

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”**

**ПРОГРАМУВАННЯ, ЧАСТИНА 2 (ОБ’ЄКТНО-
ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ)**

МЕТОДИЧНІ ВКАЗІВКИ

до виконання циклу лабораторних робіт з дисципліни
“Програмування, частина 2 (Об’єктно-орієнтоване програмування)” для
студентів базового напрямку 6.050102 “Комп’ютерна інженерія”

Затверджено
на засіданні кафедри
”Електронно-обчислювальних машини”
Протокол № 7 від 13.02.2015 р.

Львів 2015

Програмування, частина 2 (Об’єктно-орієнтоване програмування): методичні вказівки до виконання циклу лабораторних робіт з дисципліни “Програмування, частина 2 (Об’єктно-орієнтоване програмування)” для студентів базового напрямку 6.050102 “Комп’ютерна інженерія”/ Укл.: Ю.В.Морозов, М.В.Олексів, Мороз І.В. – Львів: Видавництво Національного університету “Львівська політехніка”, 2015. – 165 с.

Укладачі

Морозов Ю.В., доцент, к.т.н.

Олексів М. В., ст. викладач, к.т.н.

Мороз І.В., ст. викладач, к.т.н.

Рецензент

Дунець Р.Б., д. т. н, професор

Відповідальний за випуск:

Мельник А. О., професор, завідувач кафедри

ЗМІСТ

Вступ.....	4
Лабораторна робота №1. Перевантаження функцій.....	5
Лабораторна робота № 2. Поточний ввід-вивід	17
Лабораторна робота № 3. Засоби роботи з динамічною пам'яттю. Динамічні масиви.....	37
Лабораторна робота № 4. Класи та об'єкти.....	47
Лабораторна робота № 5. Перевантаження операторів	77
Лабораторна робота № 6. Спадкування	95
Лабораторна робота № 7. Множинне спадкування. Поліморфізм.....	110
Лабораторна робота № 8. Шаблони	135

ВСТУП

Об'єктно-орієнтоване програмування (ООП) — одна з парадигм програмування, яка розглядає програму як множину “об'єктів”, що взаємодіють між собою. Переважно ООП застосовується при розробці прикладного програмного забезпечення. Основу ООП складають три основні концепції: інкапсуляція, успадкування та поліморфізм. Одною з переваг ООП перед процедурним програмуванням є краща модульність програмного забезпечення (тисячу функцій процедурної мови, в ООП можна замінити кількома десятками класів із своїми методами). Попри те, що ця парадигма з'явилась в 1960-тих роках, вона не мала широкого застосування до 1990-тих, коли розвиток комп'ютерів та комп'ютерних мереж дозволив писати надзвичайно об'ємне і складне програмне забезпечення, що змусило переглянути підходи до написання програм. Сьогодні багато мов програмування або підтримують ООП або ж є цілком об'єктно-орієнтованими (зокрема, C++, Java, C#, Python, PHP, Ruby, Objective-C та інші).

Цикл лабораторних робіт укладений відповідно до навчальної програми “Програмування, частина 2 (Об'єктно-орієнтоване програмування)” та містить вісім робіт. Вони орієнтовані на освоєння парадигми об'єктно-орієнтованого програмування на прикладі мови C++. Мова C++ широко використовується при розробці програмного забезпечення для спеціалізованих комп'ютерних систем на базі програмованих мікроконтролерів, а також для розробки високопродуктивних фрагментів коду для програм, що створюються за допомогою кросплатформених мов програмування, таких як C#, Java, Python тощо.

Лабораторні роботи орієнтовані на наявну на кафедрі ЕОМ лабораторну базу. Методичні вказівки допоможуть студентам глибше зрозуміти парадигму об'єктно-орієнтованого програмування та ефективно використовувати відведений на виконання лабораторних робіт аудиторний час.

ЛАБОРАТОРНА РОБОТА №1. ПЕРЕВАНТАЖЕННЯ ФУНКЦІЙ

Мета: познайомитися із перевантаженням функцій.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Перевантаження функцій

У мові C++, на відміну від мови C дозволяється визначати декілька функцій з одним і тим же іменем за умови, що дані функції мають різну сигнатуру (різні типи та кількість аргументів функції). При цьому тип значення що повертається при перевантаженні до уваги не приймається. Розглянемо приклади перевантажених функцій:

```
int func(int, int);  
int func(char, double);  
int func(float, ...);    // Функція з невизначеним числом  
                        аргументів.  
int func(char*, int);  
int func(long, double);  
long func(long, double); // не є перевантаженою функцією,  
                        бо від попередньої відрізняється  
                        лише типом значення, що  
                        повертається
```

Розглянемо дії компілятора коли він зустрічає при компіляції в коді програми виклики перевантажених функцій.

При виклику функції з ім'ям `func` спершу компілятор намагатиметься знайти функцію, формальні аргументи якої відповідають фактичним аргументам без усяких перетворень типів або з використанням тільки немінучих перетворень - наприклад, імені масиву до покажчика або значення змінної до константи або навпаки.

```
char string[ ] = "Рядок - це масив символів";  
int i = func(string, 13);    // func(char*, int);  
int j = func(1995L, 36.6);  // func(long, double);
```

Якщо відповідна функція не знайдена, то здійснюється пошук такої функції, щоб для відповідності формальних і фактичних аргументів досить було використати тільки такі стандартні перетворення, що не спричиняють перетворень цілих типів до типів з плаваючою крапкою і навпаки. При цьому підбирається функція, для якої число таких перетворень було б мінімальним.

```
float a=36.6;  
j = func('a', a);  // func(char, double)
```

Третім етапом є пошук такої функції, для виклику якої досить здійснити будь-які стандартні перетворення аргументів (і знову так, щоб цих перетворень було якнайменше).

```
int k = func("PIK:", 2015.3);  // func (char*, int)
```

Далі здійснюється пошук функції, для якої аргументи можна одержати за допомогою всіх перетворень, розглянутих до цього, а також перетворень типів, визначених самим програмістом. Якщо й у цьому випадку єдина потрібна функція не знайдена, то на останньому етапі компілятор пробує знайти відповідність з урахуванням списку невизначених аргументів. Так, виклик функції `func (1, 2, 3)` може бути співставлений лише з функцією, що оголошена як `int func(float, ...)`.

Якщо компілятор не знайде жодної підходящої функції, або виклик функції не може бути однозначно співставлений з однією з оголошених функцій, то програма не скомпілюється і буде виведено повідомлення про помилку.

Зверніть увагу, що співставлення викликів функцій з оголошеними в програмі функціями відбувається на етапі компіляції, а не в процесі виконання програми.

Вбудовані (inline) функції

У мові C директива препроцесора `#define` використовується для визначення констант та макросів (макрОВизначень). Наприклад, директива

`#define sqr(x) ((x)*(x))` оголошує макрос, що дозволяє обчислювати квадрат від фактичного аргументу, який підставляється замість формального аргументу `x`:

```
#define k 5          // оголошення константи k рівної 5
#define sqr(x) ((x)*(x)) // макровизначення
...
void main()
{
    int a, i = 5;
    a = sqr(i);      // макрокоманда (виклик макросу)
}
```

У даному випадку скрізь у тексті програми під час компіляції замість `sqr(i)` буде підставлено вираз `((i)*(i))`:

```
void main()
{
    int a, i = 5;
    a = ((i)*(i)); // макророзширення
}
```

Слід зазначити, що аргумент при використанні макровизначення може бути будь-яким (змінною будь-якого значущого типу, константою, функцією, що повертає значення, іншим макросом). Це дуже схоже на виклик функції, але насправді відбувається лише текстова заміна. З цієї причини макроси працюють швидше функцій, оскільки при їх використанні не витрачаються ресурси на виклик функції. Але по цій же причині вони є потенційним джерелом помилок. У випадку `sqr(i++)` змінна `i` інкрементуватиметься не один, а два рази:

```
void main()
{
    int a, i = 5;
    a = ((i++)*(i++)); // макророзширення
}
```

Мова C++ пропонує безпечну заміну макросам – вбудовані (inline) функції. На відміну від макросів, вбудовані функції не піддаються помилкам подвійного обчислення. Крім того типи аргументів перевіряються компілятором і при потребі виконуються всі необхідні перетворення. Таким чином, якщо у вас є маленька функція (кілька рядків коду), яку доводиться часто викликати, то її можна оголосити як `inline`. Оголошена в такий спосіб функція буде проаналізована компілятором на можливість її реалізації у вигляді вбудованої функції. Якщо компілятор вважатиме доцільним реалізувати її як вбудовану, то дана функція не буде викликатися. Замість цього тіло функції підставлятиметься в те місце програми, де здійснюється виклик. При цьому підвищується ефективність програми ціною збільшення розміру коду програми. Якщо ж компілятор вважатиме, що дана функція завелика для її ефективної реалізації як вбудованої, то вона буде реалізована як звичайна функція. Таким чином кінцеве рішення у питанні робити функцію позначену як `inline` вбудованою чи ні належить компілятору.

Щоб оголосити функцію як `inline` необхідно просто поставити ключове слово `inline` перед оголошенням функції:

```
inline int func (int a, int b);
```

Класи пам'яті

<i>Клас пам'яті</i>	<i>Призначення</i>
auto	Автоматичний клас пам'яті. Зона дії автоматичної змінної обмежена блоком коду у фігурних дужках або функцією, де вона описана. Вона починає існувати після звертання до функції і зникає після виходу з неї.
external	Змінні з зовнішнім класом пам'яті - це глобальні змінні і до них можна звертатися з будь-якої функції. Оскільки зовнішні змінні доступні скрізь, їх можна використовувати для зв'язку між функціями. Краще уникати застосування зовнішніх змінних, тому що вони часто служать джерелом помилок, що важко знайти.
static	Змінні із статичним класом пам'яті, подібно автоматичним, локальні в тій функції або блоці, де вони описані, а також можуть бути доступні глобально у всій програмі. Різниця з автоматичними змінними полягає в тому, що статичні змінні не зникають, коли функція

	(блок) завершує роботу, і їхні значення зберігаються для наступних викликів функції. Крім цього статична змінна створюється і ініціалізується лише один раз за час роботи програми, та існує доти, доки програма виконується.
register	<i>Регістрові</i> змінні повинні зберігатися в надшвидкій пам'яті ЕОМ - регістрах. Використовуються аналогічно автоматичним змінним. Якщо компілятор в процесі компіляції не зможе з тих чи інших причин розмістити дану змінну в регістрі комп'ютера, тоді він її реалізує як звичайну автоматичну змінну.

Час життя й область видимості програмних об'єктів

Час життя змінної визначається за наступними правилами:

1. Змінна, оголошена глобально (тобто поза всіма блоками), існує протягом усього часу виконання програми.
2. Локальні змінні (тобто оголошені всередині блоку) із класом пам'яті register або auto, мають час життя тільки на період виконання того блоку, в якому вони оголошені. Якщо локальна змінна оголошена з класом пам'яті static або extern, то вона має час життя на період виконання всієї програми.

Видимість змінних у програмі визначається наступними правилами:

1. Змінна, оголошена або визначена глобально, видима від моменту оголошення або визначення до кінця файлу. Для того, щоб змінна була видима й в інших файлах, необхідно оголосити її з класом пам'яті extern.
2. Змінна, оголошена або визначена локально, видима від моменту оголошення або визначення до кінця поточного блоку.
3. Змінна, оголошена або визначена локально, а також змінна оголошена на глобальному рівні, видимі в усіх внутрішніх блоках. Якщо змінна, оголошена всередині блоку, має те ж ім'я, що і змінна, що оголошена в зовнішньому блоці, що включає даний блок, то змінна із зовнішнього блоку у внутрішньому блоці буде невидимою. Замість неї буде видима змінна, що оголошена у блоці.

Ініціалізація глобальних і локальних змінних:

1. Глобальні змінні завжди ініціалізуються, і якщо це не зроблено явно, то вони ініціалізуються нульовим значенням.

2. Змінна з класом пам'яті `static` може бути ініціалізована константним значенням. Ініціалізація для них виконується один раз перед початком програми. Якщо явна ініціалізація відсутня, то змінна ініціалізується нульовим значенням.

3. Ініціалізація змінних із класом пам'яті `auto` або `register` виконується кожен раз при вході в блок, у якому вони оголошені. Якщо ініціалізація змінних при оголошенні відсутня, то їхнє початкове значення не визначене.

4. Початковими значеннями для глобальних змінних і для змінних із класом пам'яті `static` повинні бути константні значення.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке перевантаження функцій?
2. Як відбувається перевантаження функцій?
3. На якому етапі визначається яку саме з перевантажених функцій слід викликати в конкретному місці програми?
4. Що таке вбудовані функції?
5. Що таке макрос?
6. В чому полягає відмінність між макросами і вбудованими функціями?
7. Як оголосити вбудовану функцію?
8. Чи завжди вбудовані функції будуть реалізовані як вбудовані і від чого це залежить?
9. Які класи пам'яті ви знаєте і в чому між ними різниця?
10. Час життя, область видимості і ініціалізація змінних різних класів пам'яті.

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.

ЗАВДАННЯ

В ході виконання завдання при оголошенні змінних використати різні класи пам'яті.

Зверніть увагу, що у рядків мови C++ ознакою кінця рядка є символ з ASCII кодом 0.

Варіанти завдань

1. Написати реалізацію перевантажених функцій :
 int func(int* arr, int length);
 double func(double * arr, int length);
 Функція func здійснює пошук максимального елемента масиву.
2. Написати реалізацію перевантажених функцій:
 long func(long* arr, int length);
 float func(float* arr, int length);
 Функція func здійснює пошук мінімального елемента масиву.
3. Написати реалізацію перевантажених функцій:
 int func(int* arr, int length, int number);
 int func(char* str, char ch);
 Функція func повертає позицію першого входження шуканого числа чи символу.
4. Написати реалізацію перевантажених функцій:
 bool func(int* arr, int length);
 bool func(char* str);
 Функція func перевіряє чи елементи масиву розташовані по зростанню.
5. Написати реалізацію перевантажених функцій:
 bool func(double* arr, int length);
 bool func(char* str);
 Функція func перевіряє чи елементи масиву розташовані по спаданню.
6. Написати реалізацію перевантажених функцій:
 int func(double* arr, int length);
 int func(char* str);
 Функція func переставляє всі елементи у зворотному порядку.
7. Написати реалізацію перевантажених функцій:
 int func(int* arr, int length, int number);
 int func(char* str, char ch);
 Функція func повертає кількість входжень числа чи символу в масив.
8. Написати реалізацію перевантажених функцій:
 int func(int* arr, int length, int number);

```
int func(char* str, char ch);
```

Функція func повертає позицію останнього входження шуканого числа чи символу.

9. Написати реалізацію перевантажених функцій:

```
int func(double* arr, int length);
```

```
int func(char* str);
```

Функція func повертає кількість елементів, які більші за своїх сусідів.

10. Написати реалізацію перевантажених функцій :

```
double func(int* arr, int length);
```

```
double func(double * arr, int length);
```

Функція func повертає середнє арифметичне елементів послідовності.

11. Написати реалізацію перевантажених функцій :

```
bool func(int* arr, int length);
```

```
bool func(double * arr, int length);
```

Функція func перевіряє чи в масиві чергуються знаки.

12. Написати реалізацію перевантажених функцій :

```
int func(int* arr, int length);
```

```
int func(double * arr, int length);
```

Функція func повертає кількість парних елементів послідовності.

13. Написати реалізацію перевантажених функцій :

```
int func(int* arr, int length);
```

```
int func(double * arr, int length);
```

Функція func повертає кількість непарних елементів послідовності.

14. Написати реалізацію перевантажених функцій:

```
long func(long* arr, int length);
```

```
float func(float* arr, int length);
```

Функція func здійснює пошук мінімального парного елемента масиву.

15. Написати реалізацію перевантажених функцій:

```
long func(long* arr, int length);
```

```
float func(float* arr, int length);
```

Функція func здійснює пошук мінімального непарного елемента масиву.

16.Написати реалізацію перевантажених функцій:

```
int func(int* arr, int length, int number);  
int func(char* str, char ch);
```

Функція func повертає позицію останнього входження шуканого числа чи символу, якщо воно більше, ніж середнє арифметичне значень масиву, інакше повертає -1.

17.Написати реалізацію перевантажених функцій:

```
bool func(int* arr, int length);  
bool func(char* str);
```

Функція func перевіряє чи елементи масиву між мінімальним і максимальним розташовані по зростанню.

18.Написати реалізацію перевантажених функцій:

```
bool func(double* arr, int length);  
bool func(char* str);
```

Функція func перевіряє чи елементи масиву між мінімальним і максимальним розташовані по спаданню.

19.Написати реалізацію перевантажених функцій:

```
int func(double* arr, int length);  
int func(char* str);
```

Функція func переставляє елементи між мінімальним і максимальним у зворотному порядку.

20.Написати реалізацію перевантажених функцій:

```
int func(int* arr, int length, int number);  
int func(char* str, char ch);
```

Функція func повертає кількість входжень числа чи символу в масив, що стоять між мінімальним і максимальним елементами масиву.

21.Написати реалізацію перевантажених функцій:

```
int func(int* arr, int length, int number);  
int func(char* str, char ch);
```

Функція func повертає позицію останнього входження шуканого числа чи символу, що стоять між мінімальним і максимальним елементами масиву.

22.Написати реалізацію перевантажених функцій:

```
int func(double* arr, int length);  
int func(char* str);
```

Функція func повертає кількість елементів, які більші за своїх сусідів, що стоять між мінімальним і максимальним елементами масиву.

23.Написати реалізацію перевантажених функцій:

```
double func(int* arr, int length);  
double func(double * arr, int length);
```

Функція func повертає середнє арифметичне елементів послідовності, що стоять між мінімальним і максимальним елементами масиву.

24.Написати реалізацію перевантажених функцій:

```
int func(int* arr, int length);  
int func(double * arr, int length);
```

Функція func перевіряє чи в масиві між мінімальним і максимальним елементами чергуються знаки.

25.Написати реалізацію перевантажених функцій:

```
long func(long* arr, int length);  
double func(double * arr, int length);
```

Функція func повертає середнє геометричне елементів масиву.

26.Написати реалізацію перевантажених функцій :

```
int func(int* arr, int length);  
int func(double * arr, int length);
```

Функція func здійснює підрахунок кількості парних елементів менших 0.

27.Написати реалізацію перевантажених функцій:

```
int func(long* arr, int length);  
int func(float* arr, int length);
```

Функція func здійснює підрахунок кількості непарних елементів менших 0.

28.Написати реалізацію перевантажених функцій :

```
int func(int* arr, int length);  
int func(double * arr, int length);
```

Функція func здійснює підрахунок кількості парних елементів більших 0.

29. Написати реалізацію перевантажених функцій:

```
int func(long* arr, int length);
```

```
int func(float* arr, int length);
```

Функція func здійснює підрахунок кількості непарних елементів більших 0.

30. Написати реалізацію перевантажених функцій:

```
int func(double* arr, int length);
```

```
int func(char* str);
```

Функція func перевіряє чи парні і непарні елементи масиву чергуються.

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application. Якщо ви використовуєте MS Visual Studio версії 2012 або вище, то для того, щоб використовувати ключове слово auto для встановлення автоматичного класу пам'яті, а не для автоматичного виведення типу змінної, в проекті слід в властивостях проекту на вкладці «Властивості конфігурації» -> «C/C++» -> «Командний рядок» у вікні «Додаткові параметри» додати параметр /Zc:auto-.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

```
// У програмі здійснюється пошук різниці між першим і останнім елементами масивів
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int func(int* a, int size)
```

```
{
```

```

    if (size > 1)
        return (a[0] - *(a + size - 1));
    else if (size == 1)
        return (*(a) - a[0]);
    else
        return 0;
}

int func (char* a)
{
    register int len = strlen(a);
    if (len > 1)
        return (a[0] - *(a + len - 1));
    else if (len == 1)
        return (*(a) - a[0]);
    else
        return 0;
}

int main()
{
    int res1 = 0;
    auto int res2 = 0;
    static int arr[5] = {1, 2, 3, 4, 5};
    char str[] = "ABCDE";
    res1 = func(arr, 5);
    res2 = func(str);
    printf("Result 1 = %d\n", res1);
    printf("Result 2 = %d\n", res2);
    return 0;
}

```

Результат виконання програми:

```

Result 1 = -4
Result 2 = -4

```


ЛАБОРАТОРНА РОБОТА № 2. ПОТОКОВИЙ ВВІД-ВИВІД

Мета: познайомитися із потоковим вводом-виводом.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Простори імен

У мові програмування C++ так само як і у багатьох об'єктно-орієнтованих мовах програмування існує поняття просторів імен, що розділяються, на відміну від мови C в якій існує один глобальний простір імен. Простори імен служать для об'єднання класів, що написані різними програмістами або мають схоже призначення в групі. Завдяки цьому з'являється можливість усувати неоднозначності, що пов'язані з використанням класів, що мають однакові імена, але різне призначення. Також простори імен дозволяють безпечно підключати бібліотеки класів не боячись співпадінь у назвах існуючих і підключених класів. Все, що оголошене в межах простору імен є видимим лише в його межах.

Для оголошення простору імен слід вжити ключове слово `namespace` після якого слід вказати назву простору імен. Після цього в фігурних дужках визначається його тіло. Тіло простору імен може містити як класи і структури, так і функції зі змінними та константами, а також вкладені простори імен. Приклад просторів імен:

```
namespace Student
{
    char* pszName;
    int* pMarks;
    double averageMark(int* Marks, int size);
}
namespace Teacher
{
    char* pszName;
    double averageMark;
}
```

Простори імен є відкритими. Це означає, що можна в будь-якому місці програми оголосити простір імен з існуючою назвою, додати в нього нові елементи і при компіляції ці простори імен об'єднуються в один, що міститиме існуючі і нововведені елементи.

Простір імен може бути безіменним, якщо його ім'я при оголошенні не вказується. Безіменні простори імен можна використовувати для оголошення глобальних статичних змінних з внутрішнім зв'язуванням, оскільки згідно стандарту ключове слово `static` є застарілим.

Для того, щоб звернутися до елементу з простору імен слід вказати назву простору імен, *оператор розширення області видимості* (`::`) та здійснити відповідне звернення до елементу простору імен:

```
Student :: averageMark(arr, size);  
  
Teacher :: averageMark = 5.2;
```

Щоб постійно не вказувати простори імен при зверненні до їх елементів, їх можна підключити до програми за допомогою ключового слова `using` частково або повністю. Для підключення окремих елементів простору імен використовується `using` оголошення. Для підключення всього простору імен з використовується `using` директива.

Синтаксис підключення окремих елементів простору імен з використанням `using` оголошення:

```
using назваПросторуІмен :: назваЕлементу;
```

```
using Student :: averageMark;  
using Teacher :: pszName;
```

Тепер до функції `averageMark` з простору імен `Student` і змінної `pszName` з простору імен `Teacher` можна звертатися звичним чином:

```
averageMark(arr, size);  
  
pszName = "Микола Григорович";
```

Синтаксис підключення всього простору імен з використанням `using` директиви:

```
using namespace назваПросторуІмен;
```

```
using namespace Student;
```

При підключенні всього простору імен ми можемо звичним чином звертатися до всіх його елементів. Проте, при підключенні кількох просторів імен, що мають елементи з однаковою назвою до яких відбувається звертання, відбудеться конфлікт імен. Для його усунення слід явно вказувати до якого простору імен належить елемент до якого іде звертання.

Простори імен можуть бути вкладеними. Оголошення вкладених просторів імен відбувається шляхом визначення простору імен в просторі імен:

```
namespace NSpace1
{
    ...
    namespace NSpace2
    {
        ...
        int value;
    }
}
```

Якщо простори імен є вкладеними, то слід вказати повний шлях до елемента розмежовуючи простори імен оператором `::`, наприклад:

```
NSpace1 :: NSpace2 :: value = 5;
```

Аналогічно відбувається підключення просторів імен та їх елементів за допомогою ключового слова `using`.

Потоки вводу-виводу в C++

Система вводу-виводу в стандартній бібліотеці C++ реалізована у вигляді потоків. Потік вводу-виводу – це логічний пристрій, який приймає та видає інформацію користувача. Кожен потік зв'язаний з фізичним пристроєм (клавіатура, монітор) або з файлом. Бібліотека потоків `iostream` реалізована

як ієрархія класів та забезпечує широкі можливості для виконання операцій вводу-виводу. Далі наведено призначення деяких класів потокового вводу-виводу:

- `istream` – підтримує операції по вводу;
- `ostream` – підтримує операції по виводу;
- `iostream` – підтримує операції по вводу-виводу;
- `ifstream` – підтримує операції по вводу з файлу;
- `ofstream` – підтримує операції по виводу у файл;
- `fstream` – підтримує операції з файлами по вводу-виводу.

Стандартні потоки

Коли запускається програма на C++, автоматично створюються чотири об'єкти, що реалізують стандартні потоки.

Таблиця 2.1. Об'єкти стандартних потоків

Потік	Призначення	Пристрій по замовчуванню
<code>cin</code>	Стандартний ввід	Клавіатура
<code>cout</code>	Стандартний вивід	Монітор
<code>cerr</code>	Стандартний вивід повідомлень про помилки	Монітор
<code>clog</code>	Стандартний вивід повідомлень про помилки (буферизований)	Монітор

Щоб мати можливість використовувати стандартні потоки необхідно підключити заголовочний файл `iostream` або `iostream.h`. Загалом різниця між стандартними заголовочними файлами з розширенням `*.h` і без нього полягає в тому, що файли з розширенням `*.h` відносяться до мови C, а без розширення – до C++. Таким чином програмуючи на мові C++ безпечніше використовувати заголовочні файли без розширення `*.h`, які орієнтовані на мову C++. Проте в цьому випадку може бути необхідним підключати додатково простори імен. При використанні стандартних бібліотек вводу-виводу таким простором імен є `std`.

Об'єкт стандартного потоку вводу `cin` класу `istream`, зв'язаний із стандартним пристроєм вводу, за звичай клавіатурою. Об'єкт стандартного потоку виводу `cout` класу `ostream`, зв'язаний із стандартним пристроєм виводу, за звичай монітором. Об'єкт `cerr` класу `ostream`, зв'язаний із

стандартним пристроєм виводу повідомлень про помилки. Потоки даних, що виводяться, для об'єкту `cerr` являються небуферизованими. Тобто кожна операція помістити в `cerr` приводить до миттєвої появи повідомлень про помилки. Об'єкт `clog` класу `ostream`, зв'язаний із стандартним пристроєм виводу повідомлень про помилки. Потоки даних, що виводяться, для об'єкту `clog` являються буферизованими. Тобто кожна операція помістити в `clog` може привести до того, що вивід буде зберігатися в буфері до тих пір, поки буфер повністю не заповниться або ж поки вмістиме буферу не буде виведене примусово.

Вивід в потік виконується за допомогою операції «помістити в потік», а саме перевантаженої операції `<<`. Дана операція перевантажена для виводу елементів даних стандартних типів, для виводу рядків та значень вказівників.

Операція `<<` повертає посилання на об'єкт типу `ostream`, для якого вона викликана. Це дозволяє будувати ланцюжок викликів операції «помістити в потік», що виконуються зліва направо.

```
int i = 5;
double d = 2.08;
cout << "i = " << i << ", d = " << d << '\n';
```

Ці оператори приведуть до виведення на екран наступного рядка:
`i = 5, d = 2.08`

Ввід потоку виконується за допомогою операції «взяти із потоку», а саме перевантаженої операції `>>`. Дана операція зазвичай ігнорує у вхідному потоці символи розділювачі та пробільні символи. Операція «взяти із потоку» повертає нульове значення (`false`), якщо зустрічає в потоці признак кінця файлу або виникає помилка при спробі читання із потоку.

Засоби форматування потоку

Система вводу-виводу дозволяє виконувати форматування даних та змінювати визначені параметри вводу інформації. Дані операції реалізовані за допомогою функцій форматування, прапорців та маніпуляторів.

Функції форматування та їх призначення приведені у табл. 2.2.

