

Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

Т. М. Сопронюк

Об'єктно-орієнтоване програмування на C++

Навчальний посібник

Рекомендовано Міністерством освіти і науки України



Чернівці
Чернівецький
національний університет
2014

УДК 004.43 C++(075.8)
ББК 32.973.2-018.2я73
С 646

Рекомендовано Міністерством освіти і науки України
(лист №1/11-17600 від 18.11.13)

Рецензенти:

Чикрій А. О., доктор фізико-математичних наук, професор, член-кореспондент НАН України, завідувач відділу оптимізації керування процесів Інституту кібернетики імені В.М. Глушкова НАН України;

Нестерук І. Г., доктор фізико-математичних наук, професор кафедри комп'ютерної інженерії університету КРОК (м. Київ), провідний науковий співробітник Інституту гідромеханіки НАН України;

Градобоєв О. В., доктор технічних наук, професор Національного дослідного Томського політехнічного університету, начальник лабораторії спецвипробувань ОАО "НДІ ПП".

Автор вдячна Чернівецькому філіалу
Нідерландської ІТ-компанії SharpMinds BV Netherlands
за матеріальну допомогу при виданні посібника.

Сопронюк Т. М.

С 646 Об'єктно-орієнтоване програмування на C++ : навч. посібник /
Т. М. Сопронюк. – Чернівці : Чернівецький національний ун-т,
2014. – 176 с.

ISBN 978-966-423-304-7

У посібнику наведено основні теоретичні відомості, необхідні для виконання лабораторних завдань із дисципліни “Об'єктно-орієнтоване програмування”. Описано операції конструювання екземплярів класу та їх знищення, статичні члени класу, дружні функції, дружні класи, перевантаження функцій та операторів для класу, шаблони функцій і класів. Дано поняття базового і породженого класів, ієрархії класів, віртуальних функцій, абстрактних класів, поліморфних функцій.

Для студентів напрямів підготовки “Прикладна математика”, “Інформатика”, “Системний аналіз”.

ББК 32.973.2-018.2я73
УДК 004.43 C++(075.8)

ISBN 978-966-423-304-7

© Сопронюк Т.М., 2014
© Чернівецький національний
університет, 2014

ЗМІСТ

ПЕРЕДМОВА.....	5
1. ЕЛЕМЕНТИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ.....	9
1.1. Об'єкт. Клас. Структура.....	9
1.2. СТАНДАРТНІ ОБ'ЄКТИ ПОТОКОВОГО ВВЕДЕННЯ/ВИВЕДЕННЯ	14
1.3. ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ ДЛЯ ТИПІВ STRUCT, UNION ТА ENUM	16
1.4. ПЕРЕВАНТАЖЕННЯ ФУНКЦІЙ	23
1.5. ПРИКЛАД ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ ДЛЯ ДВОХ СТРУКТУРНИХ ТИПІВ	25
1.6. ШАБЛони ФУНКЦІЙ	28
1.7. ПРИКЛАД ПЕРЕВАНТАЖЕННЯ ФУНКЦІЙ І ВИКОРИСТАННЯ ШАБЛОНІВ ФУНКЦІЙ	29
ЗАПИТАННЯ ДЛЯ САМОКОНТРОЛЮ	33
ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ	34
2. РОБОТА З ОБ'ЄКТАМИ В МЕЖАХ ОДНОГО КЛАСУ.....	34
2.1. УПРАВЛІННЯ ДОСТУПОМ ДО ЧЛЕНІВ КЛАСУ	34
2.2. КОНСТРУКТОРИ	38
2.3. СТАТИЧНІ ЧЛЕНИ КЛАСУ	44
2.4. ДЕСТРУКТОРИ.....	46
2.5. ДИНАМІЧНЕ СТВОРЕННЯ ТА ЗНИЩЕННЯ ОБ'ЄКТІВ.....	49
2.6. ДРУЗІ КЛАСУ	54
2.7. ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ ДЛЯ КЛАСІВ.....	57
2.8. БІНАРНІ Й УНАРНІ ОПЕРАЦІЇ	61
2.9. ОПЕРАТОР ПРИСВОЄННЯ	62
2.10. ОПЕРАТОРИ ВИКЛИКУ, ІНДЕКСУВАННЯ Й ПЕРЕТВОРЕННЯ ТИПУ..	65
2.11. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ NEW І DELETE	68
2.12. ПРИКЛАД РОЗРОБКИ КЛАСУ “МНОЖИНА ЦІЛИХ ЧИСЕЛ”	71
ЗАПИТАННЯ ДЛЯ САМОКОНТРОЛЮ	82
ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ	83
3. СТВОРЕННЯ НОВИХ КЛАСІВ ІЗ ВИКОРИСТАННЯМ ІСНУЮЧИХ.....	83
3.1. Побудова класів з членами-об'єктами. АГРЕГАЦІЯ	83
3.2. ШАБЛони класів	86
3.3. УСПАДКУВАННЯ.....	89

3.4. Ієрархія успадкування класів. Одинарне успадкування ...	92
3.5. ПЕРЕТВОРЕННЯ ВКАЗІВНИКІВ НА ОБ'ЄКТИ З ОДНІЄЇ ІЄРАРХІЇ КЛАСІВ.....	96
3.6. ПЕРЕТВОРЕННЯ ТИПІВ ДЛЯ ОБ'ЄКТІВ З ОДНІЄЇ ІЄРАРХІЇ КЛАСІВ.....	98
3.7. МНОЖИННЕ УСПАДКУВАННЯ.....	104
ЗАПИТАННЯ ДЛЯ САМОКОНТРОЛЮ	110
ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ	111
4. РОЗРОБКА СІМ'Ї КЛАСІВ. АБСТРАКЦІЯ	112
4.1. СТАТИЧНИЙ І ДИНАМІЧНИЙ ПОЛІМОРФІЗМ	112
4.2. ВІРТУАЛЬНІ ФУНКЦІЇ	113
4.3. АБСТРАКТНІ КЛАСИ.....	114
4.4. ПРОЕКТУВАННЯ І ПОБУДОВА БІБЛІОТЕКИ КЛАСІВ ІЗ ВІРТУАЛЬНИМИ ФУНКЦІЯМИ.....	117
4.5. ПОЛІМОРФНІ ФУНКЦІЇ	123
4.6. ВІРТУАЛЬНЕ УСПАДКУВАННЯ	126
ЗАПИТАННЯ ДЛЯ САМОКОНТРОЛЮ	129
ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ	130
5. ЛАБОРАТОРНІ РОБОТИ.....	131
ЛАБОРАТОРНА РОБОТА № 1	131
<i>Тема: Структури. Зовнішні функції і функції-члени структури</i>	<i>131</i>
ЛАБОРАТОРНА РОБОТА № 2	137
<i>Тема: Перевантаження операцій і функцій.</i>	<i>137</i>
<i>Шаблони функцій</i>	<i>137</i>
ЛАБОРАТОРНА РОБОТА № 3	142
<i>Тема: Структури та класи.</i>	<i>142</i>
ЛАБОРАТОРНА РОБОТА № 4	150
<i>Тема: Створення класів</i>	<i>150</i>
ЛАБОРАТОРНА РОБОТА № 5	166
<i>Тема: Створення класів з використанням існуючих класів.</i>	
<i>Успадкування. Агрегація</i>	<i>166</i>
ЛАБОРАТОРНА РОБОТА № 6	168
<i>Тема: Абстрактні класи</i>	<i>168</i>
СПИСОК ЛІТЕРАТУРИ	174

ПЕРЕДМОВА

Об'єктно-орієнтоване програмування, або ООП (**object-oriented programming**), – методологія програмування, яка ґрунтується на поданні програми у вигляді сукупності об'єктів, кожен з яких є реалізацією певного типу, описаного за допомогою класів, організованих в ієрархію успадкування. Ця методологія концентрує основну увагу на зв'язках між об'єктами, а не на деталях їхньої реалізації. Усі мови ООП, особливо **C++**, базуються на трьох основних концепціях: інкапсуляції, поліморфізмі й успадкуванні [13, 18, 19, 22-26, 28].

Інкапсуляція (encapsulation) – це механізм, що поєднує дані і код, який маніпулює цими даними, в одній конструкції. Інкапсуляція також захищає частину функцій і даних від зовнішнього втручання чи неправильного використання. В об'єктно-орієнтованому програмуванні функції і дані, якими вони маніпулюють, можуть бути об'єднані в єдине ціле. Таке об'єднання утворює класовий тип.

У **C++** інкапсуляція реалізується за допомогою класів. Інкапсуляція забезпечує захисну оболонку (капсулу) від зовнішніх функцій. Доступ до коду та до даних усередині оболонки (капсули) контролюється інтерфейсом, який надається користувачу. Отже, *клас* – це інкапсульована абстракція зі строго визначеними правилами доступу до окремих частин (секцій).

Об'єкт – це змінна класового типу (екземпляр класу). Об'єкт підтримує інкапсуляцію. Усередині об'єкта коди й дані можуть бути закритими (**private**). Закриті коди чи дані доступні тільки для інших частин цього об'єкта. Таким чином, закриті коди й дані недоступні для тих функцій, які існують поза об'єктом. Якщо ж коди та дані відкриті (**public**), то, хоча вони задаються всередині об'єкта, однак доступні й для інших частин програми. Зазвичай відкрита частина об'єкта використовується для того, щоб забезпечити контрольований доступ до закритих елементів об'єкта. Об'єкт поєднує коди й дані та є змінною визначеного користувачем класового типу. Отже, кожен об'єкт є складеною змінною.

Захищені (**protected**) елементи класу зазвичай використовуються при побудові ієрархії класів і доступні не тільки всім частинам свого об'єкта, а й об'єктам похідних класів.

Успадкування (**inheritance**) – це процес, за допомогою якого один об'єкт здобуває властивості іншого. Точніше, об'єкт може успадковувати основні властивості іншого об'єкта й додавати до них риси, характерні тільки для нього.

Успадкування дозволяє підтримувати концепцію ієрархії класів (**hierarchical classification**). Без використання ієрархії класів для кожного об'єкта довелося б задавати всі характеристики, які б його повністю визначали, а при використанні успадкування можна описати об'єкт, вказуючи загальний клас (батьківський клас), до якого він належить, і ті специфічні риси, що роблять об'єкт унікальним. Породжений клас успадковує всі, пов'язані з батьком, риси і додає до них свої власні специфічні характеристики. Успадкування відіграє визначальну роль у ООП.

Поліморфізм (від грецьк. **polymorphos**) – це властивість, що дозволяє однакові імена використовувати для позначення кількох схожих за змістом, але різних за реалізацією задач. Метою поліморфізму, стосовно ООП, є використання одного імені для задання загальних для класу дій. Виконання кожної конкретної дії буде визначатися типом даних. Типи даних, що використовуються як аргументи при виклику функції, визначають, яка конкретна версія функції дійсно викликається. Отже, у **C++** можна використовувати одне ім'я функції для реалізації ряду різних дій. Така можливість називається *перевантаженням функцій* (**function overloading**). Перевантаження функцій – це одна з форм поліморфізму. Перевагою поліморфізму є те, що він, дозволяючи використання одного інтерфейсу для задання єдиного типу дій, уможливує зниження складності програм. Вибір же конкретної дії, залежно від ситуації, відбувається компілятором. Потрібно тільки використовувати загальний інтерфейс.

У більш загальному сенсі концепція поліморфізму означає здатність об'єкта відреагувати на деякий запит (тобто виклик

функції-члена) відповідно до свого типу, навіть якщо на стадії компіляції тип об'єкта ще невідомий.

Поліморфізм застосовується також і до операторів. Фактично в мові **C** поліморфізм застосовується обмежено, зокрема, в арифметичних операторах. Наприклад, символ "+" використовується для додавання цілих, довгих цілих і дійсних чисел. У цьому випадку компілятор автоматично визначає, який тип арифметики потрібний. У **C++** можна застосувати цю концепцію і до інших заданих користувачем, типів даних. Так, символ "+" можна використати для виконання операції додавання комплексних чисел. Для цього треба визначити відповідну операторну функцію. Такий тип поліморфізму називається *перевантаженням операторів* (**operator overloading**). Основне в поліморфізмі те, що він дозволяє маніпулювати об'єктами різної складності шляхом створення для них загального стандартного інтерфейсу для реалізації схожих дій.

Посібник складається з п'яти частин. Спочатку розглянуто можливості **C++**, яких не було в мові **C** і які не мають прямого відношення до поняття класів об'єктів, зокрема, потокове введення/виведення, оператори динамічного виділення та звільнення пам'яті, перевантаження функцій, шаблони функцій. Далі, від поняття структури в мові **C** здійснено перехід до поняття класу. Описано операції конструювання екземплярів класу та їх знищення, статичні члени класу, дружні функції, інтерфейс і реалізація класу, перевизначення операторів для об'єктів даного класу, шаблони класів та дружні класи.

Розглянуто також засоби використання створених класів (агрегація й успадкування) для розширення можливостей існуючих класів. Описано поняття базового й породженого класів, множинне успадкування, побудова ієрархії класів, використання віртуальних функцій, абстрактних класів, поліморфних функцій, віртуального успадкування.

У кожній частині посібника наведено коментовані програми, які демонструють викладений теоретичний матеріал.

Кожна частина закінчується питаннями для самоконтролю і

завданнями для самостійної роботи з продемонстрованими програмами. У кінці видання наведено умови лабораторних робіт, які задаються студентам протягом семестрового курсу ООП на **C++**. Частина умов (лабораторні роботи № 1, № 3 і № 6) ґрунтується на ідеях з [11, 25]. Інші умови лабораторних робіт належать автору [17], вдосконалювались і тестувались на заняттях разом зі студентами і викладачами кафедри. Зазначимо, що для допомоги студентам у посібнику демонструються коди схожих програм для всіх типів лабораторних робіт.

У посібнику в основному наведено консольні програми, які перевірені у середовищах розробки **Microsoft Visual C++.Net** і/або **Borland C++Builder** [4, 9, 16, 27]. Робота з бібліотеками **STL** [8, 9, 11, 12] і **VCL** у даному посібнику не розглядається. Ці інструменти ООП на **C++** та деякі інші аспекти, які не увійшли до посібника, описано автором у [16].

Автор вдячна за багаторічну співпрацю при проведенні занять Якімову Ігорю Вікторовичу, Дудницькому Павлу Миколайовичу, Данилюку Івану Михайловичу, а також Чернівецькому філіалу Нідерландської ІТ-компанії SharpMinds BV Netherlands за матеріальну допомогу при виданні посібника.

SharpMinds is proud to support this beautiful and useful book to prepare yourself to become the SharpMind from tomorrow. SharpMinds is very interested to start discussions with you how to use your skills in the future in IT business. We have great apprenticeship places, provide IT and English lessons and have challenging projects. We invite you to experience the open and pleasant atmosphere in our office. Just send your CV to hrm@sharpminds.com and we invite you for an open discussion and a good cup of coffee.

Success with your study!

Edwin Zuijdendorp
hrm@sharpminds.nl

Director SharpMinds



Sharp
Minds

1. Елементи об'єктно-орієнтованого програмування

1.1. Об'єкт. Клас. Структура

Об'єкт у загальному розумінні позначає довільну дійсну або уявну реальність, тобто це абстрактна сутність, наділена характеристиками реального світу.

З точки зору програмування, створення об'єктів і маніпулювання ними – результат методології програмування, що втілюється в кодових конструкціях опису об'єктів і операцій над ними. Кожен об'єкт програми, як і будь-який реальний об'єкт, відрізняється власними атрибутами й характерною поведінкою. Таким чином, будь-який клас визначає деяку категорію об'єктів, а будь-який об'єкт є екземпляром деякого класу.

Клас узагальнює поняття типу даних. Об'єкт – це змінна класового типу. Метою введення класів у **C++** є надання програмісту можливості створення нових типів даних, таких само зручних, як і вбудовані типи. Отже, клас – це визначений користувачем тип. Кожен клас містить дані та набір функцій, що маніпулюють цими даними. Подібно до того, як сутність вбудованого типу, такого як **int**, називається змінною, сутність визначеного користувачем типу (тобто класу) називається об'єктом. Центром уваги в **C++** є не функції, а об'єкти. Нові типи потрібні для визначення концепції, яку не можна виразити вбудованими типами.

Клас не має фізичної сутності, його найближчою аналогією є структура. Пам'ять виділяється тільки тоді, коли клас використовується для створення об'єкта. Цей процес також називається створенням екземпляра класу (**class instance**).

Будь-який об'єкт мови **C++** має однакові атрибути і функціональність з іншими об'єктами того ж класу.

Працюючи в деякому середовищі, програміст одержує доступ до бібліотек класів (наприклад, до бібліотеки візуальних компонентів **C++Builder** [16] або до стандартної бібліотеки шаблонів **STL** [8,10]).

Зазвичай, об'єкт знаходиться в деякому унікальному стані, який описується поточними значеннями його атрибутів. Функціональність об'єктного класу визначається можливими операціями (діями) над екземпляром цього класу.

Визначення класу в мові **C++** містить інкапсуляцію членів даних і методів, що оперують із членами даних і визначають поведінку об'єкта. У термінології ООП дані називаються членами-даними (**data members**) об'єкта, а коди – членами-функціями або функціями-членами (**methods, member functions**).

Часто функції-члени називають методами. Метод – це функція, в яку додатково неявним параметром передається об'єкт, тобто екземпляр класу. Метод описує дію саме над тим об'єктом, для якого він викликається. Саме прихований зв'язок методу з класом, до якого він належить, виокремлює його з поняття простої функції. Під час виконання методу він має доступ до всіх даних свого класу, хоча і не вимагає явної передачі об'єкта цього класу як параметра. Об'єкт передається методу за допомогою прихованого параметра – вказівника **this** на екземпляр класу. При будь-якому звертанні методу до членів даних класу компілятор генерує спеціальний код, що використовує вказівник **this**.

Інкапсуляція дозволяє ізолювати об'єкт від зовнішнього оточення. Вона підвищує надійність розроблених програм, оскільки локалізовані в об'єкті функції обмінюються з програмою порівняно невеликими порціями даних, причому кількість і тип цих даних контролюється. У результаті заміна чи модифікація функцій і даних, інкапсульованих в об'єкті, як правило, не спричиняє наслідків для програми в цілому, що погано відстежуються. Крім того, в ООП майже не використовуються глобальні змінні. Це робиться з метою підвищення захищеності програм.

Класи в **C++** є природним продовженням структур **struct** у **C**. Тому перш ніж розглядати специфіку розробки класів н а **C++**, побудуємо визначений користувачем тип, створений з використанням структури.

Приклад 1.1.1:

```
// структура TDate із членами-даними
// d-день, m-місяць, y-рік
struct TDate
{
    int d,m,y;
};
//функція ініціалізації змінної date значеннями d,m,y
void init (TDate& date, int d, int m, int y)
{
    date.d =d;
    date.m =(m % 12)>0 ? m % 12 : 12;
    date.y =2000+y;
}
void addYear(TDate& date, int n)
{ //додавання n років до поля y структури date
    date.y +=n;
}
void addMonth(TDate& date, int n)
{ //додавання n місяців до поля m структури date
    date.m +=n;
}
void addDay(TDate& date, int n)
{ // додавання n днів до поля d структури date
    date.d +=n;
}
#include <stdio.h>
int main()
{
    // створення екземплярів (об'єктів)структури TDate
    TDate today, birthday;
    init(today,5,9,13);
    //ініціалізація змінної birthday значеннями d,m,y+2000
    init(birthday,30,13,-8);
    // створення екземпляра структури TDate з назвою
    // tomorrow, яка ініціалізується даними змінної today
    TDate tomorrow=today;
    //додавання одного дня до tomorrow
    addDay(tomorrow,1);
    //друж дня змінної tomorrow
    printf("%d\n", tomorrow.d);
    //друж змінної birthday
    printf("%d %d %d ", birthday.y, birthday.m, birthday.d);
    return 0;
}
```

Зауважимо, що у функції ініціалізації дати рік задається відносно 2000 року, а місяць може бути лише в діапазоні від 1 до 12. Іншого контролю коректності даних ні у функції ініціалізації, ні у функціях додавання не передбачено. Зазначимо, що аналог наведеного вище прикладу можна знайти в [19, 26].

Тут немає явного зв'язку між типом даних і функціями. Такий зв'язок можна встановити, визначивши функції як члени структури:

Приклад 1.1.2:

```
//структура TDate із членами-даними
//d-день, m-місяць, y-рік
struct TDate
{
    int d,m,y;
    void init (int dd,int mm, int yy);
    void addYear(int n);
    void addMonth(int n);
    void addDay(int n);
};
void TDate::init(int dd,int mm,int yy)
{ d=dd;
  m =(mm % 12)>0 ? mm % 12 : 12;
  y =2000+y;
}
void TDate::addYear(int n)
{
    y +=n;
}
void TDate::addMonth(int n)
{
    m +=n;
}
void TDate::addDay(int n)
{
    d +=n;
}
```

Тут всі змінні структурного типу будуть містити як поля **d**, **m**, **y**, так і функції **init()**, **addYear()**, **addMonth()**, **addDay()**.

Оскільки різні структури можуть мати функції-члени з

однаковими іменами, при визначенні функції-члена треба вказати ім'я структури разом з оператором прив'язки – “::” (оператором розширення або дозволу області видимості).

Якщо функції є членами структури, то об'єкт, для якого викликаються ці функції, вже не передається як параметр. Тому тіла цих функцій в прикладі 1.1.2 простіші, ніж у прикладі 1.1.1.

Функції, визначені всередині структури (класу), називаються функціями-членами і їх можна викликати тільки для змінної створеного типу, використовуючи стандартний синтаксис доступу до членів структури. Для доступу до елементів структури (чи класу) використовуються операція *точка* (.) і операція *стрілка* (->). Операція *точка* звертається до елемента структури (чи класу) за іменем змінної (об'єкта). Операція *стрілка* забезпечує доступ до елемента структури (класу) через вказівник на об'єкт.

Наведемо приклад основної програми з використанням створеної структури **TDate** з прикладу 1.1.2.

Приклад 1.1.3:

```
#include <iostream.h>
int main()
{
    // створення екземплярів (об'єктів) структури TDate
    TDate today, birthday;
    today.init(5,9,12);
    //ініціалізація змінної birthday
    birthday.init(30,12,-8);
    // створення екземпляру структури TDate з назвою
    //tomorrow, яка ініціалізується даними змінної today
    TDate tomorrow=today;
    //додавання одного дня до tomorrow
    tomorrow.addDay(1);
    //друк дня змінної tomorrow
    cout << tomorrow.d << endl;
    //друк змінної tomorrow
    cout<<tomorrow.y<<" "<< tomorrow.m <<" "<< tomorrow.d;
    //друк змінної birthday
    cout<<birthday.y <<" "<< birthday.m<<" "<< birthday.d;
    cin.get();
    return 0;
}
```

У тілі функції-члена структури або класу імена членів цього ж класу можна використовувати без явної вказівки об'єкта. Опис структури **TDate** пропонує набір функцій для роботи з датою, але зауважимо, що не тільки вони мають доступ до об'єктів класу **TDate**. Обмежити доступ можна, використавши ключове слово **class** замість **struct**.

1.2. Стандартні об'єкти потокового введення/виведення

Зауважимо, що у прикладах 1.1.1 і 1.1.2 по-різному організовані операції введення/виведення. У прикладі 1.1.1 використовуються функції з бібліотеки мови **C** **stdio**. Для роботи з ними треба підключити заголовочний файл **<stdio.h>**.

У мові **C++** існує бібліотека потокового введення/виведення **iostream**. Для роботи з нею підключається файл заголовка **<iostream.h>**.

Два важливих класи з бібліотеки **iostream** [13]:

- **istream** – уведення зі стандартного пристрою введення, містить перевантажений оператор введення **>>** і функції для неформатованого введення;
- **ostream** – вивід на стандартний пристрій виведення, містить перевантажений оператор виведення **<<** і функції для неформатованого виведення.

Стандартні об'єкти потокового введення/виведення **C++** аналогічні за призначенням до стандартних файлів введення/виведення мови **C**:

- **cin** – стандартне введення (за замовчуванням клавіатура), аналог файла стандартного введення **stdin**;
- **cout** – стандартне виведення (за замовчуванням дисплей), аналог файла стандартного виведення **stdout**;
- **cerr** – стандартне небуферизоване виведення повідомлень про помилки (за замовчуванням дисплей), аналог файла стандартного виведення інформації про помилки **stderr**;

- **clog** – стандартне буферизоване виведення повідомлень про помилки (за замовчуванням дисплей) аналог файла стандартного виведення інформації про помилки **stderr**.

Отже, як видно, потоки виведення можна розділяти за призначенням, перенаправляти на різні пристрої, робити програму більш наочною. Наприклад, якщо виведення відбувається в потік **cerr**, то це одразу сигналізує про помилкову ситуацію у програмі.

При виведенні даних можна керувати форматуванням інформації за допомогою маніпуляторів. Частина маніпуляторів визначена в бібліотеці **iostream**, а частина – в **iomanip**.

Наведемо зміст деяких маніпуляторів:

- **endl** – перехід на новий рядок;
- **dec** – виведення цілого числа в десятковій системі числення;
- **oct** – виведення цілого числа у вісімковій системі числення;
- **hex** – виведення цілого числа в шістнадцятковій системі числення;
- **setprecision(3)** – виведення дійсного числа з трьома знаками після коми.

Як видно з прикладу 1.1.3, потік виведення можна продовжувати й перенаправляти в нього змінні різних стандартних типів.

