна *XML* (використовується в *AJAX*) під час асинхронної передачі структурованої інформації між клієнтом та сервером. При цьому перевагою *JSON* перед *XML* є те, що він дозволяє використовувати складні структури в атрибутах, займає менше місця і прямо інтерпретується за допомогою *JavaScript* в об'єкти.

JSON будується на двох структурах:

- набір пар ім'я / значення. У різних мовах це реалізовано як об'єкт, запис, структура, словник, хеш-таблиця, список з ключем або асоціативним масивом;
- впорядкований список значень. У багатьох мовах це реалізовано як масив, вектор, список, або послідовність.

Це універсальні структури даних. Теоретично всі сучасні мови програмування підтримують їх у тій чи іншій формі. Оскільки *JSON* використовується для обміну даними між різними мовами програмування, то є сенс будувати його на цих структурах.

У *JSON* використовуються такі їхні форми:

- об'єкт – це послідовність пар ім'я/значення. Об'єкт починається з символу `{` і закінчується символом `}`. Кожне значення слідує за `:` і пари ім'я/значення відділяються комами;
- масив – це послідовність значень. Масив починається символом `[` і закінчується символом `]`. Значення відділяються комами.

Значення може бути рядком у подвійних лапках, або числом, або логічними *true* чи *false*, або *null*, або об'єктом, або масивом. Ці структури можуть бути вкладені одна в одну.

Рядок – це послідовність з нуля або більше символів юнікоду, обмежена подвійними лапками, з використанням *escape*-послідовностей, що починаються зі зворотної косої риски `\`. Символи подаються простим рядком.

Тип Рядок (*String*) дуже схожий на *String* у мовах *C* та *Java*. Число теж дуже схоже на *C* або *Java*-число, за винятком того, що вісімкові та шістнадцяткові формати не використовуються. Пропуски можуть бути вставлені між будь-якими двома лексемами.

Наступний приклад показує *JSON* подання об'єкта, що описує людину. В об'єкті є рядкові поля імені і прізвища, об'єкт, що описує адресу, та масив, що містить список телефонів.

```
{
  "firstName": "Іван",
  "lastName": "Іванов",
  "address": {
    "streetAddress": "вул. Кирпичова 2",
    "city": "Харків",
    "postalCode": 61002
  },
  "phoneNumbers": [
    "057 123-1234",
    "050 123-4567"
  ]
}
```

Розширювана мова розмітки (англ. *Extensible Markup Language*, скорочено *XML*) – запропонована консорціумом *World Wide Web* (W3C), стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через Інтернет. Є спрощеною підмножиною мови розмітки *SGML*. *XML*-документ складається із текстових знаків і придатний до читання людиною.

Стандарт *XML* (англ. *Recommendation*, перше видання від 10 лютого 1998, останнє, четверте видання 29 вересня 2006) визначає набір базових

лексичних та синтаксичних правил для побудови мови описання інформації шляхом застосування простих *тегів*. Цей формат достатньо гнучкий для того, аби бути придатним для застосування в різних галузях. Іншими словами, запропонований стандарт визначає метамову, на основі якої шляхом запровадження обмежень на структуру та зміст документів визначаються специфічні, предметно-орієнтовані мови розмітки даних. Ці обмеження описуються мовами схем (англ. *Schema*), такими як *XML Schema (XSD)*, *DTD* або *RELAX NG*.

Прикладами мов, заснованих на *XML*, є: *XSLT*, *XAML*, *XUL*, *RSS*, *MathML*, *GraphML*, *XHTML*, *SVG*, а також *XML Schema*.

XML-документ має ієрархічну логічну структуру, і може представлятись у вигляді дерева. Вузлами цього дерева можуть бути:

- елементи, фізична структура яких складається із:
 - коректної пари відкриваючого та закриваючого тегів (*<Назва-тега>*) та (*</Назва-тега>*), або
 - тега порожнього елемента (*<Назва-тега />*);
- атрибути, що мають вигляд пар ключ-значення (*назва атрибута="значення атрибута"*) і знаходяться або у відкриваючому, або у порожньому тезі (подібно до метаданих);
- вказівки щодо обробки документа (англ. *Processing Instruction*) (*<?Обробник параметр ?>*);
- коментарі (*<!-- Текст коментаря -->*);
- текст, або у вигляді простого тексту, або фрагментів CDATA (*<![CDATA[довільний текст]]>*).