Таблиця 2.2. Функції форматування

Функція-член	Призначення
<code>width(int wide)</code>	Дозволяє задати мінімальну ширину поля для виведення значення. При вводі задає максимальне число символів, що читаються. Якщо значення, що виводиться, має менше символів, ніж задана ширина поля, то воно доповнюється символами-заповнювачами до заданої ширини (за замовчуванням - пробілами). Якщо ж значення, що виводиться має більше символів, ніж ширина відведеного йому поля, то поле буде розширене.
<code>precision(int prec)</code>	Дозволяє прочитати або встановити точність (число цифр після десяткової крапки), з якою виводяться числа з рухомою крапкою. По замовчуванню числа з рухомою крапкою виводяться з точністю, рівною шести цифрам.
<code>fill(char ch)</code>	Дозволяє прочитати або встановити символ-заповнювач.

```

void main()
{
    double x;
    cout.precision(4);
    cout.fill('0');
    cout << " x      sqrt(x)\n";
    for (x = 1.0; x <= 6.0; x++)
    {
        cout.width(7);
        cout << x << ' ';
        cout.width(7);
        cout << sqrt(x) << '\n';
    }
}

```

Результат роботи програми наступний:

```

x      sqrt(x)
0000001 0000001

```

0000002	001.414
0000003	001.732
0000004	0000002
0000005	002.236
0000006	002.449

З кожним потоком зв'язаний набір прапорців, що керують форматуванням потоку. Вони являють собою бітові маски. Прапорці форматування і їхнє призначення приведені в табл. 2.3. Встановити значення одного або кількох прапорців можна за допомогою функції-члену `setf(long mask)`.

Таблиця 2.3. Прапорці форматування і їхнє призначення

Прапорець	Призначення
dec	Встановлюється десяткова система числення
hex	Встановлюється шістнадцяткова система числення
oct	Встановлюється вісімкова система числення
scientific	Числа з рухомою крапкою, виводяться в науковому записі (тобто <code>n.xxxEyy</code>)
showbase	Виводиться основа системи числення у виді префікса до цілого числового значення (наприклад, шістнадцяткове число <code>1FE</code> виводиться як <code>0x1FE</code>)
showpos	При виводі позитивних числових значень виводиться знак плюс
uppercase	Замінюються визначені символи нижнього регістра на символи верхнього регістра (символ "e" при виведенні чисел в науковому записі – на "E" і символ "x" при виведенні чисел в шістнадцятковій системі – на "X")
left	Дані при виведенні вирівнюються по лівому краю поля виводу
right	Дані при виведенні вирівнюються по правому краю поля виводу
internal	Додаються символи-заповнювачі між усіма цифрами і знаками числа для заповнення поля виводу
skipws	Ведучі символи-заповнювачі (знаки пробілу, табуляції і переходу на новий рядок) відкидаються

```

void main()
{
    double d = 3.124e7;
    int n = 25;
    cout << "d = " << d << '\n' ;
    cout << "n = " << n << '\n';
    cout.setf(ios::hex | ios::uppercase);
    cout.setf(ios::showpos);
    cout << "d = " << d << '\n' ;
    cout << "n = " << n << '\n';
}

```

Результат роботи програми наступний:

```

d = 3.124e+007
n = 25
d = +3.124E+007
n = 19

```

Список маніпуляторів та їхнє призначення приведені в табл. 2.4. Маніпулятори вводу-виводу являють собою вид функцій-членів класу `ios`, що, на відміну від звичайних функцій-членів, можуть розташовуватися усередині операцій вводу-виводу.

За винятком функції-члену `setw(int n)`, усі зміни в потоці, внесені маніпулятором, зберігаються до наступної установки.

Для доступу до маніпуляторів з параметрами необхідно включити в програму стандартний заголовний файл `iomanip`.

Таблиця 2.4. Маніпулятори вводу-виводу

Маніпулятор	Призначення
<code>endl</code>	Виводить символ нового рядка та очищує потік
<code>flush</code>	Видає вміст буфера потоку у пристрій
<code>dec</code>	Встановлює десяткову систему числення
<code>hex</code>	Встановлює шістнадцяткову систему числення
<code>oct</code>	Встановлює вісімкову систему числення
<code>setbase (int base)</code>	Задає основу системи числення для цілих чисел (8, 10, 16)
<code>setfill (int c)</code>	Встановлює символ-заповнювач

setprecision (int n)	Встановлює точність чисел з рухомою крапкою
setw(int n)	Встановлює мінімальну ширину поля виводу
setiosflags (iosbase::long mask)	Встановлює ios-прапорці згідно з mask

```
void main()
{
    double x = 45.12345;
    cout << "x = " << setprecision(4)
          << setfill('0') << setw(7)
          << x << endl;
}
Результат роботи програми наступний:
x = 0045.12
```

Часто застосовувані функції

Крім вже описаних функцій, бібліотека вводу-виводу C++ містить широкий набір інших функцій. Тут ми приведемо лише деякі часто вживані з них.

Для читання символу з потоку можна використовувати функцію-член `get()` потоку `istream`. Функція `get()` повертає код прочитаного символу або `-1`, якщо зустрівся кінець файлу вводу (`ctrl/z`).

Функція `get(char* str, int len, char delim)` може також використовуватися для читання рядка символів. У цьому випадку використовується її варіант, у якому ця функція читає з вхідного потоку символи в буфер `str`, поки не зустрінеться символ-обмежувач `delim` (за замовчуванням – `\n`) або не буде прочитано `(len-1)` символів чи ознаку кінця файлу. Сам символ-обмежувач не читається з вхідного потоку.

Для вставки символу в потік виведення використовується функція-член `put(char ch)`.

Через те, що функція `get()` не читає з вхідного потоку символ-обмежувач, вона використовується рідко. Набагато частіше використовується функція `getline(char* str, int len, char delim)`, що читає з вхідного потоку символ-обмежувач, але не поміщає його в буфер.

Функція `gcount()` повертає число символів, прочитаних з потоку останньою операцією неформатуючого вводу (тобто функцією `get()`, `getline()` або `read()`).

Розглянемо приклад, у якому використовуються дві останні функції:

```
void main(void)
{
    const len = 100;
    char name[len];
    int count = 0;

    cout << "Enter your name" << endl;
    cin.getline(name, len);
    count = cin.gcount();
    /* Зменшуємо значення лічильника на 1, тому що getline() не
    поміщає обмежувач в буфер*/
    cout << "Number of symbols is "
        << count - 1 << endl;
}
```

Результат роботи програми наступний:

```
Enter your name
Petro
Number of symbols is 5
```

Для того, щоб пропустити при введенні кілька символів, використовується функція `ignore(int n = 1, int delim = EOF)`. Ця функція ігнорує `n` символів у вхідному потоці. Пропуск символів припиняється, якщо вона зустрічає символ-обмежувач, яким по замовчуванню є символом кінця файлу. Символ-обмежувач читається з вхідного потоку.

Функція `peek()` дозволяє "заглянути" у вхідний потік і довідатися наступний символ, що вводиться. При цьому сам символ з потоку не читається.

За допомогою функції `putback(char ch)` можна повернути символ `ch` у потік вводу.

Файловий ввід-вивід

Робота з файлами в мові C++ як і у мові C передбачає 3 етапи: відкривання файлу (файлового потоку), обмін даними з файловим потоком, закривання файлового потоку.

Для виконання операцій з файлами в мові C++ передбачено три класи: `ifstream`, `ofstream` і `fstream`. Ці класи є похідними від класів `istream`, `ostream` і `iostream`. Всі функціональні можливості (перевантажені операції `<<` та `>>` для вбудованих типів, функції і прапорці форматування, маніпулятори й ін.), що застосовуються до стандартного вводу та виводу, можуть застосовуватися і до файлів. Існує деяка відмінність між використанням стандартних та файлових потоків. Стандартні потоки можуть використовуватися відразу після запуску програми, тоді як файловий потік спочатку слід зв'язати з файлом. Для реалізації файлового вводу-виводу потрібно підключити заголовочний файл `fstream`, що знаходиться в просторі імен `std`.

Відкрити файл для вводу чи виводу можна наступним чином:

```
// Для виводу
ofstream outfile;
outfile.open("File.txt");
або
ofstream outfile("File.txt");
або
fstream outfile("File.txt", ios::out);
// Для вводу
ifstream infile;
infile.open("File.txt");
або
ifstream infile("File.txt");
або
fstream infile("File.txt", ios::in);
```

Режими відкриття файлів та їхнє призначення наведені у табл. 2.5.

Таблиця 2.5. Режими відкриття файлів

Режим відкриття	Призначення
<code>ios::in</code>	Відкрити файл для читання
<code>ios::out</code>	Відкрити файл для запису
<code>ios::ate</code>	Відкрити файл для додавання в кінець
<code>ios::app</code>	Відкрити файл для додавання в кінець
<code>ios::trunc</code>	Усікти файл, тобто видалити його вміст
<code>ios::binary</code>	Відкрити файл у двійковому режимі

Режими відкриття файлу являють собою бітові маски, тому можна задавати два або більш режими, поєднуючи їх побітовою операцією АБО. Слід звернути увагу, що по замовчуванню режим відкриття файлу відповідає типові файлового потоку. У потоці вводу або виводу прапорець режиму завжди встановлений неявно.

Між режимами відкриття файлу `ios::ate` та `ios::app` існує певна відмінність. Якщо файл відкривається в режимі додавання (`ios::app`), весь вивід у файл буде здійснюватися в позицію, що починається з поточного кінця файлу, безвідносно до операцій позиціонування у файлі. У режимі відкриття `ios::ate` (від англійського "at end") можна змінити позицію виводу у файл і здійснювати запис, починаючи з неї. Файли, які відкриваються для виводу, створюються, якщо вони ще не існують.

Якщо при відкритті файлу не зазначений режим `ios::binary`, файл відкривається в текстовому режимі.

Якщо відкриття файлу завершилося невдачею, об'єкт, що відповідає потокові, буде повертати нуль. Перевірити успішність відкриття файлу можна також за допомогою функції-члена `is_open()`. Дана функція повертає 1, якщо потік вдалося зв'язати з відкритим файлом.

Для перевірки, чи досягнутий кінець файлу, можна використовувати функцію `eof()`. Завершивши операції вводу-виводу, необхідно закрити файл, викликавши функцію-член `close()`.

Далі наведений приклад, що демонструє файловий ввід-вивід з використанням потоків.

```
#include <fstream>
using namespace std;
int main( )
{
    int n = 50;
```

```

// Відкриваємо файл для виводу
ofstream ofile("Test.txt");
if (!ofile)
{
    cout << "Файл не відкритий. \n";
    return -1;
}
ofile << "Hello!\n" << n;
// Закриваємо файл
ofile.close();

// Відкриваємо той же файл для вводу
ifstream ifile("Test.txt");

if ( !ifile )
{
    cout << "Файл не відкритий.\n";
    return -1;
}
char str[80];
ifile >> str >> n;
cout << str << " " << n << endl;
ifile.close(); // Закриваємо файл

return 0;
}

```

Меню

Більшість прикладних програм передбачає інтерактивну взаємодію користувача з комп'ютером. Для її реалізації з метою керування процесом роботи консольної програми часто використовується меню. Воно передбачає вивід на екран варіантів функціонування програми (включно з варіантом, що передбачає вихід з програми) з подальшим вибором одного з них користувачем. Один з алгоритмів реалізації меню має наступний вигляд:

```

while (true)
{
    вивід <- вивести варіанти роботи
    ввід -> варіант роботи
    if (варіант роботи 1)
    {
        Функціонування програми згідно варіанту роботи 1
    }
    ...
    if (варіант роботи N)
    {
        Функціонування програми згідно варіанту роботи N
    }

    if (вихід)
    {
        break;    //Переривання вічного циклу
    }
}

```

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке простір імен?
2. using оголошення і директиви.
3. Які способи доступитися до елементів простору імен ви знаєте?
4. Потоки вводу-виводу в C++?
5. Що таке маніпулятор?
6. Які прапорці форматування виводу ви знаєте?
7. Які шляхи здійснення форматування виводу ви знаєте?
8. Особливості файлового вводу-виводу.
9. Класи файлового вводу-виводу і їх призначення.
10. Режими відкриття файлів і їх відмінності.

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.

2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.

ЗАВДАННЯ

Написати програму, яка буде додавати у текстовий файл введену з клавіатури інформацію (згідно варіанту). Слід передбачити можливість вибору користувачем режиму роботи: додавання чи відображення даних (меню). Забезпечити зберігання даних у файлі у вигляді структурованої таблиці за допомогою засобів форматування. При виводі на екран має відображатися шапка з інформацією в якій колонці які дані відображаються. У файлі має зберігатися лише структурована інформація без шапки.

Варіант	Завдання
1	З клавіатури вводиться прізвище, адреса та номер телефону, а у файл записується прізвище, адреса, номер телефону та сума всіх цифр номеру телефону
2	З клавіатури вводиться ціле число, у файл записується його 10-ткове, 16-ткове та 8-кове заокруглене до десятків значення
3	З клавіатури вводиться ім'я, адреса, вартість і рік заснування компанії
4	З клавіатури вводиться ім'я, день і місяць народження, а у файл записується ім'я, день і місяць народження та добуток дня і місяця народження
5	З клавіатури вводиться ціле число, у файл записується саме число та значення квадратного кореня з даного числа
6	З клавіатури вводиться число з рухомою крапкою, у файл записується саме число та його заокруглене значення
7	З клавіатури вводиться назва, кількість та ціна за одиницю товару, а у файл виводиться назва, кількість, ціна за одиницю та сумарна вартість товару
8	З клавіатури вводиться прізвище, ім'я і рік народження

9	З клавіатури вводиться прізвище, ім'я і розмір депозиту, а у файл виводиться прізвище, ім'я і розмір депозиту за 1, 3, 6 і 12 місяці з урахуванням ставки по депозитам на рівні 4% на місяць
10	З клавіатури вводиться номер поїзда, назва сполучення, час відправлення з початкової станції і час прибуття на кінцеву, а у файл записується номер поїзда, назва сполучення, час відправлення з початкової станції і час прибуття на кінцеву та час у дорозі.
11	З клавіатури вводиться найменування товару, кількість товару, прізвище експедитора, вартість одиниці товару, а у файл записується найменування товару, кількість товару, прізвище експедитора, вартість одиниці товару та загальна вартість товару + 20% податку.
12	З клавіатури вводиться кут, а в файл записуються кут та значення його синуса і косинуса
13	З клавіатури вводиться кут, а в файл записуються кут та значення його тангенса і котангенса
14	З клавіатури вводиться найменування жорсткого диску, його об'єм і відсоток зайнятого простору, а у файл записується найменування жорсткого диску, його об'єм і кількість байт зайнятого простору
15	З клавіатури вводиться найменування жорсткого диску, його об'єм і відсоток зайнятого простору, а у файл записується найменування жорсткого диску, його об'єм і кількість байт вільного простору
16	З клавіатури вводиться найменування книги, кількість сторінок та швидкість читання (сторінок/хвилину), а у файл записуються найменування книги, кількість сторінок та скільки часу в хвилинах займе прочитати книгу
17	З клавіатури вводиться найменування спорт клубу, кількість відвідувачів за день та ціна річного абонементу, а у файл записується найменування спорт клубу, кількість відвідувачів за день та прибуток за день

18	З клавіатури вводиться найменування авіалінії, кількість пасажирів за день, ціна білету, а у файл записується найменування авіалінії, кількість відвідувачів за місяць та прибуток за місяць
19	З клавіатури вводиться номер поїзду, найменування сполучення, кількість вагонів, вартість проїзду, а у файл записується номер поїзду, найменування сполучення, кількість пасажирів за рік та прибуток за рік
20	З клавіатури вводиться назва моделі маніпулятора мишка, час напрацювання на відмову та середній час експлуатації на день, а у файл виводиться назва моделі маніпулятора мишка, час напрацювання на відмову та середній час експлуатації на день та кількість днів життя мишки
21	З клавіатури вводиться назва персонажу гри, кількість очок персонажу, кількість очок для наступного рівня та швидкість набирання очок за день, а у файл записується назва персонажу гри, кількість очок персонажу, кількість очок для наступного рівня, швидкість набирання очок за день та кількість днів для досягання наступного рівня
22	З клавіатури вводиться прізвище, ім'я і по батькові, а у файл записується прізвище, ім'я і по батькові та відстань в буквах між наймолодшою і найстаршою буквами.
23	З клавіатури вводиться прізвище, ім'я, розмір депозиту та ціна користування Інтернетом за місяць, а у файл виводиться прізвище, ім'я, розмір депозиту, ціна користування Інтернетом за місяць та кількість днів доступу до інтернету абонентом для поточного розміру депозиту.
24	З клавіатури вводиться ім'я спортсмена, часи проходження кожного з 3 кіл, а у файл записується ім'я спортсмена, часи проходження кожного з 3 кіл, середній час проходження кола та найкращий результат.
25	З клавіатури вводиться назва шахти, кількість людей,

	що у ній працює на зміну, тривалість зміни у годинах, розмір видобутку за зміну, а у файл виводиться назва шахти, кількість людей, що у ній працює на зміну, тривалість зміни у годинах, розмір видобутку за зміну та розмір видобутку за місяць.
26	З клавіатури вводиться назва автомобіля, оптимальна швидкість, ємність бензобаку, споживання палива на 100 км, а у файл записується назва автомобіля, максимальна швидкість, ємність бензобаку, споживання палива на 100 км та час проходження відстані у 1 000 км.
27	З клавіатури вводиться прізвище, ім'я середня швидкість друку (стор./год.), а у файл виводиться прізвище, ім'я середня швидкість друку та кількість робочих днів друкування тексту у 1 000 сторінок з врахуванням 8-годинного робочого дня.
28	З клавіатури вводиться назва танку, максимальна відстань пострілу, відстань на якій знаходиться ціль, а у файл виводиться назва танку, максимальна відстань пострілу, відстань на якій знаходиться ціль та ймовірність попадання, якщо вважати що ймовірність попадання зменшується зі швидкістю 0.1 / 100 м.
29	З клавіатури вводиться назва резервуару, об'єм резервуару, швидкість випускання води, а у файл вводиться назва резервуару, об'єм резервуару, швидкість випускання води та час за який резервуар спорожниться.
30	З клавіатури вводиться веб-адреса файлу, розмір файлу, середня швидкість завантаження, а у файл виводиться веб-адреса файлу, розмір файлу, середня швидкість завантаження та загальний необхідний час завантаження з врахуванням надлишковості інформації через особливості протоколу обміну у 21%.

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.

3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

// З клавіатури вводиться ім'я компанії та рік її заснування і записуються у файл

```
#include<iostream>
#include<fstream>
#include<string>
#include<iomanip>
using namespace std;

int main()
{
    while(true)
    {
        cout << "Show data - press 1" << endl;
        cout << "Write data - press 2" << endl;
        cout << "Exit - press 3" << endl;

        int choice;
        cin >> choice;

        if (choice == 1)
        {
            string company;
            int year;
            ifstream infile;
            infile.open("File.txt");
            if (!infile)
            {
                cout << "Cannot open file" << endl;
                return -1;
            }

            cout << setw(9) << "Company" << setw(9) << "Year"<<endl;
            while (!infile.eof())
            {
```

```

        infile >> company;
        infile >> year;
        if(!infile.eof())
        {
            cout.width(9);
            cout << company;
            cout.width(9);
            cout << year;
            cout << endl;
        }
    }
    infile.close();
}

if (choice == 2)
{
    string company;
    int year;
    cin>>company;
    cin>>year;
    fstream outfile("File.txt", ios::app);
    if (!outfile)
    {
        cout << "Cannot open file" << endl;
        return -1;
    }
    outfile.setf(ios::left);
    outfile.width(9);
    outfile << company << ' ';
    outfile.width(9);
    outfile << year << endl;
    outfile.close();
}

if (choice == 3)
{
    break;
}

}

system("pause");
}

```

ЛАБОРАТОРНА РОБОТА № 3.

ЗАСОБИ РОБОТИ З ДИНАМІЧНОЮ ПАМ'ЯТТЮ.

ДИНАМІЧНІ МАСИВИ

Мета: познайомитися із динамічними масивами.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Динамічне виділення пам'яті

В C++ об'єкти можна розміщати статично – під час компіляції, або динамічно – під час виконання програми, шляхом виклику функцій зі стандартної бібліотеки. Основна відмінність у використанні даних методів – в їхній ефективності та гнучкості. Статичне розміщення більш ефективне, так як виділення пам'яті відбувається до виконання програми, проте воно менш гнучке, тому що необхідно наперед знати тип і розмір об'єкту. Задачі, в яких необхідно зберігати та обробляти наперед не відому кількість елементів, зазвичай потребують динамічного виділення пам'яті.

Динамічне виділення пам'яті у мові C++ здійснюється за допомогою оператора `new`. Оператор `new` здійснює пошук неперервної області пам'яті в області пам'яті, що зветься некерована куча. Некерована куча – це структура даних за допомогою якої реалізована пам'ять, що може бути виділена динамічно в ході виконання програми, а також це область пам'яті, зарезервована під цю структуру. З іншої сторони куча - це довгий відрізок адрес пам'яті, поділений на блоки різних розмірів, що йдуть підряд. Пам'ять у кучі поділяється на заняту і вільну. Перед початком роботи програми вся пам'ять у кучі позначається як вільна. При виклику оператора динамічного виділення пам'яті у кучі відбувається пошук неперервного сегменту вільної пам'яті заданого розміру. Час такого пошуку є значним і займає більшу частину часу, що необхідна для виконання операції динамічного виділення пам'яті. Якщо такий сегмент було знайдено в кучі, то він помічається як зайнятий і програмі повертається адреса його початку, інакше – програмі повертається ознака відсутності такого сегменту в пам'яті, найчастіше `NULL`. Якщо в ході виконання програми значення адреси початку цього сегменту втрачається, то зайняту пам'ять звільнити буде неможливо. Якщо ця ситуація проявляється неодноразово, то це може призвести до вичерпання вільної пам'яті в системі. Коли динамічно виділена область пам'яті стає непотрібною,

то її потрібно звільнити за допомогою оператора звільнення динамічно виділеної пам'яті. При його виклику область пам'яті на яку вказує вказівник і яка була попередньо динамічно виділена з кучі позначається як вільна і її можна буде в подальшому використовувати заново.

Оператор динамічного виділення пам'яті `new` може мати дві форми:

1. *Виділення пам'яті під одиничний об'єкт* має наступний синтаксис:

```
тип *вказівник = new тип;
```

Наприклад, при виконанні оператора

```
int *ip = new int;
```

створюються 2 об'єкти: динамічний безіменний об'єкт розміром 4 байти (значення типу `int` займає 4 байти) і вказівник на нього з ім'ям `ip` розміром також 4 байти (у 32-ох бітній системі адреса займає 32 біти), значенням якого є адреса у пам'яті динамічного об'єкта. Можна створити й інший вказівник на той же динамічний об'єкт:

```
int *other = ip;
```

Якщо вказівникові `ip` присвоїти інше значення, то можна втратити доступ до динамічного об'єкта:

```
int *ip = new int;  
int i = 0;  
ip = &i;
```

У результаті динамічний об'єкт як і раніше буде існувати, але звернутися до нього буде вже не можна. При виділенні пам'яті об'єкт можна ініціалізувати певним значенням (окрім масивів):

```
int *ip = new int (3);
```

В даному випадку об'єкт типу `int` буде ініціалізований значенням 3.

2. Виділення пам'яті під масив заданого розміру має наступний синтаксис:

```
тип *вказівник = new тип[розмір_масиву];
```

Наприклад, при виконанні оператора

```
double *mas = new double [50];
```

виділяється пам'ять під масив з 50 елементів типу double. Тепер з цією динамічно виділеною пам'яттю можна працювати як зі звичайним масивом:

```
*(mas+5) = 3.27;  
mas[6] = mas[5] + sin(mas[5]);
```

У випадку успішного завершення операція new повертає вказівник зі значенням, відмінним від нуля. Результат операції, рівний NULL, говорить про те, що безперервний вільний фрагмент пам'яті потрібного розміру не знайдено.

Оператор звільнення динамічної пам'яті delete звільняє для подальшого використання в програмі ділянку пам'яті, яка була раніше виділена оператором new. Синтаксис оператора delete має наступний вигляд:

```
delete вказівник;      // видалення одиничного  
                        динамічного об'єкту  
  
delete[] вказівник;    // видалення динамічного  
                        масиву
```

Наприклад:

```
delete ip;              // Видаляє динамічний об'єкт типу int,  
                        що створений як ip = new int;  
delete[] mas;          // видаляє динамічний масив  
                        довжиною 50, що створений як  
                        double *mas = new double[50];
```

Операції `new` і `delete` дозволяють створювати і видаляти багатомірні динамічні масиви, підтримуючи при цьому ілюзію довільної розмірності.

Для створення динамічного двовимірного масиву використовуються наступні елементи:

1. вказівник на вказівник, який містить адресу початку допоміжного масиву адрес розмір якого рівний висоті двовимірного масиву (кількості рядків);
2. допоміжний масив адрес, що зберігає адреси одновимірних масивів, які власне міститимуть дані; розмір цих масивів рівний розміру ширини двовимірного масиву (кількості стовпців);
3. множина масивів, що зберігають дані (реалізують рядки масиву).

Якщо вимірів більше, то використовується більша кількість допоміжних масивів до яких приєднуватимуться інші масиви, завдяки чому власне і утворюватимуться нові виміри. Загалом можна сказати: скільки зірочок при оголошенні базового вказівника на багатовимірний масив, стільки вимірів міститиме цей масив.

Розглянемо типовий алгоритм створення динамічного багатовимірного масиву на прикладі створення динамічного двовимірного масиву елементи якого мають тип `int`. Типова схема двовимірного масиву зображена на рис. 3.1.

На першому етапі (1 на рис. 3.1) оголошується вказівник на вказівник, який міститиме адресу початку допоміжного масиву адрес розмір якого рівний висоті двовимірного масиву (кількості рядків). На другому етапі (2 на рис. 3.1) створюється масив адрес розмір якого рівний висоті двовимірного масиву (кількості рядків). Початкова адреса цього масиву присвоюється вказівнику на вказівник. На наступних двох етапах ітераційно створюються масиви, що зберігатимуть дані двовимірного масиву (3 на рис. 3.1) і зв'язуються з відповідними комітками масиву адрес (4 на рис. 3.1). Кількість цих масивів рівна розміру допоміжного масиву адрес, а їх розмір рівний ширині цільового двовимірного масиву (кількості стовпців).

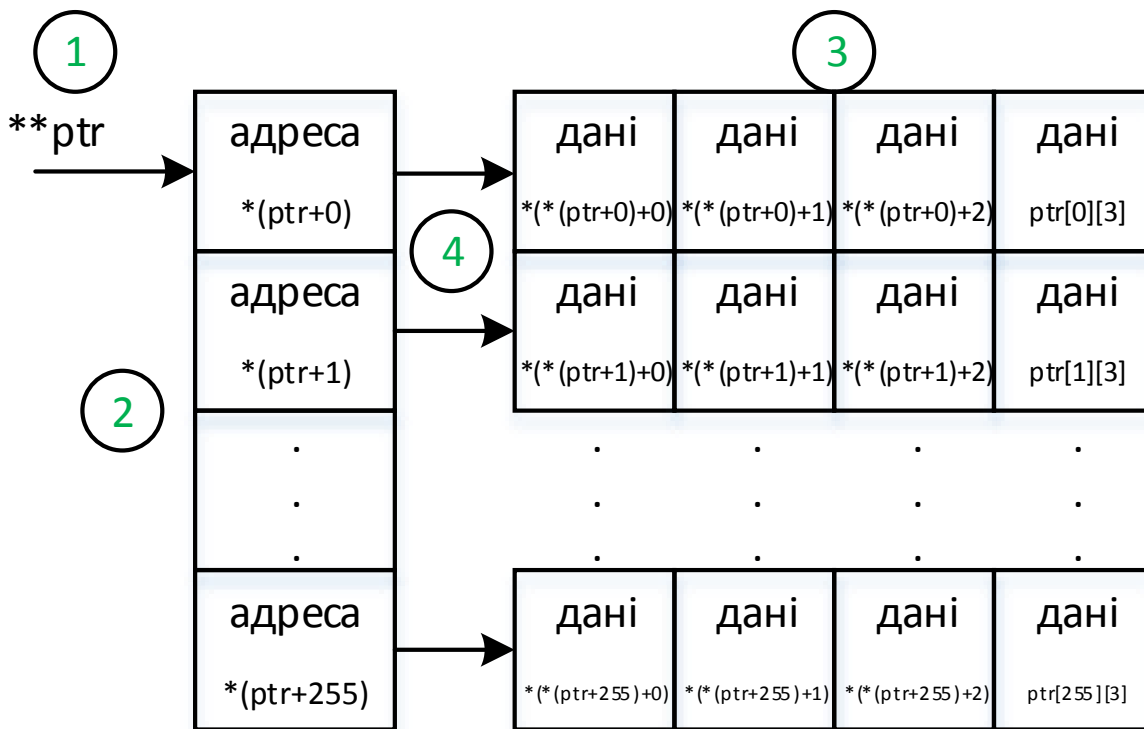


Рис. 3.1. Типова схема двовимірного масиву

Для кращого розуміння процесу створення двовимірного динамічного масиву розглянемо стилізований розподіл пам'яті кучі (див. рис. 3.2.), який відповідає розподілу пам'яті двовимірного масиву розміром 256×4 елементи типу `int`, що зображений на рис. 3.1. На рис. 3.2 зображено фрагмент адресного простору пам'яті процесу (програми, що виконується) в якому міститься вказівник на вказівник і куча в межах якої динамічно виділено пам'ять під двовимірний масив даних типу `int`. Як видно з рис. 3.2. вказівник на вказівник розташовується за межами кучі, займає 4 байти в адресному просторі і містить адресу початку масиву вказівників на масиви з елементами двовимірного масиву даних. Будь який вказівник в 32-ох розрядній системі займає 4 байти, оскільки кожна адреса має 32 розряди, тому кожен елемент масиву вказівників так як і вказівник на вказівник займає 4 байти пам'яті, тому відстань між елементами масиву вказівників рівна 4. Достаючись до масиву вказівників шляхом розіменовування першого виміру двовимірного вказівника і зміщуючись між елементами масиву адрес можна отримати адреси початку масивів з даними, що відображено на рис. 3.2 записами типу $*(ptr+0)$, $*(ptr+1)$, Розіменовуючи відповідну адресу початку відповідного масиву ми можемо прочитати нульовий елемент відповідного масиву даних, що відображено записом типу $*(*(ptr+1)+0)$. Цей запис означає, що ми зміщуємося відносно початку масиву вказівників (першого виміру) на 1 елемент, розіменовуємо його (читаємо вміст комірки, який є адресою початку

першого масиву з даними) і одержуємо адресу початку першого масиву з даними. Зміщуємося на 0 елементів відносно початку цього масиву і знову розіменовуємо його (читаємо вміст комірки пам'яті, що зміщена на 0 елементів відносно початку першого масиву з даними) одержуючи таким чином вміст 0 комірки. Кожен елемент масиву даних типу `int` займає у пам'яті стільки місця, скільки необхідно для зберігання одного значення типу `int`, а саме 4 байти, тому відстань між елементами масивів з даними рівна 4.