Аналогічно можна робити і з потоком введення.

Наприклад, у наступному фрагменті коду дані будуть аналізуватись і записуватись у змінні відповідно до їхнього типу.

```
int i, j;
double a;
char s[80];
cin>>i>>a>>j>>s;
```

Зауважимо лише, що у рядок запишеться тільки перше слово до пробільного символу. Щоб записати весь рядок, необхідно написати такий код:

```
cin.get(s, 80);
```

Зазначимо, що у програміста є можливість перевизначати операторні функції `operator<< i operator>>` для власних класових типів. При цьому необхідно дотримуватися вимог синтаксису.

Перевантажений оператор `<< (>>)` повинен повертати посилання на стандартний клас потокового виведення `ostream` (потокового введення `istream`), який визначається в файлі `iostream.h` (у мові C оператори `<< i >>` використовуються для побітового зсуву).

Крім того, операторна функція `operator<< (operator>>)` повинна мати два аргументи:

- перший параметр – посилання на `ostream` (`istream`);
- другий параметр для оператора `operator<<` – значення змінної класового типу, яку треба вивести, або посилання на цю змінну;
- другий параметр для оператора `operator>>` – посилання на змінну класового типу, яку потрібно ввести.

1.3. Перевантаження операцій для типів `struct`, `union` та `enum`

Розглянемо фрагмент основної програми з прикладу 1.1.3.

```
TDate tomorrow=today;  
//додавання одного дня до tomorrow  
tomorrow.addDay(1);  
//друк змінної tomorrow  
cout<<tomorrow.y<<" "  
    << tomorrow.m <<" " << tomorrow.d;
```

Для прикладного програміста, який користується визначеною нами структурою `TDate`, наведений нижче фрагмент коду був би більш зручним і компактним

```
TDate tomorrow=today+1;  
cout << tomorrow;
```

Для надання прикладному програмісту такої можливості необхідно застосувати механізм перевантаження операцій, який можна реалізовувати у C++ для змінних класового типу (**enum**, **union**, **struct**, **class**).

Операторні функції мають зарезервоване ім'я **operator@**, де @ – операція, що перевизначається. Розрізняють унарні (один аргумент) і бінарні (два аргументи) оператори.

Існують правила перевантаження операцій, які розглянемо нижче, коли будемо вивчати класи. А зараз наведемо приклад перевантаження операторів додавання кількості днів “+” і потокового виведення “<<” для структурного типу **TDate**.

Приклад 1.3.1:

```
#include <iostream.h>
```

```
struct TDate
```

```
{
    int d,m,y;
};
```

```
TDate operator+(TDate date, int dd)
```

```
{
    TDate rez=date;
    rez.d+=dd;
    return rez;
}
```

```
ostream& operator<<(ostream& os, TDate date)
```

```
{
    os<<"\n Pk - "<<date.y<<endl<<
    "\n Місяць - "<<date.m<<endl<<
    "\n День - "<<date.d<<endl;
    return os;
}
```

```
int main()
```

```
{
    // створення екземплярів структури TDate
    TDate today ={5,9,2013};
    TDate tomorrow=today+1; // tomorrow=operator+(today,1);
    cout << tomorrow;      // operator<<(cout,tomorrow);
    cin.get();
}
```

```
    return 0;
}
```

Результат роботи програми:

```
Рік - 2013
Місяць - 9
День - 6
```

Зауважимо, що при додаванні кількості днів не зроблено контроль коректності, тому можливими будуть випадки, коли кількість днів перевищує 31.

Не можна перевантажувати операції для неklasових типів. Звідси випливає, що принаймні один з операндів операторної функції має бути класового типу.

Далі покажемо застосування перевантаження оператора потокового виведення **operator<<** для типу об'єднання **union**.

У наведеній нижче програмі визначимо два класових типи – бітове поле й об'єднання. Накладемо введений символ на бітове поле (використаємо одну область пам'яті). Для кожного введеного символу введемо кожен біт його бітового зображення.

Приклад 1.3.2:

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

// бітове поле з іменованими бітами
struct TByte
{
    unsigned a:1, b:1, c:1, d:1, e:1, f:1, g:1, h:1;
};

// об'єднання
// (символ накладається на його бітове зображення)
union TSymbol
{
    unsigned char ch;
    struct TByte bit;
};
```

```

// перевантаження операції потокового виведення
// для об'єднання
ostream& operator<<(ostream& os, TSymbol p)
{
return os<<p.bit.h<<p.bit.g<<p.bit.f<<p.bit.e

<<p.bit.d<<p.bit.c<<p.bit.b<<p.bit.a<<endl;
}

void main()
{
TSymbol symbol;
// поки не натиснуто ESC
while (symbol.ch != 27)
    {
        // читати символ
        symbol.ch=getche();
        putch(':');
        // і виводити його бітове зображення
        cout<<symbol;
    }
return;
}

```

Результат роботи програми:

```

g:01100111
f:01100110
k:11011010

```

У наступному прикладі покажемо перевантаження операторів інкременту для типу перерахування **enum**. Інтерфейс цих операцій для деякого типу **T** повинен мати такий вигляд:

```

T& operator++(T&);           // префіксний інкремент
T operator++(int, T&);       // постфіксний інкремент
T operator++(T&, int);       // постфіксний інкремент

```

Зазначимо, що інтерфейс постфіксного інкременту залежить від версії компілятора, тому наведено дві його можливі форми.

При перевантаженні операцій ++ і -- для деякого типу **T**

буває необхідно визначити їх префіксну й постфіксну форми. Очевидно, що ім'я операторної функції, наприклад **operator++**, однакове для префіксної та постфіксної форм. Щоб компілятор міг розрізнити ці дві форми, у постфіксній формі використовується ще один параметр цілого типу, який ніяк не застосовується, а тільки визначає потрібний вибір операторної функції.

Приклад 1.3.3:

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

// тип перерахування (атрибути файла)
enum TMode
{
    mRead, mEdit, mWrite, mCreate
};

// перевантаження операції потокового виведення
// для типу перерахування enum
ostream& operator<<(ostream& os, TMode mode)
{
    os<< "Виведення режиму роботи\n";
    switch (mode)
    {
        case mRead:
            os<<"Режим читання - 0 \n";
            break;
        case mEdit:
            os<<"Режим редагування - 1 \n";
            break;
        case mWrite:
            os<<"Режим запису - 2 \n";
            break;
        case mCreate:
            os<<"Режим створення - 3 \n";
            break;
    }
    return os;
}
```

```

// перевантаження операції потокового
// виведення для типу enum
istream& operator>>(istream& is, TMode& mode)
{
    cout<< "Виберіть режим роботи з файлом\n"<<
        "Режим читання - 0\n"<<"Режим редагування -1\n"<<
        "Режим запису -2\n"<<"Режим створення - 3\n";
    int m;
    is>> m;
    mode=m%4;
    return is;
}

// префіксний інкремент
TMode& operator++(TMode& mode)
{
    switch (mode)
    {
        case mRead:
            mode=mEdit;
            break;
        case mEdit:
            mode=mWrite;
            break;
        case mWrite:
            mode=mCreate;
            break;
        case mCreate:
            mode=mRead;
            break;
    }
    return mode;
}

// постфіксний інкремент
TMode operator++(TMode& mode, int)
{
    TMode rez=mode;
    switch (mode)
    {
        case mRead:
            mode=mEdit;
            break;
        case mEdit:
            mode=mWrite;

```

```

        break;
    case mWrite:
        mode=mCreate;
        break;

    case mCreate:
        mode=mRead;
}
return rez;
}

void main()
{
// режим кирилиці для консолі (встановити властивість
// шрифту консолі - Lucida Console)
SetConsoleOutputCP(1251);
SetConsoleCP(1251);

TMode model,mode2;
cin>>model;
mode2=++model;
cout<<model;
mode2=model++;
cout<<mode2;
cout<<model;

cin.get();
return;
}

```

Результат роботи програми:

```

Виберіть режим роботи з файлом
Режим читання - 0
Режим редагування -1
Режим запису -2
Режим створення - 3
0
Виведення режиму роботи
Режим редагування - 1
Виведення режиму роботи
Режим редагування - 1
Виведення режиму роботи
Режим запису - 2

```

1.4. Перевантаження функцій

Як правило, різним функціям дають різні імена, але коли функції розв'язують концептуально аналогічні задачі для об'єктів різних типів, зручно використовувати однакові імена.

Наприклад, для мови **C** знаходження абсолютної величини числа вимагає використання трьох різних функцій: **abs**, **labs** і **fabs**. Ці функції повертають абсолютну величину цілих, довгих цілих і дійсних чисел відповідно. У мові **C++** усі ці різні функції можуть мати одне ім'я, наприклад **abs**. Компілятор їх розрізнить по типу вхідного аргумента.

Використання одного імені для операції, що виконується з різними типами даних, називається *перевантаженням* (*перевизначенням*). Цю ідею реалізовано в різних бібліотеках **C++**, її легко поширити на функції, визначені користувачем.

Наприклад, треба написати функції друку інформації про вміст змінних структурного типу **TDate** і змінних типу масив (розмірності 10 типу **float**). Такі функції можна назвати однаково. Компілятор їх розрізнить за типом і кількістю вхідних параметрів.

Приклад 1.4.1:

```
#include <iostream.h>
```

```
struct TDate
{
    int d,m,y;
};

void print(TDate date)
{
    cout<< "Структура: ";
    cout<<"\n Рік - "<<<date.y<<endl<<
        "\n Місяць - "<<<date.m<<endl<<
        "\n День - "<<<date.d<<endl;
}

void print(float* a)
{
    cout<< "Масив: ";
```

```

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

int main()
{
    TDate today={5, 9, 2013};
    float mas[10]={1.2, 44., 2.33, 6.9, -77.5};
    print(today);
    print(mas);
    cin.get();
    return 0;
}

```

Спільне, що мають функції друку, визначені в наведеному вище прикладі, – це їх імена (**print**) і призначення. Тіла ж функцій нічого спільного не мають. Перевантажені (перевизначені для різних типів вхідних параметрів) функції призначені насамперед для зручності запису. Коли викликається перевантажена функція **print()**, компілятор визначає, яку з функцій з іменем **print** використовувати. Застосовується функція з найбільш близькими за типами аргументами, і видається повідомлення про помилку, якщо такої функції не знайдено.

Процес пошуку потрібної з перевантажених функції полягає в знаходженні найкращої відповідності типів формальних і фактичних аргументів. Це здійснюється шляхом перевірки критеріїв [11]:

- точна відповідність типів;
- відповідність, що досягається неявним перетворенням типів (**char** у **int**, **int** у **unsigned int**, **bool** у **char** і т.д.) та стандартним перетворенням типів (**int** у **double**, **double** у **int**);
- відповідність, що досягається перетворенням вказівників на довільні типи в **void***, вказівників на похідні класи – у вказівники на базові.

Результат перевантаження не залежить від порядку оголошення функцій. Типи, що повертаються, не є критерієм вибору перевантажених функцій. Функції, оголошені в різних областях видимості, не перевантажуються.

1.5. Приклад перевантаження операторів для двох структурних типів

Умова задачі: Для вектора дійсних чисел і рядка різних символів перевизначити операції:

- “*” – скалярний добуток векторів;
- “*” – множення вектора на ціле число і навпаки;
- “*” – множення рядків (у рядку з результатом залишаються лише спільні символи);
- “<<”, “>>” – введення, виведення масиву.

Приклад 1.5.1:

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

struct TStr
{
    char s[80];
    int lenr;
};

struct TVec
{
    float v[50];
    int lenv;
};

float operator*(TVec v1, TVec v2)
{
    if(v1.lenv == v2.lenv)
    {
        float skal=0.;
        for(int j=0;j<v1.lenv;j++)
            skal+=v1.v[j]*v2.v[j];
    }
}
```

```

        return skal;
    }

    else
    {
        cerr<<"\nРозмірність не збігається "<<endl;
        return 0;
    }
}

TStr operator*(TStr s1, TStr s2)
{
    TStr s3;
    int k=0;

    for(int i=0;i<s1.lenr;i++)
        for(int j=0;j<s2.lenr;j++)
            if(s1.s[i]==s2.s[j])
                s3.s[k++]=s1.s[i];
    s3.s[k]='\x0';
    s3.lenr=k;
    return s3;
}

TVec operator*(int k, TVec v1)
{
    TVec v2;
    for(int i=0;i<v1.lenv;i++)
        v2.v[i]=k*v1.v[i];
    v2.lenv=v1.lenv;
    return v2;
}

TVec operator*(TVec v, int k)
{
    return k*v;
}

ostream& operator<<(ostream& os, TVec v)
{
    os<<"\nВиведення вектора"<<endl
    <<"\nРозмірність n="<<v.lenv<<endl;

    for(int i=0;i<v.lenv;i++)
        os<<v.v[i]<<" ";

```

```

return os;
}

istream& operator>>(istream& is, TVec& v)
{
    cout<<"\nВведення вектора"<<endl<<"\nРозмірність n=";
    is>>v.lenr;

    cout<<"\nВведення елементів"<<endl;
    for(int i=0;i<v.lenr;i++)
        is>>v.v[i];
    return is;
}

int main()
{
    TStr s1,s2,s3;
    TVec v1,v2,v3,v[5];

    strcpy(s1.s,"abcd");
    s1.lenr=strlen(s1.s);

    strcpy(s2.s,"aefdl");
    s2.lenr=strlen(s2.s);

    s3=s1*s2;

    cout<<s1.s<<endl;
    cout<<s2.s<<endl;
    cout<<s3.s<<endl;

    cin>>v1>>v2>>v3;

    float f=v1*v2;

    cout<<f<<endl;
    cout<<3*v3<<endl;

    int i;
    for(i=0;i<5;i++)
        cin>>v[i];

    for(i=0;i<5;i++)
        cout<<v[i]*i;

```

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

Як видно з прикладу, робота з векторами організована зручніше, ніж з рядками, за рахунок написаних операцій перевантаженого потокового введення і виведення векторів.

Для того, щоб була можливість використовувати перевантаження операторів, масив і рядок необхідно було помістити у структури.

Зауважимо, що у прикладі реалізовано чотири оператори з однаковими іменами **operator***, які розрізняються компілятором за вхідними параметрами.

1.6. Шаблони функцій

Якщо виникає необхідність у розробці кількох однакових функцій, які відрізняються тільки типом вхідних параметрів, в C++ передбачено можливість створення шаблону (**template**) таких функцій для деяких узагальнених типів параметрів. За допомогою шаблону компілятор сам генерує функції з необхідними типами вхідних параметрів, причому тип параметрів визначається при відповідному виклику функції. При описі шаблону функції запис **<class T>** або **<typename T>** означає оголошення типу **T**, який невідомий, але при виклику функції з явним аргументом компілятор автоматично підставить відповідний тип.

Приклад 1.6.1:

```
#include <iostream.h>
// оголошення шаблону функції для обміну значеннями
// між першим і останнім елементами масиву
template <typename T, int size> void firstSwapLast(T* x)
{
    T temp = x[0];
    x[0] = x[size-1];
    x[size-1] = temp;
}
```

```

int main()
{
char* s="abcdefghkl";
float* mas={1.2,44.,2.33, 6.9, -77.5};

firstSwapLast<float,5>(mas);
cout<< "Перетворений масив: ";
for(int i=0;i<5;i++)
    cout<<mas[i]<<" ";
cout<<endl;

firstSwapLast<char,10>(s);
cout<<"Перетворений рядок: "<<s;

cin.get();
return 0;
}

```

Як видно з прикладу, у параметрах налагодження шаблону можна використовувати не лише узагальнені типи, а й реальні. Крім того, параметрів налагодження шаблону може бути кілька (наприклад, `<class T1, class T2, class T3>`) і при використанні (конкретизації шаблону) їх можна опускати, якщо в кутових дужках у шаблоні всі типи задаються як узагальнені за допомогою ключового слова **class** або **typename**.

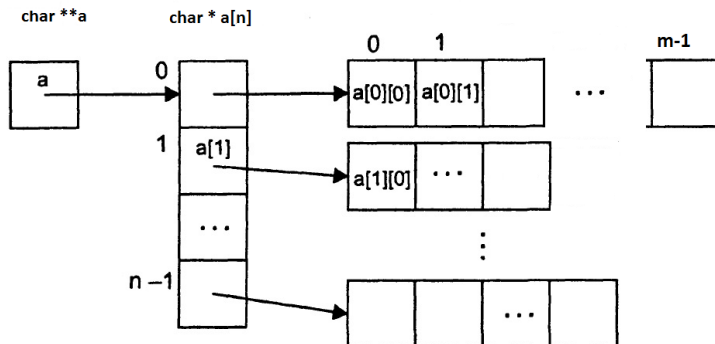
1.7. Приклад перевантаження функцій і використання шаблонів функцій

Умова задачі: Створити функції, які впорядковують динамічні масиви, що складаються з елементів типу **int**, **float** або **char***.

Довжини масивів задаються при введенні.

Використати перевантаження функцій і шаблони.

Зауважимо, що алгоритм введення динамічного масиву рядків змінної довжини можна проілюструвати за допомогою наведеної нижче схеми.



Приклад 1.7.1:

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
typedef char* str;

// шаблон для введення динамічних числових масивів
// (без вказівки типу "посилання" не буде працювати)
template <class T>int input(T*& a)
{
    cout<<"\nВведіть розмірність масиву ";
    int n;
    cin>>n;
    a=new T[n];
    cout<<"\nВведіть елементи масиву "<<endl;
    for(int i=0;i<n;i++)
        cin>>a[i];
    return n;
}

// шаблон для виведення динамічних числових і
// рядкових масивів
template <class T> void output(T* a, int n)
{
    for(int i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}

// шаблон для обміну значень двох змінних
template <class T> void swap (T & a, T & b)
{
```

```

        T c=b;
        b=a;
        a=c;
    }

// шаблон для сортування числових масивів
template <class T> void sort(T* a, int n)
{
    int check;
    do
    {
        check=0;
        for(int i=0;i<n-1;i++)
            if(a[i+1]<a[i])
            {
                swap(a[i+1],a[i]);
                check=1;
            }

        n--;
    }while(check);
}

// функція для введення динамічного
// масиву рядків змінної довжини
int input(str*& a) //без посилання не буде працювати
{
    cout<<"\nВведіть кількість і максимальну довжину рядків
    ";

    int n,m;
    cin>>n>>m;
    a=new str[n];
    str buffer=new char[m+1];
    cout<<"\nВведіть рядки "<<endl;
    for(int i=0;i<n;i++)
    {
        cin>>buffer;
        a[i]=new char[strlen(buffer)+1];
        strcpy(a[i],buffer);
    }
    delete[] buffer;
    return n;
}

// функція для сортування динамічного масиву рядків
// змінної довжини

```

```

void sort(str* a,int n)
{
    int check;
    do
        {check=0;
        for(int i=0;i<n-1;i++)
            if(strcmp(a[i+1],a[i])<0)
            {
                swap(a[i+1],a[i]);
                check=1;
            }
        n--;
    }while(check);
}

// шаблон для введення, сортування і виведення
// динамічних масивів чисел або рядків змінної довжини
template <class T> void InputSortOutput()
{
    T mas;
    int size=input(mas);
    sort (mas,size);
    output(mas,size);
}

int main(void)
{
    // режим кирилиці для консолі (встановити властивість
    // шрифту консолі - Lucida Console)
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);

    // конкретизація шаблонів
    InputSortOutput<str*>();
    InputSortOutput<int*>();
    InputSortOutput<float*>();

    cin.get();
    return 0;
}

```

У наведеній вище програмі для роботи з масивами цілих і дійсних чисел створено шаблони. Оскільки динамічний масив рядків змінної довжини вводиться й сортується не так, як

числові масиви, то для реалізації введення та сортування написано функції, імена яких збігаються з іменами відповідних шаблонів для числових масивів. Виведення ж всіх трьох типів масивів можна реалізовувати однаково, тому для цієї операції написано один шаблон.

Нарешті, створено шаблон **InputSortOutput**, який тричі конкретизується для роботи з всіма потрібними типами динамічних масивів.

Запитання для самоконтролю

1. Що таке об'єкт? Як його можна описати?
2. Чим структура відрізняється від класу?
3. Чи можна перевантажувати операції для типів "масив цілих чисел" і "масив дійсних чисел"? А функції?
4. Чи можна створити шаблон для сортування масиву слів і масиву чисел? А для друку цих масивів?
5. Що таке шаблон функції?
6. Які ключові слова можна використовувати для позначення деякого узагальненого типу в шаблоні функції?
7. Скільки може бути параметрів у шаблоні?
8. Що таке конкретизація шаблону при виклику функції?
9. Коли можна опускати фактичні значення параметрів при генерації конкретної функції?
10. Чи можна визначати кілька операторних функцій з одним іменем для одної структури?
11. Які типи даних вважаються класовими?
12. Для яких типів даних можна застосовувати перевантаження функцій? А операторів?
13. Чи можна змінювати пріоритет виконання операцій при їх перевантаженні?
14. Скільки параметрів може бути в перевантаженій операторній функції **operator+**?
15. Чи можна використовувати ім'я **operator**** як ім'я операторної функції?

▣ Завдання для самостійної роботи

1. Напишіть функцію `addWeek()` додавання одного тижня до дати для структури з п. 1.1. Реалізуйте її спочатку як зовнішню функцію, а потім як функцію-член структури.
2. Доповніть приклади 1.3.1 і 1.5.1 для структур `TDate` і `TStr` оператором `operator>>` перевантаженого потокового введення дати і рядка відповідно.
3. Реалізуйте перевантаження оператора потокового виведення `operator<<` для типу об'єднання `union`, як це було зроблено в п. 1.3. Для кожного введенного цілого числа (тип `char`) виведіть його бітове зображення.
4. Напишіть шаблон для сортування і друку масивів з п. 1.6 і продемонструйте його конкретизацію для них.
5. Реалізуйте звільнення пам'яті, виділеної під динамічні дані з п. 1.7. Чи можна це зробити в шалоні?

2. Робота з об'єктами в межах одного класу

2.1. Управління доступом до членів класу

Утворені класи можна використовувати далі для розробки складних програм та для створення нових класів з уже існуючих (*успадкування*). Тому питання доступу до членів класу має суттєве значення.

Кожне оголошення всередині класу визначає привілей доступу до імен класу залежно від того, в якій секції ім'я з'являється. Кожна секція розпочинається з одного з ключових слів: **private**, **protected** і **public**.

Конструкція вигляду

```
class <className>
{
    [private:]
        <приватні члени-дані>
        <приватні конструктори>
        <приватні члени-функції>
```

```
protected:
    <захищені члени-дані>
    <захищені конструктори>
    <захищені члени-функції>
public:
    <загальнодоступні члени-дані>
    <загальнодоступні конструктори>
    <загальнодоступний деструктор>
    <загальнодоступні члени-функції>
};
```

називається визначенням (об'явою, оголошенням, інтерфейсом) класу, тому що вона визначає новий тип.

Поля класу можуть мати будь-який тип, окрім типу цього ж класу (але можуть бути вказівниками або посиланнями на цей клас). Ініціалізація полів при описі не допускається.

Будь-яка секція може бути відсутньою або зустрітися не один раз. Порядок розміщення секцій неістотний.

Захищена секція **protected** буде використовуватися далі при успадкуванні. Якщо клас не успадковується, то секція **protected** діє аналогічно секції **private**.

Щоб пояснити дію специфікаторів доступу **public** і **private**, розглянемо тип даних, який наведений у прикладі 1.1.2, але замінимо ключове слово **struct** на **class** і члени-дані розташуємо в закритій (приватній) секції класу, а члени-функції – у відкритій (доступній) секції.

Приклад 2.1.1:

```
#include <iostream.h>
// клас TDate із членами-даними d-день, m-місяць, y-рік
class TDate
{
    int d,m,y;
public:
    void init (int dd,int mm, int yy);
    void addYear(int n){y+=n;}
    void addMonth(int n){m+=n;}
    void addDay(int n){d+=n;}

};
void TDate::init(int dd,int mm,int yy)
```

```

{ d=dd;
  m=mm;
  y=yy;
}
. . .
int main()
{
  TDate today={5,9,2013}; //помилка (члени-дані недоступні)
  TDate today;
  today.d=5;               // помилка (змінна d недоступна)
  today.init(5, 9, 2013); // правильно
  cin.get();
  return 0;
}

```

Приклад 2.1.2:

```

#include <iostream.h>
// клас TDate із членами даними d-день, m-місяць, y-рік
class TDate
{
    int d,m,y;
public:
    // функції доступу до змінної m-місяць
    void setM(int m){this->m=(m % 12)>0 ? m % 12 : 12;}
    int getM(){return m;}
    char* getM(int);
};

char* TDate::getM(int)
{
    char* season=new char[7];
    switch(m)
    {
    case 12:; case 1:; case 2: strcpy(season,"winter");
        break;
    case 3:; case 4:; case 5: strcpy(season,"spring");
        break;
    case 6:; case 7:; case 8: strcpy(season,"summer");
        break;
    case 9:; case 10:; case 11: strcpy(season,"autumn");
        break;
    }
    return season;
}

```

```

int main()
{
    TDate day;
    day.setM(12);
    cout<<day.getM()<<endl;
    cout<<day.getM(1)<<endl;
    day.setM(55);
    cout<<day.getM()<<endl;
    cout<<day.getM(1)<<endl;
    cin.get();
    return 0;
}

```

У наведеному прикладі є дві функції з іменем **getM()**. Вони розрізняються вхідним параметром і повертають номер місяця або пору року, залежно від параметра. Функція доступу **setM()** заносить коректне значення номера місяця.