XML-документ повинен мати лише один кореневий елемент. Решта елементів є піделементами цього кореневого елемента.

Деякі веб-браузери здатні безпосередньо відображати *XML*-документи. Це може досягатись шляхом застосування таблиці стилів (англ. *Stylesheet*). Вказані у таблиці стилів операції можуть призводити до перетворення *XML*-документа в інший, відмінний від *XML* формат.

Корисні посилання:

- <https://github.com/nlohmann/json/>
- <https://github.com/leethomason/tinyxml2>
- https://github.com/elios264/pakal_persist
- <https://github.com/USCiLab/cereal>

Загальне завдання

Маючи класи з прикладної області РЗ, виконати модернізацію наступним шляхом:

- існує файл конфігурації, в котрому вказується, в якому форматі серіалізувати. На основі даних цього файлу, потрібно організувати запис до файлу у звичайному вигляді (як було раніше), *JSON*-форматі або *XML*-форматі;
- кожен формат файлу має своє розширення. На базі цього розширення при десеріалізації з файлу обирається потрібний механізм.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті;
- продемонструвати роботу розроблених методів за допомогою модульних тестів.

Контрольні запитання

1. Виконайте порівняльну характеристику *JSON* та *XML*-форматів.
2. Які ще формати серіалізації об'єктів вам відомі?
3. *XML*-формат: що таке тег, параметр, значення?
4. *JSON*-формат: які примітивні типи даних підтримуються?

Лабораторна робота 18. GTK+

Тема. Розробка графічного додатку з використанням *GTK+*.

Мета – отримати знання та практичний досвід розробки графічного інтерфейсу на основі використання бібліотеки *GTK+*.

Теми для попереднього опрацювання:

- класи;
- робота з файлами;
- бібліотеки.

Загальні відомості

GTK+ (від *The GIMP ToolKit*) – кросплатформовий набір інструментів для створення графічних інтерфейсів користувача. Разом із *Qt* є одним із найпопулярніших інструментів для *X Window System*.

GTK+ було розроблено для *GNU Image Manipulation Program (GIMP)*, растрового графічного редактора, у 1997 р. Спенсером Кімбалом (*Spencer Kimball*) та Петером Матісом (*Peter Mattis*), членами *eXperimental Computing Facility (XCF)* в *UC Berkeley*. *GTK+* розвивається у рамках проекту *GNU* і є вільним програмним забезпеченням. Код *GTK+* поширюється під ліцензією *GPL*, що дозволяє використовувати *GTK+* не тільки для розробки вільного ПЗ, а і для створення власних програм, не вимагаючи від виробників закритих програм виплати роялті або купівлі спеціальної ліцензії.

До складу тулкіта входить повний набір віджетів, що дозволяють використовувати *GTK+* для проектів різного рівня і розміру. *GTK+* спеціально спроектований для підтримки не тільки C/C++, але й інших мов програмування, таких як *Perl* та *Python*, що у поєднанні з використанням візуального будівника інтерфейсу *Glade* дозволяє істотно спростити розробку і скоротити час написання графічних інтерфейсів. Організація виводу в *GTK+* абстрагована від типу віконних систем, наприклад, поставляється бекенд, що забезпечує можливість роботи поверх дисплейного сервера *Wayland*, а також бекенд, котрий дозволяє здійснювати виведення бібліотеки *GTK+* у вікні веб-браузера (запустивши *GTK*-застосунок на одній машині, можна відкрити браузер на іншій машині і отримати доступ до інтерфейсу цієї програми).

Докладну інформацію зі встановлення та настроювання *GTK+* можна знайти за посиланнями: <https://www.gtk.org/download/windows.php> та <http://irkedrants.blogspot.com.es/2011/02/configuring-eclipse-to-compile-gtk.html>

Загальне завдання

Маючи класи з прикладної області РЗ, виконати модернізацію таким шляхом:

- перетворити існуюче діалогове консольне меню на віконне;
- кожен пункт меню перетворюватиметься на кнопку (*button*) на формі;
- усі дані, що потрібно вводити користувачем виконуються у модальному вікні. Кожне поле заповнюється у відокремленому текстовому полі / списку, що випадає, тощо;
- вибір файлу для запису / читання виконується за допомогою стандартних механізмів вибору файлів;
- відображення результатів навести у табличному вигляді.

Додаткові умови виконання завдання:

- продемонструвати відсутність витоків пам'яті.