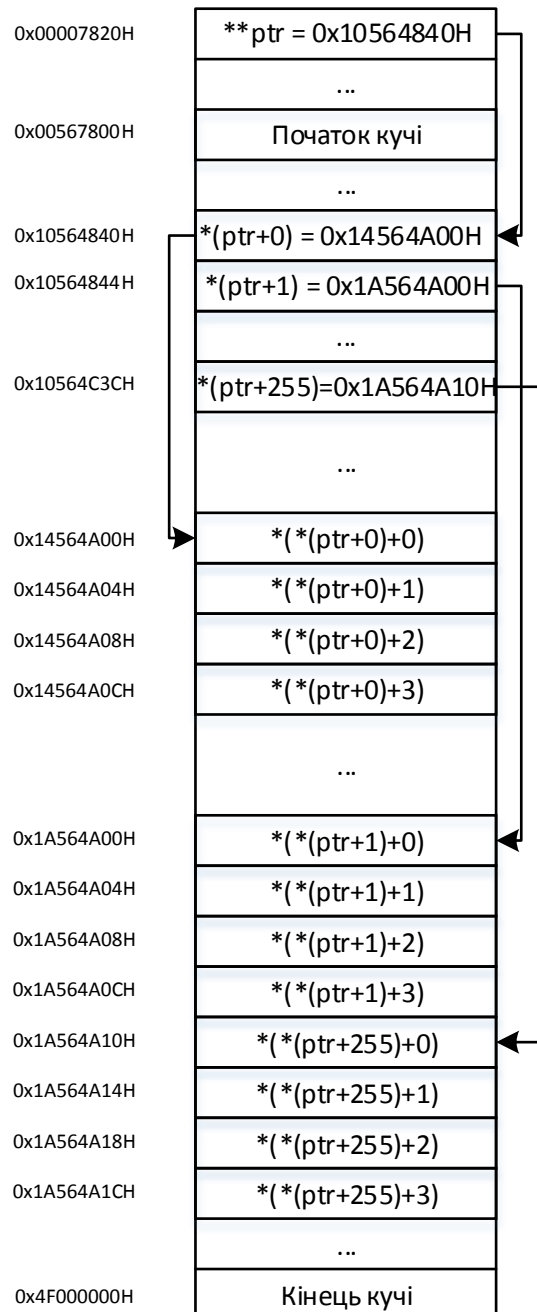


Рис. 3.2. Фрагмент пам'яті процесу з динамічно створеним двовимірним масивом даних типу `int`.

Якщо нам необхідно прочитати скажімо 3 комірку 2-го масиву даних (комірку 3-го стовпця у другому рядку за умови нумерації рядків і стовпців починаючи з нуля), то нам слід записати наступний рядок коду `* (* (ptr+2)+3)`. Цей рядок коду аналогічний запису `ptr[2][3]`. При роботі з масивами форми з дужками і вказівниками є еквівалентними, тому можна застосовувати будь-яку з них, яка є зручнішою для конкретного випадку. Проте слід зазначити що форма з вказівниками є низькорівневішою і значно потужнішою, ніж дужкова форма, проте часто вона є джерелом помилок. Для роботи з багатовимірними масивами допускається і змішана форма: `* (ptr[2]+3)`.

Розглянемо фрагмент коду, виконання якого призведе до створення двовимірного масиву типу `int` розміром `256*4`, що ми щойно розглянули.

```
// Оголошення вказівника на вказівник на тип int
int** ptr;
// Створення у кучі масиву вказівників на тип int. Тип даних у
// кожній з комірок створеного масиву є вказівник на тип int.
ptr = new int* [256];

// Створення у кучі 4 елементного масиву з даними типу int і
// запам'ятовування адреси його початку у відповідному елементі
// масиву вказівників на тип int.
for (int i = 0; i < 256; i++)
    a[i] = new int[4];
```

Звільнення пам'яті для двовимірних динамічних масивів відбувається у зворотному напрямку відносно того, як він створювався – спочатку вивільняється пам'ять зі всіх масивів з даними, а потім вивільняється пам'ять масиву вказівників.

```
for (i = 0; i < 256; i++)
{
    delete[] ptr[i];
}
delete[] ptr;
```

Зубчаті масиви

Створюючи масиви з даними різного розміру у другому вимірі динамічних двовимірних масивів можна створювати так звані зубчаті масиви, які в окремих випадках можуть значно зекономити пам'ять в порівнянні з використанням двовимірних масивів. Приклад зубчатого масиву зображено на рис. 3.3. Принцип створення зубчатих масивів відрізняється від принципу створення двовимірних динамічних масивів лише етапом створення одновимірних масивів з даними, де на відміну від двовимірних динамічних масивів масиви з даними можуть мати різний розмір утворюючи не матрицю, а зубці, завдяки чому даний вид масивів і одержав свою назву.

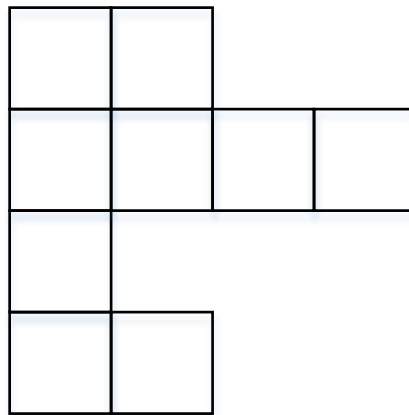


Рис.3.3 Зубчатий масив.

КОНТРОЛЬНІ ПИТАННЯ



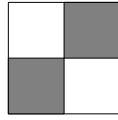
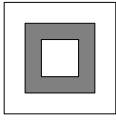
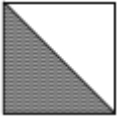

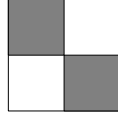
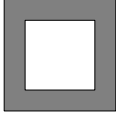



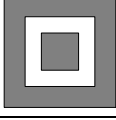
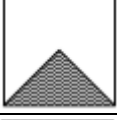
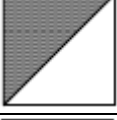
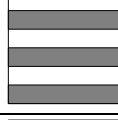
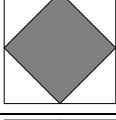
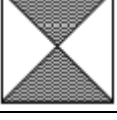
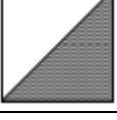
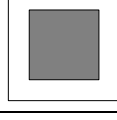
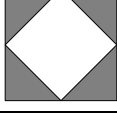
1. Що таке динамічне виділення пам'яті?
2. Як динамічно виділити пам'ять для змінної?
3. Як динамічно виділити пам'ять для одновимірного масиву?
4. Як динамічно виділити пам'ять для двовимірного масиву?
5. Як динамічно звільнити пам'ять пам'ять для змінної?
6. Як динамічно звільнити пам'ять пам'ять для одновимірного масиву?
7. Як динамічно звільнити пам'ять пам'ять для двовимірного масиву?
8. Скільки байт в пам'яті займає вказівник?
9. Що таке зубчатий масив?
10. Чим відрізняється зубчатий масив від двовимірного?

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.

ЗАВДАННЯ

Задано квадратну матрицю, всі елементи якої рівні одиниці. Написати функцію `void func (int** arr, int n)`, котра заповняє заштриховану область матриці (згідно варіанту) нулями. Розмір масиву вводиться з клавіатури.

№		№		№		№	
1		6		11		16	
2		7		12		17	
3		8		13		18	
4		9		14		19	
5		10		15		20	

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

```
#include <iostream>
#include <stdio.h>
#include <iomanip>
using namespace std;
// функція заповнює двовимірну квадратну матрицю нулями і виводить її на
екран
void func (int** arr, int n)
{
    for (int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
        {
            arr[i][j]=0;
        }
    }
    for (int i=0; i<n; i++)
    {
        for(int j=0; j<n; j++)
            cout<<setw(4)<<arr[i][j];
        cout<<endl;
    }
}
int main()
{
    int **arr, n;
    cout << "Enter order of matrix:";
    cin >> n;
    cout<<"-----"<<endl;
    arr = new int* [n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = new int [n];
    }

    func(arr,n);

    for (int i = 0; i < n; i++)
    {
        delete[] arr[i];
    }
    delete[] arr;
    return 0;
}
```

ЛАБОРАТОРНА РОБОТА № 4.

КЛАСИ ТА ОБ'ЄКТИ

Мета: познайомитися із класами та об'єктами.

КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

Визначення класу

Основна відмінність будь-якої об'єктно-орієнтованої мови програмування від інших не об'єктно-орієнтованих мов програмування є можливість створення нових визначених користувачем типів, що називаються класами. Клас – це визначений користувачем тип з даними-елементами (властивостями) та функціями-елементами (методами), що являються членами класу. Він зазвичай описує певну абстракцію якоїсь сутності реального світу з її властивостями і можливими діями. Змінна типу клас називається об'єктом. Об'єкт – це вже не абстракція, а елемент реального світу, який може володіти певними характеристиками, які задаються властивостями в класі, та діяти згідно операцій заданих методами у класі. Оголошення класу в мові C++ має наступний синтаксис (не обов'язково щоб клас мав всі секції чи спадкував базовий клас):

```
class ім'я_класу : список_базових_класів {
public:
    //      Оголошення загальнодоступних (відкритих)
    членів класу, що можуть бути доступні звідусіль
protected:
    //      Оголошення захищених членів класу, що
    доступні тільки для похідних класів
private:
    //      Оголошення закритих членів класу, що
    доступні лише членам цього класу, та не
    можуть бути змінені чи викликані з-за меж
    класу напрямую, а лише за посередництвом
    методів з інших секцій
};
```

Члени класу (властивості і методи), оголошені після ключового слова *public* стають відкритими членами. Це означає, що вони доступні:

- усім іншим членам класу;
- дружнім конструкціям (класам, методам, функціям);
- членам похідних класів;
- з-під об'єктів класу після створення об'єктів; після створення об'єкту класу до його членів можна звертатися лише якщо вони є загальнодоступними.

Члени класу, оголошені після ключового слова *private*, стають закритими членами. Вони доступні:

- для інших членів того ж класу;
- друзям класу.

Якщо в класі не оголошено секцій, але визначено методи чи властивості, то вони вважатимуться такими, що оголошені у секції *private*. Для модифікації властивостей, що оголошені в секції *private* створюються спеціальні методи, які здійснюють цю модифікацію. Назви таких методів прийнято починати зі слова *set*. На жаргоні такі методи інколи називають сетерами. Для читання властивостей, що оголошені в секції *private* створюються спеціальні методи, які безпечно повертають значення цих властивостей. Назви таких методів прийнято починати зі слова *get*. На жаргоні такі методи інколи називають гетерами.

Члени класу, оголошені після ключового слова *protected*, стають захищеними членами. Вони доступні:

- для інших членів того ж класу;
- членам похідних класів;
- друзям класу.

Таблиця 4.1. Рівні доступу членів класу

Доступ	private	protected	public
Інші члени цього класу	+	+	+
Друзі класу	+	+	+
Члени похідних класів	-	+	+
З-під об'єктів класу	-	-	+

Оголошення класу містить оголошення даних-елементів (властивостей) та функцій-елементів (методів) класу. Одна з фундаментальних основ ООП передбачає інкапсуляцію даних, тобто дані мають бути недоступними ззовні, а

лише за посередництвом загальнодоступних методів класу. Оголошення методів має вигляд оголошення прототипу функції в середині однієї з секцій класу, зазвичай у секції `public`. Якщо методи є службовими і не мають бути доступні ззовні, тоді їх розміщують в секції `private` або `protected`. Сама ж реалізація методу може знаходитись як усередині класу (тоді оголошення методу в класі має вигляд оголошення функції з тілом), так і поза оголошенням класу (тоді у класі оголошується прототип функції, а її тіло визначається за межами класу). Але один з найфундаментальніших принципів розробки програмного забезпечення мовою C++ полягає у розмежуванні інтерфейсу класу від його реалізації. Тому при побудові програми мовою C++ кожне оголошення класу зазвичай розміщують у заголовочний файл `*.h` назва якого співпадає з назвою класу, а реалізацію методів цього класу – у файл `*.cpp` із тим ж іменем, що й `*.h` файл. Заголовочні файли включаються (за допомогою директиви `#include`) у кожен файл, у якому використовується клас, а файли з вихідними кодами компілюються і компонуються з файлом, що містить головну програму (`main`-функцію).

При визначенні методу класу за межами класу необхідно зв'язати ім'я відповідного методу з ім'ям класу. Зв'язок встановлюється шляхом написання імені класу, оператора розширення області видимості (`::`) та імені методу. Оскільки різні класи можуть мати елементи з однаковими іменами, то оператор розширення області видимості дозволяє однозначно ідентифікувати методи конкретного класу.

```
тип ім'яКласу :: ім'яМетоду (списокПараметрів)
{
    // тіло методу
}
```

Областю видимості властивостей та методів класу є клас, тобто все що оголошене в середині класу є видиме і доступне за іменем елементів в середині класу без додаткових маніпуляцій.

Статичні властивості і методи

Статичні дані-елементи (властивості) оголошуються в класі, а ініціалізуються за межами класу (не у конструкторах). На відміну від інших властивостей вони належать класу, а не об'єкту. Тобто вони єдині для всіх об'єктів певного класу і зміни одним об'єктом вплинуть на всі об'єкти даного

класу. Якщо статичні властивості оголошені в секції `public`, то вони будуть завжди доступні незалежно від того чи створено хоч один об'єкт класу. За допомогою статичних властивостей зручно реалізовувати лічильники, наприклад, лічильники створених об'єктів класу.

Синтаксис оголошення і ініціалізації статичних властивостей:

```
// оголошення статичної властивості
class ім'я_класу {
    private:
        static тип назваВластивості;
};
// ініціалізація статичної властивості
тип ім'я_класу :: назваВластивості = значення;
```

Приклад оголошення і ініціалізації статичних властивостей:

```
// оголошення статичної властивості
class CMyClass
{
    private:
        static int m_counter;
};
// ініціалізація статичної властивості
int CMyClass :: m_counter = 0;
```

Статичні методи оголошуються з використанням ключового слова `static` при оголошенні методу. На відміну від нестатичних методів дані методи існують завжди, навіть якщо нема жодного екземпляру класу. Тому їх можна викликати з будь якого місця програми не створюючи при цьому об'єкту класу. Для виклику статичного методу слід вказати назву класу, оператор розширення області видимості (`::`), назву статичного методу, передати йому параметри і поставити крапку з комою в кінці виклику. Статичні методи не мають доступу до членів класу, тому вони мають реалізуватися самодостатніми, тобто такими, що не потребують для своєї роботи інших даних, ніж ті що вони містять в собі і ті, що передаються їм через параметри. Одним з прикладів використання статичних методів є методи, що обчислюють різні тригонометричні функції.

Конструктори та деструктор

При створенні об'єкта класу автоматично викликається спеціальний метод, що зветься конструктор. Конструктор – це метод класу ім'я якого співпадає з іменем класу та не повертає ніяких значень (навіть void). Основне призначення конструктора – початкова ініціалізація об'єкту класу.

У класі може бути визначено кілька перевантажених конструкторів, що відрізняються списком параметрів. При створенні об'єкта викликатися буде тільки один з них. Який саме – визначається зі способу створення об'єкта. Якщо конструктор має один параметр, то він може бути використаним для операції приведення типів. Якщо треба унеможливити використання конструктора для операції приведення типів, то його слід оголосити з використанням ключового слова `explicit`.

Конструктор без параметрів або конструктор, у якого всі аргументи – це аргументи за замовчуванням, має спеціальну назву – конструктор за замовчуванням. Конструктор за замовчуванням може бути лише один. Якщо в класі явно не визначений конструктор за замовчуванням, то він створюється автоматично і при виклику ініціалізуватиме всі члени-дані нулями. Приклади конструкторів за замовчуванням (клас може містити лише один з них):

```
CMyClass(){}  
CMyClass(int i = 5){m_i=i;}
```

Конструктор, що як свій єдиний параметр приймає посилання на константний об'єкт цього ж класу, зветься конструктором копіювання. Він призначений для створення копії існуючого об'єкта шляхом поелементного копіювання значень нестатичних членів-даних (властивостей) класу і викликається при ініціалізації в операторі оголошення об'єкта або при передачі об'єкта класу за значенням у функцію (метод). Якщо конструктор копіювання не визначений явно в класі, то він створюється автоматично і при виклику здійснює поелементне копіювання всіх властивостей класу. Даний конструктор можна не оголошувати явно в більшості випадків. Одним з випадків коли його слід обов'язково явно оголосити є ситуація коли клас містить динамічні члени-дані. У цьому випадку якщо не оголосити явно конструктор копіювання при копіюванні динамічних властивостей створюватимуться не їх копії, а копії вказівників на одне і те ж значення, що належить об'єкту оригіналу. Це призведе до помилок функціонування програми, зокрема спроби більше, ніж один раз знищити динамічно виділену властивість при знищенні об'єктів оригіналу і його копій.

Приклади конструкторів:

```
CMyClass(){} //конструктор за замовчуванням  
  
CMyClass(int i = 5){m_i=i;} //конструктор за замовчуванням  
  
CMyClass(int i){m_i=i;} //конструктор який дозволяє виконувати операцію  
приведення типу  
  
explicit CMyClass(double d){m_d=d;} //явний конструктор, який не дозволяє  
виконувати операцію приведення типу  
  
CMyClass(const CMyClass & cl){cl.m_i = m_i;} // конструктор копіювання
```

Приклади ситуацій при яких викликаються конструктори, що оголошені вище:

```
CMyClass myclass; // неявний виклик конструктора за замовчуванням  
  
CMyClass myclass = CMyClass(); //явний виклик конструктора за замовчуванням  
  
CMyClass myclass(5); //виклик конструктора, що приймає тип int  
  
CMyClass *pmyclass = new CMyClass(5); //виклик конструктора, що приймає  
тип int для динамічного створення об'єкту  
  
CMyClass myclass = 5; //виклик конструктора, що приймає тип int  
  
CMyClass myclass = 5.0; //помилка (explicit забороняє використовувати  
конструктор для операції приведення типів)
```

Приклади ситуацій при яких викликається конструктор копіювання:

`CMyClass myclass(another_myclass);` // коли створюється об'єкт і при цьому він ініціалізується іншим об'єктом такого ж класу

`CMyClass myclass = another_myclass;` // коли створюється об'єкт класу і йому при цьому присвоюється інший існуючий об'єкт такого ж класу.

`CMyClass myclass = CMyClass (another_myclass);` // коли відбувається явний виклик конструктора копіювання при створенні нового об'єкту

`CMyClass *pmyclass = new CMyClass (another_myclass);` // коли новий об'єкт створюється динамічно і при цьому ініціалізується існуючим об'єктом цього ж класу

`return myclass;` // коли відбувається повернення об'єкту з методу або функції

`void function(myclass);` // коли об'єкт передається в функцію за значенням

`myclass.method(another_myclass);` // коли об'єкт передається в метод за значенням

Деструктор – це спеціальна функція-елемент класу, яка викликається при знищенні об'єкту і не приймає параметрів та не повертає значень. Знищення об'єкту може відбуватися, наприклад, коли виконувана програма залишає область дії, у якій був створений об'єкт цього класу, або явно здійснюється знищення динамічно створеного об'єкту. Деструктор сам не знищує об'єкт, а лише виконує підготовку до знищення об'єкту (вивільняє пам'ять від динамічно створених даних-елементів об'єкту, закриває потоки обміну даними,...) перед тим, як система звільняє область пам'яті, у якій зберігався об'єкт, щоб використовувати її для розміщення нових об'єктів. Ім'я деструктора складається з символ тильда (~) та імені класу. Клас може мати тільки один деструктор – перевантаження деструктора забороняється. Якщо деструктор не визначений явно, то він буде згенерований автоматично і при виклику нічого не робитиме. Деструктор доцільно оголошувати написавши перед ним ключове слово `virtual`, яке забезпечує коректність функціонування класу при спадкуванні.

Загалом у класі автоматично визначаються наступні методи:

- конструктор за замовчуванням;
- конструктор копіювання;

- оператор присвоювання;
- деструктор;
- оператор адресації.

Конструктори глобальних об'єктів (ті що створюються за межами будь-якої функції) викликаються перед тим як будь-яка функція даного файлу (включаючи `main`) почне виконуватися. Відповідні деструктори викликаються, коли завершується `main`-функція або коли викликається функція `exit`.

Конструктори автоматичних локальних об'єктів викликаються, коли процес виконання досягає місця де створюються об'єкти класів. Відповідні деструктори викликаються, коли покидається область дії об'єктів (тобто покидається блок, у якому ці об'єкти оголошені). Конструктори і деструктори для автоматичних об'єктів викликаються щораз при вході і виході з області дії.

Конструктори статичних локальних об'єктів викликаються відразу ж, як тільки процес виконання досягає місця, де об'єкти були вперше оголошені. Відповідні деструктори викликаються, коли завершується `main`-функція або коли викликається функція `exit`.

Списки ініціалізації

В конструкторі можна здійснювати ініціалізацію властивостей класу за допомогою списків ініціалізації. Список ініціалізації має наступний вигляд:

```
ім'яКонструктора(Параметри) : властивість1(значення),
    властивість2(значення),...
{
    // тіло конструктора
}
```

Список ініціалізації слідує після двокрапки, яка ставиться після закриваючої круглої дужки в оголошенні конструктора. Після двокрапки вказуються назва властивості і в круглих дужках яким значенням слід цю властивість ініціалізувати. Якщо у списку ініціалізації кілька властивостей, то вони розмежовуються комами. Якщо властивість є об'єктом якогось класу, то в дужках вказуються параметри, за допомогою яких буде визначено який конструктор для створення даного об'єкту слід викликати. Таким способом можуть ініціалізуватися лише не статичні і не динамічні властивості класу. Список ініціалізації є аналогом тіла конструктора, якщо мова не йде про

динамічні змінні. Дозволяється поєднувати ініціалізацію властивостей в тілі конструктора і у списку ініціалізації.

Методи доступу до об'єкта

Змінна класу (об'єкт, екземпляр класу) оголошується у відповідності з наступним синтаксисом:

```
Ім'яКласу ім'яЗмінної(списокПараметрів);
```

При оголошенні змінної відбувається створення відповідного об'єкта. При цьому виконується виклик одного з конструкторів класу цього об'єкта. Для доступу до елементів об'єкта використовується оператор крапка (.).

Доступ до об'єкта можна одержати через вказівник на цей об'єкт. При цьому до членів об'єкта звертаються не за допомогою оператора крапка, а за допомогою оператора "стрілка" (->).

Синтаксис динамічного створення об'єкту класу має наступний вигляд:

```
ім'яКласу * ім'яЗмінної = new ім'яКласу(параметри);
```

У даному випадку з вільної пам'яті кучі виділяється ділянка пам'яті, достатня для збереження об'єкта зазначеного класу. Для створеного об'єкта автоматично викликається конструктор відповідно до заданих параметрів. Як результат, оператор `new` повертає адресу створеного об'єкта, яка зберігається у вказівнику. Після цього можна здійснювати доступ до динамічно створеного об'єкта за допомогою цього вказівника.

Після того, як динамічно створений об'єкт стає непотрібним, його необхідно знищити за допомогою оператора `delete`. Знищення об'єкта приводить до виклику деструктора, звільненню пам'яті, відведеної під даний об'єкт і поверненню цієї пам'яті у вільну пам'ять кучі.

Кожен об'єкт у мові C++ містить спеціальний вказівник `this`, який містить адресу об'єкту якому він належить. Крім цього цей вказівник неявно автоматично передається будь-якому нестатичному методу класу при його виклику, вказуючи цим самим на об'єкт з-під якого метод був викликаний. Як і будь-який вказівник у 32-ох бітній системі він займає 4 байти в пам'яті.

Ще одним способом доступу до об'єкта є доступ через посилання. Посилання по суті є псевдонімом (нікнеймом) назви об'єкту класу. Його зручно використовувати при передачі об'єкту або змінної методам або функціям. При цьому копіювання даної змінної чи об'єкту не відбувається, а

функція чи метод отримує сам об'єкт напряду, який просто відомий для неї під іншою назвою. Оскільки посилання є лише іншою назвою об'єкту або змінної, то в пам'яті комп'ютера воно не займає ніякого місця, що дозволяє її економити та пришвидшувати обробку даних при викликах функцій і методів. Небезпекою використання посилань є те, що функція чи метод працює з об'єктом на пряму, а не з його копією, в результаті чого є небезпека ненароком змінити стан об'єкта (значення даних-елементів об'єкту). Єдина ситуація, коли не можна використовувати посилання, а слід використовувати вказівники – це передача масивів через параметри. Синтаксис оголошення посилання має наступний вигляд:

```
ім'яКласу & ім'язмінної;
```

Приклад використання посилання:

```
CMyClass myclass;  
CMyClass & refObj = myclass;  
CMyClass *pmyclass = new CMyClass();  
refObj = *pmyclass;  
// функція void func(CMyClass & rObj);  
void func(refObj);  
void func(myclass);
```

У даному прикладі створюється об'єкт `myclass` класу `CMyClass`. Створений об'єкт присвоюється посиланню `refObj`. Тепер `myclass` і `refObj` – це один і той самий об'єкт відомий за різними назвами. Потім динамічно створюється об'єкт класу `CMyClass` адреса якого присвоюється вказівнику `pmyclass`. Після чого даний вказівник розіменовується і об'єкт на який він вказує присвоюється посиланню `refObj`. Потім здійснюються 2 виклики функції `func`, яка оголошена як `func(CMyClass & rObj);` і приймає посилання на об'єкт класу `CMyClass`. Хоча в першому випадку у функцію передається посилання на об'єкт класу, а у другому – об'єкт класу вона в обох випадках працюватиме з переданим об'єктом як з посиланням `rObj`. Перевагою такого способу передачі параметрів є економія пам'яті, відсутність необхідності автоматичного створення копії об'єкту чи змінної при передачі її через параметр, звертання до членів об'єкту відбувається як до членів об'єкту класу, а не за вказівником, оригінал переданого об'єкту змінюється в процесі обробки, завдяки чому немає потреби повертати

оброблену копію об'єкту з функції. Недоліками даного способу є неможливість передавати масиви через посилання і те, що не завжди об'єкт має змінюватися і ризик помилково його змінити може мати велику ціну. Також бувають ситуації, коли слід працювати з копією об'єкту в методі, тоді в метод слід передавати об'єкт за значенням, а не за посиланням чи вказівником.