Таким чином, оголошення базового класу на **C++** надає такі права доступу й відповідні області видимості:

- *приватні* (**private**) імена мають найбільш обмежений доступ, дозволений тільки функціям-членам і друзям даного класу. Доступ похідних класів до приватних методів базових класів заборонений;
- *захищені* (**protected**) імена мають доступ, дозволений членам-функціям даного і похідних від нього класів (і друзям);
- *загальнодоступні* (**public**) імена мають необмежений доступ, дозволений методам усіх класів й їхніх об'єктів.

Наступні правила застосовуються при утворенні різних секцій в оголошенні класу:

- секції можуть з'являтися в будь-якому порядку, а їхні назви можуть зустрічатися повторно;
- якщо секція не названа, то компілятор вважає наступні оголошення імен класу приватними (**private**). Тут виявляється відмінність оголошень класу і структури – остання розглядається за замовчуванням як загальнодоступна;
- якщо члени даних класу не використовуються в програмі, їх оголошують захищеними, щоб дозволити

доступ тільки методам похідних класів.

Використання секцій має ряд переваг.

По-перше, якщо треба змінити реалізацію класу (наприклад, по-іншому реалізувати сортування), то потрібно лише змінити члени-функції. Код же користувача класу не зміниться, оскільки він залежить тільки від відкритого інтерфейсу, і не потрібно буде його переписувати (хоча може знадобитися його перекомпіляція).

По-друге, щоб навчитися використовувати клас, його потенційному користувачу буде потрібно ознайомитися тільки з прототипами відкритих функцій-членів.

За замовчуванням усі елементи класу приватні, а елементи структури – відкриті. Тому наведені нижче конструкції еквівалентні.

```
class My {public: ...}; ~ struct My { ... };
```

Звідси можна зробити висновок, що структура – це клас, члени якого оголошені як **public**.

Об'єкт є екземпляром класу. Визначення класу не створює об'єктів даного класу. Вони створюються під час опису (відбувається виділення пам'яті). Кожен об'єкт має власний набір елементів.

2.2. Конструктори

Використання функцій типу **init()** з прикладу 2.1.1 для ініціалізації об'єктів класу часто призводить до помилок. Правила синтаксису не передбачають, що об'єкт повинен бути проініціалізований, тому програміст може помилково працювати з неініціалізованим об'єктом або ініціалізувати об'єкт двічі. В **C++** передбачено можливість оголосити спеціальну функцію, що має явне призначення – створення та ініціалізація об'єктів. Оскільки така функція створює (конструює) значення даного типу, то вона називається конструктором. Конструктор

розпізнається за іменем, що збігається з назвою самого класу. Якщо клас має конструктор, усі об'єкти цього класу будуть проініціалізовані. Клас може містити будь-яку кількість конструкторів або не містити жодного. Конструктори не можуть бути оголошені віртуальними.

Є три види конструкторів:

- *конструктор за замовчуванням* не має параметрів. Якщо клас не містить жодного конструктора (як у прикладі 2.1.1), то компілятор автоматично створить один конструктор за замовчуванням, який просто виділяє пам'ять при створенні об'єкта свого класу;

- *конструктор з аргументами (з параметрами)* дозволяє ініціалізувати об'єкт у момент його створення, використовуючи отримані параметри – викликати різні функції, виділяти динамічну пам'ять, присвоювати змінним початкові значення і т.д;

- *конструктор копіювання (генератор копій)* призначений для створення об'єктів даного класу шляхом копіювання даних з іншого, вже існуючого об'єкта цього класу. Такі конструктори особливо доцільні, навіть необхідні, для створення копій об'єктів, що моделюють динамічні структури даних.

Конструктори підкоряються тим же правилам переважання, що й інші функції. Оскільки конструктори розрізняються типами своїх аргументів, то компілятор вибирає потрібний конструктор у кожному конкретному випадку.

Локальні змінні іноді називаються автоматичними об'єктами, а об'єкти, створювані в області вільної пам'яті, – динамічними і не ініціалізуються за замовчуванням. Конструктори для автоматичних об'єктів не повертають значення, відповідають за створення екземпляра класу, а пам'ять виділяється при вході в блок зі стека.

Доповнимо приклад 2.1.1 конструкторами і функцією друку, а функцію ініціалізації вилучимо:

Приклад 2.2.1:

```
#include <iostream.h>

//структура strDate із членами-даними
// d-день, m-місяць, y-рік

struct TStrDate
{
int d,m,y;
}today={5,9,2013};
//клас TDate із членами-даними d-день, m-місяць, y-рік
class TDate
{
    int d,m,y;

public:

//конструктор, якому передаються три
// аргументи цілого типу
    TDate(int d,int m,int y);
//конструктор, якому передається один аргумент
// типу TStrDate
    TDate(TStrDate);
//конструктор за замовчуванням
    TDate(){d=today.d; y=today.y; m=today.m;};
//конструктор з параметром дата у символьному вигляді
    TDate(const char*); // поки що - тільки прототип
    TDate(const TDate& obj); //конструктор копіювання
    void print();
    void addYear(int n){y+=n;}
    void addMonth(int n){m+=n;}
    void addDay(int n){d+=n;}
};

TDate::TDate(int d,int m,int y)
{
    this->d = d;
    this->m = m;
    this->y = y;
}

TDate::TDate(TStrDate strD)
{
    d = strD.d;
    m =strD.m;
```

```

        y =strD.y;
    }

TDate::TDate(const TDate& obj)
{
    d = obj.d;
    m =obj.m;
    y =obj.y;
}

void TDate::print()
{
    cout<<"\n Рік - "<< y << endl <<
        "\n Місяць - "<< m << endl <<
        "\n День - "<< d << endl;
}

int main()
{
    // виклик конструктора з трьома параметрами
    TDate day1(30,12,2013);

    // виклик конструктора з параметром типу TStrData
    TDate day2(today);

    // виклик конструктора копіювання
    TDate day3(day2);
    day3.addMonth(2);

    // також виклик конструктора копіювання (не присвоєння)
    TDate day3=day2;
    day3.addMonth(2);

    // виклик конструктора за замовчуванням 5 разів
    TDate days[5];

    // друк результатів
    day1.print();
    day2.print();
    day3.print();
    for(int i=0; i<5; i++)
        days[i].print();
    cin.get();
    return 0;
}

```

Зазначимо, що існують випадки, коли конструктор копіювання викликається неявно. Це відбувається при передачі об'єкта у функцію за значенням і при поверненні об'єкта з функції оператором **return**.

Існують правила, за якими аргументу може бути присвоєне значення за замовчуванням. Якщо параметри за замовчуванням використовуються при виклику конструктора, то це не повинно призводити до конфліктів з викликом конструктора без параметрів чи з викликом іншого конструктора з параметрами, де також використовуються параметри за замовчуванням.

Перепишемо клас з прикладу 2.2.1. При цьому застосовуємо для деяких аргументів значення за замовчуванням і члени-дані **d**, **m**, **y** замінимо на одну змінну **date** структурного типу **TStrDate**.

Приклад 2.2.2:

```
#include <iostream.h>
//структура TStrDate із членами-даними
// d-день, m-місяць, y-рік
struct TStrDate
{
    int d,m,y;
    }today={5,9,2013};

// клас TDate із членом-даних date
class TDate
{
    TStrDate date;
public:
    //конструктор, якому передаються три аргументи
    // цілого типу;
    //значення за замовчуванням (ім'я аргумента можна не
    // вказувати)
    TDate(int d,int m=9,int = 2000);
    // конструктор, якому передається один аргумент
    // типу TStrDate
    // конфліктує з конструктором за замовчуванням
    // TDate(TStrDate=today);
    TDate(TStrDate);
    //конструктор за замовчуванням
    TDate()
```

```

        {
            date.d=date.m=1;
            date.y=2000;
        }
// конструктор з параметром дата у символному вигляді
TDate(const char*); // поки що тільки прототип
TDate(const TDate& obj); //конструктор копіювання
void print();
void addYear(int n=1){date.y+=n;}
void addMonth(int n){date.m+=n;}
void addDay(int n){date.d+=n;}
};

TDate::TDate(int d,int m,int y)
{ date.d = d;
  date.m = m;
  date.y = y;
}

TDate::TDate(TStrDate strD)
{
  date=strD;
}

TDate::TDate(const TDate& obj)
{
  date = obj.date;
}

void TDate::print()
{
  cout<<"\n Рік - "<< date.y << endl <<
    "\n Місяць - "<< date.m << endl <<
    "\n День - "<< date.d << endl;
}

int main()
{
// три виклики конструктора з трьома параметрами
// є параметри за замовчуванням
TDate day1(30,12,2013);
TDate day11(30,12);
TDate day12(25);
// TDate day12(5,,2005); помилка
day1.print();

```

```

    day11.print();
    day12.print();

    // виклик конструктора з параметром типу TStrData
    TDate day2(today);
    // виклик конструктора копіювання
    TDate day3(day2);
    day3.addMonth(2);
    // є параметр за замовчуванням
    day3.addYear();
    day3.print();

    cin.get();
    return 0;
}

```

У двох останніх прикладах об'єктні змінні типу **TDate** залежать від глобальної змінної **today**. Клас **TDate** тепер можна застосовувати тільки там, де змінна **today** визначена. А це робить клас **TDate** незручним для використання. Для розв'язання цієї проблеми існують статичні члени класу.

2.3. Статичні члени класу

Змінна, яка є частиною класу, але не є частиною об'єкта цього класу, називається статичним членом-даним класу. Існує тільки одна копія статичного члена, на відміну від звичайних членів, коли кожен об'єкт класу має свої незалежні члени. Аналогічно, функція, якій потрібен доступ до членів класу, але не потрібно, щоб вона викликала для конкретного об'єкта класу, називається статичною функцією-членом.

Для опису статичних елементів використовується ключове слово **static**.

На статичні члени класу поширюється дія специфікаторів доступу, тому статичні члени-дані, описані як **private**, можна змінити тільки за допомогою статичних членів-функцій.

Пам'ять, яка займається статичним членом-даним, не враховується при визначенні розміру об'єкта за допомогою операції **sizeof**.

Додамо в клас **TDate** статичну змінну **today** та статичні функції **setToday()** і **getToday()**.

Приклад 2.3.1:

```
#include <iostream.h>
// структура TStrDate із змінними
// d-день, m-місяць, y-рік
struct TStrDate
{
    int d,m,y;
};

// клас TDate із змінною date
class TDate
{
    TStrDate date;
    //статична змінна
    static TStrDate today;
public:
    // конструктор, який приймає один аргумент типу TStrDate
    TDate(TStrDate strD) {date=strD;}
    //конструктор за замовчуванням
    TDate() {date=today;};
    void print();
    //статичні функції
    static void setToday(int d,int m,int y);
    static TStrDate getToday(){return today; }

};

//ініціалізація статичної змінної
TStrDate TDate::today={5,9,2013};

//опис статичної функції
void TDate:: setToday(int d,int m,int y)
{
    today.d=d;
    today.m=m;
    today.y=y;
}

void TDate::print()
{
    cout<<"\n Рік - "<< date.y << endl <<
```

```

        "\n Місяць - "<< date.m << endl <<
        "\n День - "<< date.d << endl;
    }

int main()
{
//виклик конструктора за замовчуванням
    TDate day1;
    day1.print();
    TDate day2;

//виклик статичної функції через ім'я класу
    TDate::setToday(8,3,2013);
    day2.print();

//виклик статичної функції через ім'я об'єкта
    day2.setToday(23,2,2013);
    TDate day3(TDate::getToday());
    day3.print();
    cin.get();
    return 0;
}

```

З прикладу видно, що до статичних членів можна звертатися так само, як і до будь-яких інших членів. Крім того, до статичного члена можна звертатися без вказівки імені об'єкта. Замість цього як кваліфікатор його імені використовується ім'я самого класу з оператором прив'язки "::".

2.4. Деструктори

Конструктори ініціалізують об'єкт, тобто вони створюють середовище, в якому працюють функції-члени. Іноді створення такого середовища передбачає виділення ресурсів (файли, пам'ять), що повинні бути звільнені після їхнього використання. Тому деяким класам потрібна функція, яка гарантовано буде викликана при знищенні об'єкта. Такі функції називають деструкторами. Вони очищують пам'ять і звільняють ресурси.

Деструктор – це функція-член класу, що не має параметрів і не повертає значення. Ім'я деструктора складається із символу "~" та імені класу. Клас повинен містити лише один деструктор.

Якщо деструктор явно не визначено, компілятор

автоматично створює порожній деструктор.

Деструктори викликаються автоматично (неявно), коли об'єкт виходить з області видимості:

- для глобальних об'єктів – при виході з функції **main()**;
- для локальних об'єктів – при виході з блоку, в якому вони оголошені;
- для об'єктів, заданих через вказівники (створених динамічно), деструктори викликаються при виконанні оператора **delete**.

Приклад 2.4.1:

```
#include <iostream.h>
```

```
// структура TStrDate із змінними d-день, m-місяць, y-рік
struct TStrDate
{
    int d,m,y;
};
```

```
//клас TGroupDate із масивом днів народження
//студентів групи
class TGroupDate
{
    // динамічний масив днів народження студентів групи
    TStrDate* dates;
    // кількість студентів у групі
    int count;
```

```
public:
    //конструктор за замовчуванням
    TGroupDate(){count=0; dates=NULL;};
    //конструктори з параметрами
    TGroupDate(int size, TStrDate* mas);
    TGroupDate(int size){dates=new
TStrDate[count=size];}
    //деструктор
    ~TGroupDate(){delete[] dates;};
    void input();
    void print();
};
```

```
//конструктор, який створить динамічний масив
//днів народження студентів, використовуючи масив mas
```

```

TGroupDate::TGroupDate(int size, TStrDate* mas)
{
    count=size;
    dates=new TStrDate[count];
    for (int i=0;i<count;i++)
        dates[i]=mas[i];
}

void TGroupDate::print()
{
    for (int i=0;i<count;i++)
        cout<<"\n" << i+1 <<"") Пік - "<< dates[i].y <<
            " Місяць - "<< dates[i].m <<
            " День - "<< dates[i].d << endl;
}

void TGroupDate::input()
{
    delete[] dates;
    cout<< "\n Введіть кількість студентів у групі: ";
    cin>>count;
    dates=new TStrDate[count];
    cout<< "\n Введіть дні народження (день, місяць, рік)
            студентів за списком у групі: ";
    for (int i=0;i<count;i++)
        cin >> dates[i].d >> dates[i].m >> dates[i].y;
}

int main()
{
    cout<< "\n Група 201\n";
    TGroupDate days;
    days.input();
    days.print();

    cout<< "\n Група 202\n";
    TStrDate mas[]={14,4,1993},{3,11,1994},{23,10,1994}};
    TGroupDate days202(3,mas);
    days202.print();
    cin.get();
    return 0;
}

```

У даному прикладі описано клас, який працює з динамічним масивом днів народження студентів деякої групи. Конструктори класу виділяють місце під динамічний масив

dates, деструктор **~TGroupDate()** звільняє пам'ять, на яку вказує вказівник **dates**. Зазначимо, що введення даних про об'єкт і його конструювання – дві різні операції. Функція введення може викликатися лише для існуючого об'єкта, для якого було виділено пам'ять. Пам'яті може не вистачити для іншого об'єкта, тому у функції **input()** звільнено стару пам'ять і виділено нову під потрібний об'єкт.

2.5. Динамічне створення та знищення об'єктів

У мові **C++** існують оператори **new** і **delete**, які можна використовувати замість бібліотечних функцій мови **C** **malloc()**, **calloc()**, **free()**.

Оператор **new** виділяє блок пам'яті, необхідний для розміщення змінної або масиву відповідного типу (розмірності). Він може також присвоювати початкове значення.

```
// виділити пам'ять для змінної типу int
int *p1=new int;
// виділити пам'ять для змінної типу double
// та ініціалізувати її
double *p2=new double(25.7415) ;
// виділити пам'ять для рядка
char *p3=new char[80];
// виділити пам'ять для масиву дійсних
// чисел розмірності 50
float *p3=new float[50];
```

Тут вказівники **p1**, **p2**, **p3**, **p4** вказують на адреси виділених блоків пам'яті.

Звільнити пам'ять можна за допомогою оператора **delete**.

```
delete p1;
delete p2;
delete[] p3;
delete[] p4;
```

Зауважимо, що оператори **new** і **delete** працюють набагато ефективніше, ніж бібліотечні функції **malloc()**,

calloc(), **free()**. Крім того, їх, як й інші оператори, можна перевантажувати для власних класових типів (див. п. 2.11).

Використовуючи оператори **new** і **delete**, можна викликати потрібні конструктори і деструктор а, коли виникає необхідність, динамічно створювати об'єкти.

Простежити повний цикл існування екземпляра класу можна, якщо додати в тіла конструкторів і деструктора друк інформацію про початок існування об'єкта (у конструкторах) і про припинення його існування (у деструкторі).

Приклад 2.5.1:

```
#include <iostream.h>
#include <stdlib.h>
class TDemo
{
    private:
        char *name;
        static int number;
    public:
        TDemo();
        ~TDemo();
};
// реалізація конструктора
TDemo::TDemo()
{
    name=new char[10];
    char s[4];
    itoa(++number,s,10);
    strcpy(name,"Object");
    name=strcat(name,s);
    cout << "Creating " << name << endl;
}

TDemo::~TDemo() // реалізація деструктора
{
    cout << "Deleting " << name << endl;
    delete[] name;
}

int TDemo::number=0;

int main()
{
```

```

TDemo *p1,*p2;
    { // блок існування об'єктів objects i *p1
      // створення об'єктів №№1-3

TDemo objects[3];

// створення об'єкта №4
p1=new TDemo;
...
// знищення об'єкта №4
delete p1;
}

//динамічне створення і знищення об'єкта *p2
p2=new TDemo;          // створення об'єкта №5
...
delete p2;             // знищення об'єкта №5

cin.get();    // затримка вікна з результатом
return 0;
}

```

Результат роботи програми:

```

Creating Object1
Creating Object2
Creating Object3
Creating Object4
Deleting Object4
Deleting Object3
Deleting Object2
Deleting Object1
Creating Object5
Deleting Object5

```

Іноді, працюючи в режимі **Console** у середовищі **C++Builder**, виникає необхідність намалювати деякий об'єкт. Для реалізації такої задачі, як правило, застосовуються функції **Windows GDI**.

Іншим способом є використання властивості **Canvas**, наявної у багатьох компонентах бібліотеки **VCL (Visual Component Library)**.

Для доступу до властивості **Canvas** можна за допомогою оператора **new** динамічно створити форму, яка є компонентним класом, і далі через вказівник звертатися до необхідних властивостей і методів класу **Canvas**.

Приклад 2.5.2:

```
#include <stdio.h>
#include <vcl.h>
#include <iostream.h>
#include <conio.h>

int main(int argc, char* argv[])
{
    // динамічне створення форми
    TForm* pForm;
    pForm = new TForm (Application);
    pForm->Caption =
        "Графіка у режимі Console на динамічно створеній формі";
    pForm->Width=450;
    pForm->Height= 300;
    pForm->Top=400;
    pForm->Left=400;
    pForm->Visible=true;
    // колір контуру (перо)
    pForm->Canvas->Pen->Color = clRed;

    // колір заливки (пензель)
    pForm->Canvas->Brush->Color = clYellow;

    // жовтий еліпс, обведений червоним контуром
    pForm->Canvas->Ellipse(10, 10, 100, 100);

    // колір контуру (перо)
    pForm->Canvas->Pen->Color = clBlue;

    // колір заливки (пензель)
    pForm->Canvas->Brush->Color = clRed;

    // червоний прямокутник, обведений синім контуром
    pForm->Canvas->Rectangle(200, 110, 120, 220);

    // кольори - у вікні Console
    // білий текст на чорному фоні
```

```

cout<<"without color"<<endl;
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_GREEN | BACKGROUND_RED);

// зелений текст на червоному фоні
cout<<"with color"<<endl;

SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_GREEN | FOREGROUND_RED | BACKGROUND_BLUE);
// жовтий текст на синьому фоні
cout<<"with color"<<endl;

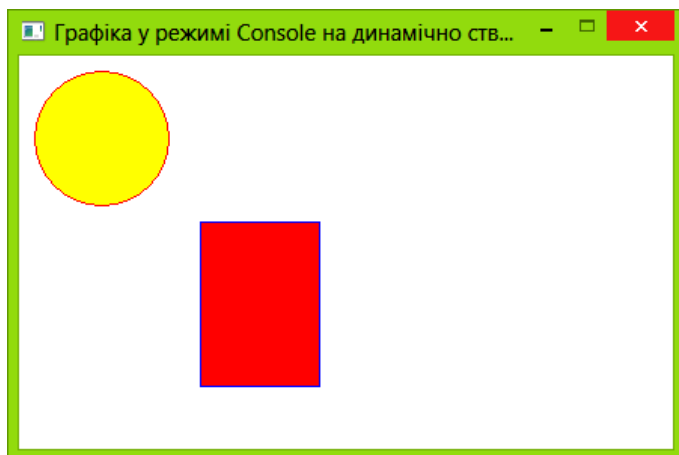
// білий текст на чорному фоні
SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),
FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE );
cout<<"without color"<<endl;

// затримка часу
getch();

// знищення форми
pForm->Hide();
pForm->Release();
return 0;
}

```

Результат графічного виведення на динамічно створеній формі:



2.6. Друзі класу

Часто виникає необхідність у написанні функцій, які мають доступ до приватних секцій кількох класів, однак членом двох класів функція бути не може. Тому в C++ передбачено засіб мови, що надає іншому класу або функції право доступу до закритої частини класу за допомогою зарезервованого слова **friend**.

Функція-не-член, що одержала право доступу до закритої частини класу, називається другом класу (дружною функцією).

Функція-друг – це звичайна функція, вона не має ніяких особливостей, крім права доступу до закритої частини класу. Зокрема, **friend**-функція не має вказівника **this**, оскільки вона не є повноправним членом-функцією. Ключове слово **friend** задається у прототипі функції перед її іменем і типом, що повертається. Опис дружньої функції може розташовуватися у закритій чи відкритій частині опису класу – де саме, значення не має. При написанні тіла дружньої функції оператор прив'язки не пишеться, тому що функція не є членом класу.

Наприклад, перевантажимо оператор, який знаходить добуток матриці (**TMatrix**) на вектор (**TVector**). Оскільки класи **TMatrix** та **TVector** приховують свої елементи, оголошені в секції **private**, то доступ до цих елементів можна відкрити для сторонніх функцій, оголосивши ці функції як **friend**.

Приклад 2.6.1:

```
#include <iostream.h>
```

```
// попереднє оголошення класу TMatrix
//(в класі TVector використовується клас TMatrix)
class TMatrix;

//клас вектор
class TVector
{
    float v[4]; //масив, який містить координати вектора
public:
    //конструктор
```

```

TVector(float x1=1, float x2=1,
        float x3=1, float x4=1)
{
    v[0] = x1;
    v[1] = x2;
    v[2] = x3;
    v[3] = x4;
}
// друк вектора
void print();
//добуток матриці на вектор (дружня функція)
friend TVector multiplay(const TMatrix&,
                        const TVector&);
};

void TVector::print()
{ // друк вектора
    cout <<v[0]<<" "<<v[1]<<" "<<v[2]<<" "<<v[3]<<endl;
}

// клас TMatrix
class TMatrix
{
    TVector v[4]; //масив векторів
public:
    TMatrix();
    TMatrix(TVector, TVector, TVector, TVector);
    void print ();
    //добуток матриці на вектор (дружня функція)
    friend TVector multiplay (const TMatrix&,
                            const TVector&);
};

TMatrix::TMatrix()//конструктор за замовчуванням
{
    // ініціалізація масиву векторів
    for (int i = 0; i<4;i++)
        v[i] = TVector(1,1,1,1);
}

// конструктор, який приймає аргументи -
// вектори для утворення матриці
TMatrix::TMatrix(TVector x1, TVector x2,
                TVector x3, TVector x4)
{

```

```

        v[0] = x1;
        v[1] = x2;
        v[2] = x3;
        v[3] = x4;
    }

    void TMatrix::print()
    { //друж матриці
    for (int i = 0; i<4;i++)
        v[i].print();
    }

    //добуток матриці на вектор (дружня функція)
    TVector multiplay(const TMatrix& m, const TVector& v)
    {
        TVector rez;
        for(int i=0;i<4;i++)
        {
            rez.v[i] = 0;
            for(int j=0;j<4;j++)
                rez.v[i] += m.v[i].v[j]*v.v[j];
        }
        return rez;
    }

    int main()
    {
        TVector v(1,2,3,4), rezv;
        TMatrix m;
        cout<<"Matrix m:\n";
        m.print();

        cout<<"Vector v:\n";
        v.print();

        // виклик дружної функції multiplay(), яка має доступ
        // до прихованих членів-даних класів
        // TVector i TMatrix
        rezv = multiplay (m,v);

        cout<<"Vector r:\n";
        rezv.print();
        cin.get();
        return 0;
    }

```

Якщо `fun()` – деяка функція-член класу `TMyClass1` і всі члени-функції класу `TMyClass2` повинні мати доступ до закритих членів даних класу `TMyClass`, то в оголошенні класу це вказується так:

```
class TMyClass2; // попереднє оголошення класу
class TMyClass
{. . .
    friend void TMyClass1::fun();
    friend class TMyClass2;
. . .
};
```

Клас `TMyClass2` тепер стає дружнім для класу `TMyClass`.