Контрольні запитання

1. Крім *GTK+* які ще існують бібліотеки розробки графічних додатків?
2. Що таке *signal* у контексті бібліотеки *GTK+*?
3. Що таке діалогове вікно та чим воно відрізняється від звичайного?
4. Які компоненти вікна відомі?
5. Що таке *widget* у контексті бібліотеки *GTK+*?

Список літератури

1. Дейтел Х.М. Как программировать на Си++ / Х.М. Дейтел, П.Дж. Дейтел М. : ЗАО БИНОМ, 1999. – 1000 с.
2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++ / Г. Буч; пер. с англ. 2-е изд. – М. : «Издательство Бином». 1998. – 560 с.
3. Павловская Т.А. С++. Объектно-ориентированное программирование: Практикум. / Т.А. Павловская, Ю.А. Щупак – С.Пб: Питер, 2004 – 288 с.
4. Павловская Т.А. С++. Структурное программирование: Практикум. / Т.А. Павловская, Ю.А. Щупак – С.Пб: Питер, 2003 – 240 с.
5. Павловская Т.А. С/С++ Программирование на языке высокого уровня. – С.Пб: Питер, 2007. – 461 с.
6. Вандервуд, Джосаттис - Шаблоны С++. Справочник разработчика. / Пер. с англ. - М.: Вильямс, 2008. – 536 с.
7. Страуструп Б. Дизайн и эволюция С++ / Б. Страуструп; пер. с англ. – М. : ДМК Пресс; С.Пб: Питер, 2007. – 445 с.
8. Остерн Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки шаблонов С++ / Остерн; Пер. с англ. – С.Пб: Невский Диалект, 2004. – 544 с.

Зміст

ВСТУП.....	3
Лабораторна робота 1. КЛАСИ.....	5
Лабораторна робота 2. ПЕРЕВАНТАЖЕННЯ МЕТОДІВ.....	13
Лабораторна робота 3. ПОТОКИ	19
Лабораторна робота 4. РЕГУЛЯРНІ ВИРАЗИ.....	25
Лабораторна робота 5. АГРЕГАЦІЯ ТА КОМПОЗИЦІЯ.....	29
Лабораторна робота 6. СПАДКУВАННЯ	33
Лабораторна робота 7. ПОЛІМОРФІЗМ	39
Лабораторна робота 8. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ.....	47
Лабораторна робота 9. ВИКЛЮЧЕННЯ	53
Лабораторна робота 10. ШАБЛОННІ ФУНКЦІЇ.....	63
Лабораторна робота 11. ШАБЛОННІ КЛАСИ	67
Лабораторна робота 12. STL	71
Лабораторна робота 13. АЛГОРИТМИ ПЕРЕМІЩЕННЯ ТА ПОШУКУ ..	77
Лабораторна робота 14. СОРТУВАННЯ.....	81
Лабораторна робота 15. РОЗУМНІ ВКАЗІВНИКИ.....	85
Лабораторна робота 16. РОБОТА З ДИНАМІЧНОЮ ПАМ'ЯТТЮ	91
Лабораторна робота 17. СЕРІАЛІЗАЦІЯ	95
Лабораторна робота 18. GTK+	99
Список літератури	101

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт
з дисципліни «Програмування»
для студентів першого курсу всіх форм навчання
спеціальності «Комп'ютерна інженерія» та «Кібербезпека»
(частина 2)

Укладачі: ДАВИДОВ Вячеслав Вадимович
ДАЛЕКА Валентина Дмитрівна
МОЛЧАНОВ Георгій Ігорович
СЕМЕНОВ Сергій Геннадійович

Відповідальний за випуск проф. Семенов С. Г.

Роботу до видання рекомендував проф. Заполовський М. Й.

Редактор Н.В. Верстюк

План 2018 р., поз. 396

Підп. до друку 03.06.2019. Формат 60x84 1/16. Папір офсетний.

Друк – ризографія. Гарнітура Times New Roman. Ум.-друк. арк. 5,7.

Наклад 100 прим. Зам. № _____. Ціна договірна.

Видавничий центр НТУ «ХП»

Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.

61002, Харків, вул. Кирпичова, 2

Надруковано ТОВ «Видавництво «Форт»

Свідоцтво про внесення до Державного реєстру видавців

ДК №333 від 09.02.2001 р.

61023, м. Харків, а/с 10325. Тел. (057)714-09-08