Особливості класів з динамічними властивостями

Робота з класами, що використовують динамічно створювані властивості має деяку особливість незнання якої призводить до фатальних наслідків. Тож, якщо клас містить динамічно створювані дані-елементи, то в ньому обов'язково мають перевизначатися наступні методи:

1. Конструктор за замовчуванням в якому має здійснюватися коректна ініціалізація динамічно створюваних властивостей (даних-елементів) об'єкту та інших властивостей об'єкту;
2. Конструктор копіювання в якому мають створюватися копії динамічно створюваних властивостей класу, які копіюються повністю, а не створюються копії вказівників, та копії інших властивостей об'єкту, що копіюється.
3. Деструктор в якому має відбуватися коректне звільнення пам'яті динамічно створюваних властивостей класу.
4. Оператор присвоєння, який має перевіряти на самоприсвоєння об'єкт, що присвоюється і у разі, якщо об'єкту присвоюється інший об'єкт даного класу, то він має знищувати існуючі динамічно створені властивості класу і замінити їх копіями динамічно створених властивостей об'єкту, що присвоюється, а також скопіювати інші властивості об'єкту, що присвоюється.

Приклад реалізації класу, що містить динамічні властивості.

```
// файл Str.h
class CStr
{
private:
    char * m_pstr; //дані-елементи (властивості) класу
    int m_len;
public:
    explicit CStr(const char * s); //конструктор, якому заборонено
```

використання для операції приведення типу `const char *` до об'єкту класу `CStr`

```
CStr(); // конструктор за замовчуванням
CStr(const CStr & st); // конструктор копіювання
~CStr(); //деструктор
CStr & operator=(const CStr & st); //оператор присвоєння

// методи класу
void SetString(const char * str);
char* GetString();
void Show();
};
```

// файл Str.cpp

```
#include<iostream>
#include "str.h"
using namespace std;

CStr::CStr(const char * s)    //конструктор класу CStr
{
    m_len=strlen(s);
    m_pstr=new char[m_len+1];
    strcpy(m_pstr,s);
    cout<<"CStr(const char * s). CStr object created" <<endl;
}
CStr::CStr()
{
    m_len=0;
    m_pstr=new char[1];
    m_pstr[0]='\0';
    cout<<"CStr(). CStr object created"<<endl;
}
CStr::CStr(const CStr & st)
{
    m_len=st.m_len;
    m_pstr=new char[m_len+1];
```

```

        strcpy(m_pstr,st.m_pstr);
        cout<<"CStr(const CStr & st). CStr object created" <<endl;
    }
    CStr::~CStr()
    {
        delete [] m_pstr;
        cout<<"CStr object destroyed"<<endl;
    }

    void CStr::Show()
    {
        cout<<m_pstr<<endl;
    }

    void CStr::SetString(const char * str)
    {
        m_len=strlen(str);
        delete [] m_pstr;
        m_pstr=new char[m_len+1];
        strcpy(m_pstr,str);
    }

    char* CStr::GetString()
    {
        return m_pstr;
    }

    CStr & CStr::operator=(const CStr & st)
    {
        // перевірка на самоприсвоєння
        if ( this==&st )
        {
            return *this;
        }
        delete [] m_pstr;
        m_len=st.m_len;

```

```

        m_pstr=new char[m_len+1];
        strcpy(m_pstr,st.m_pstr);
        cout<<"CStr::operator=(const CStr & st) working" <<endl;
        return *this;
    }

```

// файл Stock.h

```

class CStock
{
private:
    CStr m_company;
    int m_shares;
    double m_share_val;
    double m_total_val;
    inline void set_tot()
    {m_total_val = m_shares*m_share_val; }
    static unsigned int m_unbjects;
public:
    CStock();
    CStock( const char * co, int n=0, double pr=0 );
    ~CStock();
    CStock & CStock::operator=(const CStock & stc);
    void buy( int num, double price );
    void sell( int num, double price );
    void update( double price );
    void show();
    const CStock & CStock::topval( const CStock & s )const;
};

```

// файл Stock.cpp

```

#include<iostream>
#include "Stock.h"
using namespace std;

```

```

unsigned int CStock::m_unbjects=0; //ініціалізація статичного елементу класу

```

```

CStock::CStock() : m_company() //ініціалізація об'єкту класу CStr за
допомогою списків ініціалізації. Викликає конструктор за замовчуванням

```

для объекта m_company.

```
{
    m_shares=0;
    m_share_val=0.0;
    m_total_val=0.0;
    m_unbjects++;
    cout<<"CStock(). CStock Object: "<<m_unbjects <<endl;
}
```

CStock::CStock(const char * co, int n, double pr):m_company(co)

```
{
    m_shares=n;
    m_share_val=pr;
    set_tot();
    m_unbjects++;
    cout<<"CStock(const char * co, int n, double pr). CStock Object:
"<<m_unbjects<<endl;
}
```

CStock::~CStock() //деструктор

```
{
    cout<<"CStock Object destroyed"<<endl;
    m_unbjects--;
}
```

void CStock::buy(int num, double price)

```
{
    m_shares+=num;
    m_share_val=price;
    set_tot();
}
```

void CStock::show()

```
{
    m_company.Show();
    cout<<" Shares:"<<m_shares<<"\n"
```

```

        <<" Share Price: $"<<m_share_val
        <<" Total Worth: $"<<m_total_val
        <<" Total Objects created: "<<m_unbjecks<<"\n';
    }

const CStock & CStock::topval( const CStock & s ) const // вживання const в
кінці оголошення методу гарантує, що код методу не змінить значення
властивостей об'єкту з-під якого викликано метод
{
    if ( (s.m_total_val>m_total_val) )
    {
        return s;
    }
    else
    {
        return *this; //повертає об'єкт, який визивав метод
    }
}

CStock & CStock::operator=(const CStock & stc)
{
    // перевірка на самоприсвоювання
    if ( this==&stc )
    {
        return *this;
    }
    m_share_val=stc.m_share_val;
    m_shares=stc.m_shares;
    m_total_val=stc.m_total_val;
    m_company=stc.m_company;
    cout<<"CStock::operator=(const CStock & stc) working "<<endl;
    return *this;
}

// файл main.cpp
#include<iostream>
#include<cstdlib>

```

```

#include<cstring>
#include "stock.h"
#include "str.h"

using namespace std;

int main()
{
    cout<<endl<<endl<<"creating empty share" <<endl;
    CStock MyShare;
    MyShare.show();

    {
        cout<<endl<<endl<<"creating MS share" <<endl;
        CStock MSShare("Microsoft", 5, 5.0);
        MSShare.show();
    }

    cout<<endl<<endl<<"creating IBM share"<<endl;
    CStock IBMShare("IBM", 7, 5.5);
    IBMShare.show();

    cout<<endl<<endl<<"MyShare = IBMShare;"<< endl;

    MyShare = IBMShare;

    cout<<endl<<endl<<"MyShare = \"Hello\";" <<endl;
    MyShare = "Hello";

    return 0;
}

```

Угорська нотація

Угорська нотація - метод найменування змінних в програмуванні, який отримав найбільшого поширення у 90-х роках у програмістів компанії

Microsoft у яких вона була внутрішнім стандартом. Угорська нотація полягає в тому, що до ідентифікатора змінної або функції додається префікс, що вказує на його тип. Префікси, що додаються до типів відображені в табл. 4.2.

Перевагою угорської нотації є системність, що полегшує читання програм і зменшує ймовірність неправильного використання змінної.

Недоліком є те, що при зміні типу змінної назва втрачає зміст і не вказує на тип змінної, що потребує введення нової змінної.

Таблиця 4.2. Префікси типів.

Префікс	Скорочення від	Сенс	Приклад
s	string	рядок	sClientName
sz	zero-terminated string	рядок, обмежений нульовим символом	szClientName
n	int	цілочисельна змінна	nSize
i	int	цілочисельний індексатор	iCntr
l	long	довге ціле	lAmount
b	boolean	булева змінна	bIsEmpty
a	array	масив	aDimensions
t, dt	time, datetime	час і дата	tDelivery, dtDelivery
p	pointer	вказівник	pBox
lp	long pointer	дальній вказівник	lpBox
r	reference	посилання	rBoxes
h	handle	дескриптор	hWindow
m_	member	змінна-член класу	m_sAddress
g_	global	глобальна змінна	g_nSpeed
C	class	клас	CString
T	type	тип	TObject
I	interface	інтерфейс	IDispatch
v	void	відсутність типу	vReserved
dbl	double	змінна з рухомою комою подвійної точності	dblSquare
fp	float	змінна з рухомою комою одинарної точності	fSquare

ch	char	буква або однобайтна змінна	chSymbol
u	unsigned	беззнакова змінна (застосовується сумісно з іншими цілочисельними типами)	uiCntr
fn	function	Функція	fnCreate
f	flag	Прапорець	fArrived
s_	static	Статичний член класу	s_Cntr
c_	static	Статичний член функції	c_Cntr

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке клас і з чого він складається?
2. Що таке private, public, protected і в чому між ними різниця?
3. Конструктор.
4. Деструктор.
5. Статичні властивості.
6. Статичні методи.
7. Списки ініціалізації.
8. Методи доступу до об'єкта.
9. Особливості класів, що містять динамічно створювані члени.
10. Угорська нотація.

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.
3. Угорська нотація [Електронний ресурс]. – режим доступу: http://uk.wikipedia.org/wiki/Угорська_нотація. – Назва з екрану.

ЗАВДАННЯ

Спроектувати клас, що реалізує предметну область згідно варіанту, та розробити програму-драйвер (main-функцію), яка демонструє роботу класу, а

саме: можливі способи створення об'єктів класів і коректність функціонування розроблених методів класу (запустити кожен метод класу хоча б 2 рази).

Варіант	Завдання
1	Клас CBox (Коробка). Клас має атрибути m_dbLength (довжина), m_dbWidth (ширина) і m_dbHeight (висота), кожний з яких за замовчуванням дорівнює 10. Він має: 1) методи запису та читання для атрибутів; методи запису повинні перевіряти, чи атрибути – числа з рухомою крапкою, що знаходяться в межах від 0.0 до 50.0. 2) методи, що обчислюють об'єм та площу поверхні; 3) три методи для визначення обхвату коробки.
2	Клас CCylinder (Циліндр). Клас має атрибути m_nRadius (радіус) і m_nheight (висота), кожний з яких за замовчуванням дорівнює 10. Він має: 1) методи запису та читання для атрибутів; методи запису повинні перевіряти, що атрибути – числа з плаваючою крапкою, що знаходяться в межах від 1.0 до 50.0. 2) методи, що обчислюють об'єм та площу поверхні; 3) три методи для визначення обхвату коробки.
3	Клас CQuadrangle (Чотирикутник). Клас зберігає декартові координати чотирьох кутів чотирикутника. Конструктор приймає чотири групи координат. Повинні бути передбачені методи, що обчислюють периметр і площу, а також метод, що перевіряє чи передані координати визначають прямокутник. Довжиною повинне вважатися більше із двох вимірів.
4	Клас CCircle (Коло). Клас зберігає декартові координати центра кола, а також значення точки на колі. Повинні бути передбачені методи, що обчислюють довжину кола, площу та діаметр, вивід на екран поточного стану об'єкта.
5	Клас CTriangle (Трикутник). Клас зберігає декартові координати трьох кутів трикутника. Конструктор приймає три групи координат. Повинні бути передбачені методи, що обчислюють периметр і площу, а також метод, що перевіряє чи трикутник є

	прямокутним і метод, що здійснює вивід на екран поточного стану об'єкта.
6	Клас CRobot (Робот). Клас зберігає назву моделі робота, поточний заряд енергії, завантаженість вантажного відсіку. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, забезпечувати рух в різні сторони зменшуючи заряд енергії, завантажувати і розвантажувати вантажний відсік враховуючи його фізичні обмеження, виводити на екран поточний стан об'єкта..
7	Клас SCoins (Монети). Клас зберігає масив із 3 монет та загальну кількість лицевих сторін після останнього підкидування. Повинні бути передбачені методи, що реалізують ініціалізацію та підкидування монет, вивід на екран поточного стану об'єкта.
8	Клас CDice (Гральні кості). Клас зберігає 3 гральні кості та загальну кількість очків після останнього кидання. Повинні бути передбачені методи, що реалізують ініціалізацію та кидання гральних костей, вивід на екран поточного стану об'єкта.
9	Клас CPlane (Літак). Клас зберігає назву рейсу, кількість палива, кількість пасажирів. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, забезпечувати рух в різні сторони зменшуючи кількість палива, посадку і висадку пасажирів, враховуючи фізичні обмеження літака і різні позаштатні ситуації (падіння через нехватку палива), виводити на екран поточний стан об'єкта.
10	Клас CMine (Шахта). Клас зберігає глибину, об'єм запасів шахти, кількість шахтарів, продуктивність праці одного шахтаря за зміну. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, збільшувати глибину шахти збільшуючи запаси руди, добувати руду за змінну враховуючи продуктивність шахтарів, виводити на екран поточний стан об'єкта.

11	Клас CTank (Танк). Клас зберігає розмір боєкомплекту, кількість палива, кількість вражених цілей, ймовірність попадання в ціль. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, робити постріл, визначати чи попали в ціль чи ні, переміщати танк зменшуючи при цьому кількість палива, виводити на екран поточний стан об'єкта.
12	Клас CBrowser (Інтернет-браузер). Клас зберігає адресу поточної відкритої сторінки, розмір сторінки, швидкість під'єднання до інтернету (завантаження і відвантаження). Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, завантажувати і відвантажувати файли з і на сторінку повертаючи при цьому час обміну інформації, виводити на екран поточний стан об'єкта.
13	Клас CCar (Машина). Клас зберігає назву машини, кількість палива, пробіг, вартість машини, вартість пробігу 1 км, на яку зменшується вартість машини. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, здійснювати рух машини зменшуючи при цьому запас палива, модифікувати вартість машини в залежності від пробігу, виводити на екран поточний стан об'єкта.
14	Клас CGasStation (Бензозаправка). Клас зберігає об'єм кожного з 4-ох сховищ пального, об'єм запасів кожного з 4-ох видів пального, пропускну здатність заправочного пістолета. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, збільшувати запаси палива для кожного виду, заправляти паливом зменшуючи при цьому об'єм палива і повертаючи час заправки, виводити на екран поточний стан об'єкта.
15	Клас CPhone (Телефон). Клас зберігає 5 останніх набраних номерів в послідовності їх набирання, номер, що набирається, час останнього дзвінка, вартість хвилини розмови. Конструктор має ініціалізувати ці властивості. Методи дозволяють

	модифікувати і читати властивості, здійснювати дзвінок на номер та проводити розмову з заданою тривалістю, розраховувати вартість останнього дзвінка, виводити на екран інформацію про розмови.
16	Клас CTV (Телевізор). Клас зберігає список каналів, поточний канал, гучність, ресурс телевізора, стан ввімкнення. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, вмикати чи вимикати телевізор, перемикає канал, збільшувати/зменшувати гучність, зменшувати ресурс телевізора при перегляді телепередач, виводити на екран інформацію про поточний стан телевізора.
17	Клас CLake (Озеро). Клас зберігає площу озера, об'єм води, глибину, швидкість випаровування води. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, наповнювати водойму, розраховувати час необхідний для її осушення, виводити на екран поточний стан об'єкта.
18	Клас CHuman (Людина). Клас зберігає ім'я людини, рік народження, вік, опановані вміння, зарплату, яка залежить від кількості вмінь. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, опановувати людині нові знання змінюючи при цьому її вік і зарплату, розраховувати скільки часу людині необхідно для заробляння певної суми грошей, виводити на екран поточний стан об'єкта.
19	Клас CShip (Корабель). Клас зберігає назву корабля, кількість палива, кількість пасажирів, вартість корабля, вартість пробігу 1 милі, на яку зменшується вартість корабля. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, здійснювати рух корабля зменшуючи при цьому запас палива і його вартість, садити і висаджувати пасажирів, виводити на екран поточний стан об'єкта.
20	Клас CBank (Банк). Клас зберігає назву банку,

	<p>поточну ставку по депозитах і кредитах у гривні на місяць, розмір виданих кредитів на місяць, вкладених депозитів на місяць і власних коштів банку, розмір відсотків які мають сплатити банку по кредитах і які має сплатити банк по депозитах. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, відкривати і забирати депозит з відсотками, брати і повертати кредит з відсотками, змінюючи при цьому прибуток/збиток банку, виводити на екран поточний стан об'єкта.</p>
21	<p>Клас CLocomotive (Локомотив). Клас зберігає модель локомотиву, кількість палива, споживання палива, відстань яку треба подолати, максимальну швидкість локомотиву, крок зменшення максимальної швидкості і збільшення споживання палива на кожен причеплений вагон, кількість причеплених вагонів, час в дорозі. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, здійснювати рух локомотива зменшуючи при цьому запас палива, розраховувати час, за який локомотив зможе пройти відповідну відстань з заданою кількістю вагонів і палива, дочіпляти і відчіпляти вагони, виводити на екран поточний стан об'єкта.</p>
22	<p>Клас CMachinegun (Автомат). Клас зберігає розмір боєкомплекту, ресурс, кількість вражених цілей, ймовірність попадання в ціль. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, заряджати/розряджати автомат, робити постріл, зменшуючи при цьому ресурс автомата, визначати чи попали в ціль чи ні, виводити на екран поточний стан об'єкта.</p>
23	<p>Клас CSoldier (Солдат). Клас зберігає ім'я солдата, звання, номер частини в якій він служить, дані про те чи він воював колись, кількість знищених ворогів, термін служби, бойовий досвід (днів). Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, змінювати звання</p>

	солдата в залежності від терміну служби і бойового досвіду, змінювати термін служби солдата і його бойового досвіду, виводити на екран поточний стан об'єкта.
24	Клас CComputer (Комп'ютер). Клас зберігає модель процесора, його продуктивність, розмір жорсткого диску, оперативної пам'яті, модель відеокарти, час експлуатації, кількість встановлених програм. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, встановлювати і видаляти програми, розраховувати скільки часу необхідно для виконання обчислень програмою, якщо відома кількість операцій, яку має виконати програма, виконувати програму, виводити на екран поточний стан об'єкта.
25	Клас CFlat (Квартира). Клас зберігає площу квартири, кількість людей, що в ній живуть і їх імена, термін експлуатації квартири, кількість спожитої електроенергії, води і газу. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, заселяти і відселяти мешканців, розраховувати вартість спожитих ресурсів, виводити на екран поточний стан об'єкта і вартість спожитих ресурсів.
26	Клас CTranslator (Перекладач). Клас зберігає ім'я перекладача, швидкість перекладу, знання мов. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, здійснювати переклад тексту, якщо перекладач знає дану мову, розраховувати час перекладу заданого тексту, додавати мови, які знає перекладач, виводити на екран поточний стан об'єкта.
27	Клас CPhotocamera (Фотоапарат). Клас зберігає розмір пам'яті фотоапарату, роздільну здатність знімку, кількість знімків, що зберігається на фотоапараті, коефіцієнт стиснення фотографії. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, здійснювати фотографування, якщо є вільна пам'ять,

	розраховувати степінь заповненості пам'яті фотоапарату, витирати знімки, виводити на екран поточний стан об'єкта.
28	Клас CPlant (Рослина). Клас зберігає назву рослини, її вік, висоту, стан спить/цвіте, швидкість росту, кількість генерації кисню за одиницю часу, час цвітіння. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, рости змінюючи при цьому вік і висоту, розраховувати скільки рослина генерує кисню за заданий проміжок часу, якщо вона цвіте, виводити на екран поточний стан об'єкта. Цвісти рослина може не довше, ніж задано часом цвітіння, після чого вона засинає на цей же час і потім знову просинається і цвіте.
29	Клас CBed (Ліжка). Клас зберігає назву моделі, кількість місць, кількість пружин і час експлуатації кожної пружини, кількість заміненних пружин, рівень комфорту у відсотках. Конструктор має ініціалізувати ці властивості. Методи дозволяють модифікувати і читати властивості, спати на ліжку впродовж заданого зменшуючи цим самим час експлуатації пружин, визначати на скільки відсотків комфортним був сон враховуючи, що кожна пружина, термін експлуатації якої вийшов зменшує рівень комфорту на 5%, замінювати пружини, виводити на екран поточний стан об'єкта.
30	Клас (Стадіон). Клас зберігає максимальну кількість місць, кількість проданих білетів, кількість проданих річних абонементів, ціну річного абонементу, середню ціну білету, стан газону у відсотках, вартість ремонту кожного відсотку газону, прибуток стадіону. Конструктор має ініціалізувати ці властивості, причому кількість річних абонементів має бути меншою максимальної кількості місць. Методи дозволяють модифікувати і читати властивості, продавати квитки і абонементи, проводити ігри на полі зношуючи поле. Якщо стан поля не задовільний (<50%), то гра не проводиться. Крім цього методи

	мають забезпечувати ремонт газону за рахунок прибутку стадіону.
--	---

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

```
// Rectangle.h: interface for the Rectangle class.
class CRectangle
{
private:
    double m_dblLength;
    double m_dblWidth;
public:
    //конструктор за замовчуванням
    CRectangle(double l = 1, double w = 1);
    // конструктор копіювання
    CRectangle(const CRectangle & rect);
    // деструктор
    virtual ~CRectangle();

    double GetLength();
    double GetWidth();
    void SetLength(double);
    void SetWidth(double);

    double Area();
    double Perimeter();
};

// Rectangle.cpp: implementation of the Rectangle class.
#include "Rectangle.h"
CRectangle::CRectangle(double l, double w )
```

```

{
    m_dblLength = l;
    m_dblWidth = w;
}
CRectangle::CRectangle(const CRectangle & rect)
{
    m_dblLength = rect.m_dblLength;
    m_dblWidth = rect.m_dblWidth;
}

CRectangle::~CRectangle()
{
}
double CRectangle::GetLength()
{
    return m_dblLength;
}

double CRectangle::GetWidth()
{
    return m_dblWidth;
}
void CRectangle::SetLength(double l)
{
    m_dblLength = l;
}
void CRectangle::SetWidth(double w)
{
    m_dblWidth = w;
}
double CRectangle::Area()
{
    return m_dblLength * m_dblWidth;
}
double CRectangle::Perimeter()
{
    return 2.0 * (m_dblLength + m_dblWidth);
}

// main.cpp

#include<iostream>
#include"Rectangle.h"
using namespace std;

```

```

int main()
{
    CRectangle rec1;
    cout << "rec1" << endl;
    cout << "length = " << rec1.GetLength() << endl;
    cout << "width = " << rec1.GetWidth() << endl;
    cout << "area = " << rec1.Area() << endl;

    CRectangle rec2(10,10);
    cout << "rec2" << endl;
    cout << "length = " << rec2.GetLength() << endl;
    cout << "width = " << rec2.GetWidth() << endl;
    cout << "area = " << rec2.Area() << endl;

    CRectangle rec5(rec2);
    cout << "rec5" << endl;
    cout << "length = " << rec5.GetLength() << endl;
    cout << "width = " << rec5.GetWidth() << endl;
    cout << "area = " << rec5.Area() << endl;

    CRectangle rec6;
    rec6 = CRectangle (6,3);
    cout << "rec6" << endl;
    cout << "length = " << rec6.GetLength() << endl;
    cout << "width = " << rec6.GetWidth() << endl;
    cout << "area = " << rec6.Area() << endl;

    CRectangle * rec7;
    rec7 = new CRectangle (2,3);
    cout << "rec6" << endl;
    cout << "length = " << rec7->GetLength() << endl;
    cout << "width = " << rec7->GetWidth() << endl;
    cout << "perimeter = " << rec7->Perimeter() << endl;
    delete rec7;

    CRectangle & rec8 = rec1;
    cout << "rec8" << endl;
    cout << "length = " << rec8.GetLength() << endl;
    cout << "width = " << rec8.GetWidth() << endl;
    cout << "perimeter = " << rec8.Perimeter() << endl;

    return 0;
}

```

Результати виконання:

```
rec1
length = 1
width = 1
area = 1
rec2
length = 10
width = 10
area = 100
rec5
length = 10
width = 10
area = 100
rec6
length = 6
width = 3
area = 18
rec6
length = 2
width = 3
perimeter = 10
rec8
length = 1
width = 1
perimeter = 4
```

ЛАБОРАТОРНА РОБОТА № 5. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

Мета: познайомитися із перевантаженням операторів.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Перевантаження операторів

Кожному оператору мова C++ ставить у відповідність ім'я функції, що складається з ключового слова `operator`, власне оператору та аргументів відповідних типів:

```
тип operator символОператору (списокПараметрів)
{
    //тіло методу
}
```

Щоб використовувати операцію над об'єктами класів, ця операція повинна бути перевантажена, але є два виключення. Операції присвоювання (=) і взяття адреси (&) створюються в класі автоматично за замовчуванням, тому їх можна використовувати без явного перевантаження. За замовчуванням операція присвоювання зводиться до побітового копіювання даних-елементів класу. Проте таке побітове копіювання небезпечне для класів з елементами, що вказують на динамічно виділені області пам'яті, масиви, рядки, оскільки в цьому випадку відбувається копіювання не даних (глибоке копіювання), а лише вказівників на дані (поверхневе копіювання). Для таких класів слід явно перевантажувати операцію присвоювання і здійснювати у ній глибоке копіювання. Операція адресації також може бути використана з об'єктами будь-яких класів без перевантаження. Вона просто повертає адресу об'єкта в пам'яті. Але операцію адресації можна також і перевантажувати.

Перевантаження операцій підпорядковується наступним правилам:

- При перевантаженні зберігаються кількість аргументів, пріоритети операцій та правила асоціації, що використовуються у стандартних типах даних;
- Для стандартних типів даних операції не підлягають перевизначенню;

- Перевантажена функція-оператор не може мати параметрів по замовчуванню, не успадковується та не може бути визначеною як `static`;
- Функція-оператор може бути визначена трьома способами – метод класу, дружня функція або звичайна функція. В останніх двох випадках вона повинна приймати хоча б один аргумент, що має тип класу, вказівника або посилання на клас.

При перевантаженні операцій `()`, `[]`, `->` та `=` функція перевантаження операції може бути оголошена лише як метод класу. Для інших операцій функції перевантаження операцій можуть не бути методами класу.

Оператори, які можна перевантажити:

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Оператори, які не можна перевантажити:

- `sizeof`
- `.` (селектор елемента структури або класу)
- `*` (оператор доступу до елемента за вказівником)
- `::` (оператор дозволу видимості)
- `? :` (тернарний оператор)
- `typeid`
- `const_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `static_cast`
- `# i ##` (символи препроцесору)

Коли операція реалізована як метод класу, тоді:

- якщо операція є унарною (передбачає один операнд, наприклад, інкременту або []), то лівим операндом вважається об'єкт, до якого застосовується операція, а правий операнд відсутній, тому метод, що реалізує даний оператор не приймає аргументів, за виключенням кількох операторів (наприклад, постфіксна форма інкременту або декременту).

- якщо операція є бінарною (передбачає 2 операнди, наприклад додавання або присвоєння) і лівий операнд є об'єктом класу у якому перевантажується операція, то крайній лівий операнд вважається об'єктом з-під якого здійснюється виклик даного методу (оператора), а правий операнд – передається як параметр, тому для нього слід вказати тип аргумента у методі; перевантажених операторів одного виду може бути кілька в залежності від типу аргументу, що передається методу;

Якщо операція є бінарною але лівий операнд не є об'єктом класу у якому перевантажується операція, то дана операція не може бути методом класу, а має бути реалізована як дружня функція, якщо ця функція повинна мати прямий доступ до закритих або захищених елементів цього класу, або звичайна функція в протилежному випадку.

Перевантажена операція << повинна мати лівий операнд типу ostream & (такий, як cout), так що вона не може бути функцією-елементом. Аналогічно, перевантажена операція >> повинна мати лівий операнд типу istream & (такий, як cin), так що вона теж не може бути функцією-елементом. До того ж кожна з цих перевантажених функцій-операцій може забажати доступу до закритих елементів-даних об'єкта класу, так що ці перевантажені функції-операції роблять функціями-друзями класу.

Будь-яку бінарну операцію можна перевантажувати як нестатичний метод з одним аргументом, або як функцію, що не є елементом, із двома аргументами (один з цих аргументів повинен бути або об'єктом класу, або посиланням на об'єкт класу).

Унарну операцію класу можна перевантажувати як метод без аргументів, або як функцію, з одним аргументом; цей аргумент повинен бути або об'єктом класу, або посиланням на об'єкт класу. Функції-елементи, що реалізують перевантажені операції, повинні бути нестатичними, щоб вони могли мати доступ до даних класу. Нагадаємо, що статичні методи можуть мати доступ тільки до статичних даних-елементів класу.