2.7. Перевантаження операцій для класів

У `C++` існує можливість перевантаження стандартних операцій:

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

Крім того, синтаксис дозволяє перевантажувати оператори `new`, `delete`.

Операції, які не можна перевантажувати:

- `.` (вибір члена класу);
- `.*` (вибір члена класу через вказівник);
- `::` (оператор дозволу області видимості або оператор прив'язки);
- `?:` (умовний оператор).

Ім'я операторної функції починається з ключового слова **operator** і символу `@`, де `@` – одна з наведених у таблиці операцій. Операторна функція **operator@** використовується для того, щоби визначити нову, перевантажену дію конкретного оператора мови. Аналогічно, як й у випадку з перевантаженими

функціями, компілятор розрізняє різні функції за кількістю і типом входних параметрів. Це дає можливість використовувати об'єкти у виразах замість того, щоб передавати їх як параметри у функції. Кожному оператору мова C++ ставить у відповідність ім'я функції, що складається з ключового слова **operator**, знака операції та аргументів відповідних типів:

```
<тип результату> operator <символ оператора>
                        (<вхідні аргументи>);
```

Якщо у прикладі 2.6.1 дружню функцію переназвати іменем **operator***, то замість виклику функції

```
rezv = multiplay (m,v);
```

можна використовувати вираз з операцією множення двох об'єктів, а саме:

```
rezv = m*v; // скорочений виклик операторної функції
rezv = operator*(m,v); //явний виклик операторної
функції
```

Як бачимо, операторна функція може бути викликана як будь-яка інша функція. Використання операторної функції як оператора є просто скороченою формою її явного виклику.

Як і у випадку структур, розрізняють унарні (один аргумент) або бінарні (два аргументи) оператори. Якщо для структур є сенс визначати операторні функції лише як звичайні сторонні функції, то для класів, як правило, використовуються три способи перевантаження операторних функцій [11, 25]:

- як член-функція класу;
- як дружня функція;
- як звичайна функція.

Розглянемо приклад класу **TComplex** (комплексне число), для якого визначимо три операторні функції трьома різними способами:

- **operator+** – член-функція класу;

- **operator-** – дружня функція;
- **operator*** – звичайна функція.

Приклад 2.7.1:

```
#include <iostream.h>
class TComplex
{
    float re,im;// дійсна та уявна частини відповідно
public:
    TComplex(float r=0., i=0.) //конструктор
    {
        re=r;
        im=i ;
    }
    // функції доступу
    float getRe(){return re;}
    float getIm(){return im;}

    //перевантаження оператора + (член класу)
    TComplex operator+(const TComplex&)const;

    //перевантаження оператора - (друг класу)
    friend TComplex operator-(const TComplex&,
                             const TComplex&);
};

//перевантаження оператора * (звичайна функція)
TComplex operator*(TComplex& one, TComplex& two)
{
    float re1=one.getRe();
    float im1=one.getIm();
    float re2=two.getRe();
    float im2=two.getIm();
    return TComplex(re1*re2-im1*im2, re1*im2+im1*re2);
}

TComplex operator-(const TComplex& one, const TComplex&
two)
{
    return TComplex(one.re-two.re,one.im-two.im);
}

TComplex TComplex::operator+(const TComplex& two)const
{
```

```

        return TComplex(re+two.re,im+two.im);
    }

int main(int argc, char* argv[])
{
    TComplex op1(1.5,-3.32),op2(5.2,7.02);
    TComplex suma, rizm, rez;

    suma=op1+op2;
    rizm=op2-op1;
    rez=op1*op2-suma*rizm;
    cout<<rez.getRe()<<" + i*(" <<rez.getIm()<<' ' <<endl;
    cin.get();
    return 0;
}

```

Перевантаження операцій підпорядковується таким правилам:

- зберігається кількість аргументів: їх кількість строго визначена (крім операторів виклику), 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 за замовчуванням, не успадковується та не може бути оголошена статичною;

- оператори `+`, `-`, `*`, `&` можуть бути як унарними, так і бінарними;

- оператори `[]`, `()`, `=`, `->` можуть бути лише членами класу;

- неможливо визначити нові лексичні символи операцій (наприклад, не можна перевизначити операцію `**` для піднесення до степеня);

- не можна перевантажувати операції для некласових типів (є дві можливості: щоб операторна функція була членом якогось класу, створеного користувачем, або щоби хоча б один з операндів цієї операції був класового типу);

- обмежень на тип, що повертає перевантажений оператор, немає.

2.8. Бінарні й унарні операції

Бінарна операція може бути визначена як функція-член, що одержує один параметр, або як функція-друг чи звичайна функція, що одержує два параметри [13]. Таким чином, для будь-якої бінарної операції `@` вираз `one@two` може інтерпретуватися як `one.operator@(two)` чи як `operator@(one, two)`. Якщо визначені обидві операторні функції, то запис `one@two` помилковий. Унарна операція, префіксна чи постфіксна, може бути визначена як функція-член, що не одержує параметрів, або як функція-друг чи звичайна функція, що одержує один параметр. Таким чином, для будь-якої унарної операції `@` вирази `obj@` або `@obj` можуть інтерпретуватися як `obj.operator@()` або як `operator@(obj)`. Якщо визначено й те, й інше, то вирази `obj@`, і `@obj` помилкові.

Наведемо приклади оголошення операцій для деякого класу `T`. При цьому позначимо некоректні описи.

Приклад 2.8.1:

```
class T
{ . . .
  // дружні оператори

  // унарний мінус
  friend T operator-(T);
  // бінарний мінус
  friend T operator-(T,T);
  // помилка: немає операндів
  friend T operator-();
  // помилка: тернарна операція
  friend T operator-(T,T,T);
  // унарний префіксний декремент
  friend T& operator--(T&);
  //унарний постфіксний декремент
  friend T operator--(int,T&);
  //унарний постфіксний декремент
```

```

        friend T operator--(T&,int);
        . . .
public:
//оператори члени класу
// (з неявним першим параметром: this)

// унарне & (узяття адреси)
    T* operator&();
//бінарне & (логічне побітове множення)
    T operator&(T);
// помилка: тернарна операція
    T operator&(T,T);
//унарний префіксний декремент
    T& operator--();
//унарний постфіксний декремент
    T operator--(int);
    . . .
};

```

Коли операції ++ і -- перевантажуються, буває необхідно визначити обидві їх форми (префіксну й постфіксну). Ім'я операторної функції, наприклад, **operator--**, однакове для префіксної та постфіксної форм. Тому для того, щоб розрізнити дві форми, у постфіксній формі використовується ще один (для дружнього оператора – перший або другий, в залежності від версії компілятора) параметр цілого типу, який не застосовується, а лише визначає потрібний вибір операторної функції.

2.9. Оператор присвоєння

Операторна функція присвоєння зазвичай повертає посилання на об'єкт, для якого вона викликана, і приймає як параметр один аргумент – посилання на об'єкт, який присвоюється. Дозволяється також як тип результату використовувати **void**.

Синтаксис двох форм оператора присвоєння:

```

<класовий тип &> operator=(<[const]класовий тип &>);
<void> operator=(<[const]класовий тип &>);

```

Випадок, у якому тип результату – **void**, простіший у написанні, але працює лише при одноразовому використанні присвоєння і не працює (видає помилку компіляції) при ланцюговому присвоюванні об'єктів (**ob1=ob2=ob3;**).

У тілі операторної функції присвоєння потрібно звільнити пам'ять, задіяну для старого об'єкта, і виділити нову під об'єкт, який присвоюється. При знищенні старого об'єкта потрібно контролювати операцію самоприсвоєння. Після цього контролю необхідно скопіювати вміст об'єкта у виділену область пам'яті.

Приклад 2.9.1:

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

class TString
{
    char* s;
public:
    TString(char* t)
    {
        s=strdup(t);
    }
    TString(const TString&);
    ~TString()
    {
        delete[] s;
    }
    TString& operator=(const TString&);
    void print() const;
};

void TString::print() const
{
    cout<<s<<endl;
}

TString::TString(const TString& a)
{
    s=new char[strlen(a.s)+1];
    strcpy(s,a.s);
}
```

```

TString& TString::operator=(const TString& a)
{
    if (this != &a) // контроль самоприсвоєння
    {
        delete[] s;
        s=new char[strlen(a.s)+1];
        strcpy(s,a.s);
    }
    return *this;
}

void main()
{
    // виклик конструкторів з параметром
    TString s1("abcabc6"),s2("123");

    // виклик конструктора копіювання
    TString s3(s2);
    s3.print();

    // знову виклик конструктора копіювання (не присвоєння)
    TString s4=s2;
    s4.print();

    // виклик оператора присвоєння (одноразове присвоєння)
    s2=s1;

    // виклик оператора присвоєння (ланцюгове присвоєння)
    s2=s4=s1;

    // друк результатів
    s1.print();
    s2.print();
    s3.print();
    s4.print();

    cin.get();
}

```

Зауваження: Якщо серед членів даних класу є вказівники, яким у конструкторі динамічно виділяється пам'ять, то для коректної роботи такого класу потрібно визначити, крім конструкторів, деструктор, оператор присвоєння і конструктор копіювання.

Приклад 2.9.2:

```
class TDemo
{
    . . .
public:
    // створює об'єкт
    TDemo ([<параметри>]);

    // створює копію об'єкта
    TDemo (const TDemo &);

    // знищує об'єкт (звільняє пам'ять)
    ~TDemo ();

    // звільняє пам'ять, виділяє нову під потрібний об'єкт,
    // копіює містиме об'єкта у виділену область пам'яті
    TDemo& operator=(const TDemo&);

    . . .
};
```

2.10. Оператори виклику, індексування й перетворення типу

Оператори виклику, індексування і перетворення типу можна перевизначати лише як члени класу. Друзями класу вони бути не можуть. Крім того, не можна використовувати параметри зі значеннями за замовчуванням.

- Оператор перетворення типу можна оголошувати для перетворення у будь-який тип даних. Він викликається без аргументів, а тип значення, що повертається, визначається за іменем оператора. Явно тип, що повертається, не вказується.

Синтаксис:

```
operator <тип для перетворення>();
```

Оператори перетворення типу зазвичай явно не викликаються, оскільки компілятор часто сам може визначити, коли вони повинні викликатися.

- Оператор індексування `operator[]` може бути перевантажений тільки з одним аргументом довільного типу й повертати може також довільний тип як своє значення. Якщо передбачається використовувати `operator[]` з лівого боку від знака присвоєння, то треба повертати тип посилання.

Синтаксис:

`<тип результату> operator[](<тип вхідного параметра>);`

- Оператор виклику `operator()`. Якщо перевантажується оператор виклику, то дужки зліва від імені об'єкта інтерпретуються як функція, що викликається. Аргументи оператора є формальними аргументами, які передаються об'єкту-функції при виклику. Оператор виклику може повертати будь-який тип і приймати аргументи будь-яких типів довільної кількості. При цьому на нього поширюються ті ж правила, що й на будь-яку іншу функцію.

Синтаксис:

`<тип результату> operator()(<вхідні параметри>);`

Приклад 2.10.1:

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

class TString
{
    char* s;
public:
    TString(char* t){s=strdup(t);}
    ~TString(){delete[] s;}

    // оператори виклику
    int operator() (int pos, char c);

    //int operator() (int pos=0, char c); так писати не можна
```

```

        int operator() (char c)  {return operator() (0,c); }

// оператори індексування
    char& operator[] (int i){return s[i];}
    int operator[] (char c)  {return operator() (0,c); }

// оператори перетворення типу
    operator int() {return atoi(s);}
    operator char*(){return s;}
};

int TString::operator() (int pos, char c)
{
    for(int i=pos+1;i<strlen(s);i++)
        if(s[i]==c)return i;
    return -1;
}

void main()
{
    TString s1("abcabc6"),s2("123");

    // operator char* (char*)s-явне перетворення
    // не конфліктує з (int)s
    cout<<" s1="<<(char*)s1;

    // виклик оператора operator() (int pos, char c)
    cout<<"\n s1[3]="<<s1(2,'a')

    //виклик оператора operator() (char c)
    <<"\n s1[1]="<<s1('b')

    //виклик оператора operator[] (char c)
    <<"\n s1[1]="<<s1['b'];

    // виклик оператора operator[] (int i) два рази
    s2[2]=s1[6];

    // неявний виклик оператора operator int
    int i=s2;
    cout<<"\n i="<<i;

    cin.get();
}

```

Результат роботи програми:

```
S1=abcabc6
S1[3]=3
S1[1]=1
S1[1]=1
i=126
```

2.11. Перевантаження операторів new і delete

Для виконання нестандартних операцій керування пам'яттю використовується перевантаження операторів **new** і **delete**.

Оператор **new** може перевантажуватися як у формі зовнішньої функції, так і у формі функції-члену класу. Оператор **new** за замовчуванням має такий інтерфейс:

```
void* operator new(size_t <кількість байтів>);
```

Аргумент задає кількість байтів, що виділяється. Повертається адреса виділеної області.

Після обов'язкового параметра можуть розміщуватись необов'язкові додаткові параметри. При виклику оператора **new** додаткові аргументи вказуються між ключовим словом **new** і перед іменем класу; вони передаються перевантаженому оператору після розміру блока, що надається компілятором.

Наприклад,

```
MyClass* p = new (<список додаткових аргументів>)
                MyClass;
```

Для такого виклику оголошення члена-функції оператора **new** має вигляд

```
void* operator new(size_t <кількість байтів>,
                  <список додаткових аргументів>);
```

Тут обов'язковий перший параметр надається компілятором.

Для виконання нестандартних операцій керування пам'яттю разом з оператором **new** зазвичай перевантажується оператор **delete**. Існують два інтерфейси, які неявно викликаються компілятором при видаленні об'єкта:

```
void operator delete(void* <адреса>);
```

```
void operator delete(void* <адреса>,
                    size_t <кількість байтів>);
```

Приклад 2.11.1:

Умова задачі: Ввести дані в асоціативний масив пар (пара – це слово і кількість його входжень). Для створення нової пари перевантажити оператор **new**. Написати функцію введення у масив однієї пари і функцію виведення всіх пар масиву.

```
#include <iostream.h>
```

```
struct TPair {
    char* name;
    int val;
    void* operator new(size_t bytes, char* slovo)
    {
        TPair* p= ::new TPair;
        p->name=strdup(slovo);
        p->val=1;return p;
    };
};
```

```
class TAssoc {
    TPair vec[100]; // вектор пар
    int max;        // довжини max
    // індекс першого невикористаного елемента
    int ifree;
```

```

public:
    TAssoc() {
        ifree=0;
        max=100;
    };
    ~TAssoc();
    void input(char*);
    void print_all();
};

TAssoc::~TAssoc()
{
    for (int i = 0; i<ifree;i++)
        delete[] vec[i].name;
    ifree=0;
}

void TAssoc::input(char* slovo)
    /* робота з множиною пар TPair:пошук слова slovo серед
    пар і збільшення цілої частини TPair (кількості слів);
    створення нової пари TPair, якщо slovo не зустрічалося у
    масиві.
    */
    {
        if (ifree==max)
        {
            cerr<< " переповнення\n" ;
            exit(1);
        }
        if (ifree)
            for (TPair* p =&vec[ifree-1]; vec<=p; p--)
                if (strcmp(slovo,p->name)==0)
                {
                    p->val++;
                    return;
                }
            TPair* temp=new (slovo)TPair;
            vec[ifree++] = *temp;
            delete temp;
        }

//функція друку всіх пар
void TAssoc::print_all()
{
    for (int i = 0; i<ifree;i++)

```

```

    cout << vec[i].name << ": " << vec[i].val << "\n";
}

int main()
{
    {
        TAssoc vec;
        char buf[80];

//кінець вводу Ctrl-z
        while (cin>>buf) vec.input(buf);
        vec.print_all();
    } // викликається деструктор

    cin.get();
    return 0;
}

```

2.12. Приклад розробки класу “Множина цілих чисел”

Умова задачі: Створити клас **TSetInt** “Множина цілих чисел” цілих чисел. Створити конструктори, деструктор, генератор копій.

Визначити операції:

- '*' – перетин двох множин;
- '+' – об'єднання двох множин;
- '-' – різниця двох множин;
- '=' – присвоєння;
- '<=', '>=', '!=', '==' – операції відношення.

Визначити функції:

- друку множини;
- визначення кількості елементів у множині;
- порівняння з порожньою множиною;
- впорядкування множини;
- визначення, чи належить число множині;
- визначення максимального елемента в множині.

Створити потрібну кількість об'єктів множин для ілюстрації всіх операцій та функцій. Частина функцій і операторів зробити дружніми, частину – членами класу.

Приклад 2.12.1:

```
#include <windows.h>
#pragma hdrstop
#pragma argsused
#include <tchar.h>
#include <stdio.h>

// бібліотека, що містить класи введення-виведення
// cin, cout, cerr та ін.
// дає можливість користуватися перевантаженими
// операторами введення/виведення
#include <iostream.h>
#include <conio.h>

typedef int Bool;
const Bool NO=0;
const Bool YES=1;

class TSetInt
{
// закриті члени-дані
// доступ дозволено тільки членам та друзям класу
    int* V;           // елементи множини
    int Sz;           // кількість елементів V[]
    int Num;          // номер об'єкта
    Bool Copy;        // копія об'єкта, чи ні

// кількість об'єктів-множин, одна статична змінна
// для всіх об'єктів
    static int KilOb;

// відкриті члени-дані
public:
    TSetInt(int);      // конструктор з аргументом
    TSetInt(int,int*); // конструктор з аргументами
    TSetInt();         // конструктор за замовчуванням
    TSetInt(const TSetInt&); // конструктор копіювання
    ~TSetInt();        // деструктор
```

```

// бінарні оператори-члени класу
// об'єкт *this – перший параметр (передається неявно)
// перший і другий параметр не змінюються (const)
    TSetInt operator*(const TSetInt&) const;
    TSetInt operator-(const TSetInt&) const;
    Bool operator<=(const TSetInt&) const;
    Bool operator==(const TSetInt&) const;
    Bool operator>=(const TSetInt&) const;
    Bool operator!=(const TSetInt&) const;

// оператор присвоєння
// (тільки другий параметр не змінюється)
    TSetInt& operator=(const TSetInt&);

// дружній оператор
friend TSetInt operator+(const TSetInt&,
                        const TSetInt&);

// функції-члени класу
    void printSet()const;
    int kilElem(){return Sz;}

// функція, що підставляється
    Bool isEmpty();
    void order();
TSetInt& inSet(int); // можна так: void inSet(int);
    Bool ifInSet(int) const;

// функція-друг класу
friend int maxElem(const TSetInt&);
};

// ініціалізація статичної змінної
int TSetInt::KilOb=0;

TSetInt::TSetInt(int sz)
{
    V=new int[sz];
    Copy=NO;
    Sz=sz;
    Num=++KilOb;
    int i=0,e1;

// запис випадкових елементів у масив-множину тільки
// по одному разу

```

```

        while(i<Sz)
        {
            el=random(50);
            if (!ifInSet(el))V[i++]=el;
        }
cout<<"\nСтворено об'єкт (int) №"<< Num <<endl;
    }

TSetInt::~TSetInt()
{
    cout<<"\nЗнищено об'єкт №"<< Num <<
    " Copy "<<Copy<<
    " Кількість об'єктів "<<--KilOb <<endl;
    delete[] V;
}

TSetInt::TSetInt()
{
    Sz=0;
    V=NULL;
    Copy=NO;
    Num=++KilOb;
    cout <<"\nСтворено об'єкт() №"<< Num <<endl;
}

//Конструктор записує елементи масиву
//тільки по одному разу
TSetInt::TSetInt(int sz,int* vek)
{
    int* temp=new int[sz];
    Copy=NO;
    Sz=1;
    Num=++KilOb;
    temp[0]=vek[0];
    for(int i=1;i<sz;i++)
    {
        Bool flag=YES;
        for(int j=0;j<Sz;j++)
            if (vek[i]==temp[j]) flag=NO;
        if(flag)temp[Sz++]=vek[i];
    }
    V=new int[Sz];
    for(int i=0;i<Sz;V[i]=temp[i],i++);
    delete[] temp;
    cout<<"\nСтворено об'єкт (int,int*) №"

```

```

        <<Num<<endl;
    }

TSetInt::TSetInt(const TSetInt& Obj) :

    //список ініціалізаторів
    Num(Obj.Num) ,
    Copy(YES) ,
    Sz(Obj.Sz)
{
    V=new int[Obj.Sz];
    for(int i=0;i<Obj.Sz;i++)V[i]=Obj.V[i];
    KilOb++;
    cout<<"\nСтворено копію об'єкта №"<<Num<<endl;
}

TSetInt TSetInt::operator*(const TSetInt& Ob) const
{
    TSetInt temp(*this); //виклик генератора копій
    int k=0;
    for(int i=0;i<Ob.Sz;i++)
        for(int j=0;j<Sz;j++)
            if(Ob.V[i]==V[j])
                temp.V[k++]=Ob.V[i];
    temp.Sz=k;
    return temp;
}

TSetInt TSetInt::operator-(const TSetInt& Ob) const
{
    TSetInt temp(*this);
    int k=0;
    for(int j=0;j<Sz;j++)
        if(!Ob.ifInSet(V[j]))
            temp.V[k++]=V[j];
    temp.Sz=k;
    return temp;
}

TSetInt& TSetInt::operator=(const TSetInt& Ob)
{
    if (this==&Ob)
    {
        cerr<<"\nПомилка присвоєння"<<endl;
        return *this;
    }
}

```

```

        }
        delete[] V;
        V=new int[Sz=Ob.Sz];
        for(int i=0;i<Sz;i++)V[i]=Ob.V[i];
        return *this;
    }

    // немає області видимості,
    // оскільки цей оператор – не член класу, а друг
    TSetInt operator+(const TSetInt& Ob1,const TSetInt& Ob2)
    {
        int* v=new int[Ob1.Sz+Ob2.Sz];
        for(int i=0;i<Ob1.Sz;i++)v[i]=Ob1.V[i];
        int sz=Ob1.Sz-1;
        for(int i=0;i<Ob2.Sz;i++)
    // необов'язково перевіряти,
    // бо перевірка є в конструкторі
    //if (!(Ob1.ifInSet(Ob2.V[i])))
        v[sz++]=Ob2.V[i];
        TSetInt rez(sz,v);
        delete[] v;
        return rez;
    }

    Bool TSetInt::operator<=(const TSetInt& Ob) const
    {
        if(Ob.Sz < Sz ) return NO;
        if(Sz==0)return YES;
        for(int i=0;i<Sz;i++)
            if (!(Ob.ifInSet(V[i])))return NO;
        return YES;
    }

    Bool TSetInt::operator>=(const TSetInt& Ob) const
    {
        return Ob<=(*this);
    }

    Bool TSetInt::operator==(const TSetInt& Ob) const
    {
        if(Ob.Sz != Sz) return NO;
        return(operator<=(Ob)); // краще так: *this<=Ob;
    }

    Bool TSetInt::operator!=(const TSetInt& Ob) const

```

```

    {
        return (! (operator==(Ob))) ;
    }

TSetInt& TSetInt::inSet(int a)
{
    int i;
    if(ifInSet(a))return *this;
    TSetInt Ob(*this);
    delete[] V;
    V=new int[++Sz];
    for(i=0;i<Ob.Sz;i++)
        V[i]=Ob.V[i];
    V[i]=a;
    return *this;
}

void TSetInt::printSet() const
{
    cout<<"\nОб'єкт №"<<Num<<
    " Copy " <<Copy<<endl<< "Елементи: ";
    for(int i=0;i<Sz;i++)
        cout<<V[i]<<" ";
    cout<<"\nКількість об'єктів " <<KilOb<<endl;
    return;
}

Bool TSetInt::isEmpty()
{
    if(Sz==0)return YES;
    return NO;
}

void TSetInt::order()
{
    for(int i=1;i<Sz;i++)
    {
        for(int j=Sz-1;j>=i;j--)
            if(V[j-1]>V[j])
            {
                int a=V[j-1];
                V[j-1]=V[j];
                V[j]=a;
            }
    }
}

```

```

        return ;
    }

    Bool TSetInt::ifInSet(int a) const
    {
        for(int i=0;i<Sz;i++)
            if(a==V[i])return YES;
        return NO;
    }

    int maxElem(const TSetInt& Ob) // дружня функція
    {
        int a,i;
        for(a=Ob.V[0],i=1; i<Ob.Sz; i++)
            if(Ob.V[i]>a)a= Ob.V[i];
        return a;
    }

    int _tmain()
    {
        // режим кирилиці для консолі
        SetConsoleOutputCP(1251);
        SetConsoleCP(1251);

        TSetInt Ob1(4),Ob2(5);
        Ob1.printSet();
        cout<<"\nКількість елементів "<<Ob1.kilElem()
                <<" в об'єкті N1"<<endl;
        cout<<"\nifInSet 4 - "<<Ob1.ifInSet(4)<<" ifInSet 1-"
                <<Ob1.ifInSet(1)<<endl;
        cout<<"\nisEmpty "<<Ob1.isEmpty()<<endl;

        Ob2.printSet();

        TSetInt Ob3(Ob2);
        Ob3.printSet();
        cout<<"\nmax Ob3 "<<maxElem(Ob3)<<endl;

        TSetInt Ob4(0);
        Ob4.printSet();
        cout<<"\nisEmpty Ob4 "<<Ob4.isEmpty()<<endl;

        int vek[5];
        for(int i=0;i<5;vek[i]=6+i,i++);
        vek[1]=8;
    }