При перевантаженні унарних операцій переважно створюють методи класу, замість дружніх функцій, що не є членами класу. Дружніх функцій краще уникати доти, поки вони не стануть абсолютно необхідними. Використання друзів порушує інкапсуляцію класу.

Щоб перевантажити операцію інкремента та декремента для одержання можливості використання і префіксної, і постфіксної форм, кожна з цих двох перевантажених функцій-операцій повинна мати різну сигнатуру, щоб компілятор мав можливість визначити, яка версія маєтсья на увазі в кожному конкретному випадку. Префіксний варіант перевантажується як будь-яка інша префіксна унарна операція. Для постфіксної форми вводиться додатковий параметр цілого типу у список аргументів, щоб зробити функцію для постфіксного варіанту відмінною від функції для префіксної форми.

Зауваження щодо перевантаження операцій:

- Неможливим є введення власних операторів.
- Компілятор C++ не розуміє семантики перевантаженого оператора, а отже, не нав'язує жодних математичних концепцій. Можна перевантажити, скажімо, оператор інкременту в якості зменшення аргументу, проте навряд чи в цьому є сенс.
- Не існує виведення складних операторів з простих: якщо ви перевантажили оператори **operator+** та **operator=**, це зовсім не означає, що C++ обчислить вираз $a += b$, оскільки ви не перевантажили **operator +=**.
- Перевантаження бінарних операторів не тотожно відносно перестановки аргументів місцями, тим більше, якщо вони різного типу.

Синтаксис виклику операторів реалізований у мові C++ таким чином, щоб програміст міг записати операцію над об'єктом класу з звичному для нього вигляді, наприклад `object1 += object2`. Проте для компілятора такий запис стосовно об'єктів не є звичним, бо дана операція у такому вигляді визначена лише для простих типів, а для об'єкту класу даний виклик має бути перетворений у виклик методу, що оголошений в класі, або у виклик відповідної функції. Тому при компіляції компілятор перетворює даний виклик у виклик відповідного методу або функції за наступним принципом:

- якщо оператор реалізований у вигляді методу, то лівий операнд перетворюється в об'єкт з-під якого викликається метод, що реалізує вказаний оператор, а правий операнд, якщо він присутній, передається методу в якості аргументу.

- якщо оператор реалізований у вигляді функції або дружньої функції, то виклик операції перетворюється у виклик функції, яка приймає своїм лівим параметром лівий операнд операції, а правим, якщо такий присутній, - правий операнд операції.

Розглянемо приклади викликів операторів. Виклик оператору має наступний вигляд:

```
змінна символОператору [змінна];
```

Наприклад:

```
obj1 = obj2;  
obj1++;  
obj1 + 5;  
2 * obj1;
```

При цьому, якщо оператор визначений як метод класу, то перші 3 виклики будуть перетрансьювані компілятором в наступні виклики:

```
obj1.operator=(obj2);  
obj1.operator++();  
obj1.operator+(5);
```

Останній виклик реалізується виключно у вигляді дружньої або звичайної функції і після обробки компілятором буде перетворений у наступний виклик дружньої функції:

```
operator*(2, obj1);
```

Типові підходи до реалізації операторів

Розглянемо деякі стандартні зразки перевантаження операторів. Наступні приклади операторів реалізуються у вигляді методів класу.

```
// Оператор присвоєння  
Тип& operator= (const Тип& obj)  
{  
    if(&obj == this) // перевірка на самоприсвоєння  
    {  
        return *this;  
    }  
    else  
    {  
        // алгоритм присвоєння  
    }  
}
```

```

        return *this;
    }

// Оператор додавання
Тип operator+(const Тип & obj) const
{
    Тип об'єкт;
    об'єкт.властивість1 = властивість1 + obj.властивість1;
    об'єкт.властивість2 = властивість2 + obj.властивість2;
    ...
    об'єкт.властивістьN = властивістьN + obj.властивістьN;
    return об'єкт;
}

// Оператор віднімання
Тип operator-(const Тип & obj) const
{
    Тип об'єкт;
    об'єкт.властивість1 = властивість1 - obj.властивість1;
    об'єкт.властивість2 = властивість2 - obj.властивість2;
    ...
    об'єкт.властивістьN = властивістьN - obj.властивістьN;
    return об'єкт;
}

// Оператор множення
Тип operator*(const Тип & obj) const
{
    Тип об'єкт;
    об'єкт.властивість1 = властивість1 * obj.властивість1;
    об'єкт.властивість2 = властивість2 * obj.властивість2;
    ...
    об'єкт.властивістьN = властивістьN * obj.властивістьN;
    return об'єкт;
}

// Оператор ділення
Тип operator/(const Тип & obj) const
{
    Тип об'єкт;
    об'єкт.властивість1 = властивість1 / obj.властивість1;
    об'єкт.властивість2 = властивість2 / obj.властивість2;
    ...
    об'єкт.властивістьN = властивістьN / obj.властивістьN;
    return об'єкт;
}

```

```

// Оператор додавання з присвоєнням
Тип operator+=(const Тип & obj)
{
    властивість1 = властивість1 + obj.властивість1;
    властивість2 = властивість2 + obj.властивість2;
    ...
    властивістьN = властивістьN + obj.властивістьN;
    return *this;
}

// Оператор віднімання з присвоєнням
Тип operator-=(const Тип & obj)
{
    властивість1 = властивість1 - obj.властивість1;
    властивість2 = властивість2 - obj.властивість2;
    ...
    властивістьN = властивістьN - obj.властивістьN;
    return *this;
}

// Оператор унарного мінусу
Тип operator-() { return -властивість1;}

// Оператор модуль від числа
int operator%(int iVal) { return властивість1 % iVal;}

// Оператор менше
bool operator<(Тип val) { return властивість1 < val;}

// Оператор більше
bool operator>(Тип val) { return властивість1 > val;}

// Оператор менше рівне
bool operator<=(Тип val) { return властивість1 <= val;}

// Оператор більше рівне
bool operator>=(Тип val) { return властивість1 >= val;}

// Оператор І
Тип operator&&(Тип val) { return властивість1 && val;}

// Оператор Або
Тип operator||(Тип val) { return властивість1 || val;}

// Оператор заперечення
bool operator!() { return властивість1 == 0;}

```

```

// Оператор звертання за адресою
Тип* operator->()
{
    return m_ptr;    // властивість-вказівник, яка
                    // зберігається в об'єкті
}

// Оператор звертання до елементу масиву
Тип& operator[] (int index)
{
    assert (index >= 0 && index < m_size);
    return m_arr[index];
}

// Оператор постфіксного інкременту
Тип& operator++ ()
{
    // інкрементування
    return *this;
}

// Оператор префіксного інкременту
Тип operator++ (int)
{
    Тип result(*this);    // робимо копію об'єкту
    ++(*this);    // використовуємо префіксну форму для інкременту
    return result;    // повернути старе значення.
}

// Оператор префіксного декременту
Тип& operator-- ()
{
    // декрементування
    return *this;
}

// Оператор постфіксного декременту
Тип operator-- (int)
{
    Тип result(*this);    // робимо копію об'єкту
    --(*this);    // використовуємо префіксну форму для декременту
    return result;    // повернути старе значення.
}

```

```

// Оператор створення елементу
void* operator new(size_t size)
{
    // дії
    return ptr;
}

// Оператор створення масиву
void* operator new[](size_t size)
{
    // дії
    return ptr;
}

// Оператор видалення елементу
void operator delete(void* ptr)
{
    if (ptr)
    {
        // дії по видаленню.
    }
}

// Оператор видалення масивів
void operator delete[] (void* ptr)
{
    if (ptr)
    {
        // дії по видаленню.
    }
}

// Оператор приведення типу
operator Тип()
{
    return властивість1;
}

// Оператор (), який дозволяє створити об'єкт-функцію
Тип operator (Аргументи)
{
    // дії над аргументами
    return значення;
}

```

Наступні приклади операторів реалізуються тільки у вигляді дружніх функцій, оскільки з лівої сторони оператора стоїть не об'єкт класу, для якого перевантажується оператор.

```
ostream& operator<<(ostream& output, const Тип& obj)
{
    // вивід на екран
    return output;
}

istream& operator>>(istream& input, Тип& obj)
{
    // ввід з клавіатури
    return input;
}
```

Дружні конструкції

Дріжніми до класу можуть бути: класи, методи та функції. Якщо якийсь з даних елементів має бути дружнім до класу, от в цьому класі його слід оголосити як дружній. Для цього слід зробити:

- У випадку якщо дружнім має бути клас, то цей клас треба оголосити дружнім в тілі класу, до якого він має бути дружнім за допомогою ключового слова `friend`, наступним чином

```
friend class someClass;
```

У цьому випадку всі елементи дружнього класу матимуть доступ до закритих і захищених елементів класу, до якого вони є дружніми. Оскільки тут `someClass` ідентифікується за допомогою ключового слова `class` як клас, то випереджуючого оголошення для компілятора робити не потрібно. Приклад оголошення дружнього класу:

```
class a
{
    friend class b;
    ...
};
```

```
class b
{
    ...
};
```

- У випадку якщо дружнім має бути метод класу, то цей метод треба оголосити дружнім в тілі класу, до якого він має бути дружнім за допомогою ключового слова `friend`, наступним чином

```
friend Тип ДружнійКлас::НазваМетоду(Аргументи);
```

У цьому випадку лише вказаний метод класу матиме доступ до закритих і захищених елементів класу, до якого він є дружнім. Оскільки з одної сторони класи з дружніми методами переважно оголошуються після класів, до яких вони мають бути дружніми, але з іншої сторони клас, що містить дружній метод, має бути відомим до моменту оголошення методу дружнім, то перед оголошенням класу, до якого метод класу має бути дружніми слід вжити випереджаюче оголошення класу, що містить дружній метод. Інакше компілятор видасть помилку, бо для нього ім'я класу, що містить дружній метод буде невідомим. Наприклад

```
class b;      // випереджаюче оголошення класу b

class a
{
    friend int b::activate(a& obj, int val);
    ...
};

class b
{
    int activate(a& obj, int val)
    ...
};
```

- У випадку якщо дружньою має бути функція, то цю функцію треба оголосити дружньою в тілі класу, до якого вона має бути дружньою за допомогою ключового слова `friend`, наступним чином

```
friend Тип НазваФункції(Аргументи);
```

У цьому випадку вказана функція матиме доступ до закритих і захищених елементів класу. Приклад оголошення дружньої функції:

```
class a
{
    friend int deactivate(a& obj, int val);
    ...
};

int deactivate(a& obj, int val)
{
    //Тіло функції
}
```

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке перевантаження оператора?
2. Синтаксис оголошення перевантаженого оператора?
3. Які оператори можна перевантажувати?
4. Які оператори не можна перевантажувати?
5. Які способи перевантаження операторів ви знаєте?
6. Які оператори можна перевантажити лише як методи класу?
7. Які оператори можна перевантажити лише як дружні функції?
8. Наведіти приклади перевантаження операторів.
9. Які дружні конструкції ви знаєте?
10. В чому відмінності між дружніми класами, методами та функціями?

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. – 800 с.

ЗАВДАННЯ

Розширити функціональність розроблених у 4 лабораторній роботі класів за допомогою операторів, що задані варіантом та оператора присвоювання. Конкретні функції операторів реалізувати на власний розсуд (крім оператора присвоювання). Організувати виведення та введення даних за допомогою класів-потоків `cin`, `cout` та перевантажених операторів вводу/виводу. Написати програму, яка демонструє роботу з об'єктами цього класу.

Варіант	Оператори
1	+, -, <, >
2	+=, -=, <=, >=
3	*, +, -, >
4	+, -, ==, !=
5	(), ==, !=, ++
6	+=, -=, (), --
7	%, ++, (), ==
8	++, +=, >, <
9	+, -, (), >
10	*, /, --, ==
11	+=, -=, ==, >
12	+=, -=, >, <
13	--, +=, -=, ==
14	+, --, !=, []
15	+, ==, (), []
16	++, >, <, +=
17	+, -=, <, >
18	*, +=, >, <
19	+=, -=, ==, >
20	+=, -=, >=, <
21	--, +=, >, <
22	+=, -=, ==, >
23	+=, %, ==, !
24	*, ++, ==,
25	+=, --, !=, &&

26	+=, ++, ==, []
27	++, -=, >, <
28	++, !, , +=
29	+, -, !, ==
30	+=, -=, ==,

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

```
// Array.h: interface for the Array class.
#include<iostream>
#include<stdlib.h>
#include<time.h>
using namespace std;

class Array
{
    int* ptr;
    int size;
public:
    Array(int s = 10);
    Array(Array& arr);
    virtual ~Array();
    friend ostream& operator<< (ostream& output, Array& arr);
    void Rindomize(int num = 10);
    Array& operator= (Array& arr);
    Array& operator+= (Array& arr);
    Array operator+ (Array& arr) ;
    int operator!= (Array& arr);
    Array operator++ ();
    Array operator++ (int);
};
```

```
// Array.cpp: implementation of the Array class.  
#include "Array.h"
```

```
Array::Array(int s)  
{  
    size = s;  
    ptr = new int[size];  
    for(int i = 0; i < size; i++)  
    {  
        ptr[i] = 0;  
    }  
    cout << "Constructor" << endl;  
}
```

```
//-----
```

```
Array::Array(Array& arr)  
{  
    size = arr.size;  
    ptr = new int[size];  
    for(int i = 0; i < size; i++)  
    {  
        ptr[i] = arr.ptr[i];  
    }  
    cout << "Copy Constructor" << endl;  
}  
//-----
```

```
Array::~~Array()  
{  
    delete[] ptr;  
    cout << "Destructor" << endl;  
}
```

```
//-----
```

```
void Array::Rindomize(int num)  
{  
    for(int i = 0; i < size; i++)  
    {  
        ptr[i] = rand() % num;  
    }  
}
```

```
//-----
```

```
ostream& operator<< (ostream& output, Array& arr)  
{  
    for(int i = 0; i < arr.size; i++)  
    {
```

```

        output << arr.ptr[i] << " ";
    }
    output << endl;
    return output;
}

```

```

//-----
Array& Array::operator= (Array& arr)
{

```

```

    if(this != &arr)
    {
        delete[] ptr;
        size = arr.size;
        ptr = new int[size];
        for(int i = 0; i < size; i++)
        {
            ptr[i] = arr.ptr[i];
        }
    }
    cout << "Operator =" << endl;
    return *this;
}

```

```

//-----
int Array::operator!= (Array& arr)
{

```

```

    if(size != arr.size)
        return 1;
    for(int i = 0; i < size; i++)
    {
        if(ptr[i] != arr.ptr[i])
            return 1;
    }
    return 0;
}

```

```

//-----
Array Array::operator+ (Array& arr)
{

```

```

    int mins = (size < arr.size) ? size : arr.size;
    Array temp;
    if(mins == arr.size)
    {
        temp = *this;
        for(int i = 0; i < mins; i++)
        {
            temp.ptr[i] += arr.ptr[i];
        }
    }
}

```

```

    }
    else
    {
        temp = arr;
        for(int i = 0; i < mins; i++)
        {
            temp.ptr[i] += ptr[i];
        }
    }
    cout << "Operator +" << endl;
    return temp;
}
//-----
Array& Array::operator+= (Array& arr)
{
    Array temp;
    temp = *this + arr;
    *this = temp;
    return *this;
}
//-----
Array Array::operator++ ()
{
    for(int i = 0; i < size; i++)
        ptr[i] += 1;
    return *this;
}
//-----
Array Array::operator++ (int)
{
    Array temp = *this;
    for(int i = 0; i < size; i++)
        ptr[i] += 1;
    return temp;
}

// Main.cpp : main program
#include "Array.h"
int main()
{
    srand(time(NULL));

    Array a(5);
    a.Rindomize(5);
    Array b(7);

```

```

    b.Rindomize(5);
    Array c;

    cout << "Array a:" << endl << a;
    cout << a++;
    cout << a;

    cout << "Array b:" << endl << b;

    c = a + b;
    cout << "Array c:" << endl << c;
    return 0;
}

```

Результати виконання:

```

Constructor
Constructor
Constructor
Array a:
0  1  4  0  3
Copy Constructor
Copy Constructor
Destructor
0  1  4  0  3
Destructor
1  2  5  1  4
Array b:
2  0  1  3  2  2  4
Constructor
Operator =
Operator +
Copy Constructor
Destructor
Operator =
Destructor
Array c:
3  2  6  4  6  2  4
Destructor
Destructor
Destructor
Press any key to continue

```

ЛАБОРАТОРНА РОБОТА № 6. СПАДКУВАННЯ

Мета: познайомитися із спадкуванням класів.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Спадкування (ієрархія "is a")

Спадкування – це механізм, за допомогою якого один клас може одержувати атрибути та функціональність іншого. Спадкування дозволяє створювати ієрархію класів.

При створенні нового класу, що повністю дублює існуючий клас і дещо розширяє його новими властивостями і функціональністю програміст може не дублювати існуючий клас і дописувати в нього розширену функціональність, а вказати, що новий клас є спадкоємцем елементів попередньо визначеного класу і визначити у ньому лише необхідну нову функціональність. В цьому випадку існуючий клас, функціональність якого розширюється у новому класі, називається **базовим класом** (base class). Новостворений клас називається **похідним класом** (derived class), або **спадкоємцем**. Кожен похідний клас може бути використаним у ролі базового класу для майбутніх похідних класів створюючи при цьому **дерево спадкування**, яке ще називають **ієрархією спадкування класів** (class hierarchy). Спадкування прийнято відображати у вигляді графу (дерева) у напрямку зверху-вниз. При цьому клас, що є у самому верху є самим першим базовим класом і називається **кореневим класом** або **коренем дерева спадкування класів**. Похідний клас, через проміжний, може наслідувати характеристики базового класу. У цьому випадку говорять, що **базовий клас є непрямым базовим класом (indirect base class) для похідного**. Зокрема, корінь дерева наслідувань є непрямым базовим класом для усіх класів, які знаходяться нижче першого рівня ієрархії. Клас, який При **одиначному спадкуванні** (single inheritance) клас породжується одним базовим класом. При **множинному спадкуванні** (multiple inheritance) похідний клас успадковує властивості декількох базових класів, причому можлива ситуація коли один базовий клас буде успадкований кілька разів по кількох гілках. При створенні об'єкта похідного класу в пам'яті зберігаються копії усіх класів, які **становлять вітку, що породила даний клас**.

Похідний клас наслідує характеристики базового через **специфікатор доступу (access specifier) : "довкранка"**. Нижче наведено синтаксис спадкування базового класу:

```
class ім'яПохідногоКласу : [public/protected/private] ім'яБазовогоКласу
{
    // тіло похідного класу
}
```

За допомогою специфікатора доступу можна визначити, яким чином елементи базового класу будуть успадковуватися похідним класом. При відкритому спадкуванні (використанні специфікатора `public`) у похідному класі члени базового класу мають ті ж специфікатори доступу, що й у базовому класі. При захищеному спадкуванні (використанні специфікатора `protected`) у похідному класі відкриті члени базового класу стають захищеними, а інші зберігають своє початкове значення специфікатора доступу. Нарешті, при закритому спадкуванні (використанні специфікатора `private`) у похідному класі всі члени базового класу стають закритими. Узагальнена таблиця впливу специфікатора доступу наведена у табл. 6.1.

Таблиця 6.1. Особливості різних видів спадкування

Властивість	Відкрите спадкування	Захищене спадкування	Закрите спадкування
Загальнодоступні елементи стають	Загальнодоступними елементами похідного класу	Захищеними елементами похідного класу	Закритими елементами похідного класу
Захищені елементи стають	Захищеними елементами похідного класу	Захищеними елементами похідного класу	Закритими елементами похідного класу
Закриті елементи стають	Доступними через інтерфейс базового класу	Доступними через інтерфейс базового класу	Доступними через інтерфейс базового класу
Неявне висхідне перетворення	Так	Так (але в межах породженого класу)	Ні

Кожного разу, коли з батьківського класу створюється похідний клас, то похідний клас успадковує окремі або усі риси батьківського, **додаючи свої** або **розширюючи батьківські**. Усі спадковані характеристики похідний клас використовує як свої власні. Кінцевий користувач класу не розрізняє успадкованих характеристик і маніпулює ними за допомогою оператора (.) як звичайними членами об'єкту, наприклад,

```
object.metod(parameters);
```

Об'єкт похідного класу оперує з власними та наслідуваними характеристиками через оператор (: :) без жодних обмежень, згідно з правами доступу до них. Якщо є потреба явно вказати клас, якому належить даний член, то використовують оператор (: :) з іменем класу, наприклад:

```
ім'яБазовогоКласу::Метод(параметри);  
ім'яПохідногоКласу::Метод(параметри);
```

Не успадковуються:

1. Конструктор
2. Деструктор
3. Оператор присвоєння
4. Дружні конструкції
5. Конструкції створені за допомогою оператора new.

Конструктори при одинарному спадкуванні

При створенні об'єкту класу *автоматично викликається його конструктор*. Якщо об'єкт породжується від похідного класу, то, очевидно, при створенні об'єкту похідного класу повинен викликатись і конструктор базового класу. У цьому випадку порядок виклику і виконання конструкторів є таким:

1. при створенні об'єкта похідного класу генерується виклик конструктора цього класу, але сам конструктор не виконується;
2. після генерації виклику конструктора похідного класу генерується виклик і виконується конструктор базового класу;
3. по закінченню виконання тіла конструктора базового класу виконується тіло конструктора похідного класу.

У випадку багатократного спадкування наведений порядок виклику і виконання конструкторів зберігається. Глибина вкладень конструкторів при цьому може бути довільною. Порядок виклику конструкторів є послідовним по мірі ієрархічного спадкування і строго фіксується.

За допомогою специфікатора доступу у визначенні конструктора похідного класу *можна передати параметри конструктору базового класу*. Очевидно, при цьому базовий клас повинен містити відповідний конструктор, який здатний прийняти такий параметр. Синтаксис оголошення конструктора похідного класу з передачею параметрів конструктору базового класу наступний:

```
ім'яКласу :: ім'яКласу (параметри) : ім'яБазовогоКласу(параметри)
{
    // тіло конструктора
}
```

Конструктор базового типу не спадкується, проте він викликається компілятором на етапі побудови об'єкта похідного класу. Явний виклик конструктора базового класу не допускається. Можна лише за допомогою специфікатора доступу і списку параметрів базового конструктора керувати вибором конструктора базового класу, який запуститься на виконання при створенні об'єкту похідного класу. При визначенні конструктора можна передавати аргументи конструктору лише *верхнього сусіднього класу*. Конструкторам класів, які не є сусідніми в ієрархії, аргументи можна передавати лише *через визначення конструкторів проміжних класів*. Після того як об'єкт побудовано конструктор базового класу стає недоступним.

Деструктори при одинарному спадкуванні

Деструктори викликаються подібно до конструкторів, а виконуються у зворотному порядку. Порядок виклику і виконання деструкторів є таким:

1. для знищення об'єкта похідного класу викликається деструктор похідного класу і виконується його основне тіло;
2. по закінченню виконання основного коду деструктор похідного класу не закривається, а генерує виклик деструктора базового класу;
3. після виклику виконується тіло деструктора базового класу,
4. по закінченню виконання деструктор базового класу закривається і передає керування деструктору похідного класу;
5. останнім закривається деструктор похідного класу.

Деструктори викликаються автоматично у встановленому порядку. Порядок викликів і виконання деструкторів зберігається в будь-яких ієрархіях класів.

Деструктори мають обов'язково бути віртуальними, якщо тільки клас не має використовуватися як базовий. У протилежному випадку буде застосовуватися статичне зв'язування при поліморфізмі і оперуванні похідними класами через вказівники чи посилання на базовий клас. Це призведе до того, що при знищенні об'єкта похідного класу деструктор похідного класу не буде викликатися, що в свою чергу може призводити до помилок функціонування програми.

Приведення типів

Класи, що входять до складу дерева спадкування не є розрізненими, а ієрархічно між собою зв'язані. Тому стосовно них легко здійснювати операцію приведення типів визначену правилом приведення класів при простому спадкуванні: *класи, які знаходяться ієрархічно нижче, автоматично приводяться до класів, які знаходяться ієрархічно вище на одній вітці дерева спадкування*. Наведене правило справджується для явного та неявного приведення типів.

Якщо оголошено вказівник на базовий клас, то його можна розглядати як вказівник на будь-який похідний клас породжений від базового і у цьому випадку помилка приведення типів не буде генеруватись. При цьому об'єкт похідного класу, на який посилається вказівник базового класу, автоматично приводиться до типу похідного класу.

Після присвоєння адреси об'єкта похідного класу вказівнику на базовий клас, ним можна скористатись як вказівником на цей об'єкт. При цьому забезпечується доступ лише до членів базового класу і членів класів, що знаходяться у ієрархії вище, ніж клас типу якого є вказівник. Доступ до членів похідного класу з-під такого вказівника чи посилання є заборонений.

C++ допускає використання вказівника на базовий клас чи посилання на базовий клас з об'єктом похідного класу, ***але зворотній напрям застосування вказівників і посилань є забороненим*** (див. лістинг нижче). Це обмеження можна обійти використовуючи явне приведення типів (див. лістинг нижче). Операція інкрементація вказівника базового класу, який вказує на об'єкт похідного класу, приведе до посилання на інший об'єкт базового, а не похідного класу.

Лістинг демонстраційної програми

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

class CBase{ // базовий клас

private:
    int m_bnum;
    int m_common_num;

protected:
    int m_prot;

public:
    void SetNum(const int n) {m_bnum = n;}
    int GetNum() const { return m_bnum;}
    void Show() const {cout<< m_bnum;}
    void SetCommonNum(const int n) { m_common_num = n;}
    int GetCommonNum() const { return m_common_num;}
    void ShowCommon() const { cout<<m_common_num;}

    CBase(int n=0){ m_bnum =n;} //нема сенсу робити віртуальним,
    бо конструктор не успадкується
    virtual ~CBase(){} // для коректного виклику деструктора при
    використанні вказівників базового класу зв'язаних з об'єктами
    похідного класу
};

class CDeriv: public CBase // похідний клас від CBase
{
private:
    int m_dnum;
    int m_my_common;

public:
    CDeriv(int n=0, int k=0): CBase(k), m_dnum(n) {} // ініціуючий
    конструктор
    CDeriv(const CBase & ob) {//копіювання}
    virtual ~CDeriv() {}

    void SetNum(const int n) { m_dnum = n;}
    int GetNum() const { return m_dnum;}
    void Show() const {cout<< m_dnum;}

    void SetCommonNum(const int n) { m_my_common = n;}
```

```

int GetCommonNum() { return m_my_common; }
void ShowCommon() const { cout<< m_my_common; }

//void SetBaseNum(int n) { m_bnum = n; } //- помилка
//void SetBaseNum(int n) { CBase::m_bnum = n; } //- помилка
//void SetBaseNum(int n) { SetNum( n ); } //- помилка
void SetBaseNum(int n) { CBase::SetNum( n ); } /* - ОК.
Керування доступом через оператор доступу (::)*/

void SetProtected(const int n) { m_prot = n; }
};

void main ()
{
    CDeriv der;
    CBase bas;

    CBase & bs = der;
    CBase * pbs = &der;
    //CDeriv * pder = &bas; //- error
    CDeriv * pder = (CDeriv*)&bas; //- ОК

    bas.SetNum(5); // CBase::m_bnum = 5
    bas.SetCommonNum(7); // CBase::m_common_num = 7

    der.SetNum(5); //CDeriv::m_dnum = 5
    der.CBase::SetNum(5); //CBase::m_bnum = 5
    der.SetCommonNum(7); //CDeriv::m_my_common = 7

    bs.SetNum(3); // CBase::m_dnum = 3
    bs.SetCommonNum(4); // CBase::m_common_num = 4

    pbs->SetProtected(6); // error
    pbs->SetNum(5); // CBase::m_dnum = 5

    pder->SetProtected(6); // CBase::m_prot=6
}

```

Перевизначення елементів класів

Перевизначення членів класу (overriding) полягає у визначенні членів класу (методів, властивостей) у похідному класі, що мають такі ж сигнатури, що й члени базового класу. У цьому випадку компілятор при визначенні того для якого класу (одного з базових чи похідного) згенерувати код виклику елементу класу у тій чи іншій ситуації користується **правилом домінування**.

Воно полягає в тому, що для знаходження потрібного члена компілятор переглядає дерево ієрархії спадкування знизу догори і генерує код для виклику члену класу, який є найближчим до похідного класу у дереві спадкування. Це відбувається завдяки тому, що ідентифікатор в похідному класі перекриває (або перевизначає) однаковий ідентифікатор оголошений в базовому класі. Якщо потрібно, здійснити виклик елементу класу з класу, який не є найближчим до похідного класу у дереві спадкування, то для цього слід перед викликом цього елементу вказати назву класу і оператор розширення області видимості (: :):

```
БазовийКлас :: metod(parameters);  
БазовийКлас :: властивість;
```

Але С++ не допускає використання декількох операторів (: :) при виклику одного члена. Це означає, що звертаючись до члена несусіднього класу однієї вітки не можна записати усю послідовність класів аж до похідного. Якщо мова іде про методи класу, то для забезпечення правила домінування вони повинні мати однакові сигнатури.

Якщо слід звернутися до елементу, який знаходиться на рівень вище від поточного домінуючого елементу, а не через кілька рівнів, то достатньо вказати лише оператор розширення області видимості (: :) перед назвою елементу.

Переозначення (overriding) не є перевантаженням, оскільки однаковість сигнатур не забезпечує виконання умов перевантаження. Якщо сигнатура функції-члена похідного класу є відмінною від сигнатури функції базового класу, то одержуємо звичайне перевантаження функції-елемента похідного класу.

Аби запобігти неоднозначностям, рекомендовано перевизначати лише віртуальні методи класів. Функція оголошується віртуальною за допомогою ключового слова `virtual`, що передує прототипу функції в базовому класі.

Рекомендації до застосування спадкування:

- **Перш ніж будувати новий клас необхідно визначитись**, чи може він бути похідним від наперед існуючого. Іноді вже існують класи, які мають необхідні характеристики. Очевидно, що такі класи можна використати як батьківські, а наслідування при цьому може бути вибіркоким.

- **Рекомендується спадкувати максимальну кількість характеристик.** Це суттєво зменшує обсяг коду, полегшує відлагодження та зменшує кількість синтаксичних і логічних помилок.

Абстрактні базові класи і конкретні класи

Коли ми думаємо про клас як про тип, ми припускаємо, що будуть створюватися об'єкти цього типу. Однак, існують випадки, в яких корисно визначати класи, для яких програміст не має наміру створювати об'єкти. Такі класи називаються **абстрактними класами**. Оскільки вони застосовуються лише як базові класи в процесі спадкування і не можуть бути використані для створення об'єктів напряду без спадкування, то їх прийнято називати **абстрактними базовими класами**. Таким чином об'єкти абстрактного базового класу не можуть бути створені.

Єдиним призначенням абстрактного класу є створення відповідного базового класу, від якого інші класи можуть успадкувати інтерфейс і реалізацію. Класи, об'єкти яких можуть бути реалізовані, називаються **конкретними класами**.

Клас робиться абстрактним шляхом оголошення хоча б однієї його віртуальної функції чисто віртуальною. **Чисто віртуальна функція** - це функція, у якої в її оголошенні тіло визначене як «= 0». Наприклад:

virtual тип ім'яМетода (параметри) = 0;

Для того, щоб об'єкт класу, що спадкується від абстрактного класу можна було створити слід перевизначити усі чисто віртуальні функції абстрактних базових класів. В іншому випадку, якщо клас є похідним від класу з чистою віртуальною функцією і якщо ця чисто віртуальна функція не описана в похідному класі, то функція залишається чисто віртуальною й у похідному класі. А отже і похідний клас залишається абстрактним класом.

Ієрархія не вимагає обов'язкового використання абстрактних класів. Але на практиці багато програм, що використовують об'єктно-орієнтований підхід, мають ієрархію класів, породжену абстрактним базовим класом. У деяких випадках абстрактні класи складають кілька верхніх рівнів ієрархії.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке спадкування?
2. Різниця між закритим, захищеним і відкритим спадкуванням?

3. Різниця між одинарним і множинним спадкуванням.
4. Конструктор при одинарному спадкуванні.
5. Деструктор при одинарному спадкуванні.
6. Що таке базовий клас?
7. Що таке похідний клас?
8. Перевизначення елементів класів.
9. Приведення типів.
10. Що таке абстрактний базовий клас?

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.

ЗАВДАННЯ

Створити абстрактний базовий клас і похідний від нього клас, які реалізують модель предметної області згідно варіанту. Кожен клас має мати мінімум 3 власні елементи даних один з яких створюється динамічно, методи встановлення і читання характеристик елементів-даних класу (Set і Get), та мінімум 2 абстрактні методи обробки даних і мінімум 2 методи обробки даних у похідному класі. Крім цього клас має містити перевантаження оператора присвоєння, конструкторів по замовчуванню і копіювання та віртуальний деструктор. Для розроблених класів реалізувати програму-драйвер, яка демонструє роботу класів.

Варіант	Завдання	
	Абстрактний базовий клас	Похідний клас
1	Магазин	Супермаркет
2	Людина	Лікар
3	Комаха	Комар
4	Тварина	Скунс
5	Машина	Вантажна машина
6	Літак	Бомбардувальник

7	Комп'ютер	Ноутбук
8	Годинник	Будильник
9	Рослина	Дерево
10	Будинок	Офісний центр
11	Пристрій	Монітор
12	Водойма	Море
13	Пристрій	Телефон
14	Пристрій	Телевізор
15	Корабель	Круїзний лайнер
16	Пристрій	Диктофон
17	Пристрій	Відеоплеєр
18	Пристрій	Копіювальний апарат
19	Машина	Бронетранспортер
20	Костюм	Військовий костюм
21	Трактор	Комбайн
22	Двері	Двері з кодовим замком
23	Мікросхема	Процесор
24	Кип'ятильник	Електрочайник
25	Кондиціонер	Пристрій кліматконтролю
26	Шлюпка	Сторожовий човен
27	Грядка	Теплиця
28	Сумка	Чемодан
29	Село	Місто
30	Вагон	Рейковий автобус

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

```
// Device.h
#include <stdlib.h>
#include <string.h>

class CDevice
{
public:
    CDevice(char* fName);
    virtual ~CDevice();
    virtual bool Open() = 0;
    virtual bool Close() = 0;
    virtual bool Execute(char* cmd, void* prm) = 0;
    virtual bool Status(int ext=0)
    {
        return m_bIsOpened;
    }

protected:
    char* m_pszDeviceName;
    char* m_pszFriendlyName;
    bool m_bIsOpened;
};

//Device.cpp
#include "Device.h"

CDevice::CDevice(char* fName)
{
    m_pszFriendlyName = new char[strlen(fName)+1];
    strcpy(m_pszFriendlyName, fName);
    m_pszDeviceName = 0;
}

CDevice::~~CDevice()
{
    if(m_pszDeviceName!=0)
    {
        delete [] m_pszDeviceName;
    }
    if(m_pszFriendlyName!=0)
    {
        delete [] m_pszFriendlyName;
    }
}
```

```

    }
}

//Scanner.h
#include <stdlib.h>
#include <string.h>
#include "Device.h"

class CScanner : public CDevice
{
public:
    CScanner(char* fName);
    virtual ~CScanner();
    virtual bool Open();
    virtual bool Close();
    virtual bool Execute(char* cmd, void* prm);
private:
    char* m_pszScrBuf;
};

```

```

//Scanner.cpp
#include "Scanner.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

CScanner::CScanner(char* fName) : CDevice(fName)
{
    m_pszScrBuf = 0;
    srand(time(0));
}

CScanner::~~CScanner()
{
    if(m_pszScrBuf!=0)
    {
        delete [] m_pszScrBuf;
    }
}

bool CScanner::Open()
{
    m_pszDeviceName = new char[30];
    cin >> m_pszDeviceName;
}

```

```

        m_bIsOpened = true;
        cout << "Device is ready for work." << endl;
        return true;
    }

    bool CScanner::Close()
    {
        m_bIsOpened = false;
        cout << "Device finished the work." << endl;
        return true;
    }

    bool CScanner::Execute(char* cmd, void* prm)
    {
        if(strcmp(cmd, "Scan") == 0)
        {
            m_pszScrBuf = new char[128];
            for(int i = 0; i < 127; i++)
            {
                m_pszScrBuf[i] = rand() % 26 + 65;
            }
            m_pszScrBuf[127] = '\0';
        }

        if(strcmp(cmd, "Read") == 0)
        {
            strcpy((char*)prm, m_pszScrBuf);
        }

        if(strcmp(cmd, "Clear") == 0)
        {
            m_pszScrBuf[0] = '\0';
        }
        return true;
    }

//main.cpp
#include "Scanner.h"
#include <iostream>
using namespace std;

int main()
{
    char* strinf = new char[128];
    strcpy(strinf, " ");

```

```
CScanner scanner("My Scanner");

cout << "Enter Scanner's model: ";
scanner.Open();

cout << scanner.Status() << endl;
scanner.Execute("Scan",strinf);

cout << strinf << endl;

scanner.Execute("Read",strinf);

cout << strinf << endl;

scanner.Execute("Clear",strinf);

scanner.Execute("Read",strinf);
cout << strinf << endl;

scanner.Close();
cout << scanner.Status() << endl;

}
```

ЛАБОРАТОРНА РОБОТА № 7. МНОЖИННЕ СПАДКУВАННЯ. ПОЛІМОРФІЗМ

Мета: познайомитися із множинним спадкуванням класів та поліморфізмом.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Множинне спадкування

Якщо спадкування здійснюється від декількох батьківських класів одночасно, тоді воно називається *множинним спадкуванням*. Визначальним для похідного класу породженого множинним спадкуванням є те, що він явно чи неявно повинен успадковувати характеристики декількох базових класів.

Основні принципи одинарного спадкування, зокрема спадкування членів, модифікаторів доступу до членів базових класів, розширення та обмеження характеристик, без жодних доповнень можуть бути перенесені на множинне спадкування.

Неявним множинним спадкуванням можуть бути випадки змішаного спадкування. Результатом цих спадкувань є ієрархія, в якій похідний клас неявно (через один проміжний) успадкував характеристики двох базових класів.

Якщо похідний клас породжується від декількох базових, то в декларації класу треба вказати усі базові класи, розділяючи їх комою, разом зі специфікаторами спадкування. У загальному випадку синтаксис множинного спадкування має вигляд:

```
class Ім'яПохідногоКласу : [модифікатор] Ім'яБазовогоКласу1, ...,  
[модифікатор] Ім'яБазовогоКласуN {тіло класу};
```

Розглянемо узагальнений приклад множинного спадкування:

```
class A {оголошення класу};  
class B { оголошення класу };  
class C : public A { оголошення класу };  
class D: public B, public C { оголошення класу };    // аналогічно  
оголошенню class D: public B, C{ оголошення класу };
```

Відповідна схема утвореної ієрархії класів матиме наступний вигляд (див. рис. 7.1).

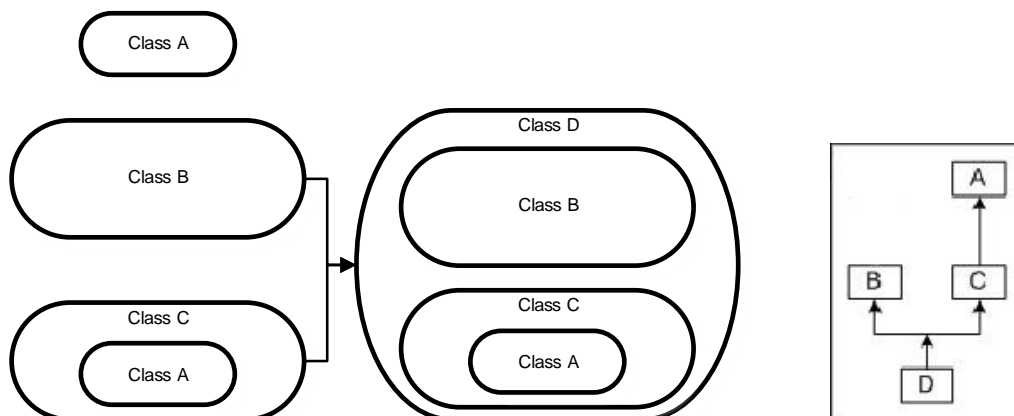


Рис.7.1. Схема ієрархії класів при множинному спадкуванні.

Як видно з даної схеми клас В є самостійним класом, клас А успадковується одинарно класом С, а клас D множинно успадковує класи В і С, при цьому клас D, опосередковано через клас С успадковує клас А. Тож при створенні об'єкту класу D він міститиме в собі всі характеристики об'єктів класів А, В і С. При цьому оскільки кожен з об'єктів входить в об'єкт D один раз, то ніяких конфліктів при створенні об'єкту D не буде.

Конструктори при множинному спадкуванні

Розглянемо алгоритм роботи конструкторів при множинному спадкуванні. При створенні об'єкта класу, який множинно породжений, після виклику конструктора похідного класу **викликатиметься конструктор найпершого батьківського класу**. Якщо він є похідним від ще одного класу, то викликатиметься і буде виконуватись конструктор останнього. По закінченню роботи усіх конструкторів по гілці дерева від найпершого класу, розпочне виконуватись гілка від другого батьківського класу і т.д. Після того, як відпрацюють конструктори усіх батьківських класів, виконається тіло конструктора похідного класу.

Механізм передавання аргументів конструкторам базових класів повністю підтримується при множинному спадкуванні, наприклад:

```
C::C(char c1, char c2, char c3): B(c2), A(c3)
{ m_cPub=c1; Show(mes_cosntr, m_cPub); }
```

Тут аргументи c_2 і c_3 передаються конструкторам базових класів В і А відповідно.

Порядок передавання аргументів конструкторам в оголошенні конструктора похідного класу може бути довільним, оскільки виклики і виконання конструкторів визначаються порядком спадкування в оголошенні класу.

Як і у випадку одинарного спадкування при визначенні конструктора можна передавати аргументи лише конструкторам виключно базових класів. Конструкторам класів, які не є безпосередньо в ієрархії, аргументи передаються лише через визначення конструкторів проміжних класів. Наприклад, наступне визначення конструктора є неправильним, якщо $O()$ є батьківським класом тільки класу А або В.

```
C::C(char c1, char c2, char c3): B(c2), A(c3), O()  
{  
    m_cPub=c1;  
    Show(mes_cosntr, m_cPub);  
}
```

Деструктори при множинному спадкуванні

Порядок виклику деструкторів є таким як у конструкторів, а виконання - зворотнім. Найпершим почне виконуватись деструктор похідного класу, а далі - деструктори гілки породженої останньою в оголошенні батьківським класом. У порядку зворотному до декларації батьківських класів відпрацюють деструктори класів усіх гілок від них породжених. Лише по закінченню роботи і закритті батьківських деструкторів закриється деструктор похідного класу.

Віртуальні базові класи

Розглянемо наступний приклад множинного спадкування:

```
class A { /*оголошення класу*/ };  
class B : public A { /*оголошення класу*/ };  
class C : public A { /*оголошення класу*/ };  
class D: public B, public C { /*оголошення класу*/ };
```


Відповідна схема утвореної ієрархії класів матиме наступний вигляд (див. рис. 7.2).

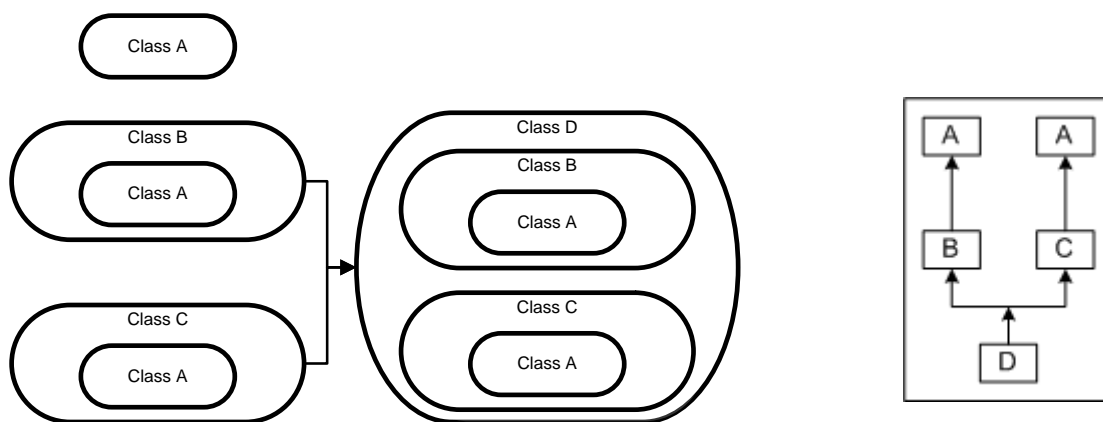


Рис.7.2. Схема ієрархії класів при дублюванні входження класу А у клас D.

Як видно з даної схеми клас А успадковується по кожній з двох гілок класом D. Таким чином клас D містить 2 об'єкти класу А. Дана ситуація є неоднозначною, оскільки при зверненні до членів класу А невизначено по якій з гілок слід звертатися до об'єкту класу А. Тому клас D не можна безпечно створити і компілятор в цій ситуації видасть помилку компіляції.

Однак, C++ дає змогу уникнути дублювання копій класів в множинних ієрархіях через механізм **віртуальних базових класів** (virtual base class). Віртуальні базові класи використовуються лише при множинному спадкуванні. Їх використання дає змогу полегшити контролювання над складними взаємозв'язками, які виникають при множинному спадкуванні.

Тип базового класу як віртуального визначає наявність модифікатора `virtual` в оголошенні похідного класу. Наявність зарезервованого слова `virtual` попереджає компілятор, що похідний клас є породжений від проміжних класів, які мають спільний батьківський клас. Це дає можливість уникнути неоднозначних викликів і зекономити пам'ять на створенні лише одного об'єкта класу А.

При використанні віртуальних класів треба явно викликати в полі ініціалізації конструктора класу, що породжений множинним спадкуванням, необхідний конструктор базового віртуального класу і передавати туди необхідні параметри (що є не обов'язковим при звичайному одиночному чи множинному наслідуванні).

Розглянемо приклад використання віртуальних базових класів.

```

class A { /*оголошення класу*/ };
class B : virtual public A { /*оголошення класу*/ };
class C : virtual public A { /*оголошення класу*/ };
class D: public B, public C {
    ...
    D(char c1, char c2): A(c2)    // явний виклик конструктора класу A
    {
        m_cPub=c1;
        Show(mes_cosntr, m_cPub);
    }
    ...
    // оголошення класу
};

```

Відповідна схема утвореної ієрархії класів матиме наступний вигляд (див. рис. 7.3).

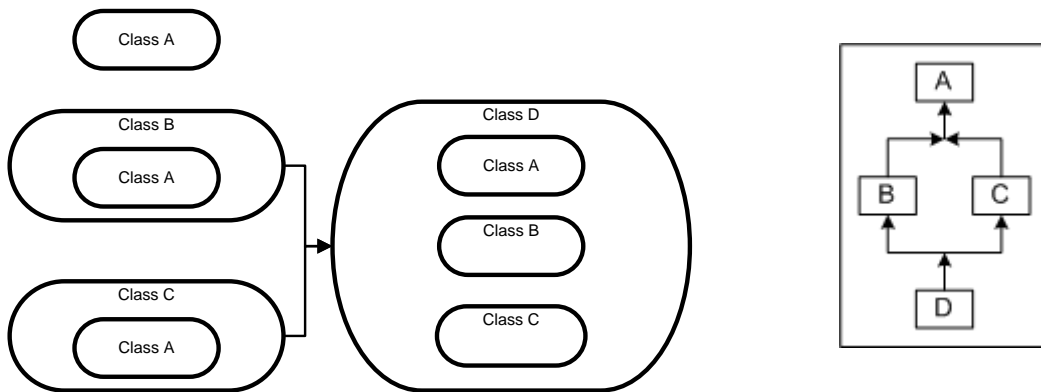


Рис.7.3. Схема ієрархії класів при використанні віртуального класу А.

Розглянемо особливості віртуальних базових класів. **Конструктори віртуальних класів завжди викликаються першими** незалежно від порядку оголошення. Проте, коли віртуальних класів є декілька, то порядок їх оголошення визначатиме порядок виклику і виконання їх конструкторів.

C++ допускає створювати ієрархій, в яких **сумісно можна використовувати віртуальні і невіртуальні класи**. Наявність невіртуального класу визначатиме принаймні дві копії об'єкта цього класу. Оголошувати

віртуальні і невіртуальні класи можна в будь-якому порядку, але найпершим викликатиметься конструктор віртуального базового класу.

Виклики членів базових класів

Для звертання до будь-яких членів базових класів при множинному спадкуванні як і при одинарному спадкуванні слід використати оператор (::) з іменем класу. Цей підхід є найпростішим і, очевидно, найефективнішим. Більшість логічних суперечностей, які виникають при маніпуляціях з успадкованими членами в похідних класах, розв'язуються за допомогою оператора (::).

Правило домінування, яке визначало порядок виклику успадкованих членів при простому спадкуванні, в цілому підтримується при множинному спадкуванні, але його застосування вимагає додаткових розумових зусиль. ***Правило домінування застосовується лише стосовно тих класів, які породжують одне одного***, тобто є сусідніми (Подібно до одинарного спадкування при множинному спадкуванні здійснюється автоматичне створення зв'язків від об'єктів нижчих класів до об'єктів вищих. Єдиною проблемою можуть стати неоднозначності батьківських зв'язків). При множинному спадкуванні цю умову треба особливо відслідковувати. В іншому випадку правило домінування не можна застосовувати і треба повертатись до використання оператора (::).

Якщо два класи породжені один від одного, то член похідного класу є домінуючим. Правило пошуку відповідного члена для виклику відбувається на стадії компіляції. У випадку, коли правило домінування не спрацьовує треба використовувати механізм (::) і явно вказувати ідентифікатор класу, якому належить даний член.

Поліморфізм

Поліморфізм – це здатність коду при постійному інтерфейсі змінювати свою поведінку в залежності від ситуації, яка виникає на момент виконання. Іншими словами один і той же метод може бути визначений для об'єктів різних класів, що є між собою в ієрархії спадкування, при цьому метод якого класу викликати вирішується під час виконання програми.

Поліморфізм широко застосовується при створенні складних бібліотек класів та програмних систем. Один продуманий інтерфейс може бути з успіхом використаний у різних ситуаціях, що зменшує складність програмної структури завдяки уніфікації інтерфейсу. Спрощення створення програмного

інтерфейсу дає можливість розробнику "тримати" в пам'яті меншу кількість інформації.

Загалом розрізняють два види поліморфізму: *статичний (static або compilation-time polymorphism)* і *динамічний (run-time polymorphism)*.

Статичний поліморфізм реалізовується за допомогою так званого *раннього зв'язування* через механізм перевантаження функцій, методів та операторів і віртуальні класи. Він притаманний класичним структурним мовам програмування. При використанні статичного поліморфізму вибір функції чи методу, що буде викликатися здійснюється компілятором при компіляції програми (раннє зв'язування). Вибір функції чи методу в даному випадку залежить від типу вказівника чи посилання і не залежить від типу реального об'єкту на який вказує вказівник чи посилається посилання. Тобто при звертанні до методу об'єкту похідного класу, використовуючи вказівник чи посилання на базовий клас, викликається буде метод базового класу. Раннє зв'язування реалізується наступним чином. Компілятор на основі вихідного коду використовує фіксовані ідентифікатори функцій і методів. На етапі компонування ці ідентифікатори замінюються фізичними адресами відповідних функцій і методів. Раннє зв'язування характеризується високою швидкістю виконання програми, оскільки єдиними витратами в період виконання є: передавання аргументів, виконання виклику функції або методу та очищення стеку. Основним недоліком раннього зв'язування є втрати гнучкості виконуваного коду.

Динамічний поліморфізм реалізовується за допомогою так званого *пізнього зв'язування* через механізм віртуальних функцій. Він притаманний об'єктно-орієнтованим мовам програмування і може бути застосований лише до методів класів. При використанні динамічного поліморфізму вибір методу, що буде викликатися здійснюється в процесі виконання програми (пізнє зв'язування). Вибір методу в даному випадку залежить від типу реального об'єкту на який вказує вказівник чи посилається посилання. Тобто при звертанні до методу об'єкту похідного класу, використовуючи вказівник чи посилання на базовий клас, викликатися буде метод похідного класу. Динамічний поліморфізм дозволяє значно підвищити ефективність і гнучкість поведінки програми, але ціною за це є сповільнення виконання програми, оскільки в процесі виконання програмі додатково необхідно приймати рішення який метод у якій ситуації викликати. Тому використання механізму пізнього зв'язування має сенс лише за умови існування ієрархії класів. В іншому випадку єдине що ми отримаємо – це сповільнення виконання програми.

Пізнє зв'язування при одинарному спадкуванні реалізується за допомогою таблиці віртуальних функцій (ТВФ) та вказівника `vptr`. Для

кожного класу, що містить віртуальні функції в процесі роботи програми створюється тільки одна ТВФ. Вказівник `vptr` в компіляторах компанії Microsoft завжди міститься за нульовим зміщенням відносно початку об'єкта в усіх об'єктах, що мають віртуальні функції незалежно від того чи то об'єкти базового чи похідного класу і часто називається `__vfptr`, а сама ТВФ називається `vftable`. Таблиця віртуальних функцій – це масив, кожен елемент якого містить вказівник на метод. Отримати доступ до вказівника `vptr` засобами мови C++ неможливо. ТВФ створюється за допомогою конструктора класу. Якщо класи знаходяться в одинарній ієрархії, то спочатку конструктор створює ТВФ для базового класу і в неї включаються всі віртуальні функції базового класу, а вказівнику `vptr` присвоюється адреса цієї таблиці. Віртуальні методи у ТВФ вносяться в порядку слідування, тому компілятор завжди може однозначно визначити де знаходиться той чи інший метод не зберігаючи його назви. Далі викликається конструктор для похідного класу і створюється копія ТВФ базового класу в якій при необхідності змінюються адреси перевизначених віртуальних методів та додаються нові знову ж таки в порядку слідування починаючи з індекса на якому закінчилася індексація віртуальних методів базового класу, а `vptr` присвоюється адреса цієї ТВФ. Процес продовжується доти, поки не створиться останній об'єкт похідного класу. При виклику віртуального методу компілятор генерує код звернення за вказівником `vptr` до ТВФ, за допомогою черговості оголошення методу визначає індекс методу в ТВФ і в необхідне місце програми вставляє код непрямого виклику методу за адресою, що міститься в ТВФ за відповідним зміщенням (у відповідній комірці масиву, який реалізує ТВФ).

При знищенні об'єкту процес відбувається у зворотньому порядку відносно створення об'єкту – спочатку знищується об'єкт похідного класу, а `vptr` присвоюється адреса об'єкту на рівень вище в ієрархії класів і т.д. поки не знищиться найвищий в ієрархії об'єкт класу.

У випадку якщо використовується множинне спадкування, то для кожного класу, що містить множинне спадкування додаються стільки вказівників `vptr`, скільки класів спадкує похідний класу. Це робиться для того, щоб кожен з базових класів можна було використати в якості вказівника чи посилання.

Мова C++ є гібридною мовою програмування в плані підтримки поліморфізму, оскільки в ній реалізована підтримка як статичного так і динамічного поліморфізму. Тож відповідно до типу зв'язування в C++ розрізняють звичайні функції і методи – функції і методи раннього зв'язування і поліморфні або віртуальні методи (*virtual methods*) - методи пізнього зв'язування.

Для того, щоб оголосити віртуальний метод треба перед оголошенням методу поставити ключове слово `virtual`. Для того, щоб метод став віртуальним достатньо вжити ключове слово `virtual` лише раз у всій ієрархії спадкування. Зазвичай це робиться у базовому класі, а в похідних класах цей метод перевизначається як звичайний метод класу.

Віртуальна функція є членом класу, а тому може викликатись як звичайний метод класу.

Розглянемо приклад статичного і динамічного поліморфізму.

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

class A
{
public:
    A() {}
    virtual ~A() {}
    void meth1() {cout<<"A class"<<endl;}
    virtual void meth2 () {cout<<"A class"<<endl;}
};

class B : public A
{
public:
    B() {}
    virtual ~B() {}
    void meth1 () {cout<<"B class"<<endl;}
    void meth2() {cout<<"B class"<<endl;}
};

int main()
{
    A* a = new A();
    B* b = new B();
    A* d = b;
```

```

cout << "Overloaded method" << endl;
a-> meth1 ();
b-> meth1 ();
d-> meth1 ();
cout << endl << "Virtual method" << endl;
a-> meth2();
b-> meth2 ();
d-> meth2 ();

delete a;
delete b;
return 0;
}

```

Результати виконання:

Overloaded method

A class

B class

A class

Virtual function

A class

B class

B class

Press any key to continue

У наведеному прикладі при виклику методу `meth1()` за вказівником на об'єкт вибір методу буде залежати **тільки від типу вказівника**. Метод `meth2()` у базовому класі оголошений як віртуальний. Тому при його виклику за вказівником на об'єкт вибір методу буде залежати **від типу об'єкта, на який вказує вказівник**, і аж ніяк не від типу вказівника.