```

```

TSetInt Ob5(5,vek) ;
Ob5.printSet();
TSetInt Ob6;
{
// Блок існування тимчасових об'єктів,
// створених конструктором копіювання
Ob6=Ob5+Ob1;
}

Ob6.inSet(3);
Ob6.order();
Ob6.printSet();

{
// Блок існування тимчасових об'єктів,
// створених конструктором копіювання
Ob2=Ob6-Ob1;
}
Ob2.printSet();

Ob2.inSet(27);

{
// Блок існування тимчасових об'єктів,
// створених конструктором копіювання
Ob6=Ob1*Ob2;
}
Ob6.printSet();
Ob2.printSet();

cout<<"  Ob6<=Ob2 ? "<< (Ob6<=Ob2) <<endl;
cout<<"  Ob2<=Ob6 ? "<< (Ob2<=Ob6) <<endl;
cout<<"  Ob2==Ob1 ? "<< (Ob2==Ob1) <<endl;
cout<<"  Ob2>=Ob6 ? "<< (Ob2>=Ob6) <<endl;

Ob6.printSet();
Ob6.inSet(9);

TSetInt Ob7;
Ob7=Ob2;
Ob7.printSet();
cin.get();
return 0;
}

```

Результат роботи програми:

Створено об'єкт (int) №1

Створено об'єкт (int) №2

Об'єкт №1 Сору 0

Елементи: 27 44 17 46

Кількість об'єктів 2

Кількість елементів 4 в об'єкті N1

ifInSet 4 - 0 ifInSet 1 - 0

isEmpty 0

Об'єкт №2 Сору 0

Елементи: 20 35 31 40 29

Кількість об'єктів 2

Створено копію об'єкта №2

Об'єкт №2 Сору 1

Елементи: 20 35 31 40 29

Кількість об'єктів 3

max Ob3 40

Створено об'єкт (int) №4

Об'єкт №4 Сору 0

Елементи:

Кількість об'єктів 4

isEmpty Ob4 1

Створено об'єкт (int,int*) №5

Об'єкт №5 Сору 0

Елементи: 6 8 9 10

Кількість об'єктів 5

Створено об'єкт() №6

Створено об'єкт (int,int*) №7

Знищено об'єкт №7 Сору 0 Кількість об'єктів 6

Створено копію об'єкту №6

Знищено об'єкт №6 Сорту 1 Кількість об'єктів 6

Об'єкт №6 Сорту 0

Елементи: 3 6 8 9 17 27 44 46

Кількість об'єктів 6

Створено копію об'єкта №6

Створено копію об'єкта №6

Знищено об'єкт №6 Сорту 1 Кількість об'єктів 7

Знищено об'єкт №6 Сорту 1 Кількість об'єктів 6

Об'єкт №2 Сорту 0

Елементи: 3 6 8 9

Кількість об'єктів 6

Створено копію об'єкта №2

Знищено об'єкт №2 Сорту 1 Кількість об'єктів 6

Створено копію об'єкта №1

Створено копію об'єкта №1

Знищено об'єкт №1 Сорту 1 Кількість об'єктів 7

Знищено об'єкт №1 Сорту 1 Кількість об'єктів 6

Об'єкт №6 Сорту 0

Елементи: 27

Кількість об'єктів 6

Об'єкт №2 Сорту 0

Елементи: 3 6 8 9 27

Кількість об'єктів 6

Ob6<=Ob2 ? 1

Ob2<=Ob6 ? 0

Ob2==Ob1 ? 0

Ob2>=Ob6 ? 1

Об'єкт №6 Сорту 0

Елементи: 27

Кількість об'єктів 6

Створено копію об'єкта №6

Знищено об'єкт №6 Сорту 1 Кількість об'єктів 6

Створено об'єкт () №7

Об'єкт №7 Сору 0

Елементи: 3 6 8 9 27

Кількість об'єктів 7

Запитання для самоконтролю

1. Які назви секцій використовуються у описі класу?
2. Чим відрізняються привілеї доступу до імен в класі й структурі?
3. Що таке інкапсуляція?
4. Які є способи звертання до відкритих членів класу?
5. Чи можна звертатися до закритих членів класу?
6. Навіщо потрібні конструктори і скільки їх може бути у класі?
7. Чим відрізняється функція ініціалізації від конструктора?
8. У яких випадках конструктор викликається неявно?
9. Коли можна не писати конструктор і деструктор?
10. Як можна звертатися до статичних членів класу?
11. Як ініціалізується статичний член класу?
12. Чи треба явно викликати деструктор при знищенні динамічно створеного об'єкта?
13. Навіщо потрібні дружні функції?
14. Коли є сенс призначати дружній клас?
15. Якими способами можна перевантажити бінарні оператори для класів? А унарні?
16. Як задається інтерфейс постфіксних і префіксних унарних операцій?
17. У яких випадках необхідно перевантажувати оператор присвоєння?
18. Від чого залежить необхідність повертання типу посилання в операторі індексування?
19. Чи можна перевантажувати кілька операторів індексування для одного класу?
20. У якій формі найчастіше використовується оператор перетворення типу?

Завдання для самостійної роботи

1. Напишіть тіло конструктора `TDate(const char*)` у класі з п. 2.2 і застосуйте його у функції `main()`, створюючи об'єкти як автоматичні та динамічні змінні.
2. Доповніть клас `TDate` з п. 2.2 оператором перетворення типу у `char*` і застосуйте його для неявного виклику при друці дати.
3. Доповніть клас `TDemo` з п. 2.5 оператором виклику, який буде повертати значення одного з членів-даних.
4. Перепишіть функції `addYear()`, `addMonth()`, `addDay()` з прикладу 2.1.1 і функцію `print_all()` з прикладу 2.11.1, оголосивши їх дружніми.
5. Перепишіть операції відношення класу `TSetInt` з прикладу 2.12.1, оголосивши їх дружніми, а операцію об'єднання двох множин переробіть на оператор-член класу.

3. Створення нових класів із використанням існуючих

3.1. Побудова класів з членами-об'єктами. Агрегація

Використання класу або його різновиду – структури всередині іншого класу називається агрегацією.

Агрегація забезпечує відношення ціле – частина. Частиною може бути об'єкт локального класу (такий вид агрегації називається композицією) або на частину може вказувати вказівник.

Розглянемо приклад агрегації для реалізації стека [17], який буде зберігати будь-які типи даних.

Приклад 3.1.1:

```
// Файл з інтерфейсною частиною класу Stack.h
#ifndef STACKN //якщо не визначена директива STACKN
#define STACKN // визначаємо її
// оголошення класу TStack
class TStack
```

```

{
    struct TLink
    { // структура, яка оголошена у класі
        void* data;
        TLink* next;
        TLink(void* dat, TLink* nxt);
        ~TLink();
    } * head; // агрегація з використанням вказівника
public:
    TStack();
    ~TStack();
    void push(void* dat);
    void* peek();
    void* pop();
};
#endif // кінець визначення директиви STACKN

// Файл реалізації Stack.cpp
#include "Stack.h"
#include <iostream.h>
TStack::TLink::TLink(void* d, TLink* n)
{
    data = d;
    next = n;
}

TStack::TLink::~TLink() { }
TStack::TStack() { head = NULL; }

void TStack::push(void* d)
{
    head = new TLink(d, head);
}

void* TStack::peek()
{
    return head->data;
}

void* TStack::pop()
{
    if(head == NULL) return NULL;
    void* rez = head->data;
    TLink* oldHead = head;
    head = head->next;
}

```

```

        delete oldHead;
        return rez;
    }

TStack::~TStack()
{
    while(head)pop();
    cout<<"\nTStack empty";
}

```

Програма складається з трьох файлів: файла заголовка **Stack.h**, в якому знаходиться опис класу **TStack**, файла **Stack.cpp**, в якому знаходиться реалізація цього класу, і файла **Main.cpp** з основною функцією **main**.

У класі **TStack** у приватній секції оголошена структура **TLink**, яка зберігає вказівник на об'єкт (змінна **data**) та вказівник на наступний елемент (змінна **next**); вказівник **head**, який вказує на структуру **TLink**; конструктор **TStack()** та деструктор **~TStack()**.

Доступні функції-члени класу **TStack**:

- **push** для додавання об'єкта до стека;
- **peek** для повернення об'єкта;
- **pop** для вилучення об'єкта зі стека.

Конструктор **TLink::TLink()** присвоює вказівник на об'єкт змінній **data** та вказівник на наступний елемент **next**. Рядок **head = new TLink(dat, head)** у функції **TStack::push** створює новий елемент **TLink** і додає його до стека.

Зуваження: деструктор **~TLink()** не звільняє місце в пам'яті, тому що:

- оператор **delete** не може звільнити правильно вказівник на **void**;
- якщо звільняти місце в деструкторі, то функція-член **pop** повертала б вказівник на об'єкт, який уже знищено, і це було б помилкою.

Протестуємо клас **TStack** для елементів цілого типу.

Приклад 3.1.2:

```
// Файл: Main.cpp
#include <iostream.h>
#include "Stack.h"

int main(int argc, char* argv[])
{
{
TStack my;
int e11,e12,e13;
cout<<"Input elements\n";
cin>>e11>>e12>>e13;

my.push(&e11);
my.push(&e12);
my.push(&e13);

// вилучення e13
my.pop();

// друк e12
// (перетворення типу й розіменування вказівника)
cout<<*(int*)my.peek();
} // тут спрацює деструктор класу TStack

cin.get(); // затримка вікна з результатом
return 0;
}
```

Класи бувають глобальними й локальними. Якщо один клас вкладений в інший клас (агрегація), то обидва класи не мають жодних особливих прав доступу до елементів один одного.

У локальному класі забороняється використовувати автоматичні змінні з області, в якій він описаний. Локальний клас не може мати статичних елементів і методи цього класу можуть бути описані тільки всередині класу.

3.2. Шаблони класів

Шаблони класів часто називають генераторами класів, або узагальненими чи родовими класами, або параметризованими типами. Вони дозволяють визначити структуру цілої сім'ї

класів, по якій компілятор утворює конкретні класи в залежності від вказаних параметрів.

Шаблони класів використовуються тоді, коли виникає необхідність створити кілька дуже подібних, але все ж таки різних класів. Існує бібліотека шаблонів класів **STL – Standart Template Library** [8, 10, 16, 25], в якій розроблено багато корисних контейнерів.

При описі шаблону класу (ключове слово **template**), як і при описі шаблону функції (див. п. 1.6), використовується запис **<class T>** або **<typename T>**, що означає оголошення типу **T**, який невідомий, але при генерації конкретного класу з явним аргументом компілятор автоматично підставить відповідний тип.

Окрім узагальненого типу, як параметр може використовуватися вбудований тип або знову шаблон. Параметрів шаблону може бути кілька, і вони відділяються комами. Для будь-яких параметрів шаблону можуть бути задані значення за замовчуванням.

У загальному випадку шаблон класу має вигляд:

```
template <опис параметрів шаблону> <опис класу>
```

Методи шаблону класу автоматично стають шаблонами для функцій. Якщо метод з шаблону описується поза шаблоном, його заголовок повинен мати такий вигляд:

```
template <опис параметрів шаблону>  
<тип, що повертається функцією> <ім'я класу>  
<параметри налагодження шаблону>::<ім'я функції>  
    (<список параметрів функції>)
```

Перепишемо приклад 3.1.1 так, щоб можна було працювати з будь-якими типами даних без явного їх перетворення. Для цього створимо два шаблони: для структури **TLink** і для класу **TStack**. За допомогою цих шаблонів згенеруємо об'єкти-стеки з цілими і символьними елементами. Нагадаємо, що в п. 1.6 уже розглядалася конструкція **template** для створення шаблонів функцій.

Приклад 3.2.1:

```
#include <iostream.h>
template <class T>
struct TLink
{
    T* data;
    TLink* next;
    TLink(T* d, TLink* n){data = d; next = n;}
    ~TLink(){};
};

template <class T >
class TStack
{
    // агрегація з використанням вказівника
    TLink<T>* head;
public:
    TStack(){ head = NULL; }
    ~TStack();
    void push(T* d){head = new TLink<T>(d,head);}
    T* peek(){return head->data;}
    T* pop();
};

// опис шаблонної функції за межами класу
template <class T>
T* TStack<T>::pop()
{
    if(head == NULL) return NULL;
    T* rez = head->data;
    TLink<T>* oldHead = head;
    head = head->next;
    delete oldHead;
    return rez;
}

//опис шаблонної функції за межами класу
template <class T>
TStack<T>::~~TStack()
{
    while(head)pop();
    cout<<"\nTStack empty";
}
```

```

int main(int argc, char* argv[])
{
TStack <int> my1; //конкретизація шаблону
TStack <char> my2; //конкретизація шаблону
int e11,e12,e13;
cout<<"Input elements\n";
cin>>e11>>e12>>e13;

my1.push(&e11);
my1.push(&e12);
my1.push(&e13);
// вилучення e13
my1.pop();
char c1='a',c2='b';
my2.push(&c1);
my2.push(&c2);
// друк e12 і c2 (розіменування вказівників)
cout<<*my1.peek()<<endl<< *my2.peek();

cin>>c1; // затримка вікна з результатом
return 0;
}

```

Зауваження [25]:

- шаблони класів можуть містити статичні елементи, дружні функції і класи;
- всередині шаблону не можна визначати дружні шаблони;
- локальні класи не можуть містити шаблони як свої елементи.

3.3. Успадкування

Окрім агрегації, існує ще один спосіб використання існуючих класів при побудові нових – це успадкування.

Успадкування дозволяє, практично без обмежень, послідовно будувати й розширювати класи. Починаючи з найпростіших класів, можна створювати похідні класи ускладнюючи, розширюючи чи обмежуючи їх функціональність.

Використання успадкування, особливо при розробці великих програмних проектів, добре узгоджується з технікою

структурного програмування (від загального – до часткового) і багато в чому стимулює такий підхід. При цьому складність коду програми в цілому істотно скорочується. Похідний клас (нащадок) успадковує всі члени-функції і члени-дані свого базового класу (батька) і всіх його попередників в ієрархії класів.

При успадкуванні базовий клас набуває нові атрибути та функції. У похідному класі зазвичай з'являються нові члени-дані. При роботі з об'єктами програміст, як правило, підбирає найбільш придатний існуючий клас для розв'язання конкретної задачі і створює один або кілька нащадків від нього, розширюючи можливості батьківських класів. Дружні функції дозволяють похідному класу одержати доступ до всіх членів даних зовнішніх класів.

Крім того, похідний клас може перевантажувати успадковані функції-члени в тому випадку, коли їхня реалізація в базовому класі не влаштовує нащадка.

C++ дає можливість оголосити похідний клас, що успадковує дані та функції всіх своїх попередників в ієрархії класів, а також може оголошувати нові характеристики й перевантажувати деякі з успадкованих функцій. Наслідуючи характеристики базового класу, можна змусити породжений клас розширити, звзвити, змінити чи знищити їх або залишити без змін.

Успадкування дозволяє повторно використовувати код базового класу в екземплярах похідного класу.

Порядок перерахування секцій рекомендується задавати так, щоб він відповідав зростанню привілеїв доступу областей видимості розташованих у них елементів: від найбільш обмежених – до найдоступніших.

Синтаксис оголошення похідного класу (одинарне успадкування)

```
class Name : [специфікатор доступу] Parent
{
<Оголошення дружніх класів>
<Оголошення дружніх функцій>

[private:]
    <приватні члени-дані>
```

```

        <приватні конструктори>
        <приватні члени-функції >
protected:
        <захищені члени-дані>
        <захищені конструктори>
        <захищені члени-функції>
public:
        <загальнодоступні члени-дані>
        <загальнодоступні конструктори>
        <загальнодоступний деструктор>
        <загальнодоступні члени-функції>
};

```

Parent – базовий клас, **Name** – клас-нащадок базового класу (похідний клас).

При створенні екземплярів похідних класів спочатку викликаються конструктори базових класів, а потім – конструктори похідних класів (у тій послідовності, в якій класи розташовані в ієрархії успадкування).

При виділенні пам'яті для екземпляра **Name** до цієї області додається область пам'яті під екземпляр **Parent**.

Якщо клас породжується від базового й необов'язковий специфікатор доступу опущений, то всі його імена в похідному класі автоматично стають приватними за замовчуванням. Але це можна змінити за допомогою специфікаторів доступу базового класу.

Специфікатор **private** означає, що успадковані (тобто захищені й доступні) імена базового класу стають недоступними в екземплярах похідного класу.

Специфікатор **public** підтверджує, що доступні імена базового класу та його попередників будуть доступними і в екземплярах похідного класу, а всі захищені імена залишаться захищеними.

І, нарешті, специфікатор **protected** усі успадковані (захищені й доступні) імена базового класу робить захищеними в екземплярах похідного класу.

Можна породжувати класи, які розширюють можливості базового класу. Для цього треба перевизначити в похідному класі ту функцію базового класу, яку необхідно змінити. У подібний

спосіб можна породжувати класи, що обмежують можливості базового класу.

Отже, успадкування дає змогу виключити з програми повторювані фрагменти коду і спростити модифікацію програми.

Крім того, успадкування – єдина можливість використовувати для подальшої розробки класи, вихідний код яких недоступний, а доступний лише протокол доступу до захищених і загальнодоступних секцій.

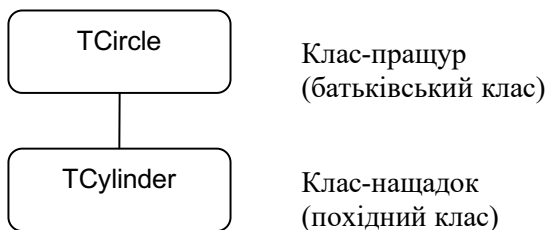
3.4. Ієрархія успадкування класів. Одинарне успадкування

При побудові нового класу є можливість успадковувати один або кілька базових класів. Успадкування буває одинарне і множинне; безпосереднє (напрямку від вказаних класів) і через інші класи. Кожен з базових класів, у свою чергу, може бути похідним від інших базових для нього класів. Таким чином, може бути побудована складна ієрархічна структура, яка відображатиме шляхи успадкування класів. Така структура має назву *ієрархія успадкування класів*.

Розглянемо спочатку приклад розробки базових і похідних класів у найпростішому випадку, коли оголошена ієрархія класів лише двох простих геометричних об'єктів: круга і циліндра (одинарне успадкування).

У класах внутрішні значення змінних r (радіус круга) і h (висота циліндра) задають параметри створюваних об'єктів. Базовий клас **TCircle** (циліндр з висотою 0) визначає круг, а похідний клас **TCylinder** – циліндр.

Розглянемо ієрархію класів.



У програмі ілюструється безпосереднє одинарне відкрите успадкування (клас **TCylinder** успадковується від класу **TCircle**).

Приклад 3.4.1:

```
#include <math.h>
#include <iostream.h>
#include <windows.h>

// клас-пращур
class TCircle
{ protected:
    double r ;
public:
    TCircle (double R = 0) {r = R;}
    void setRadius(double R) {r = R;}
    double getRadius() {return r;}
    double area() { return M_PI*r*r; }
    void show();
};

// клас-нащадок
class TCylinder : public TCircle
{
private:
    double h;
public:
// конструктор
    TCylinder(double H = 0, double R = 0):
        TCircle(R)
        {h = H;}
//встановити висоту циліндра
    void setHeight(double H) { h = H;}
//отримати висоту циліндра
    double getHeight() { return h;}
//отримати площу поверхні циліндра
    double area(){return 2*TCircle::area()+2*M_PI*r*h;}
//отримати об'єм поверхні циліндра
    double volume() { return TCircle::area()*h; }
//вивести на екран інформацію про циліндр
    void show();
};
```

```

void TCircle::show()
{
    cout << "\nРадіус круга = " << getRadius() << endl
    << "Площа круга = " << area() << endl << endl;
}

void TCylinder::show()
{
    cout << "\nрадіус основи = " << getRadius()
    << "\t\tВисота циліндра = " << getHeight() << endl
    << "Площа поверхні = " << area()
    << "\tОб'єм циліндра = " << volume() << endl;
}

void main()
{
    // режим кирилиці для консолі
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);

    // створення об'єктів відповідними конструкторами
    TCircle circle(3);
    TCylinder cylinder(5, 1);

    // виведення інформації про круг
    circle.show();

    // виведення інформації про циліндр
    cylinder.show();

    // виведення інформації про основу циліндра
    cylinder.TCircle::show();

    // затримка часу
    cin.get();
}

```

Оголошення класу **TCircle** містить єдиний член даних **r**, конструктор і ряд функцій-членів класу. При створенні об'єкта конструктор ініціалізує член даних **r** початковим значенням радіуса круга.

Функції доступу **setRadius()** установлює, а **getRadius()** повертає значення члена даних **r**. Метод (член-функція) **area()** повертає площу круга. Метод **show()** виводить на екран значення

радіуса та площі круга.

Клас **TCylinder**, оголошений як похідний від **TCircle**, містить єдиний член даних **h**, конструктор і ряд функцій-членів класу. Конструктор **TCylinder** явно викликає базовий конструктор для ініціалізації радіуса. Цей клас успадковує член даних **r** – радіус основи циліндра і функції-члени **setRadius()** і **getRadius()**. При створенні об'єкта конструктор ініціалізує члени даних **r** та **h** початковими значеннями. У конструкторі член даних **h** ініціалізується значенням аргументу **H**, а член даних **r** отримує значення за допомогою виклику конструктора базового класу **TCircle** з аргументом **R**.

Функція **setHeight()** установлює, а **getHeight()** повертає значення члена даних **h** (це функції доступу).

Метод **TCircle::area()** перевантажує успадковану функцію базового класу, щоб повертати площу поверхні циліндра. Функція **volume()** обчислює об'єм циліндра, а функція **show()** друкує значення радіуса основи, висоти і площі поверхні циліндра.

Функція **main()** створює круг **circle** класу **TCircle** з радіусом 2 і циліндр **cylinder** класу **TCylinder** з висотою 5 і радіусом основи 1, а потім звертається до **show()** для друку параметрів створених об'єктів. Як видно з програми, об'єкт-нащадок **cylinder** може звертатися як до своїх методів, так і до методів базового класу, навіть якщо ці методи закриті власними однойменними методами. Такі методи можна викликати за допомогою оператора прив'язки до імені класу (**TCircle::**).

Результат роботи програми:

Радіус круга = 3

Площа круга = 28.2743

Радіус основи = 1

Площа поверхні = 37.6991

Висота циліндра = 5

Об'єм циліндра = 15.708

Радіус круга = 1

Площа круга = 3.14159

Зауваження:

- до відкритих членів базового класу можна звертатися в такий самий спосіб, як і до звичайного члена похідного класу;
- при створенні об'єктів похідного класу спочатку викликаються конструктори базових класів, а потім конструктор, який визначається в похідному класі. При знищенні об'єкта деструктори викликаються в оберненому порядку;
- для того, щоб передати аргументи конструкторам базових класів, використовується список ініціалізаторів.

3.5. Перетворення вказівників на об'єкти з однієї ієрархії класів

Припустимо, що похідний клас утворюється з базового за допомогою відкритого успадкування. Тоді відбувається неявне перетворення типів для вказівників на об'єкти з одної ієрархії, якщо ці перетворення – перетворення вгору (від сина до батька). Перетворення ж вниз (від батька до сина) треба здійснювати явно.

Продемонструємо такі перетворення для ієрархії класів з п. 3.4. Перепишемо основну функцію **main()**, в якій динамічно створимо два об'єкти різних типів з одної ієрархії та зробимо різні перетворення відповідних вказівників на них.

Приклад 3.5.1:

```
void main()
{
// режим кирилиці для консолі
SetConsoleOutputCP(1251);
SetConsoleCP(1251);

// створення об'єктів відповідними конструкторами
TCircle* pCircle=new TCircle(3),*p1 ;
TCylinder* pCylinder= new TCylinder(5, 1);
TCylinder* p2;
```

```
// виклик рідних методів через вказівники
pCircle->show();
pCylinder->show();

// виклик рідних методів через об'єкти,
// на які вказують вказівники
(*pCircle).show();
(*pCylinder).show();

// виклик методу класу TCircle для циліндра
// через вказівник
// неявне перетворення типу
p1 = pCylinder;
p1->show();

// виклик методу класу TCylinder для круга
// через вказівник
// явне перетворення типу
p2 = (TCylinder*)pCircle;
p2->show();

// затримка часу
cin.get();
}
```

Результат роботи програми:

Радіус круга = 3
Площа круга = 28.2743

Радіус основи = 1
Площа поверхні = 37.6991

Висота циліндра = 5
Об'єм циліндра = 15.708

Радіус круга = 3
Площа круга = 28.2743

Радіус основи = 1
Площа поверхні = 37.6991

Висота циліндра = 5
Об'єм циліндра = 15.708

Радіус круга = 1
Площа круга = 3.14159

Радіус основи = 3
Площа поверхні = 56.5487

Висота циліндра = 3.16621e-299
Об'єм циліндра = 8.95224e-298

Як видно з результатів роботи програми, неявні перетворення вказівників виконуються у напрямку від нащадка до батька, а в напрямку від батька до нащадка тип вказівника треба вказувати за допомогою операції перетворення типу.