Відзначимо відмінності між перевантаженням методу `meth1()` і перевизначенням методу `meth2()`. Вибір перевантаженого методу здійснюється під час компіляції у залежності від набору параметрів. При перевантаженні методи можуть мати різні типи й набори параметрів та відмінні типи значень, що будуть повертати. А от якщо метод оголошений як

virtual, то всі його перевизначення в похідних класах повинні мати однаковий набір параметрів і однаковий тип значення, що буде повертатись. *Інакше, якщо в похідному класу оголосити метод з відмінною сигнатурою від тої, що визначена в базовому класі, то метод похідного класу «сховає» відповідний метод базового класу і звернутися до нього за старою сигнатурою буде неможливо, тому **перевизначення успадкованих методів не є різновидом перевантаження**.*

Віртуальний метод не обов'язково повинен перевизначатися в похідному класі. У такому випадку для цього класу використовується версія методу, визначена в базовому класі.

Віртуальний механізм працює лише за допомогою вказівників і посилань на об'єкти. Об'єкт, що містить віртуальні функції, та визначений через вказівник або посилання, носить назву поліморфного. У даному випадку поліморфізм полягає у тому, що за допомогою одного й того ж звертання до методу виконуються різні дії в залежності від типу об'єкта, на який посилається вказівник у даний момент часу.

Розглянемо наступний приклад:

```
#include <iostream>
#include <cstring>
using namespace std;

class Animal
{
protected:
    char* m_pszName;
public:
    Animal() {m_pszName = new char[100];}
    virtual ~Animal() { delete[] m_pszName;}
    virtual void call() = 0;
};

class Dog : public Animal
{
public:
    Dog(const char* name) { strcpy(this-> m_pszName, name);}
```



```

    virtual ~Dog() {}
    virtual void call() {cout<< m_pszName <<": Gav - Gav"<<endl;}
};

class Cat : public Animal
{
public:
    Cat(const char* name) {strcpy(this-> m_pszName, name); }
    virtual ~Cat() {}
    virtual void call() {cout<< m_pszName <<": May - May"<<endl;}
};

int main()
{
    Animal* animals[5];
    animals[0] = new Dog("Bobik");
    animals[1] = new Cat("Murka");
    animals[2] = new Dog("Juchka");
    animals[3] = new Dog("Mos'ka");
    animals[4] = new Cat("Simka");

    cout<<"Disturb all animals ..."<<endl;
    for(int i = 0; i < 5; i++)
    {
        if(animals[i] != NULL)
            animals[i]->call();
    }
    for(i = 0; i < 5; i++)
    {
        if(animals[i] != NULL)
            delete animals[i];
    }
    return 0;
}

```

Результати виконання:

Disturb all animals ...

Bobik: Gav - Gav

Murka: May - May

Juchka: Gav - Gav

Mos'ka: Gav - Gav

Simka: May - May

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке множинне спадкування?
2. Синтаксис множинного спадкування.
3. Конструктори при множинному спадкуванні.
4. Деструктор при множинному спадкуванні.
5. Віртуальні базові класи.
6. Виклики членів базових класів при множинному спадкуванні.
7. Що таке поліморфізм?
8. Статичний поліморфізм.
9. Динамічний поліморфізм.
10. Що таке таблиця віртуальних функцій?

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.

ЗАВДАННЯ

Спроектувати і реалізувати ієрархію класів, що описують предметну область згідно варіанту, яка реалізується класом 1. Клас 1 в свою чергу утворюється шляхом множинного спадкування класів 2 і 3 кожен з яких в свою чергу успадковує клас 4. Додаткові вимоги:

1. Базовий клас містить мінімум один віртуальний метод, один невіртуальний метод і одну динамічно створювану властивість.

2. Забезпечити механізми коректної роботи конструкторів і деструкторів.
3. Перевантажити оператор присвоєння з метою його коректної роботи.
4. Кожен з класів має містити мінімум одну властивість і 4 методи.
5. Написати `main()` функцію де створити об'єкт класу 1 і продемонструвати різницю між статичним і динамічним поліморфізмом.

Варіант	Завдання				
	Предметна область	Клас 1	Клас 2	Клас 3	Клас 4
1	Відеодвійка	CTVVideo	CTV	CVideoPlayer	CDevice
2	Електронна книжка	CEBook	CAudioBook	CBook	CDevice
3	Граючий тренер	CPlayingCoach	CCoach	CPlayer	CHuman
4	Багатофункціональний пристрій	CMFU	CPrinter	CScanner	CDevice
5	Хворий лікар	CPatientDoctor	CDoctor	CPatient	CHuman
6	Машина-амфібія	CAmphibian	CCar	CBoat	CTransport
7	Літак-штурмовик	CAssaultPlane	CFighter	CBomber	CPlane
8	Аудіо-відео плеєр	CAudioVideoPlayer	CVideoPlayer	CAudioPlayer	CDevice
9	Батько-працівник	CFatherWorker	CWorker	CFather	CHuman
10	Штурмова гвинтівка	CAssaultRifle	CBazooka	CMachineGun	CGun
11	Багатофункціональна машина	CMultiCar	CTruck	CCar	CTransport
12	Смартфон	CSmartphone	CComputer	CPhone	CDevice
13	Радіогодинник	CRadioClock	CRadio	CClock	CDevice
14	Розкладний стіл	CFoldingTable	CCommode	CTable	CFurniture
15	Гарнітура	CHeadSet	CHeadPhone	CMicrophone	CDevice
16	Військовий водолаз	SMilitaryDiver	CSoldier	CDiver	CHuman
17	Годинник з зозулькою	CCuckooClock	CCuckoo	CClock	CDevice
18	Гібридний двигун	CMixedEngine	CGasEngine	CDieselEngine	CEngine
19	Ліхтарик з електрошокером	CShockingLight	CElectroshocker	CFlashlight	CDevice
20	Клавіатура з мишкою	CKeyboardMouse	CMouse	CKeyboard	CDevice
21	Трансформер	CTransformer	CCar	CRobot	CMachine
22	Торгово-розважальний центр	CShoppingLeisureCenter	CEntertainmentCenter	CShoppingCenter	CBuilding
23	Колонка з флешплеєром	CMixedSpeaker	CFlashPlayer	CLoudspeakers	CDevice
24	Газоелектричний бойлер	CMixedBoiler	CElectricBoiler	CGasBoiler	CBoiler
25	Шорти-штани	CPantsShorts	CShorts	CPants	CCloth
26	Пральна машина з годинником	CClockedWasher	CClock	CWasher	CDevice
27	Кікбоксер	CKickBoxer	CKicker	CBoxer	CSportsman
28	Саджально-копальна машина	CDiggingDibblingMachine	CDigMachine	CDibblingMachine	CMachine
29	Бармен-офіціант	CBarmanWaiter	CWaiter	CBarman	CEmployee
30	Водій-охоронець	CBodyGuardDriver	CDriver	CBodyGuard	CEmployee

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

Спроекувати і реалізувати ієрархію класів, що описують предметну область «Автомобілогонщик» (CAutoMotoRacer), що змагається як у авто гонках (описується класом CAutoRacer) і мотоциклетних гонках (описується класом CMotoRacer). Класи CAutoRacer і CMotoRacer мають спільний батьківський клас CRacer.

Реалізація

```
//Racer.h
#include <stdlib.h>
#include <iostream>
#include <cstring>
using namespace std;

#ifndef Racer
#define Racer
class CRacer
{
protected:
    char* m_pszName;
    int m_nExperience;
public:
    CRacer();
    CRacer(char* name);
    CRacer(const CRacer& obj);
    virtual ~CRacer();

    void SetExperience(int);
    int GetExperience();
};
```

```

    void ShowProperty();
    virtual void ShowSkill() = 0;
    virtual void ShowFullInformation() = 0;
    CRacer& operator=(const CRacer&);
};
#endif

```

```

//AutoRacer.h
#include "Racer.h"
#include <cstring>
#ifndef AutoRacer
#define AutoRacer
class CAutoRacer : virtual public CRacer
{
protected:
    char* m_pszCarName;
public:
    CAutoRacer();
    CAutoRacer(char* name, char* carName );
    CAutoRacer(const CAutoRacer& obj);
    virtual ~CAutoRacer();

    void ShowProperty();
    virtual void ShowSkill();
    virtual void ShowFullInformation();
    CAutoRacer& operator=(CAutoRacer&);
};
#endif

```

```

//MotoRacer.h
#include "Racer.h"
#include <cstring>
#ifndef MotoRacer
#define MotoRacer
class CMotoRacer : virtual public CRacer
{
protected:
    char* m_pszBikeName;
public:
    CMotoRacer();
    CMotoRacer(char* name, char* bikeName );
    CMotoRacer(const CMotoRacer& obj);
    virtual ~CMotoRacer();

    void ShowProperty();

```

```

        virtual void ShowSkill();
        virtual void ShowFullInformation();
        CMotoRacer& operator=(CMotoRacer&);
};
#endif

//AutoMotoRacer.h
#include "AutoRacer.h"
#include "MotoRacer.h"
#ifndef AMRacer
#define AMRacer
class CAutoMotoRacer : public CAutoRacer, CMotoRacer
{
public:
    CAutoMotoRacer();
    CAutoMotoRacer(char* name, char* bikeName, char* carName) ;
    CAutoMotoRacer(const CAutoMotoRacer& obj);
    virtual ~CAutoMotoRacer();

    void ShowProperty();
    virtual void ShowSkill();
    virtual void ShowFullInformation();
    CAutoMotoRacer& operator=(CAutoMotoRacer&);
};
#endif

//Racer.cpp
#include "Racer.h"

CRacer::CRacer(): m_pszName (new char[20]), m_nExperience (0)
{
    cout << "CRacer Default Constructor" << endl;
    strcpy(m_pszName, "No name");
}

CRacer::CRacer(char* name)
{
    cout << "CRacer Constructor" << endl;
    m_pszName = new char[strlen(name)+1];
    strcpy(m_pszName, name);
}

CRacer::CRacer(const CRacer& obj)
{
    cout << "CRacer Copy Constructor" << endl;

```

```

    m_pszName = new char[strlen(obj.m_pszName)+1];
    strcpy(m_pszName, obj.m_pszName);
}

CRacer::~CRacer()
{
    cout << "CRacer Destructor" << endl;
    delete[] m_pszName;
}

void CRacer::SetExperience(int exp)
{
    m_nExperience = exp;
}

int CRacer::GetExperience()
{
    return m_nExperience;
}

void CRacer::ShowProperty()
{
    cout << "I have no properties" << endl;
    cout << "-----" << endl;
}

CRacer& CRacer::operator=(const CRacer& obj)
{
    if (this != &obj)
    {
        cout << "CRacer operator=" << endl;
        delete[] m_pszName;
        m_pszName = new char[strlen(obj.m_pszName)+1];
        strcpy(m_pszName, obj.m_pszName);
    }
    return *this;
}

//AutoRacer.cpp
#include "AutoRacer.h"

CAutoRacer::CAutoRacer() : CRacer()
{
    cout << "CAutoRacer Default Constructor" << endl;
    m_pszCarName = new char[20];
}

```

```

        strcpy(m_pszCarName, "No name");
    }

CAutoRacer::CAutoRacer(char* name, char* carName ): CRacer(name)
{
    cout << "CAutoRacer Constructor" << endl;
    m_pszCarName = new char[strlen(carName)+1];
    strcpy(m_pszCarName, carName);
}

CAutoRacer::CAutoRacer(const CAutoRacer& obj): CRacer(obj)
{
    cout << "CAutoRacer Copy Constructor" << endl;
    m_pszCarName = new char[strlen(obj.m_pszCarName)+1];
    strcpy(m_pszCarName, obj.m_pszCarName);
}

CAutoRacer::~CAutoRacer()
{
    cout << "CAutoRacer Destructor" << endl;
    delete[] m_pszCarName;
}

void CAutoRacer::ShowProperty()
{
    cout << "I have car: " << m_pszCarName << endl;
}

void CAutoRacer::ShowSkill()
{
    cout << "I can ride a car" << endl;
}

void CAutoRacer::ShowFullInformation()
{
    cout << "My name is " << m_pszName << ", my experience is " <<
m_nExperience << " years" << endl;
    ShowProperty();
    ShowSkill();
    cout << "-----" << endl;
}

CAutoRacer& CAutoRacer::operator=(CAutoRacer& obj)
{
    if (this != &obj)

```



```

{
    cout << "CAutoRacer operator=" << endl;
    delete[] m_pszCarName;
    m_pszName = new char[strlen(obj.m_pszCarName)+1];
    strcpy(m_pszCarName, obj.m_pszCarName);
    CRacer::operator=(obj);
}
return *this;
}

//MotoRacer.cpp
#include "MotoRacer.h"
CMotoRacer::CMotoRacer() : CRacer()
{
    cout << "CMotoRacer Default Constructor" << endl;
    m_pszBikeName = new char[20];
    strcpy(m_pszBikeName, "No name");
}

CMotoRacer::CMotoRacer(char* name, char* bikeName ): CRacer(name)
{
    cout << "CMotoRacer Constructor" << endl;
    m_pszBikeName = new char[strlen(bikeName)+1];
    strcpy(m_pszBikeName, bikeName);
}

CMotoRacer::CMotoRacer(const CMotoRacer& obj): CRacer(obj)
{
    cout << "CMotoRacer Copy Constructor" << endl;
    m_pszBikeName = new char[strlen(obj.m_pszBikeName)+1];
    strcpy(m_pszBikeName, obj.m_pszBikeName);
}

CMotoRacer::~CMotoRacer()
{
    cout << "CMotoRacer Destructor" << endl;
    delete[] m_pszBikeName;
}

void CMotoRacer::ShowProperty()
{
    cout << "I have a bike:" << m_pszBikeName << endl;
}

```

```

void CMotoRacer::ShowSkill()
{
    cout << "I can ride a bike" << endl;
}

void CMotoRacer::ShowFullInformation()
{
    cout << "My name is " << m_pszName << ", my experience is " <<
m_nExperience << " years" << endl;
    ShowProperty();
    ShowSkill();
    cout << "-----" << endl;
}

CMotoRacer& CMotoRacer::operator=(CMotoRacer& obj)
{
    if (this != &obj)
    {
        cout << "CMotoRacer operator=" << endl;
        delete[] m_pszBikeName;
        m_pszName = new char[strlen(obj.m_pszBikeName)+1];
        strcpy(m_pszBikeName, obj.m_pszBikeName);
        CRacer::operator=(obj);
    }
    return *this;
}

//AutoMotoRacer.cpp
#include "AutoMotoRacer.h"

CAutoMotoRacer::CAutoMotoRacer() : CRacer(), CAutoRacer(), CMotoRacer()
{
    cout << "CAutoMotoRacer Default Constructor" << endl;
}

CAutoMotoRacer::CAutoMotoRacer(char* name, char* bikeName, char*
carName) : CRacer(name), CAutoRacer(name, carName), CMotoRacer(name,
bikeName)
{
    cout << "CAutoMotoRacer Constructor" << endl;
}

```

```

CAutoMotoRacer::CAutoMotoRacer(const CAutoMotoRacer& obj) : CRacer(obj),
CAutoRacer(obj), CMotoRacer(obj)
{
    cout << "CAutoMotoRacer Copy Constructor" << endl;
}

CAutoMotoRacer::~CAutoMotoRacer()
{
    cout << "CAutoMotoRacer Destructor" << endl;
}

void CAutoMotoRacer::ShowProperty()
{
    cout << "I have car " << m_pszCarName << " and bike " <<
m_pszBikeName << endl;
}

void CAutoMotoRacer::ShowSkill()
{
    cout << "I can ride a car and a bike" << endl;
}

void CAutoMotoRacer::ShowFullInformation()
{
    cout << "My name is " << m_pszName << ", my experience is " <<
m_nExperience << " years" << endl;
    ShowProperty();
    ShowSkill();
    cout << "-----" << endl;
}

CAutoMotoRacer& CAutoMotoRacer::operator=(CAutoMotoRacer& obj)
{
    if (this != &obj)
    {
        cout << "CAutoMotoRacer operator=" << endl;
        CAutoRacer::operator=(obj);
        CMotoRacer::operator=(obj);
    }
    return *this;
}

```

```

//main.cpp
#include "AutoMotoRacer.h"

int main()
{
    cout << "----- Creating objects -----" << endl;
    CAutoRacer a1("Ivan", "Lazy");
    a1.SetExperience(1);
    a1.ShowFullInformation();

    CMotoRacer a2("Ivan", "Lion");
    a2.SetExperience(2);
    a2.ShowFullInformation();

    CAutoMotoRacer a3 ("Ivan", "Crazy", "Bull");
    a3.SetExperience(3);
    a3.ShowFullInformation();

    CAutoMotoRacer a4 = a3;
    a4.ShowFullInformation();

    CAutoMotoRacer a5(a4);
    a5.ShowFullInformation();

    a1 = a5;
    a1.ShowFullInformation();

    {
        CAutoMotoRacer a6(a4);
        a6.ShowFullInformation();
    }

    //-----
    cout << "----- Creating reference -----" << endl;

    cout << "----- CRacer &ref1 = a1; -----" << endl;
    CRacer &ref1 = a1;
    ref1.ShowFullInformation();

    cout << "----- ref1 = a2; -----" << endl;
    ref1 = a2;
    ref1.ShowFullInformation();

    cout << "----- CRacer &ref2 = a2; -----" << endl;
    CRacer &ref2 = a2;

```

```

ref2.ShowFullInformation();

cout << "----- CRacer &ref3 = a2; -----" << endl;
CRacer &ref3 = a3;
ref3.ShowFullInformation();
//-----
cout << "----- Creating dynamic objects -----" << endl;

cout << "----- CRacer* ptr1 = new CAutoRacer(\"Taras\", \"Flash\") -
-----" << endl;
CRacer* ptr1 = new CAutoRacer("Taras", "Flash");
ptr1->SetExperience(1);
ptr1->ShowProperty();
ptr1->ShowSkill();
ptr1->ShowFullInformation();

cout << "----- CRacer* ptr2 = new CMotoRacer(\"Oleg\", \"Chiky\"); -
-----" << endl;
CRacer* ptr2 = new CMotoRacer("Oleg", "Chiky");
ptr2->SetExperience(2);
ptr2->ShowProperty();
ptr2->ShowSkill();
ptr2->ShowFullInformation();

cout << "----- CRacer* ptr3 = new CAutoMotoRacer(\"Yura\", \"Bull\",
\"Bulldog\"); -----" << endl;
CRacer* ptr3 = new CAutoMotoRacer("Yura", "Bull", "Bulldog");
ptr3->SetExperience(3);
ptr3->ShowProperty();
ptr3->ShowSkill();
ptr3->ShowFullInformation();

cout << "----- CAutoRacer *ptr4 = new CAutoRacer(\"Ostap\",
\"Little\"); -----" << endl;
CAutoRacer *ptr4 = new CAutoRacer("Ostap", "Little");
ptr4->SetExperience(4);
ptr4->ShowProperty();
ptr4->ShowSkill();
ptr4->ShowFullInformation();

cout << "----- CMotoRacer *ptr5 = new CMotoRacer(\"Igor\",
\"MadDog\"); -----" << endl;
CMotoRacer *ptr5 = new CMotoRacer("Igor", "MadDog");
ptr5->SetExperience(5);
ptr5->ShowProperty();

```

```

ptr5->ShowSkill();
ptr5->ShowFullInformation();

cout << "----- CAutoMotoRacer *ptr6 = new CAutoMotoRacer(\"Mark\",
\"Flame\", \" Lover\"); -----" << endl;
CAutoMotoRacer *ptr6 = new CAutoMotoRacer("Mark", "Flame", "Lover");
ptr6->SetExperience(6);
ptr6->ShowProperty();
ptr6->ShowSkill();
ptr6->ShowFullInformation();

//-----
cout << "delete ptr1 " << endl;
delete ptr1;
cout << "-----" << endl << "delete ptr2 " <<
endl;
delete ptr2;
cout << "-----" << endl << "delete ptr3 " <<
endl;
delete ptr3;
cout << "-----" << endl << "delete ptr4 " <<
endl;
delete ptr4;
cout << "-----" << endl << "delete ptr5 " <<
endl;
delete ptr5;
cout << "-----" << endl << "delete ptr6 " <<
endl;
delete ptr6;
cout << "-----" << endl;

system("pause");
return 0;
}

```

ЛАБОРАТОРНА РОБОТА № 8.

ШАБЛони

Мета: познайомитися із створенням шаблонів.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Шаблони являють собою схематичний опис побудови класів та функцій. Використовуючи шаблони, з'являється можливість створювати узагальнені специфікації для класів та функцій, що найчастіше носять назву параметризованих класів (generic classes) та параметризованих функцій (generic functions). Шаблони не прив'язані до конкретних типів даних і описують алгоритми, незалежно від типів даних. Дані алгоритми мають функціонувати однаково для різних типів даних. Такий опис дозволяє описати один раз функції, методи чи класи і на їх базі генерувати функції, методи і класи для кожного конкретного набору параметрів, що економить зусилля і час розробки програмного забезпечення. Після визначення загального шаблону, якщо для одного, кількох або всіх параметрів поведінка класу чи функції відрізнятиметься від описаної в загальному шаблоні, то створюється спеціалізація для конкретного набору параметрів. Спеціалізація може бути звичайною (неявною), явною або частковою.

Призначенням шаблонів є створення екземплярів (instantiating) шаблону, які вже є реальними функціями чи класами. При цьому відбувається прив'язування параметрів шаблону до даних визначеного типу. Цей процес називається **конкретизацією**. Спроба компілятором створити екземпляр шаблону є генерацією програми. Тому зустрічаючи спробу створити екземпляр шаблону компілятор перемикається в режим його вивчення та запам'ятовування, а це - часові витрати.

Типи загального призначення, якими оперують шаблони, **називаються шаблонними типами** (template type), а їх сукупність **параметрами шаблону** (template parameters). Параметри шаблону як множина шаблонних типів може містити також преозначені і вбудовані типи C++.

Шаблонний тип T є невизначеним узагальненим типом. По мірі використання шаблонів компілятор автоматично замінить тип T іменем реального типу. Як правило, для імені шаблонного типу використовують ідентифікатори T чи $Type$. Проте це не обов'язково: ім'я можна декларувати будь-яким допустимим в C++ ідентифікатором. Шаблонний тип можна повноцінно використовувати в тілі шаблону, але це не є строгою вимогою.

Шаблон допускає використання параметрів, *які ініціалізуються аргументами за замовчуванням*, згідно з методологією оголошення і використання таких аргументів. Типи аргументів по замовчуванню можуть бути лише преозначеними або вбудованими. Використання шаблонних типів як аргументів по замовчуванню не допускається.

Під шаблон пам'ять не виділяється. Якщо екземпляр шаблону не створюється, то компілятор навіть не буде транслювати код шаблону. Це спричинює труднощі з використанням файлів заголовків, які містять лише оголошення шаблонів, а їх реалізація знаходиться у сrr-файлі. Для подолання цих недоліків треба підключати сrr-файл, а не файл заголовку, або код шаблону вносити у файл заголовку. Ранні версії С++ компіляторів не перевіряли синтаксис тіла незалежно від створення екземпляра шаблону. Сучасні компілятори відразу аналізують синтаксис коду тіла при першому знаходженні оголошення шаблону, а тому позбавлені цих недоліків.

Використання шаблонів може значно скоротити час створення програми. Це досягається тим, що з'являється можливість перенести незалежний від типу даних, один раз написаний і перевірений код спільний для множини різнорідних функцій в одну програмну конструкцію - шаблон.

Основним застереженням при роботі з шаблонами є *правильне використання операцій до змінних шаблонних типів*. Тобто усі операції, які використовувались до змінної шаблонної типу, повинні мати місце для того типу, яким буде заміщений даний шаблонний тип.

Таким чином, за допомогою реалізації узагальнених функцій можна зменшити розмір та складність програми. Особливо корисними шаблони є саме в бібліотеках класів – тут вони вказують програмісту необхідні специфікації, приховуючи при цьому деталі справжньої реалізації.

Параметризовані функції

Для виконання схожих операцій над різними типами даних часто використовуються перевантажені функції. Якщо ж для кожного типу даних повинні виконуватися ідентичні операції, то більш компактним і зручним рішенням є використання параметризованих (шаблонних) функцій. При цьому програміст повинен написати лише один опис шаблону функції. Базуючись на типах аргументів, використаних при виклику цієї функції, компілятор буде автоматично генерувати об'єктні коди функцій, що оброблятимуть кожен тип даних.

Параметризовані функції декларуються за допомогою ключового слова **template**. Це слово використовується для створення шаблону (каркасу), що в

загальних рисах описує призначення функції та надає опис операцій – сутність алгоритму, що може застосовуватися до даних різних типів. Синтаксис оголошення функції-шаблону має вигляд:

```
template <class T1, class T2, ..., class Tn> тип ім'яФункції  
(аргументи)  
{  
    // тіло функції  
}
```

За ключовим словом **template** слідує не порожній список параметрів шаблону, який складається з ідентифікаторів типу T, кожному з яких передують ключові слова **class** або **typename** (згідно новішого стандарту). Коли компілятор створюватиме конкретну версію функції, то автоматично замінить параметри конкретними типами даних. Цей процес носить назву *інстанціювання шаблону*.

Синтаксис виклику шаблонної функції є таким самим як і виклик звичайної функції і має наступний вигляд:

```
ім'яФункції (аргументи);
```

Кожен формальний параметр з опису шаблону функції повинен з'явитися в списку параметрів функції принаймні один раз. Ім'я формального параметра може використовуватися в списку параметрів заголовка шаблону тільки один раз. Те ж ім'я формального параметра шаблону функції може використовуватися декількома шаблонами.

Шаблон функції може бути перевантажений, а саме можна визначити інші шаблони, що мають те ж ім'я функції, але різні набори параметрів. Також можна ввести не шаблонну функцію з тим же ім'ям та іншим набором параметрів функції.

Компілятор виконує процес узгодження, щоб визначити, який екземпляр функції відповідає конкретному викликові. Спочатку компілятор намагається знайти і використати функцію, що точно відповідає по імені та типам параметрів функції, що викликається. Якщо на цьому етапі компілятор зазнає невдачі, то він шукає шаблон функції, за допомогою якого він може згенерувати параметризовану функцію з точною відповідністю типів параметрів та імені функції; автоматичне перетворення типів не

забезпечується. І як останню спробу, компілятор послідовно виконує процес підбору перевантаженої функції.

Нижче наведений приклад використання шаблону функції.

```
#include<iostream>
#include<cstring>
using namespace std;
template<class T> void printInv(T* ptr, int num)
{
    for(int i = num-1; i >= 0; i--)
    {
        cout << ptr[i] << " ";
    }
    cout << endl;
}
void printInv(char* str)
{
    int num = strlen(str);
    for(int i = num-1; i >= 0; i--)
    {
        cout << str[i];
    }
    cout << endl;
}

int main()
{
    int arr[6] = {1,2,3,4,5,6};
    double farr[6] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
    char* carr = "7654321";

    printInv<int>(arr, 6);
    printInv(farr, 6);
    printInv(carr);

    return 0;
}
```

Результат роботи:

6 5 4 3 2 1

6.6 5.5 4.4 3.3 2.2 1.1

1234567

Press any key to continue

Параметризовані методи

Методи класів можуть бути параметризованими при цьому вони стають дуже схожими до шаблонних функцій, але з особливостями, які накладаються на методи. Зазначимо, що не усі компілятори підтримують можливість оголошення шаблонних функцій як методів класу. Шаблонна функція не може оголошуватись дружньою до класу. Синтаксис оголошення шаблонного методу класу має наступний вигляд:

```
class НазваКласу
{
    template <class T1, class T2, ..., class Tn> тип ім'яМетоду1
    (аргументи) // оголошення і визначення методу
    {
        // тіло методу
    }
    template <class T1, class T2, ..., class Tn> тип ім'яМетоду2
    (аргументи); // оголошення методу
};

template < class T1, class T2, ..., class Tn > тип НазваКласу::
ім'яМетоду2 (аргументи) // визначення методу
{
    // тіло методу
};
```

Синтаксис виклику шаблонного методу має наступний вигляд:

```
назваОб'кту.ім'яМетоду <тип1, тип2, ..., типN> (аргументи);
```

Спеціалізації шаблонних функцій

Варіант шаблонного класу чи шаблонної функції, які створюються компілятором в результаті конкретизації параметрів, називається **спеціалізацією**.