При друці висоти й об'єму циліндра, який був здійснений функцією **show()** для екземпляра класу **TCircle**, виводяться числа (висота та об'єм), які можна вважати нулями. Це зрозуміло, оскільки для круга змінна **h** не визначалась, а отже, її значення дорівнює 0, що узгоджується з тим, що з математичної точки зору круг можна вважати циліндром з висотою 0.

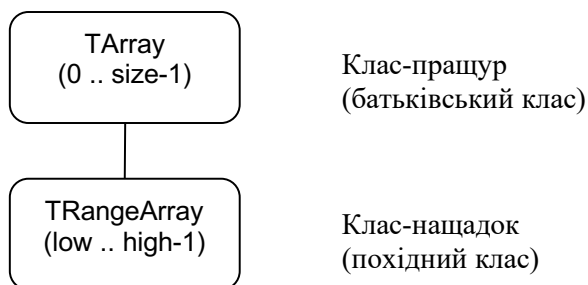
3.6. Перетворення типів для об'єктів з однієї ієрархії класів

Зауважимо, що похідний клас можна вважати спеціалізованим екземпляром базового, тому перетворення об'єкта похідного класу в тип базового класу часто відбувається неявно. Якщо ж необхідно виконувати обернені перетворення об'єктів, то доцільно реалізувати такі перетворення в конструкторі копіювання.

Тому далі наведемо приклад більш складного успадкування й покажемо використання в нащадку двох різних конструкторів копіювання. За рахунок цих конструкторів забезпечується робота методів прашурів і нащадків без необхідності написання всього набору методів для кожного з них. Ряд методів, написаних для похідних класів, з успіхом може використовуватися для екземплярів базових класів і навпаки.

Як базовий клас використаємо клас **TArray** – динамічний масив з межами зміни індексу елементів від 0 до **size-1**. Похідний клас – **TRangeArray** з межами зміни індексу елементів від **low** до **high-1**. В обох класах перевантажимо оператор індексування.

Ієрархія класів:



Приклад 3.6.1:

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

// власний простір імен з типами
// TArray, TName, TNameArray
namespace My
{
    // базовий клас - динамічний вектор
    // (індексування - з 0 до size-1)

    class TArray {
        int* v;
        int size;

    protected:
        // функції доступу для похідних класів
        int getSize(){return size;}
        int& getV(int i) const { return v[i]; }

    public:
        TArray(int); // конструктор
        TArray(const TArray&); // генератор копій
        ~TArray() {delete[] v;}; // деструктор
        TArray operator+(const TArray&) ;
        TArray& operator=(const TArray&);
        int& operator[] (int);
    };
};
```

```

TArray::TArray(int n)
{
    if (n<=0)
    {
        cerr<<"\nНеправильний розмір вектора";
        return;
    }
    size=n;
    v=new int[n];
}

TArray::TArray(const TArray& a)
{
    v=new int[size=a.size];
    for(int i=0;i<size;i++)
        v[i]=a.v[i];
}

int& TArray::operator[](int i)
{
    if(i<0 || size<=i)
    {
        cerr<<"індекс виходить за границі";
        return v[0];
    }
    return v[i];
}

TArray TArray::operator+(const TArray& two)
{
    TArray sum=*this;
    if (size != two.size)cerr<<"\nРозміри не співпадають";
    else for(int i=0;i<size;i++)
        sum[i]+=two.getV(i);
    return sum;
}

TArray& TArray::operator=(const TArray& a)
{
    if(&a == this)return *this;
    delete[] v;
    v=new int[size=a.size];
    for(int i=0;i<size;i++)
        v[i]=a.v[i];
}

```

```

    return *this;
}

// клас-нащадок з границями зміни індексу low .. high-1
class TRangeArray : public TArray
{
    int low, high;
public:
    TRangeArray (int,int);

// два генератора копій
    TRangeArray (const TRangeArray &);
    TRangeArray (const TArray& a):TArray(a)
    {
        low=0;
        high=getSize();
    }
//TRangeArray& operator=(const TRangeArray &); - можна
// не писати
    int& getV(int i)const
    {
        return TArray::getV(i-low);
    }
    int& operator[] (int i)
    {
        return TArray::operator[] (i-low);
    }

// буде працювати і для класу TArray
// за рахунок генератора копій TRangeArray (TArray&)
friend ostream& operator<<(ostream& os, TRangeArray a);

/* operator+ буде працювати і для класу TRangeArray за
рахунок генератора копій TArray(TRangeArray &),
який створюється автоматично */
};

TRangeArray ::TRangeArray (int left, int right):
    TArray(right-left)
{
    low = left;
    high = right;
    if (high < low) high = low;
}

```

```

TRangeArray ::TRangeArray (const TRangeArray & a):
    TArray(a.high-a.low)

{
    low=a.low;
    high=a.high;
    for(int i=low;i<high;i++)
//до вектора v напрямую звертаться не можна
        (*this)[i]=a.getV(i);
}

ostream& operator<<(ostream& os, TRangeArray a)
{
for(int i=a.low;i<a.high;i++)
    os<<a[i]<<" ";
return os;
}

int main(int argc, _TCHAR* argv[])
{
    TArray a(3),b(3),c(3);
    TRangeArray d(5,8),e(0,3),s(2,5);
for (int i=0;i<3;i++)
{
    a[i]=i+1;    // оператор індексування з класу TArray
    b[i]=i+4;    // оператор індексування з класу TArray

//оператор індексування з класу TRangeArray
    d[5+i]=(i+1)*2;

//викликається оператор індексування з класу TRangeArray
    e[i]=i*3;
}

TRangeArray f(a);    // TRangeArray (TArray&)

//викликається оператор << з класу TRangeArray
cout<<"\n f(a): "<<f;

//викликаються TRangeArray(TArray&), оператор <<
// з класу TRangeArray
cout<<"\n a: "<<a;

```

```

//викликаються TArray(TRangeArray&),
//оператор << з класу TRangeArray
cout<<"\n b: "<<b; cout<<"\n d[5-8]: "<<d;

// викликається оператор << з класу TRangeArray
cout<<"\n e[0-3]: "<<e;

// викликається TRangeArray(TRangeArray &)
TRangeArray j(d);

// викликається оператор << з класу TRangeArray
cout<<"\n j(d): "<<j;

// викликаються operator+(a,b), TRangeArray (TArray&),
// operator= з класу TRangeArray
s=c=a+b;

// викликається оператор << з класу TRangeArray
cout<<"\n (a+b)[2-5]: "<<s;

TRangeArray g(s); // викликається TRangeArray(TArray&)

// викликається оператор << з класу TRangeArray
cout<<"\n g()[0-3]: "<<g;

// генератор копій TArray(TRangeArray &) нами не
//написаний, він працює автоматично
TRangeArray k(a);
cout<<"\n k(a): "<<k;

d=a;
cout<<"\n d=a d[5-8]: "<<d;

// операторів operator+(TRangeArray, TArray) i
// operator+(TRangeArray,TRangeArray) немає
// (не написані), але за рахунок генераторів копій
// наступний код працює:
d=d+a;
cout<<"\n d=a[3]+d[5-8]: "<<d;
a=e+d;
cout<<"\n a=e[0-3]+d[5-8]: "<<a;
getch();
return 0;
}
}

```

Результат роботи програми:

```
a: 1 2 3
b: 4 5 6
d[5-8]: 2 4 6
e[0-3]: 0 3 6
f(a): 1 2 3
j(d): 2 4 6
(a+b)[2-5]: 5 7 9
g()[0-3]: 5 7 9
k(a): 1 2 3
d=a d[5-8]: 1 2 3
d=a[3]+d[5-8]: 2 4 6
a=e[0-3]+d[5-8]: 2 7 12
```

3.7. Множинне успадкування

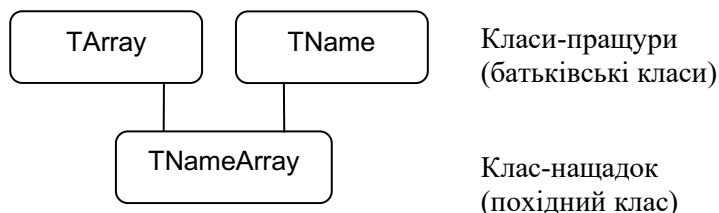
Як уже зазначалось, при побудові похідного класу є можливість успадковувати не один базовий клас, а кілька. У цьому випадку відбувається множинне успадкування. Щоб вказати, по відношенню до яких базових класів новий клас є похідним, в оголошенні класу перед списком його членів наводиться список базових класів (класів-пращурів).

Кожен з базових класів, у свою чергу, може бути похідним від інших базових класів. Отже, ієрархія успадкування класів може бути дуже складною.

Розглянемо далі приклад розробки двох базових і одного похідного класів. Як один з базових класів використаємо спрощений варіант класу **TArray**, розглянутий вище в п. 3.4. Другий базовий клас **TName** – динамічний рядок. Їх спільний нащадок **TNameArray** – динамічний масив із динамічним іменем.

При конструюванні об'єктів типу **TNameArray** будемо викликати конструктори базових класів, деструктор же залишимо порожнім (деструктори базових класів при цьому викликаються автоматично).

Ієрархія класів:



Приклад 3.7.1:

```
#include <vcl.h>
#include <windows.h>

#include <string.h>
#include <conio.h>
#include <iostream.h>

//власний простір імен з типами
// TArray, TName, TNameArray
namespace My
{
// базовий клас - динамічний вектор
// індексування - з 0 до size-1)
class TArray
{
protected:
    int size;
    int* v;

public:
    TArray(int); // конструктор
    TArray(const TArray&); // генератор копій
    ~TArray() {delete[] v;}; // деструктор
    TArray& operator=(const TArray&);
    void print() const;
};

TArray::TArray(int n)
{
    size=n;
    v=new int[n];
```

```

        for(int i=0;i<n;i++)
            v[i]=random(10);
    }

TArray::TArray(const TArray& a)
{
    v=new int[size=a.size];
    for(int i=0;i<size;i++)
        v[i]=a.v[i];
}

TArray& TArray::operator=( const TArray& a)
{
    if(&a == this)return *this; // контроль самоприсвоєння
    delete[] v;
    v=new int[size=a.size];
    for(int i=0;i<size;i++)
        v[i]=a.v[i];
    return *this;
}

void TArray::print()const
{
    for(int i=0;i<size;i++)
        cout<<v[i]<<" ";
}

// базовий клас - динамічний рядок з іменем
class TName
{
protected:
    char* s;
public:
    TName(char* str){s=strdup(str);}
    TName(const TName&);
    ~TName(){delete[] s;}
    TName& operator=(const TName&);
    void print()const;
};

void TName::print()const
{
    cout<<s<<endl;
}

```

```

TName::TName(const TName& a)
{
    s=new char[strlen(a.s)+1];
    strcpy(s,a.s);
}

TName& TName::operator=(const TName& a)
{
    if (this != &a) // контроль самоприсвоєння
    {
        delete[] s;
        s=new char[strlen(a.s)+1];
        strcpy(s,a.s);
    }
    return *this;
}

// множинне успадкування
// клас-нащадок (масив з іменем)
class TNameArray: public TArray, public TName
{
public:
    TNameArray(int, int*, char*); // конструктор
    TNameArray(TArray, TName); // конструктор
    TNameArray(const TNameArray&); // генератор копій
    ~TNameArray() {}; // деструктор
    void print()const;
};

void TNameArray::print()const
{
    cout<<"\nМасив з іменем: ";
    TName::print(); //область видимості базового класу TName
    TArray::print(); //область видимості базового класу
    TArray
}

TNameArray::TNameArray(const TNameArray& a):
    TArray(a), TName(a)
{
}

TNameArray::TNameArray(int n, int * mas, char* str ):
    TArray(n), TName(str)

```

```

    {
        for(int i=0;i<size; i++)
            v[i]=mas[i];
    }

    TNameArray::TNameArray(TArray mas, TName name):
        TArray(mas), TName(name)
    {}

int main(int argc, _TCHAR* argv[])
{
    // режим кирилиці для консолі (встановити властивість
    // шрифту консолі - Lucida Console)
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);

    TArray a(3),b(3),c(a), d(3);

    cout<<"\nЧислові масиви\n";
    cout<<"\na: "; a.print();
    cout<<"\nb: "; b.print();
    cout<<"\nc: "; c.print();

    d=a;
    cout<<"\nd=a: "; d.print();

    // виклик конструкторів з параметром
    TName s1("abcabc6"), s2("123"), s4("NewRow");

    cout<<"\n\nРядки\n";
    // виклик конструктора копіювання
    TName s3=s2;
    s3.print();

    // виклик оператора присвоєння (ланцюгове присвоєння)
    s2=s3=s1;

    // друк результатів
    s1.print();
    s2.print();
    s3.print();
    s4.print();

```

```

int v[]={5,4,3,2,1,6,7,8,9,10};
// виклик конструкторів з параметром
TNameArray ns1(4,v,"SmallRow"),ns2(6,v+3,"BigRow"),
              ns3(5,v+2,"Row"), ns4(5,v+2,"Row");

// виклик конструктора копіювання
TNameArray ns5(ns1);

cout<<"\nЧислові масиви з іменами\n";
ns1.print();
ns2.print();
ns3.print();
ns5.print();

// викликаються оператори присвоєння батьківських класів
ns4=ns2;
ns2.print();

// явна передача конструктору екземплярів
// батьківських класів
TNameArray ns6(a,ns3);
ns6.print();

cin.get();
return 0;
}

};

```

Результат роботи програми:

Числові масиви

```

a: 7 4 7
b: 7 6 0
c: 7 4 7
d=a: 7 4 7

```

Рядки

```

123
abcabc6
abcabc6
abcabc6
NewRow

```

Числові масиви з іменами

```

Масив з іменем: SmallRow
5 4 3 2
Масив з іменем: BigRow
2 1 6 7 8 9
Масив з іменем: Row
3 2 1 6 7
Масив з іменем: SmallRow
5 4 3 2
Масив з іменем: BigRow
2 1 6 7 8 9
Масив з іменем: Row
7 4 7

```

Зауважимо, що при множинному успадкуванні часто виникають побічні ефекти й помилки, пов'язані з використанням однакових імен у різних прашкурах. Для їх часткового усунення використовується оператор прив'язки, в якому задається область видимості члена базового класу. Так зроблено в тілі функції **print()** класу-нащадка **TNameArray**.

У більш складних ситуаціях (ромбоподібна ієрархія) ця проблема розв'язується за допомогою віртуального успадкування, яке розглядається в п. 4.6.

Запитання для самоконтролю

1. Що таке агрегація? Які бувають типи агрегації?
2. Які приклади агрегації можете навести?
3. Чи можна використовувати як члени класу вказівники на інші класи?
4. Навіщо потрібні шаблони класів?
5. Які ключові слова можна використовувати для позначення деякого узагальненого типу в шаблоні класу?
6. Параметри якого типу дозволяється використовувати при описі шаблону?
7. Скільки може бути параметрів у шаблоні?
8. Що таке конкретизація шаблону?
9. Навіщо потрібне успадкування?

10. Які специфікатори використовуються при успадкуванні класів?
11. Скільки може бути базових класів у похідному класі?
12. Як взаємодіють специфікатори доступу з назвами секцій базових класів?
13. Навіщо потрібна секція **protected**?
14. Чому не рекомендується оголошувати всі секції базових класів як **public**?
15. Коли потрібне множинне успадкування?
16. Які проблеми можуть виникнути при множинному успадкуванні?
17. Як зазвичай викликаються конструктори базових класів у конструкторах похідних класів?
18. Які бувають типи конструкторів копіювання у похідних класах?
19. Які перетворення вказівників на об'єкти з одної ієрархії класів відбуваються неявно, а які необхідно перетворювати явно?
20. Чи успадковується оператор присвоєння?

Завдання для самостійної роботи

1. Успадкуйте клас **TCylinder** з ієрархії класів п. 3.4 і створіть клас **TNameCylinder** "циліндр з іменем".
2. Успадкуйте класи **TName** з ієрархії класів п. 3.7 і **TCircle** з ієрархії класів п. 3.4 та створіть клас **TNameCircle** "коло з іменем" з використанням множинного успадкування.
3. Доповніть основну функцію з прикладу 3.6.1 динамічним створенням об'єктів з ієрархії класів і викликом відповідних методів через вказівники.
4. Розробіть клас **TCylinder** з прикладу 3.4.1 не методом успадкування, а використовуючи агрегацію з використанням вказівника на об'єкт типу **TCircle**.
5. Розробіть клас **TNameArray** з прикладу 3.7.1 не методом множинного успадкування, а використовуючи агрегацію за значенням (об'єкти типу **TArray** і **TName**).

4. Розробка сім'ї класів. Абстракція

4.1. Статичний і динамічний поліморфізм

Поліморфізм – це остання із трьох основних концепцій, на якій базується об'єктно-орієнтоване програмування.

У перекладі з грецької мови поліморфізм означає багатоформність. Щодо ООП на мові **C++**, то цей термін означає здатність об'єкта відреагувати на деякий запит (тобто виклик функції-члена) відповідно до свого типу, навіть якщо на стадії компіляції тип об'єкта ще невідомий.

У **C++** поліморфізм підтримується двома способами.

По-перше, при компіляції він підтримується за допомогою перевантаження функцій і операторів. Такий вид поліморфізму називається *статичним поліморфізмом*, оскільки він реалізується ще до виконання програми, шляхом раннього зв'язування ідентифікаторів функцій з фізичними адресами на стадії компіляції та компонування.

По-друге, під час виконання програми він підтримується за допомогою віртуальних функцій. Зустрівши в коді програми виклик віртуальної функції, компілятор (точніше – компоновщик) тільки позначає цей виклик, залишаючи зв'язування ідентифікатора функції з її адресою до стадії виконання. Такий процес називається *пізнім зв'язуванням*.

Віртуальна функція – це функція, виклик якої (і виконувани при цьому дії) залежить від типу об'єкта, для якого вона викликана. Об'єкт визначає, яку функцію потрібно викликати вже під час виконання програми. Цей вид поліморфізму називається *динамічним поліморфізмом*.

Якщо в програмі планується працювати одночасно з різними типами об'єктів ієрархії або планується додавання в ієрархію нових об'єктів, метод оголошується як віртуальний за допомогою ключового слова **virtual**. Усі віртуальні методи ієрархії з одним і тим самим іменем повинні мати однаковий список аргументів.

Клас, що містить хоча б одну віртуальну функцію, називається *поліморфним*.

4.2. Віртуальні функції

Підкреслимо далі ряд важливих фактів, суттєвих при використанні поліморфних класів.

- Якщо в деякому класі є функція, описана **virtual**, тоді до такого класу компілятором додається захований член даних – вказівник на таблицю віртуальних функцій, а також згенерується спеціальний код, який дозволяє здійснити вибір віртуальної функції для цього об'єкта *під час виконання програми* (а не під час компіляції).
- Якщо викликається віртуальна функція, то код, згенерований компілятором, знаходить вказівник на таблицю віртуальних функцій, потім переглядає цю таблицю і знаходить адресу відповідної функції. Після цього виконується перехід на зазначену адресу й виклик функції. Як було сказано вище, при створенні об'єкта похідного класу спочатку викликається конструктор базового класу. Саме на цій стадії створюється таблиця віртуальних функцій і вказівник на неї. Після виклику конструктора похідного класу вказівник на таблицю віртуальних функцій настраюється так, що тепер він посилається на перевизначений варіант віртуальної функції, якщо такий є.
- Якщо деяка функція оголошена в базовому класі як віртуальна, тоді функція з таким самим іменем, списком параметрів і типом значення, яке повертається, в похідному класі автоматично стає віртуальною (слово **virtual** можна не повторювати).
- Якщо віртуальна функція не була перевизначена в похідному класі, то коли вона викликається для об'єкта похідного класу, буде відбуватися виклик відповідної функції з найближчого за ієрархією базового класу, де

вона була визначена.

- Членом похідного класу може бути функція з іменем, яке збігається з іменем віртуальної функції-члена базового класу, але з іншим списком параметрів. Тоді вона буде звичайною невіртуальною функцією.
- Конструктор класу не може бути віртуальним, тому що він викликається тільки при створенні об'єкта і тип цього об'єкта повинен бути відомим компілятору.
- Деструктор може бути віртуальним.

4.3. Абстрактні класи

Якщо при визначенні базового класу деяка віртуальна функція оголошена членом цього класу, то вона повинна бути визначена подібно звичайним функціям або бути чистою віртуальною функцією, тобто має бути оголошена *абстрактною* за допомогою значення `= 0`. Це означає, що функція поки що не має тіла, але відомий її інтерфейс.

```
class TBase
{
...
virtual void f(int) = 0; //абстрактна функція
...
};
```

Клас, який містить хоча б одну абстрактну функцію, називається *абстрактним класом*. Абстрактний клас може бути тільки базовим для інших класів. Об'єкт цього класу створити неможливо. В похідних класах абстрактна функція повинна бути визначена або знову оголошена абстрактною. Якщо клас, похідний від абстрактного, не визначає всі чисті віртуальні функції, то він залишається абстрактним.

Абстрактний клас не можна застосовувати як параметр функції або як тип значення, що повертається. Його не можна використовувати при явному приведенні типів. Але можна визначати посилання і вказівники на абстрактні класи.

Приклад 4.3.1:

Умова задачі: Створити абстрактний базовий клас **TNumber** (число, записане в рядку символів) з чистими віртуальними арифметичними операцією "+"=" і функцією **sum**.

Створити похідні класи **TIntNumber** і **TCharNumber** з власними реалізаціями віртуальних операцій і функцій. У класі **TIntNumber** операція "+"=" знаходить арифметичну суму двох чисел, а функція **sum** – суму цифр одного числа.

У класі **TCharNumber** операція "+"=" знаходить конкатенацію двох чисел-рядків, а функція **sum** – кількість цифр одного числа-рядка.

Програмна реалізація:

```
#include <vcl.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>

class TNumber {
protected :
    char a[50];
public:
    TNumber(char A[]){strcpy(a,A);}
    virtual void operator+=(TNumber&)=0 ;
    virtual void print()=0;
    virtual int sum()=0;
};

class TIntNumber :public TNumber
{
public:
    TIntNumber(char A[]):TNumber(A) {}
    virtual void operator+=(TNumber&) ;
    virtual void print();
    virtual int sum();
};

int TIntNumber::sum()
{
```

```

int i=0,s=0;
while(i<strlen(a))s+=a[i++]-'0';
return s;
}

void TIntNumber::operator+=(TNumber& ob)
{
TIntNumber* p=(TIntNumber*)&ob;
int s=atoi(a)+atoi(p->a); // atoi(ob.a) - не працює
itoa(s,a,10);
}

void TIntNumber::print()
{cout<<"\nTIntNumber:"<<a;}

class TCharNumber: public TNumber
{
public:
TCharNumber (char A[]):(A) {}
virtual void operator+=(TNumber&) ;
virtual void print();
int sum(){return strlen(a);}
};

void TCharNumber::operator+=(TNumber& ob)
{ TCharNumber* p=(TCharNumber*)&ob;
strcat(a,p->a);
}

void TCharNumber::print()

{cout<<"\nTCharNumber:"<<a;}

int main()
{
TIntNumber a("12"),b("345"),c("35");
TCharNumber A("12"),B("345"),C("35");
TNumber* mas[6]={&a,&b,&c,&A,&B,&C};

b+=a;
b.print();

C+=A;
C.print();

for(int i=0;i<6;i++)

```

```

{
mas[i]->print();
cout<<"    sum="<<mas[i]->sum();
}

getch();
return 0;
}

```

Результат роботи програми:

```

TIntNumber:357
TCharNumber:3512
TIntNumber:12    sum=3
TIntNumber:357   sum=15
TIntNumber:35    sum=8
TCharNumber:12   sum=2
TCharNumber:345  sum=3
TCharNumber:3512 sum=4

```

4.4. Проектування і побудова бібліотеки класів із віртуальними функціями

Розглянемо процес створення графічної бібліотеки класів. Класи повинні маніпулювати з фігурами, які обертаються навколо заданої точки.

У процесі проектування перед нами стоїть задача – розробка ієрархії класів. Очевидно, що класи повинні якомога ближче відповідати модельованим об'єктам. Тому в загальному випадку у предметній області виділяються поняття, які можна використовувати як класи. Крім класів з прикладної області, як правило, з'являються класи, пов'язані із середовищем розробки. Будемо створювати проект у середовищі розробки **C++Builder** і використовувати компонентний клас **TForm1** (нащадок класу **TForm**) для роботи з формою та її обробниками подій.

Визначимо операції над класами, які згодом стануть методами класу. У загальному випадку їх можна розбити на групи:

- пов'язані з конструюванням і копіюванням об'єктів;
- для підтримки зв'язків між класами, які існують у прикладній області;
- дозволяють представити роботу з об'єктами в зручному вигляді.

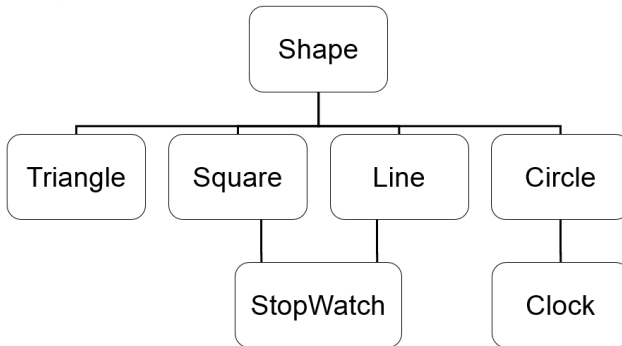
Далі визначаються функції, які будуть віртуальними.

Процес створення ієрархії класів – ітераційний. Наприклад, можна у двох класах виділити загальну частину й перемістити її в базовий клас, а потрібні класи зробити його похідними.

Враховуючи вищесказане, у корінь ієрархії винесемо абстрактний клас **Shape**, в якому визначимо характеристики, спільні для майбутньої сім'ї. Зокрема, у захищеній секції розмістимо центр обертання майбутньої фігури та її колір.

У відкритій секції оголосимо дві реальні і дві абстрактні функції. Реальні функції **where()** і **move()** будуть відповідно визначати й пересувати центр обертання фігури. Абстрактні ж функції **draw()** і **rotate()** лише задають прототипи функцій малювання та повороту, які будуть реалізовані в нащадках.

Нижче наведено можливу ієрархію успадкування класів.



Продемонструємо спочатку розробку таких реальних класів:

- **Line** (стрілка годинника);
- **Circle** (циферблат);
- **Clock** (годинник з двома стрілками).

Зазначимо, що у класі **Clock** буде реалізовано успадкування класу **Circle** для створення циферблата й агрегація класу **Line** для створення стрілок годинника.

Для наочної демонстрації роботи з об'єктами класів **Clock** створимо проект у середовищі **C++Builder**, в якому реалізуємо обробник події **Click** для форми: намалюємо окремі об'єкти, створимо поліморфні функції для маніпуляції об'єктами різних типів з ієрархії класів і запустимо годинники з двома стрілками.

За допомогою елемента керування типу **Timer** синхронізуємо роботу годинників з роботою таймера (початок роботи одного з годинників визначається системним часом, а другого – дванадцятю годиною дня).

Годинники для наочності працюватимуть у прискореному режимі.

Приклад 4.4.1:

```
// підключення класу з формою
#include "Abstract.h"

#include <vcl.h>
#include <math.h>

// опис вказівника на форму
TForm1 *Form1;

// абстрактний клас (фігура)
class Shape
{
protected:
    TPoint center; // тип, визначений у бібліотеці vcl
    TColor col;    // тип, визначений у бібліотеці vcl

public:
    TPoint where () {return center;}
    void move (TPoint newCenter)
        {
            center= newCenter;
            draw ();
        }

// абстрактні функції
```

```

        virtual void draw ()=0;
        virtual void rotate (int)=0 ;
};

// реальний клас (коло)
class Circle: public Shape
{
    int radius;
public:
    Circle(){radius=100; col= clDkGray;
            center.x=700; center.y=300;};
    Circle(int r, TPoint p, TColor c)
        {radius=r; center=p; col=c;}
    void draw ();
    void rotate (int) {} // порожня функція
};

// реальний клас (відрізок)
class Line: public Shape
{
    int alpha;
    int lenght;
public:
    Line(){center= TPoint(700,300);
          alpha=0; lenght=90; col=clDkGray;}
    Line(TPoint cen, int a, int len, TColor c)
        {center=cen; alpha=a; lenght=len; col=c;}
    void draw ();
    void rotate (int a){alpha+=a;};
    friend class Clock;
};

__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{}

TPoint operator+(TPoint p,int r)
{
    p.x+=r;
    p.y+=r;
    return p;
}

```

```

void Circle::draw()
{
    TPoint p1,p2;
    p1=center+(-radius);
    p2=center+radius;
    Form1->Canvas->Brush->Style=bsClear;
    Form1->Canvas->Pen->Color=col;
    Form1->Canvas->Ellipse(TRect(p1,p2)) ;
}

void Line::draw()
{
    TPoint p;
    float fi=alpha*M_PI/180;
    p.x=center.x+sin(fi)*lenght;
    p.y=center.y-cos(fi)*lenght;

    Form1->Canvas->Pen->Color=col;
    Form1->Canvas->MoveTo(center.x,center.y) ;
    Form1->Canvas->LineTo(p.x,p.y) ;
}

// реальний клас (годинник)
class Clock : public Circle // успадкування
{
    Line hour, minute; //агрегація
public:
    Clock(){set_now();};
    Clock(int radius,TPoint cen, TColor color):
        Circle( radius,cen,color)
    {
        minute=Line(cen, 0, radius*0.9, color);
        hour=Line(cen, 0, radius*0.6, color);
    }
    // малювання годинника
    void draw () ;
    // зсув хвилинної стрілки
    void rotate(int fi){minute.rotate(fi) ;}
    // встановлення часу за параметрами
    void set (int fi, int psi)
    {
        minute.rotate(fi);
        hour.rotate(psi);
    }
}

```

```

// встановлення поточного часу
void set_now();

};

// малювання годинника
void Clock::draw ()
{
    Circle::draw(); // циферблат (успадкування)
    minute.draw(); // малювання стрілок
    hour.draw(); // агрегація
}

// встановлення поточного часу
void Clock::set_now()
{
    Word Hour, Min, Sec, MSec;
    DecodeTime(Now(), Hour, Min, Sec, MSec);
    if (Hour>=12) Hour-=12;
    set( Min*6, Hour*30+Min/2);
    hour.lenght=minute.lenght*0.7;
}

// обробник події Click для форми
void __fastcall TForm1::FormClick(TObject *Sender)
{
    TPoint T1(300,400), T2(200,100) ;;
    Circle C(100,T1,clRed);
    C.draw ();
    C.move (T2) ; // виклик функції з абстрактного класу
    C.draw () ;
    Shape* p[2];
    p[0] = &C;
    p[1]=new Line(T2,25,100,clBlue);
    p[1]->draw(); // віртуальна функція
    p[1]->rotate(45); // віртуальна функція

    Sleep(5000);
    Clock timeNow, time(200, TPoint(300,300),clRed);
    timeNow.draw(); // віртуальна функція
    time.draw();
    Timer1->Enabled=true;
}

// обробник події від таймера на формі
// демонстрація роботи двох годинників

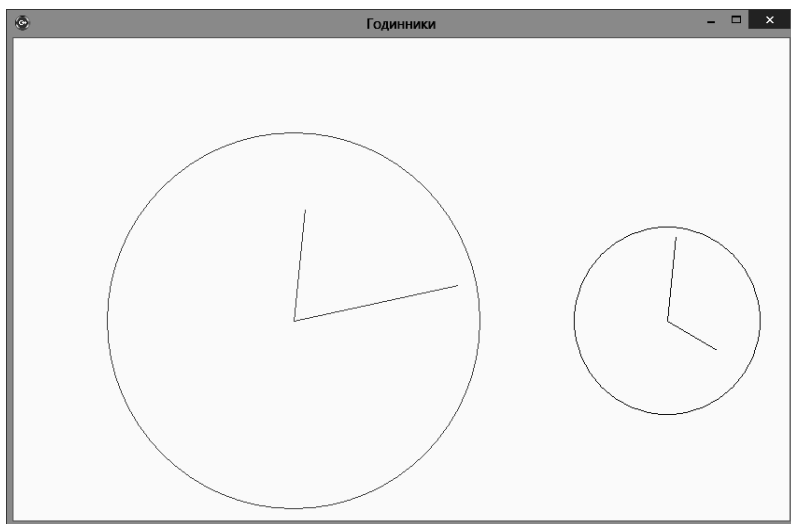
```

```

void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    static Clock timeNow,time(200, TPoint(300,300),clRed);
    static int flag=0;
    timeNow.set(6,flag);
    time.set(6,flag);
    flag=!flag;
    Form1->Canvas->Brush->Color=Color;
    Form1->Canvas->FillRect(TRect(0,0,Form1->Width,
                                   Form1->Height));
    timeNow.draw(); // віртуальна функція
    time.draw();   // віртуальна функція
}

```

Нижче наведено скріншот програми у той момент, коли демонструються два годинники (годинники працюють у прискореному режимі).



4.5. Поліморфні функції

Функції, які однаково маніпулюють з різними типами об'єктів з однієї ієрархії класів за допомогою механізму віртуальних функцій, називаються *поліморфними*.

Такі функції за допомогою таблиці віртуальних функцій визначають тип об'єкта й викликають потрібні функції під час виконання програми.

Доповнимо розглянутий у п.4.4. приклад поліморфними функціями **rotate_all()** і **move_all()**, які будуть відповідно повертати й пересувати всі фігури, вказівники на які передаються цим функціям у динамічно створеному масиві вказівників типу **Shape****.

Крім того, продемонструємо використання динамічного створення об'єктів різних типів з однієї ієрархії класів.

Приклад 4.5.1:

```
// обернути всі елементи вектора v
// параметри size (розмір вектора), alpha (кут
// повертання)
// поліморфна функція

void rotate_all (Shape** v, int size, int alpha)
{
    for (int i=0; i<size; i++)
        // викликається для кожного об'єкта
        //своя віртуальна функція
        v[i]->rotate(alpha) ;
}

// пересунути всі елементи вектора v у центр p
// поліморфна функція

void move_all (Shape** v, int size, TPoint p)
{
    for (int i=0; i<size; i++)
        // викликається для кожного об'єкта
        //своя віртуальна функція
        v[i]->move(p) ;
}

// обробник події Click для форми
void __fastcall TForm1::FormClick(TObject *Sender)
{
    TPoint T1(300,400), T2(200,100) ;;
    Circle C(100,T1,clRed);
    C.draw ();
}
```

```

C.move (T2) ; // виклик функції з абстрактного класу
C.draw () ;
Shape* p[5];

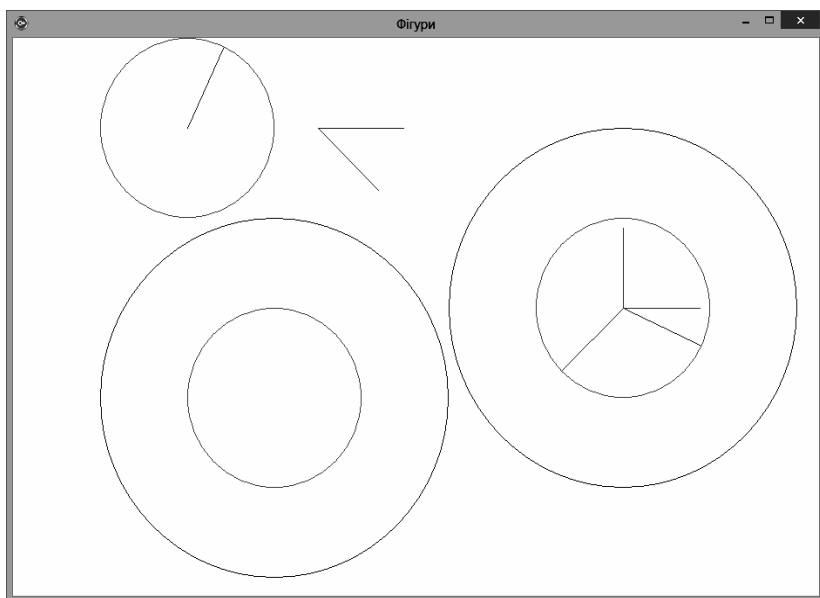
p[0] = &C;
p[1]=new Circle(200,T1,clBlack);
p[2]=new Line (T2,25,100,clBlue);
p[3]= new Line;
p[4]= new Line(TPoint(350,100),90,100,clGreen);
p[4]->draw(); // віртуальна функція
p[4]->rotate(45); // віртуальна функція

for(int i=0;i<5;i++)
    p[i]->draw();

rotate_all(p,5,90); // поліморфна функція
move_all(p,5,p[3]->where()); // поліморфна функція
}

```

Нижче наведено скріншот програми, який демонструє роботу віртуальних і поліморфних функцій.



4.6. Віртуальне успадкування

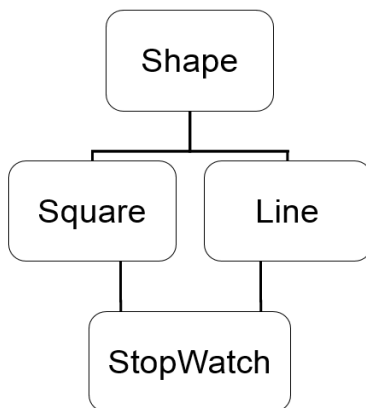
Зауважимо, що у п. 4.4 при побудові класу **Clock** (годинник з двома стрілками) клас **Line** був агрегований у клас **Clock** двічі, а клас **Circle** (циферблат) – успадкований. Якщо ж замість агрегації використати невіртуальне множинне успадкування, то виникне ряд проблем, пов'язаних з появою успадкованих членів класів з однаковими іменами. Таких проблем можна уникнути, успадкувавши абстрактний клас **Shape** віртуально. У такому випадку нащадки будуть звертатися до успадкованих членів даних класу **Shape** через неявні вказівники, які будуть вказувати на одну спільну область пам'яті для однойменних членів даних з віртуально успадкованого класу.

Продемонструємо такий підхід для побудови класу **StopWatch** (секундомір).

Як тип для стрілки секундоміра використаємо клас **Line**, а як тип для циферблата – клас **Square** (квадрат).

Нагадаємо, що класи **Shape** і **Line** вже розроблені в п. 4.4. Залишилося створити класи **Square** і **StopWatch**.

Наведемо фрагмент ієрархії класів, яка тепер має ромбоподібну структуру.



У корені ієрархії, як і раніше, – абстрактний клас **Shape** (тепер успадковується віртуально), а для побудови класу **StopWatch** будемо використовувати множинне успадкування.

Приклад 4.6.1:

```
// віртуальне успадкування (реальний клас - квадрат)
class Square: virtual public Shape
{
    int radius; // радіус вписаного кола
public:
    Square()
    {
        radius=100,
        col= clDkGray;
        center.x=700;
        center.y=300;
    };
    Square(int r, TPoint p, TColor c)
    {
        radius=r;
        center=p;
        col=c;
    }
    void draw ();
    void rotate (int) {} //порожня функція
};

// малювання циферблата (квадрата)
void Square::draw()
{
    TPoint p1,p2;
    p1=center+(-radius);
    p2=center+radius;
    Form1->Canvas->Brush->Style=bsClear;
    Form1->Canvas->Pen->Color=col;
    Form1->Canvas->Rectangle(TRect(p1,p2)) ;
}

// множинне успадкування (реальний клас - секундомір)
class StopWatch : public Square, public Line
{
public:
    StopWatch(){set();};
    StopWatch(int radius,TPoint cen,  TColor color):
```

```

        // список ініціалізаторів
        Square(radius,cen,color),
        Line(cen, radius,radius*0.9, color)
    {}

// малювання секундоміра
    void draw ();

// зсув секундної стрілки
    void rotate( int fi)
    {
        Line::rotate(fi);
    }

// встановлення секундної стрілки
    void set ()
    {
        Line::rotate(0);
    }
};

// малювання секундоміра
void Stopwatch::draw ()
{
    Square::draw(); // малювання циферблата
    Line::draw();   // малювання стрілки
}

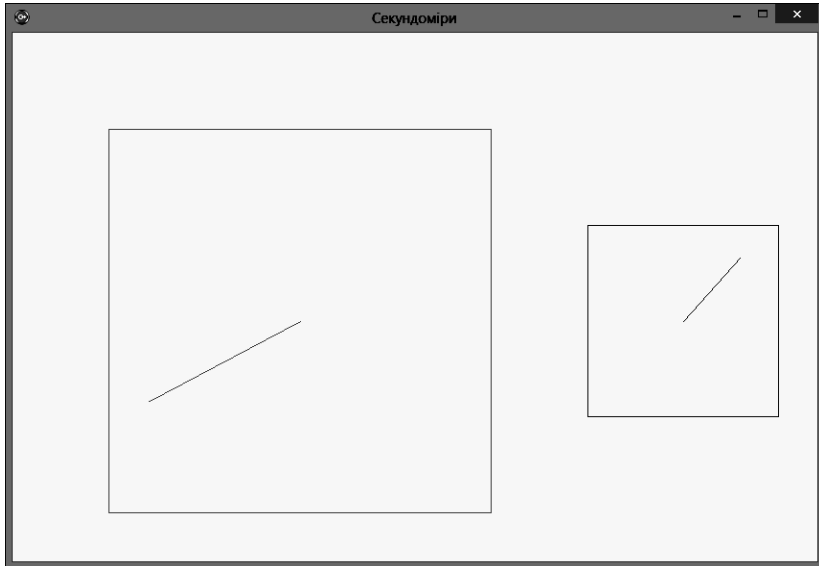
// демонстрація роботи двох секундомірів
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
    static Stopwatch secNow,
        sec(200, TPoint(300,300),clRed);
    static int flag=0;

    secNow.rotate(6);
    sec.rotate(6);
    flag=!flag;
    Form1->Canvas->Brush->Color=Color;
    Form1->Canvas->FillRect(TRect(0,0,Form1->Width,
        Form1->Height));