Існують 3 види спеціалізації – **явна, неявна (повна), часткова** спеціалізації:

- Неявною спеціалізацією називається варіант шаблонної функції, що генерується в залежності від типу її аргументів.

```
//Шаблонна функція неявна спеціалізація
template <class T, class T2> bool func1(T v1, T2 v2)
{ return v1 == v2 ? true : false; };
```

- Явною спеціалізацією називається варіант шаблонної функції, де всі параметри типу задані явним чином. Явна спеціалізація оголошується за допомогою наступної конструкції:

```
template <> типПовер Ім'яФункції<тип1,... типN>(аргументи)
{ /* тіло функції */ }
```

Наприклад

```
// Шаблонна функція явна спеціалізація
template <> bool func1<char*, char*>(char* pszStr1, char* pszStr2)
{return strcmp(pszStr1, pszStr2) ? false : true;};
```

- Частковою спеціалізацією називається варіант шаблонної функції, де частина параметрів типу задана явним чином. Основна ідея часткової спеціалізації полягає в сумісному використанні шаблонних і наперед визначених типів.

```
// Шаблонна функція часткова спеціалізація
template <class T> bool func1(T v1, double v2)
{
    if((double)(v2-v1)==0.0)
        return true;
```

```
else  
    return false;  
};
```

Виклик спеціалізації шаблонної функції як і звичайної шаблонної функції відбувається таки самим чином як і виклик звичайної нешаблонної функції.

Параметризовані класи

Визначаючи параметризований клас, ми створюємо його каркас (шаблон), що описує усі алгоритми, які використовуються класом. Фактичний тип даних, над яким проводитимуться маніпуляції, буде вказаний в якості параметру при конкретизації об'єктів цього класу. Компілятор автоматично згенерує відповідний об'єкт на основі вказаного типу. Іншими словами класи породжують об'єкти, а шаблонні класи - класи. Загальна форма декларування параметризованого класу така:

```
template < списокПараметрівШаблону > class ім'яКласу  
{  
    // протокольна частина класу  
}
```

Список параметрів шаблону може містити елементи двох видів:

- специфікації формальних констант, що складаються з імені деякого типу з наступним ідентифікатором, наприклад, `int i`;
- специфікації формальних типів, що складаються з ключового слова `class` або `typename` (згідно новішого стандарту), за яким слідує ідентифікатор; вони аналогічні параметрам шаблону функції, наприклад, `class T`.

Методи шаблонного класу автоматично стають шаблонами. Визначення методів, розташованих в тілі шаблону, нічим не відрізняються від визначення вбудованих методів звичайного класу. Визначення методів, розташовуваних поза тілом шаблону, має наступний вид:

```
template < списокПараметрівШаблону >  
тип ім'яКласу <параметриШаблону>:: ім'яМетоду (аргументи)  
{...}
```

Шаблонний клас і визначення його шаблонних методів мають знаходитися в одному файлі якщо тільки в компіляторі не визначено ключове слово **export**. Якщо дане слово визначене в компіляторі, то визначення методів шаблонного класу можна розміщати в інших файлах вживаючи перед визначенням тіла методу ключове слово **export**.

Подібно до шаблонних функцій, шаблон класу можна перевизначити для якогось конкретного типу аргументу. Це значить, що після визначення загального шаблону можна визначити спеціалізований шаблон класу і передбачити перевизначення всіх його методів і статичних елементів даних. Загалом, повне перевизначення шаблону доцільно тоді, коли необхідна спеціалізація більшості його методів. В іншому випадку достатньо написати явні реалізації необхідних методів шаблону для конкретних типів.

Розрізняють неявну, явну і часткову спеціалізацію шаблонів класів.

Неявна спеціалізація має місце, коли шаблон класу описується в термінах загальних типів, наприклад:

```
template template <class T1, class T2, int i>
class Pair
{
private:
    T1 a;
    T2 b;
public:
    T1 & first(const T1 & f);
    T2 & second(const T2 & s);
    T1 first()const{return a;}
    T2 second()const{return b;}
    Pair(const T1 & f, const T2 & s):a(f), b(s){cout << i;}
};
```

Явна спеціалізація використовується, коли клас має створюватися не за загальним шаблоном класу. При цьому всі типи є визначеними. Опис такого класу схожий на опис явних спеціалізацій шаблонів функцій. Явна спеціалізація може мати параметром не тільки тип, а й шаблонний клас і контейнер. Наприклад:

```

#include <iostream>
using namespace std;

template <class T> class A {public: T var;};
template <class T1,class T2,template <class T4> class T3> class Pair {};

template <> class Pair <char*, A<int>, A>  // явна спеціалізація
{
private:
    char a[20];
    A<int> b; /*ініціалізуватиметься контейнером типу A<int> в
конструкторі*/
    A<char> c;    /* створити контейнер на базі шаблону класу A*/
public:
    char* setfirst(const char* f);
    A<int> & setsecond(const A<int> & s);
    A<char> & setthird(const A<char> & t);
    char* first(){return a;}
    int second()const{return b.var;}
    char third()const{return c.var;}
    explicit Pair(const char * f, const A<int> s)
    {strcpy(a,f);b=s;c.var='A';}
};

char* Pair<char*,A<int>,A>::setfirst(const char* f)
{
    strcpy(a,f);
    return a;
}

A<int> & Pair<char*,A<int>,A>::setsecond(const A<int> & s)
{
    b=s;
    return b;
}

```

```

A<char> & Pair<char*,A<int>,A>::setthird(const A<char> & t)
{
    c=t;
    return c;    }

int main()
{
    A<int> oblnt;
    oblnt.var=5;
    Pair<char*,A<int>,A>ratings[]=
    {
        Pair<char*,A<int>,A>("The Purple Duke", oblnt),
        Pair<char*,A<int>,A>("Jake's Frisco Cafe", oblnt),
        Pair<char*,A<int>,A>("Mont Souffle", oblnt),
        Pair<char*,A<int>,A>("Gertie's Eats", oblnt)
    };
    int joints=sizeof(ratings)/sizeof(Pair<char *, A<int>,A>);
    cout<<"Rating:\tEatery\n";
    for (int i=0;i<joints;i++)
        cout<<ratings[i].second()<<":\t"
        <<ratings[i].first()<<":\t"
        <<ratings[i].third()<<"\n";
    cin.get();
    return 0;
}

```

Результат роботи

```

Rating:   Eatery
5:   The Purple Duke:   A
5:   Jake's Frisco Cafe:   A
5:   Mont Souffle:   A
5:   Gertie's Eats:   A

```

Шаблонними параметрами можуть бути самі шаблони. В цьому випадку говорять про подвійні шаблонні параметри (див. явне створення екземпляру класу):


```

template <class T1,class T2,template <class T4> class T3> class
Pair
{
public:
    T1 var1;
    T2 var2;
    T3 <T1> var3;
    T3 <T2> var4;
    T3 <int> var5;
};

```

Часткова спеціалізація має місце, коли лише частина параметрів ШК є наперед визначеного типу, а решта – мають абстрактний тип. Наприклад:

```

//Шаблон
template <class T1, class T2, int i>
class CP {};

//Часткова спеціалізація
template <class T1, int i>
class CP < T1, T1*, i>
{
private:
    T1 a;
    T1* ptr;
public:
    T1 & first(const T1 & f){return a;};
    T1 first()const{return a;}
    CP(const T1 & f):a(f){ptr=0;}
};

//Часткова спеціалізація
template <class T1,class T2>
class CP < T1, T2, 2>
{
private:
    T1 a;

```

```

    T2 b;
public:
    T1 & first(const T1 & f){return a;};
    T2 & second(const T2 & s){return b;};
    T1 first()const{return a;}
    T2 second()const{return b;}
    CP(const T1 & f, const T2 & s):a(f), b(s){}
};

void main(void)
{
    int n=7;
    CP <char, char*, 5> obj1;      /*error. No appropriate default
    constructor available*/
    CP <char, char*, 1> obj1('v'); // class CP < T1, T1*, i>
    CP <char, char*, n> obj1('v'); /* error. template parameter 'i' : 'n'
    : a local variable cannot be used as a non-type argument */
    CP <char, int, 2> obj2 ('v',5); // class CP < T1, T2, 2>
}

```

При наявності вибору для створення класу компілятор використовує найузагальненіший шаблон. За умови наявності всіх видів спеціалізацій найвищий пріоритет має **явна спеціалізація**, дещо нижчий – часткова, і найнижчий – неявна. Синтаксис створення об'єктів шаблону класу однаковий для всіх видів спеціалізацій.

Щоб створити із шаблону екземпляр конкретного класу, потрібно оголосити об'єкт, вказавши для його типу ім'я шаблону з набором конкретних аргументів (типів і констант). Кожен формальний тип у списку параметрів шаблону потрібно замінити на ім'я конкретного типу. Кожна формальна константа замінюється на константу зазначеного в шаблоні типу. Після того, як представник шаблонного класу створений, з ним можна поводитись як і з будь-яким об'єктом, що належить до звичайного класу.

Створення екземпляру шаблонного класу може бути здійснено в явному і неявному вигляді.

Неявне створення екземпляру класу відбувається тоді, коли вперше знадобиться об'єкт даного класу. Наприклад:

```
Pair<char *, int> * ptr; /* оголошення вказівника. Екземпляр  
                        класу не генерується*/  
Pair<char *, int> rating; /* неявне створення екземпляру класу  
                        і об'єкту на його базі*/
```

Явне створення екземпляру класу здійснюється компілятором завжди, коли він створюється явно з використанням ключового слова *template*. Наприклад:

```
template Pair<char *, int>;
```

Коли при обробці вихідного файлу компіляторіві зустрічається створення об'єкта на основі деякого шаблону класу, він насамперед генерує представника шаблону, або шаблонний клас. Власне кажучи при цьому генеруються і компілюються коди усіх методів шаблону для даного набору його аргументів (і коди деяких допоміжних функцій). Після цього компілятор може конструювати об'єкт шаблонного класу, викликати потрібні методи об'єкта і т.д..

Якщо створюється шаблонний об'єкт, аргументи якого збігаються з аргументами об'єкта, раніше створеного в поточному модулі компіляції, то новий представник шаблону не генерується. Даний шаблонний клас вже існує, залишається тільки сконструювати об'єкт.

Для шаблону класу можна визначити шаблон дружньої функції. Такий шаблон буде породжувати окрему дружню функцію для кожного шаблонного класу, що буде генеруватися. Типовим прикладом шаблону дружньої функції може служити операція передачі об'єкта в потік.

Розглянемо приклад закінченого шаблону класу, що містить дружні функції:

```
#include<iostream>  
#include<stdlib.h>  
#include<ctime>  
using namespace std;  
  
template<class T, int size>  
class Array  
{
```

```

        friend ostream& operator<< (ostream& output,
Array<T,size>& arr);
        T ptr[size];
public:
        Array( ){
                for(int i = 0 ; i < size; i++)
                {
                        ptr[i] = 0;
                }
        };
        ~Array(){ };
        void Rindomize(int num);
        T Sum();
};

template <class T, int size>
void Array<T,size>::Rindomize(int num)
{
        for(int i = 0; i < size; i++)
        {
                ptr[i] = T(rand() % num * 1.0 / (num / 10));
        }
}

template<class T, int size>
T Array<T,size>::Sum()
{
        T sum = 0;
        for(int i = 0 ; i < size; i++)
        {
                sum += ptr[i];
        }
        return sum;
}

```

```

template <class T,int size>
ostream& operator<< (ostream& output, Array<T,size>& arr)
{
    for(int i = 0; i < size; i++)
    {
        output << arr.ptr[i] << " ";
    }
    output << endl;
    return output;
}

int main()
{
    srand(time(NULL));
    Array<int,10> iarr;
    iarr.Rindomize(10);
    cout << iarr << "Sum = " << iarr.Sum() << endl;

    Array<double,10> farr;
    farr.Rindomize(100);
    cout << farr << "Sum = " << farr.Sum() << endl;
    return 0;
}

```

Результати виконання:

```

1 5 0 8 8 9 9 9 9 1
Sum = 59
1.9 1 3.7 7.4 1.7 0.7 4.2 5.7 7.2 8.6
Sum = 42.1
Press any key to continue

```

Аргументи за замовчуванням для шаблонних типів і значення по замовчуванню для вбудованих типів

Аргументи шаблонних класів, тобто значення шаблонних типів, можна задавати за замовчуванням. Синтаксис оголошення шаблонного класу із аргументами за замовчуванням є такий:

```
template <class T = напередВизначенийТип [,іншіШаблТипи]>
class Ім'я_Класу {};
```

Значенням для шаблонного типу може бути будь-який наперед визначений тип. Завдяки йому може бути опущене значення для шаблонного типу при оголошенні контейнера. Проте трикутні дужки повинні бути обов'язково. Обмеження, які накладаються на використання шаблонних класів з аргументами за замовчуванням є такими як для функцій з аргументами за замовчуванням. Це, по-перше, означає що усі шаблонні типи, які мають значення за замовчуванням повинні йти останніми у списку шаблонних типів. По-друге, при створенні контейнера список шаблонних типів повинен неперервно заповнюватись зліва направо. Змінна-параметр вбудованого типу також може входити до списку параметрів шаблону. Значення цього параметра задається значення при оголошенні контейнера. Аргументи шаблонного класу не можна задавати за замовчуванням у часткових спеціалізаціях.

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

template <class T1,class T2=int, int n=25 >
class Pair
{
private:
    T1 a;
    T2 b;
public:
    T1 & first(const T1 & f);
    T2 & second(const T2 & s);
    T1 first()const{return a;}
    T2 second()const{return b;}
    Pair(const T1 & f, const T2 & s):a(f), b(s){std::cout<<n<<endl;}
};

template <class T1,class T2, int n>
T1 & Pair<T1,T2,n>::first(const T1 & f)
{
```

```

    a=f;
    return a;
}
template <class T1,class T2, int n>
T2 & Pair<T1,T2,n>::second(const T2 & s)
{
    b=s;
    return b;
}

int main()
{
    Pair<char *>ratings[]=    // no [4]
    {
        Pair<char *>("The Purple Duke", 5),
        Pair<char *>("Jake's Frisco Cafe", 4),
        Pair<char *>("Mont Souffle", 5),
        Pair<char *>("Gertie's Eats", 3)
    };
    int joints=sizeof(ratings)/sizeof(Pair<char *, int>);
    cout<<"Rating:\tEatery\n";
    for (int i=0;i<joints;i++)
        cout<<ratings[i].second()<<":\t"
        <<ratings[i].first()<<"\n";
    ratings[3].second(6);
    cout<<"Oops! Revised rating:\n";
    cout<<ratings[3].second()<<":\t"
        <<ratings[3].first()<<"\n";
    cin.get();
    return 0;
}

```

Результат виконання

25
25
25

25

Rating: Eatery

5: The Purple Duke

4: Jake's Frisco Cafe

5: Mont Souffle

3: Gertie's Eats

Oops! Revised rating:

6: Gertie's Eats

Спадкування в шаблонах класів

Шаблони класів, як і класи, підтримують механізм спадкування. Всі основні ідеї спадкування при цьому залишаються незмінними, що дозволяє побудувати ієрархічну структуру шаблонів, аналогічну ієрархії класів. Розглянемо приклад, на якому продемонструємо, яким чином можна створити шаблон класу, похідний шаблону класу `Pair`. Нехай це буде клас `Trio`, в якому до пари елементів `a` і `b` з `Pair`, додамо елемент `c`.

```
template <class T>
class Pair
{
    T a, b;
public:
    Pair (T t1, T t2);
    T Max();
    T Min ();
    int isEqual ();
};

template <class T>
T Pair <T>::Min()
{
    return a < b ? a : b;
}
```



```

// конструктор
template <class T>
Pair <T>::Pair (T t1, T t2) : a(t1), b(t2) {}

// метод Max
template <class T>
T Pair <T>::Max()
{
    return a>b ? a : b;
}

// метод isEqual
template <class T>
int Pair <T>::isEqual()
{
    if (a==b) return 1;
    return 0;
}

template <class T>
class Trio: public Pair <T>
{
    T c;
public:
    Trio (T t1, T t2, T t3);
    ...
};

template <class T>
Trio<T>::Trio (T t1, T t2, T t3): Pair <T> (t1, t2), c(t3)
{}

```

Зауважимо, що виклик батьківського конструктора також супроводжується передачею типу T в якості параметра.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке шаблон?
2. Що таке шаблон функції і які його особливості?
3. Що таке шаблон методу і які його особливості?
4. Що таке шаблон класу і які його особливості?
5. Що таке спеціалізації і які їх особливості?
6. Що таке шаблонний тип?
7. Коли відбувається створення екземпляру шаблонного класу і які шляхи його створення ви знаєте?
8. Скільки пам'яті займає шаблон?
9. Для чого використовується ключове слово `export`?
10. Як правильно задавати параметризовані типи за замовчуванням?

ЛІТЕРАТУРА

1. Прата С. Язык программирования C++. Лекции и упражнения. 6-е издание / Стивен Прата; пер. с англ. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.: ил. – Парал. тит. англ.
2. Полный справочник по C++, 4-е издание / Герберт Шилдт — М.: «Вильямс», 2011. — 800 с.

ЗАВДАННЯ

Контейнерний клас описує та забезпечує набір дій над даними параметризованого масиву, розмірність якого визначається під час роботи програми. Усі обчислення та перетворення повинні бути реалізовані у вигляді методів класу.

Варіант 1

В масиві обчислити різницю елементів масиву, що розташовані між першим від'ємним та другим додатним елементами.

Варіант 2

Дана прямокутна матриця. Визначити кількість від'ємних елементів в тих рядках, які містять хоча б один нульовий елемент.

Варіант 3

У довільній матриці обчислити яке з середніх арифметичних парних чи непарних елементів є більшим.

Варіант 4

В одновимірному масиві елементів, обчислити суму модулів елементів, які розташовані після першого додатного елемента.

Варіант 5

У масиві обчислити суму елементів масиву, що розташовані між першим і другим додатними елементами.

Варіант 6

Дана прямокутна матриця. Визначити номер рядка, в якому знаходиться найдовша серія з однакових елементів.

Варіант 7

В масиві обчислити різницю середніх арифметичних від'ємних і додатніх елементів масиву.

Варіант 8

Дана прямокутна матриця. Визначити кількість рядків матриці, в яких елементи розташовані по зростанню.

Варіант 9

У довільній матриці визначити у якому рядку сума додатніх елементів є найбільшою.

Варіант 10

В одновимірному масиві елементів, обчислити кількість елементів масиву, що розташовні по зростанню підряд.

Варіант 11

У матриці розташувати додатні елементи в перших рядках, а від'ємні – в останніх.

Варіант 12

Дана прямокутна матриця. Визначити номер рядка, в якому знаходиться найдовша серія з різних елементів.

Варіант 13

В масиві замінити всі елементи, що більші за середнє арифметичне значення, середнім арифметичним значенням.

Варіант 14

Дана прямокутна матриця у якій розташувати елементи в рядках матриці по зростанню.

Варіант 15

У довільному масиві обчислити кількість елементів масиву, що розташовані по спаданню.

Варіант 16

В одновимірному масиві елементів, обчислити суму модулів елементів, які розташовані на непарних позиціях.

Варіант 17

У матриці обчислити суму елементів матриці, що розташовані на непарних позиціях у парних рядках.

Варіант 18

Дана прямокутна матриця. Визначити номер рядка, в якому знаходиться найдовша серія з елементів у яких чергуються знаки.

Варіант 19

В масиві обчислити різницю середніх геометричних від'ємних і додатніх елементів масиву.

Варіант 20

Дана прямокутна матриця. Визначити кількість рядків матриці, в яких елементи розташовані по спаданню.

Варіант 21

У довільній матриці визначити у якому рядку різниця від'ємних елементів є найменшою.

Варіант 22

В одновимірному масиві елементів, обчислити кількість елементів масиву, що розташовані по спаданню через один.

Варіант 23

У матриці розташувати додатні елементи в перших стовпцях, а від'ємні – в останніх.

Варіант 24

Дана прямокутна матриця. Визначити номер рядка, в якому знаходиться найдовша серія з однакових додантів непарних елементів.

Варіант 25

У довільному масиві обчислити кількість елементів масиву, що розташовані по спаданню і стоять на парних позиціях.

Варіант 26

В одновимірному масиві елементів, обчислити суму модулів елементів, які менші за середнє арифметичне значення елементів масиву.

Варіант 27

У матриці обчислити суму елементів матриці, що розташовані на парних позиціях у непарних стовпцях.

Варіант 28

Дана прямокутна матриця. Визначити номер рядка, в якому знаходиться найдовша серія з елементів у яких нечергуються знаки.

Варіант 29

В масиві обчислити середнє геометричне елементів масиву, що менші за середнє геометричне елементів масиву.

Варіант 30

Дана прямокутна матриця. Визначити кількість стовпців матриці, в яких елементи розташовані по зростанню.

ПОРЯДОК ВИКОНАННЯ

1. Запустити Microsoft Visual Studio.
2. Створити новий порожній проект в стилі Visual C++ Win 32 Console application.
3. Створити новий *.cpp файл та включити його в проект.
4. Написати і скомпілювати код програми, що реалізує поставлене завдання, та виводить результат виконання на екран.
5. Оформити та захистити звіт.

ПРИКЛАД ВИКОНАННЯ

Завдання: у довільній матриці обчислити кількість елементів масиву, рівних нулю.

Реалізація

```
// main.cpp
#include <iostream>
#include <iomanip>
#include "matrix.h"
using namespace std;

int main(void)
{
    Matrix<int> m;
    m.count_null();
    int row, col;
    cin >> row;
    cin >> col;
    Matrix<int> m2(row, col);
    cout << m2;
    m2.count_null();
    m2.init_matrix();
    cout << m2;
    m = m2;
    cout << m;
    Matrix<int> m3(m2);
    cout << m3;
    return 0;
}

// matrix.h
#ifndef matrix_h__
#define matrix_h__

#include <iostream>
#include <iomanip>

using namespace std;

template<class T>
class Matrix
{
private:
    T **m_iArr;
    int size_row;
```

```

    int size_col;
public:
    //конструктор за замовчуванням
    Matrix();
    //конструктор з параметрами
    Matrix(int size_row_BuUser, int size_col_BuUser);
    //конструктор копіюванням
    Matrix(const Matrix& init);
    //деструктор
    ~Matrix();
    //перевантаження оператора виводу
    friend ostream& operator<< <> (ostream&, Matrix<T>&);
    //перевантаження оператора присвоєння
    Matrix& operator=(Matrix<T>& rhs);
    //ініціалізація матриці вручну
    void init_matrix();
    //підрахунок нулів в матриці
    int count_null();

};

//конструктор за замовчуванням
template <class T>
Matrix<T>::Matrix()
{
    size_col = 0;
    size_row = 0;
    m_iArr = new T *[size_row];
    for (int i = 0; i < size_row; i++)
    {
        m_iArr[i] = new T[size_col];
        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = 0;
        }
    }
}

//конструктор з параметрами
template <class T>
Matrix<T>::Matrix(int size_row_BuUser, int size_col_BuUser)
{
    size_row = size_col_BuUser;
    size_col = size_row_BuUser;
    m_iArr = new T *[size_row];

    for (int i = 0; i < size_row; i++)
    {
        m_iArr[i] = new T[size_col];
    }
}

```

```

        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = 0;
        }
    }
    cout << "Matrix for random enter - 1\nMatrix for user enter -2"<<
endl << "Please enter:";
    int l = 1;
    cin >> l;
    switch (l)
    {
        case 1:
            for (int i = 0; i < size_row; i++)
                for (int j = 0; j < size_col; j++)
                {
                    m_iArr[i][j] = rand() % 99;
                }
            break;
        case 2:
            for (int i = 0; i < size_row; i++)
                for (int j = 0; j < size_col; j++)
                {
                    cout << "a[" << i << "]" << "[" << j << "]" << "=";
                    cin >> m_iArr[i][j];
                }
            break;
    }
}

//контруктор копіювання
template<class T>
Matrix<T>::Matrix(const Matrix<T>& init)
{
    if (&m_iArr != 0)
    {
        for (int j = 0; j < size_row; j++)
        {
            delete m_iArr[j];
        }
        delete[] m_iArr;
    }
    size_row = init.size_row;
    size_col = init.size_col;
    m_iArr = new T *[size_row];

    for (int i = 0; i < size_row; i++)
    {

```



```

        m_iArr[i] = new T[size_col];
        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = 0;
        }
    }
    for (int i = 0; i < size_row; i++)
        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = init.m_iArr[i][j];
        }
}

//деструктор
template<class T>
Matrix<T>::~~Matrix()
{
    for (int j = 0; j < size_row; j++)
    {
        delete m_iArr[j];
    }
    delete[] m_iArr;
}

//перевантаження оператора виводу
template <class T>
ostream& operator<<(ostream& output, Matrix<T>& matr)
{
    for (int i = 0; i < matr.size_row; i++)
    {
        for (int j = 0; j < matr.size_col; j++)
        {
            output << matr.m_iArr[i][j] << " ";
        }
        output << endl;
    }
    output << endl;
    return output;
}

//перевантаження оператора присвоєння
template<class T>
Matrix<T>& Matrix<T>::operator= (Matrix<T>& matr)
{
    if (this != &matr)
    {
        if (&m_iArr != 0)

```

```

    {
        for (int j = 0; j < size_row; j++)
        {
            delete m_iArr[j];
        }
        delete[] m_iArr;
    }
    size_row = matr.size_row;
    size_col = matr.size_col;
    m_iArr = new T *[size_row];

    for (int i = 0; i < size_row; i++)
    {
        m_iArr[i] = new T[size_col];
        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = 0;
        }
    }
    for (int i = 0; i < size_row; i++)
        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = matr.m_iArr[i][j];
        }
}
cout << "Operator =" << endl;
return *this;
}

```

//ініціалізація матриці вручну

```

template<class T>
void Matrix<T>::init_matrix()
{
    //видалення існуючої
    if (&m_iArr != 0)
    {
        for (int j = 0; j < size_row; j++)
        {
            delete m_iArr[j];
        }
        delete[] m_iArr;
    }
    //створення нової
    cout << "enter row:";
    cin >> size_row;
    cout << "enter col:";
    cin >> size_col;
}

```

```

m_iArr = new T *[size_row];

for (int i = 0; i < size_row; i++)
{
    m_iArr[i] = new T[size_col];
    for (int j = 0; j < size_col; j++)
    {
        m_iArr[i][j] = 0;
    }
}

cout << "Matrix for random enter - 1\nMatrix for user enter -
2\nPlease enter:";
int l = 1;
cin >> l;
switch (l)
{
case 1:
    for (int i = 0; i < size_row; i++)
        for (int j = 0; j < size_col; j++)
        {
            m_iArr[i][j] = rand() % 99;
        }
    break;
case 2:
    for (int i = 0; i < size_row; i++)
        for (int j = 0; j < size_col; j++)
        {
            cout << "a[" << i << "]" << "[" << j << "]" << "=";
            cin >> m_iArr[i][j];
        }
    break;
}
}

template<class T>
int Matrix<T>::count_null()
{
    int count_null = 0;
    for (int i = 0; i < size_row; i++)
        for (int j = 0; j < size_col; j++)
        {
            if (m_iArr[i][j] == 0)
            {
                count_null++;
            }
        }
}

```

```
        cout << count_null << endl;
        return count_null;
    }
#endif // matrix_h__
```

НАВЧАЛЬНЕ ВИДАННЯ

**ПРОГРАМУВАННЯ, ЧАСТИНА 2 (ОБ’ЄКТНО-ОРІЄНТОВАНЕ
ПРОГРАМУВАННЯ)**

МЕТОДИЧНІ ВКАЗІВКИ

**до виконання циклу лабораторних робіт з дисципліни
“ Програмування, частина 2 (об’єктно-орієнтоване програмування)” для студентів
базового напрямку 6.050102 “Комп’ютерна інженерія”**

Укладачі:

Морозов Юрій Васильович
Олексів Максим Васильович
Мороз Іван Володимирович

Редактор

Комп’ютерне верстання

Здано у видавництво . Підписано до друку
Формат 70x100/16. Папір офсетний. Друк на різнографі
Умовн. друк. арк. Обл.-вид. арк..
Тираж прим. Зам..

Видавництво Національного університету “Львівська політехніка”
Ресстраційне свідоцтво ДК №751 від 27.12.2001 р.

Поліграфічний центр Видавництва
Національного університету “Львівська політехніка”

Вул. Ф. Колесси, 2. Львів, 79000