    secNow.draw(); // віртуальна функція
    sec.draw();    // віртуальна функція
}

```

Нижче наведено скріншот програми, який демонструє застосування віртуального множинного успадкування (робота двох секундмірів).



Запитання для самоконтролю

1. Що таке статичний поліморфізм? Наведіть реальні приклади.
2. Що таке динамічний поліморфізм? Які функції забезпечують механізм динамічного поліморфізму?
3. Що таке таблиці віртуальних функцій? Як вона створюється і використовується?
4. Які проблеми виникають і як вони розв'язуються, якщо у класах з однієї ієрархії з'являються члени з однаковими іменами?
5. Що таке чиста віртуальна функція?
6. Що таке абстрактний і реальний класи?
7. Чи можна створити об'єкти абстрактного класу? А реального?
8. Навіщо потрібні абстрактні класи?

9. Чи може похідний клас бути абстрактним?
10. Чи можна оголосити бінарний оператор чистою віртуальною функцією-членом абстрактного базового класу? А унарний оператор?
11. Чи можна застосовувати тип "абстрактний клас" для передачі параметра в деяку функцію? А для повернення результату?
12. Що таке поліморфні функції?
13. Коли виникає необхідність застосовувати віртуальне успадкування?
14. Чи можна оголошувати абстрактні класи, якщо планується використання множинного успадкування?
15. Що таке ромбоподібна ієрархія? Які проблеми виникають при побудові такої ієрархії?

Завдання для самостійної роботи

1. Реалізуйте клас **Triangle** з ієрархії класів п. 4.4 і створіть годинник із трикутним циферблатом без використання множинного успадкування.
2. Реалізуйте клас **Triangle** з ієрархії класів п. 4.4 і створіть секундомір з трикутним циферблатом з використанням множинного віртуального успадкування.
3. Доповніть клас **StopWatch** з п. 4.6 оператором постфіксного унарного збільшення і зменшення секунд на секундомірі.
4. Доповніть клас **Clock** з п. 4.4 оператором префіксного унарного збільшення і зменшення хвилин на годиннику.
5. Успадкуйте класи **Clock** і **Square** з ієрархії класів п. 4.4 і побудуйте клас **FrameClock** (годинник із рамкою).

5. Лабораторні роботи

Лабораторна робота № 1

Тема: Структури. Зовнішні функції і функції-члени структури

Завдання

Структурою-парою прийнято називати структуру, яка складається з двох полів з іменами **first** і **second**.

Необхідно реалізувати власний тип даних за допомогою цієї структури [11].

Створити кілька об'єктів (**ob**, **ob1**, **ob2**,...) такого структурного типу.

В усіх завданнях реалізувати:

- введення структурної змінної (функція **input()**);
- функцію **check()**, яка повинна забезпечувати перевірку введених функцією **input()** значень на коректність;
- виведення на екран уведеної інформації і результату роботи (функція **output()**).

При введенні помилкових значень параметрів слід вивести відповідне повідомлення і знов запропонувати ввести потрібні значення.

Програму реалізувати у двох варіантах:

- 1) всі необхідні функції описати як звичайні зовнішні функції засобами мови **C**;
- 2) функції, які оперують *одним об'єктом*, описати як функції-члени структури (мова **C++**); для введення/виведення інформації використовувати об'єкти **cin**, **cout**.

Зауваження: для реалізації деяких власних структурних типів може виникнути необхідність у створенні ще одного допоміжного структурного типу, за допомогою якого будуть визначатися поля **first** і **second**.

Варіанти завдань

1. Квадратична функція $y = -x^2 + Ax + B$. Поле **first** – дійсне число, коефіцієнт **A**; поле **second** – дійсне число, коефіцієнт **B**. Реалізувати методи **argument(ob)** – обчислення для заданого **y** відповідних значень **x**, **isPeretyn(ob)** – визначення, чи перетинається парабола з віссю **OX**, **sim(ob)** – перетворення параболи у симетричну до неї параболу відносно осі **OY**.
2. Лінійна функція $y = Ax + B$. Поле **first** – дійсне число, коефіцієнт **A**; поле **second** – дійсне число, коефіцієнт **B**. Реалізувати методи **function()** – обчислення для заданого **x** значення функції **y**, **isPeretyn(ob1,ob2)** – визначення, чи перетинаються дві прямі.
3. Комплексне число. Поле **first** – дійсна частина комплексного числа, поле **second** – уявна частина. Реалізувати методи знаходження кореня **n**-го степеня з комплексного числа, знаходження аргумента комплексного числа, **compare(ob1,ob2)** – порівняння двох об'єктів за модулем.
4. Поле **first** – дріб (чисельник, знаменник); поле **second** – ціле число – показник степеня. Реалізувати метод **power()** – піднесення числа **first** до степеня **second**. Метод повинен працювати правильно для всіх допустимих значень **first** та **second**. Не використовувати функцію **pow()** з бібліотеки **math**.
5. Поле **first** – додатне ціле число, чисельник; поле **second** – додатне ціле число, знаменник. Реалізувати метод **ipart()** – виділення цілої частини дробу **first/second** і метод **add()** – знаходження дробу, при додаванні якого отримується найближче до значення **first/second** ціле

число. Функція **check()** повинна перевіряти рівність знаменника нулю.

6. Поле **first** – додатне ціле число, номінал купюри; номінал може набувати значення 1, 2, 5, 10, 20, 50, 100. Поле **second** – додатне ціле число, кількість купюр цього номіналу. Реалізувати методи **sum(ob)** – обчислення грошової суми одного об'єкта, **summa(ob1,ob2)** – обчислення сумарної грошової суми двох об'єктів.
7. Поле **first** – додатне дійсне число, ціна товару; поле **second** – додатне ціле число, кількість одиниць товару. Реалізувати метод **cost(ob)** – обчислення вартості одного типу товару, **costTwo(ob1,ob2)** – обчислення сумарної вартості двох типів товару.
8. Поле **first** – додатне ціле число, калорійність 100 г. продукту; поле **second** – додатне дробове число, маса продукту в кілограмах. Реалізувати методи обчислення загальної калорійності одного продукту і **compareTwo(ob1,ob2)** – порівняння загальної калорійності двох продуктів.
9. Поле **first** – дробове число (чисельник, знаменник), ліва межа діапазону, поле **second** – дробове число (чисельник, знаменник) – права межа діапазону. Реалізувати метод **rangeCheck()** – перевірку заданого дробового числа на належність діапазону [**first**, **second**].
10. Поле **first** – ціле число, ліва межа діапазону, належить діапазону; поле **second** – ціле число, права межа діапазону, не належить діапазону. Пара чисел задає напіввідкритий інтервал [**first**, **second**). Реалізувати методи **rangeCheck()** – перевірку заданого числа на належність діапазону, **rangePrintUnion(ob1,ob2)** – виведення всіх цілих чисел об'єднання двох діапазонів.

11. Часовий проміжок задається полями: **first** – додатне ціле число, години; поле **second** – додатне ціле число, хвилини. Реалізувати методи **minutes()** – переведення часу у хвилини, **shiftForward()** і **shiftBack()** – збільшення зменшення часового проміжку на 20 хвилин.
12. Поле **first** – дійсне число; поле **second** – ціле число, показник степеня. Реалізувати метод **power()** – піднесення числа **first** до степеня **second**. Метод повинен працювати правильно для всіх допустимих (за означенням) значень **first** та **second**. Не використовувати функцію **pow()** з бібліотеки **math**.
13. Лінійне рівняння $Ax+B=0$. Поле **first** – дробове число (чисельник, знаменник), коефіцієнт **A**; поле **second** – дробове число, коефіцієнт **B** (чисельник, знаменник). Реалізувати метод **root()** – обчислення кореня лінійного рівняння. Метод повинен перевірити рівність коефіцієнта **A** нулеві.
14. Поле **first** – дійсне число, координата **x** точки на площині; поле **second** – дійсне число, координата **y** точки на площині. Реалізувати методи **distance()** – відстань від точки до початку координат, **distanceTwo(ob1,ob2)** – відстань між двома точками.
15. Поле **first** – додатне дійсне число, катет **a** прямокутного трикутника; поле **second** – додатне дійсне число, висота **h**, опущена на гіпотенузу прямокутного трикутника. Реалізувати методи **hypotenuse(ob1)** – обчислення гіпотенузи, **compare(ob1,ob2)** – порівняння площ двох об'єктів.
16. Поле **first** – додатне дробове число, оклад; поле **second** – додатне ціле число, кількість відпрацьованих днів у

місяці. Реалізувати метод **summa()** – обчислення нарахованої суми за дану кількість днів для заданого номера місяця:
оклад / кількість днів місяця * відпрацьовані дні.

17. Поле **first** – додатне ціле число, тривалість телефонної розмови у хвиликах; поле **second** – додатне дробове число, вартість однієї хвилини в гривнях. Реалізувати методи **cost()** – обчислення загальної вартості розмови, **compare(ob1,ob2)** – порівняння вартості двох розмов.
18. Поле **first** – ціла частина числа; поле **second** – додатне ціле число, дробова частина числа. Реалізувати метод **multiply()** – множення його на довільне дійсне число з двома знаками після коми. Результат множення – змінна визначеного структурного типу. Метод повинен працювати коректно для будь-яких допустимих значень **first** і **second**.
19. Поле **first** – додатне ціле число, координати курсору по горизонталі; поле **second** – додатне ціле число, координати курсора по вертикалі. Реалізувати метод **changex()** – зміни горизонтальної координати курсора; реалізувати метод **changeey()** – зміни вертикальної координати курсора, **changexy()** – зміни обох координат курсора. Методи повинні перевіряти вихід за межі екрана.
20. Поле **first** – додатне ціле число, ціла частина числа; поле **second** – додатне ціле число, дробова частина числа. Реалізувати метод **multiply()** – множення на довільне ціле число типу **int**. Результат множення – змінна визначеного структурного типу. Метод повинен працювати правильно для всіх допустимих значень **first** та **second**.
21. Пара різних натуральних чисел. Реалізувати методи **nsd(ob)** – найбільший спільний дільник чисел **first** і **second**, **nsk(ob)** – найменше спільне кратне їх же,

compareTwo(ob1,ob2) – порівняння на збіг найбільших спільних дільників двох об'єктів.

22. Елемент a_j спадної геометричної прогресії обчислюється за формулою

$$a_j = a_0 r^j, j = 0, 1, 2, \dots$$

Поле **first** – додатне дійсне число, перший елемент прогресії a_0 ; поле **second** – дійсне число r в межах від 0 до 1. Реалізувати методи **element()** – для обчислення j -го елемента прогресії, **sum()** – для обчислення суми перших j елементів прогресії, **sumAll()** – для обчислення суми елементів прогресії.

23. Поле **first** – додатне ціле число, рік; поле **second** – додатне ціле число від 1 до 12 – номер місяця. Реалізувати методи **daysCount(ob)** – для обчислення кількості днів заданого місяця (врахувати, що рік може бути високосним), **monthsCount(ob1,ob2)** – для обчислення кількості місяців між двома датами.

24. Поля **first** і **second** – дійсні числа, які визначають координати точки в полярній системі координат. Реалізувати методи **getX()** та **getY()**, які обчислюють координати в прямокутній системі координат по відповідних осях, **distanceTwo(ob1,ob2)** – відстань між двома точками.

25. Число з околom задається полями **first** і **second** (дійсні числа, які визначають відповідно число та його окіл). Реалізувати методи **isDiapazon(ob1,ob2)**, який перевіряє, чи попадає одне число з околom в окіл іншого числа з околom, **distanceTwo(ob1,ob2)** – мінімальну відстань між двома об'єктами (їх границями).

Лабораторна робота № 2

Тема: Перевантаження операцій і функцій.

Шаблони функцій

Завдання

Перевантажити функції для різних типів вхідних даних, або створити шаблони функцій, або перевизначити операції для означених користувачем типів (нові типи задавати у вигляді структур) [17].

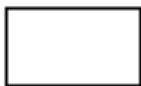
Варіанти завдань

1. Створити функцію, яка на вхід одержує текст (тип **char***) і перетворює його: відкидає повторні пробіли. Функція повинна повертати кількість відкинутих пробілів. Текст передавати у функцію по посиланню. Перевантажити операцію "<<" виведення перетвореного рядка, в якій викликати створену функцію.
2. Створити структуру, що визначає трикутник на площині. Перевизначити операції:
 - "++" – зсув трикутника на 10 пікселів вправо;
 - " – зсув трикутника на 10 пікселів вліво;
 - "<" – порівняння площ двох трикутників .
3. Створити структуру, що визначає вектор на площині. Перевизначити операції:
 - "!" – обчислення вектора, який отримується з даного поворотом на 90 градусів проти годинникової стрілки;
 - " – обчислення вектора, який протилежний даному і має з ним однакову довжину;
 - "++" – нормування вектора.
4. Створити функції, які інвертують масиви, що складаються з елементів типу **int**, **double**, а також – функцію, яка переставляє слова в реченні у зворотному порядку. Використати перевантаження функцій і створити шаблон.

5. Створити функції, які порівнюють цілі числа, рядки символів, норми векторів і повертають
 - 0, якщо дані рівні;
 - 1, якщо перше більше/довше другого;
 - 1, якщо перше менше/коротше другого.
 Використати перевантаження функцій.
6. Створити функції, які друкують квадратні корені з дійсного і комплексного числа, норму вектора. Використати перевантаження функцій.
7. Створити функції, які знаходять максимальний елемент масивів, що складаються з елементів типу `int`, `double`, а також – функцію, яка знаходить максимальну довжину слова в рядку (тип `char*`). Використати перевантаження функцій і створити шаблон функції.
8. Визначити операції “++” обчислення визначника матриці методом Гаусса, обчислення норми вектора.
9. Перевантажити операції “<<”, “>>” для введення матриць і виведення транспонованих до них матриць. Матриці прямокутні розмірністю `nxm` з цілими та дійсними елементами. Використати шаблони.
10. Визначити операцію логічного віднімання двох рядків символів `s3=s1-s2`. Рядок `s3` складається з символів, які належать рядку `s1` і не належать `s2`. Врахувати кількість однакових символів в обох рядках. Використати перевантаження операторів.
11. Визначити операції “+”, “-”, “*”, “/” для комплексного числа. Протестувати кожен оператор окремо. Обчислити вираз $z = (x+y)/(x-y) + xy/(x+y)$. Використати перевантаження операторів.

12. Визначити операції “+”, “-“, “*” для матриць ($n \times n$). Протестувати кожну операцію окремо, порівняти з результатами, отриманими у середовищі **MathCad**. Обчислити вираз $D=A*B+C*A-A*C$. Використати переваження операторів.
13. Визначити операції “+”, “-“, “*” для векторів розмірності n . Протестувати кожну операцію окремо. Обчислити вираз $d=(a+b)(a-b)$. Використати переваження операторів.
14. Визначити операцію логічного додавання двох рядків символів $S3=S1+S2$. Рядок $S3$ складається з символів, які належать хоча б одному рядку $S1$ або $S2$. Врахувати кількість однакових символів в обох рядках. Використати переваження операторів.
15. Визначити операцію логічного множення двох рядків символів $S3=S1*S2$. Рядок $S3$ складається з символів, які належать рядкам $S1$ і $S2$ одночасно. Врахувати кількість однакових символів в обох рядках. Використати переваження операторів.
16. Трійка чисел (x,y,r) визначає круг на площині. Визначити операції “+”, “-“, “*” для двох кругів:
 “+” – площа об’єднання;
 “*” – площа перетину;
 “-” – площа різниці.
 Використати переваження операторів.
17. Четвірка чисел $(x1,y1,x2,y2)$ визначає прямокутник на площині.

$(x1,y1)$



$(x2,y2)$

Визначити операції “+”, “-“, “*” для двох прямокутників:

“+” – площа об’єднання;

“*” – площа перетину;

“-” – площа різниці.

Використати перевантаження операторів.

18. Масив чисел **A(n)** містить коефіцієнти многочлена $a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$. Визначити операції “+”, “-“, “*” многочленів. Використати перевантаження операторів.
19. Масив чисел **A(n)** визначає многочлен $a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$. Визначити операцію “++” – обчислення значення многочлена в точці. Коефіцієнти зберігаються в пам’яті у вигляді списку. Використати перевантаження операторів.
20. Перевантажити операції “<<” і “>>” для введення і виведення комплексного числа, а також операцію “==” порівняння модулів двох комплексних чисел.
21. Перевантажити операції “<<” і “>>” для введення і виведення структури. В структурі зберігаються відомості про кількість пар у кожний день тижня в комп’ютерному класі.
22. Перевантажити операцію “*” для множення двох комплексних чисел, множення дійсного і комплексного числа, обчислення скалярного добутку двох векторів.
23. Для рядків символів перевантажити операції:
“-” – унарний мінус (перевертає рядок на своєму місці);
“*” – множення цілого числа **k** на рядок **s**.
Якщо **k>0**, то в результаті одержується рядок із **k** повторень рядка **s**. Якщо **k<0**, то в результаті одержується рядок із **k** повторень перевернутих рядків **s**.
24. Визначити операції:
a & **b** – векторний добуток векторів;

! **a** – нормування вектора.

Обчислити вирази **a & a**, **a & b**, ! **a**.

25. Перевантажити операції “<<” та “>>” для введення і виведення многочленів степеня **n** з дійсними коефіцієнтами.
26. Визначити операції “+”, “-“, “*”, “/” для дробів $\mathbb{P}/\mathbb{Q}$, де $\mathbb{P}$ – ціле, а $\mathbb{Q}$ – натуральне число. Протестувати всі операції окремо. Обчислити вираз $\mathbf{a/d-(a+b)/(cd)}$, **a**, **b**, **c**, **d** – дробки.
27. Визначити структуру, яка визначає прямокутник на площині. Визначити операції “<<”, “>>” для прямокутників:
“>>” – введення інформації про координати і назву прямокутника,
“<<” – виведення інформації про прямокутник (назва, координати, площа).
28. Масив чисел **A(n)** визначає многочлен з цілими коефіцієнтами $a_0+a_1x+a_2x^2+\dots+a_{n-1}x^{n-1}$. Визначити операції
! – обчислення похідної від многочлена, & – знаходження первісної многочлена.
29. Для множин, які зберігаються в пам’яті у вигляді бінарних векторів розмірності **n**, перевизначити операції:
“+” – об’єднання множин;
“*” – перетину множин;
“>>” – ввід множини.
Одиничний елемент бінарного вектора означає наявність елемента, що відповідає його індексу, у множині; нульовий – його відсутність.
30. Для множин, які зберігаються в пам’яті у вигляді бінарних векторів розмірності **n** (див. варіант 29), перевизначити операції:
“-” – різниці множин;

"!" – доповнення множин;

"<<" – вивід множини.

31. Створити функції, які знаходять довжину вектора на площині, довжину вектора у просторі, довжину речення (кількість слів). Використати перевантаження функцій.
32. Створити функції, які знаходять висоту трапеції, більшу з висот паралелограма, мінімальну з висот трикутника. Використати перевантаження функцій.
33. Визначити операції добутку "*" матриці розмірності $n \times n$ на вектор розмірності n та навпаки (вектора на матрицю), скалярний добуток векторів розмірності n .
34. Масиви чисел визначають многочлени з цілими коефіцієнтами $A(n) - a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$ та $B(m) - b_0 + b_1 x + b_2 x^2 + \dots + b_{m-1} x^{m-1}$. Визначити операції "/", "%" – ділення цих многочленів. Результат – два многочлени $P(x)$ та $Q(1)$, такі, що $A(n) = P(x) * B(m) + Q(1)$.
35. Для правильного многокутника на площині (n – кількість вершин, a – довжина сторони) перевизначити такі операції:
 - "++" – визначення радіуса описаного навколо многокутника кола;
 - " – визначення радіуса вписаного у многокутник кола;
 - "<" – порівняння площ двох многокутників.

Лабораторна робота № 3

Тема: Структури та класи

Завдання

У всіх завданнях описати класовий тип (спочатку – структуру, потім – клас з використанням структури). Окрім вказаних у завданні функцій, для об'єкта заданого класового типу повинні бути також реалізовані наступні функції [11]:

- ініціалізація об'єкта **init()**;
- введення з клавіатури **input()**;
- виведення на екран **output()**;
- перетворення об'єкта в рядок **toPChar()**.

Усі завдання повинні бути реалізовані двома способами:

- 1) тип даних задається структурою з необхідними полями, а операції реалізуються як зовнішні неоператорні функції, які отримують об'єкти даного типу як аргументи;
- 2) тип даних задається класом. Член-даних класу – одна приватна змінна структурного типу, який описаний у п.1, члени-функції класу – відкриті неоператорні функції **init()**, **input()**, **output()**, **toPChar()** і функції доступу **setStruct()**, **getStruct()** або **setVar()**, **getVar()**.

Варіанти завдань

1. Визначити тип: квадратична функція з коефіцієнтами **a**, **b**, **c** ($y=ax^2+bx+c$). Визначити функції:
 - **shift()** – зсуву параболи на одиницю вверх;
 - **odd()** – визначення парності функції;
 - **roots()** – знаходження дійсних коренів рівняння $ax^2+bx+c=0$;
 - друку координат вершини параболи $y=ax^2+bx+c$;
 - визначення кількості точок перетину з віссю OX.
2. Визначити тип: ромб, що задається довжиною сторони і гострим кутом. Визначити функції:
 - **compare()** – операцію порівняння двох об'єктів за радіусами вписаних кіл;
 - знаходження діагоналей ромба;
 - знаходження площі.

3. Визначити тип: лінійна функція з коефіцієнтами **b**, **c** ($y=bx+c$). Визначити функції:

- **shiftRight()** – зсуву прямої на одиницю праворуч;
- **shiftUp()** – зсуву прямої на одиницю вгору;
- визначення кута нахилу прямої до осі абсцис;
- визначення точки перетину двох прямих.

4. Визначити тип: **TCircle** – круг на площині. Круг визначається центром і радіусом. Визначити функції:

- **compare()** – порівняння на співпадання кругів при переносі;
- **isIn()** – визначення, чи покриває один круг другий;
- знаходження довжини дуги для заданого центрального кута;
- обчислення площі сегмента для заданого центрального кута.

5. Комплексне число задається парою дійсних чисел (**a**, **b**), де **a** – дійсна частина, **b** – уявна частина. Реалізувати тип **TComplex** для роботи з комплексними числами. Обов'язкова наявність функцій:

- додавання **add()**, $(a, b) + (c, d)$;
- віднімання **sub()**, $(a, b) - (c, d)$;
- множення **mul()**, $(a, b) * (c, d)$;
- ділення **div()**, $(a, b) / (c, d)$;
- порівняння **equ()**, $(a, b) = (c, d)$;
- спряжене число **conj(a, b)**.

6. Визначити тип **TVector3D**, який задається трійкою координат. Обов'язково повинні бути реалізовані: додавання й віднімання векторів, скалярне множення векторів, множення на скаляр, порівняння векторів, обчислення довжини вектора, порівняння довжин векторів.

7. Визначити тип **TModelWindow** для роботи з моделями екранних вікон. Як поля задаються: заголовок вікна, координати верхнього лівого кута, розмір по горизонталі, розмір по вертикалі, колір вікна, стан "видиме/невидиме", стан "з рамкою/без рамки". Координати і розміри вказуються в цілих числах. Реалізувати операції: переміщення вікна по горизонталі, по вертикалі; зміна висоти та/або ширини вікна, зміна кольору; зміна стану, опитування стану. Операції переміщення та зміни розміру повинні виконувати перевірку на перетинання границь екрана. Функція виведення на екран має демонструвати стан полів об'єкта.

8. Визначити тип **TMoney** для роботи з грошовими сумами. Число повинно бути представлене двома полями: типу **long** для гривень та типу **unsigned char** – для копійок. Дробова частина (копійки) при виводі на екран повинна бути відділена від цілої частини комою. Реалізувати додавання, віднімання, ділення сум, ділення суми на дробове число, множення суми на дробове число та операції порівняння.

9. Визначити тип **TTriangle** для представлення трикутника. Поля даних повинні включати кути та сторони. Необхідно реалізувати операції: отримання і зміни полів даних, обчислення площі, обчислення периметра, обчислення висот, а також визначення типу прямокутника (рівносторонній, рівнобедрений або прямокутний).

10. Визначити тип **TAngle** для роботи з кутами на площині, які задаються величиною в градусах та хвилинах. Обов'язково повинно бути реалізовано: переведення в радіани, приведення до діапазону 0-360, збільшення та зменшення кута на задану величину, отримання синусу, порівняння кутів.

11. Визначити тип **TPoint** для роботи з точками на площині. Координати точки – декартові. Обов'язково повинно бути реалізовано: переміщення точки по осі **X**, переміщення по осі **Y**, визначення відстані до початку координат, відстані між двома точками, перетворення в полярні координати, порівняння на збіг та неспівпадання двох точок.

12. Раціональний (нескоротний) дріб задається парою цілих чисел (**a**, **b**), де **a** – чисельник, **b** – знаменник. Реалізувати тип **TRational** для роботи з раціональними дробами. Обов'язково повинні бути реалізовані операції:

- додавання **add()**, $(a, b) + (c, d)$;
- віднімання **sub()**, $(a, b) - (c, d)$;
- множення **mul()**, $(a, b) * (c, d)$;
- ділення **div()**, $(a, b) / (c, d)$;
- порівняння **equ()**, **bigger()**, **smaller()**.

Повинна бути реалізована приватна функція скорочення дробу **reduce()**, яка обов'язково викликається при виконанні арифметичних операцій.

13. Створити тип **TDate** для роботи з датами у форматі "рік.місяць.день". Дата задається структурою з трьома полями типу **unsigned int**: для року, місяця і дня. Тип повинен включати не менше трьох функцій ініціалізації: числами, рядком вигляду "рік.місяць.день" (наприклад, "2013.08.28") та датою. Обов'язковими операціями є: обчислення дати через задану кількість днів, віднімання заданої кількості днів від дати, визначення високосності року, присвоєння та отримання окремих частин (рік, місяць, день), порівняння дат (рівно, до, після), обчислення кількості днів між датами.

14. Визначити тип **TTime** для роботи з часом у форматі "година:хвилина:секунда". Тип повинен включати в себе не менше чотирьох функцій ініціалізації: числами, рядком (наприклад, "23:59:59"), секундами та часом. Обов'язковими

операціями є: обчислення різниці між двома моментами часу в секундах, додавання часу й заданої кількості секунд, віднімання від часу заданої кількості секунд, порівняння моментів часу, переведення у секунди, переведення у хвилини (з округленням до цілої хвилини).

15. Реалізувати тип **TFuzzyNumber** для роботи з нечіткими числами, які представляються трійками чисел ($x - e_1, x, x + e_2$). Для чисел $Ob1 = (A - a_1, A, A + a_r)$ та $Ob2 = (B - b_1, B, B + b_r)$ арифметичні операції виконуються за такими формулами:

$$Ob1 + Ob2 = (A + B - a_1 - b_1, A + B, A + B + a_r + b_r);$$

$$Ob1 - Ob2 = (A - B - a_1 - b_1, A - B, A - B + a_r + b_r);$$

$$Ob1 * Ob2 = (A * B - B * a_1 - A * b_1 + a_1 * b_1, A * B, A * B + B * a_r + A * b_r + a_r * b_r);$$

$$1/Ob1 = (1/(A + a_r), 1/A, 1/(A - a_1)), A > 0;$$

$$Ob1/Ob2 = ((A - a_1)/(B + b_r), A/B, (A + a_r)/(B - b_1)), B > 0.$$

16. Реалізувати тип **TAccount**, що являє собою банківський рахунок. У типі повинні бути чотири поля: прізвище власника, номер рахунка, процент нарахування і сума в гривнях. Відкриття нового рахунка виконується операцією ініціалізації. Необхідно виконувати такі операції: змінити власника рахунка, зняти деяку суму з рахунка, покласти гроші на рахунок, нарахувати проценти, перевести суму в долари, перевести суму в євро, отримати суму прописом.

17. Номінали українських гривень можуть набувати значення 1, 2, 5, 10, 20, 50, 100, 200, 500. Копійки задати як 0.01 (1 копіяка), 0.02 (2 копійки), 0.05 (5 копійок), 0.1 (10 копійок), 0.25 (25 копійок), 0.5 (50 копійок). Створити тип **TMoney** для роботи з грошовими сумами. Суму слід представляти полями-номіналами, значеннями яких повинна бути кількість купюр даного номіналу. Реалізувати додавання сум, віднімання сум,

ділення двох сум, ділення суми на ціле число, множення на ціле число та операції порівняння. Дробову частину (копійки) при виведенні на екран треба відділити від цілої частини комою.

18. Реалізувати тип **TBankomat**, який моделює роботу банкомата. У типі повинні міститися поля для зберігання ідентифікаційного номера банкомата, інформації про поточну суму грошей, яка залишилася в банкоматі, мінімальну та максимальну суми, які дозволяється зняти клієнту в один день. Сума грошей подається полями номіналами 1-500 (див. завдання 13). Реалізувати метод ініціалізації банкомата, метод завантаження купюр у банкомат та метод зняття визначеної суми грошей. Метод зняття грошей має виконувати перевірку на коректність суми, яка знімається: вона не повинна бути меншою за мінімальне значення і більшою за максимальне значення. Метод **toString()** повинен перетворювати в рядок суму грошей, яка залишилася в банкоматі.

19. Створити тип **TFraction** для роботи з дробовими числами. Число повинно бути задано двома полями: ціла частина – довге ціле зі знаком, дробова частина – беззнакове коротке ціле. Реалізувати арифметичні операції додавання, віднімання, множення на ціле число та операції порівняння.

20. Створити тип **TGoods** (товар). У типі повинні бути присутні поля: назва товару, дата оформлення, ціна товару, кількість одиниць товару, номер накладної, за якою товар надійшов на склад. Реалізувати методи зміни ціни товару, зміни кількості товару (збільшення та зменшення), обчислення вартості товару. Метод **toString()** має видавати у вигляді рядка вартість товару.

21. Визначити тип **TBitString** для роботи з 64-бітовими рядками. Бітовий рядок повинен задаватися двома полями типу **unsigned long**. Мають бути реалізовані всі традиційні операції для роботи з бітами: **and**, **or**, **xor**, **not**. Реалізувати зсув вліво **shiftLeft()** та зсув вправо **shiftRight()** на задану кількість бітів.

22. Визначити тип **TLongLong** для роботи з цілими числами з 64 біт. Число повинно бути представлене двома полями: **long** – старша частина, **unsigned long** – молодша частина. Повинні бути реалізовані арифметичні операції, наявні в **C++** (без присвоювання), та порівняння.

23. Визначити тип: правильний многокутник на площині. Многокутник визначається кількістю сторін, центром описаного кола і одною вершиною на описаному колі. Визначити операції:

- порівняння двох многокутників за периметром;
- знаходження радіуса вписаного у многокутник кола;
- перевірка належності заданої точки вписаному у многокутник колу;
- знаходження площі многокутника.

24. Визначити тип: відрізок на площині. Відрізок задається двома точками – кінцями відрізка. Визначити операції:

- обчислення довжини відрізка;
- обчислення кута між двома відрізками;
- перевірки перетину двох відрізків;
- перевірки належності двох відрізків одній прямій.

25. Реалізувати тип **TCursor**. Полями є координати курсора по горизонталі та вертикалі – цілі додатні числа, вигляд курсора – горизонтальний чи вертикальний, розмір курсора – ціле від 1 до 15. Реалізувати методи зміни координат курсора, зміни розміру курсора, метод гасіння та відновлення курсора.

Лабораторна робота № 4

Тема: Створення класів

Завдання

У кожному варіанті розробити вказаний клас [17] і протестувати його, для чого в основній програмі створити необхідну кількість об'єктів, використовуючи які, продемонструвати всі розроблені оператори і функції. Частину операторів і функцій зробити дружніми, а частину – членами класу. Використати статичні члени класу.

Програму здавати у три етапи, розробивши функції у такій послідовності (другий і третій етапи можна міняти місцями):

- 1) конструктори, деструктор, оператор присвоєння, функції введення/виведення об'єктів;
- 2) неоператорні функції-члени (у тому числі – статичні) і функції-друзі класу;
- 3) перевантажені оператори.

Варіанти завдань

1. Визначити клас: трикутник на площині. Він визначається трьома вершинами $A(x_1, y_1)$, $B(x_2, y_2)$, $C(x_3, y_3)$. Створити конструктори й деструктор.

Визначити операції:

"<<" – друку інформації про об'єкт;
"==" , "!=" – порівняння двох об'єктів.

Визначити функції:

- знаходження площі і периметра трикутника;
- знаходження радіуса вписаного й описаного кіл;
- знаходження кутів при вершинах;
- знаходження висот і медіан;
- визначення типу трикутника (прямокутний, гострокутний, є тупий кут; рівнобедрений, рівносторонній);
- графічного зображення об'єкта на екрані;
- знаходження центрів описаного і вписаного кіл;

- повертання на заданий кут навколо центра описаного кола;
- зсуву об'єкта вправо, вліво.

2. Визначити клас многочленів степеня n з дійсними коефіцієнтами $P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$, $x \in \mathbf{R}$. Для збереження коефіцієнтів многочлена використати масив. Створити конструктори і деструктор.

Визначити операції:

"+", "-", "*" – додавання, віднімання і множення многочленів.

Визначити функції:

- друку многочлена;
- створення копії многочлена;
- знаходження значення многочлена в точці;
- знаходження частки і залишку від ділення многочлена на многочлен;
- обчислення многочленів $P'(x)$, $P(Q(x))$, $(x-b)P(x)$, $P(x+b)$ (b задається).

3. Визначити клас: довге ціле число. Для збереження числа використати масив (один елемент масиву – одна цифра числа). Створити конструктори й деструктор.

Визначити операції:

"+", "-", "*" – додавання, віднімання, множення чисел;
 "/" – цілочислове ділення;
 "%" – залишок від ділення;
 "==", "!=" , ">", "<", "<=", ">=" – порівняння.

Визначити функції:

- друку числа;
- логічну функцію, яка визначає, чи дорівнює число нулю.

Визначити, чи є число $2^{50}+1$ простим. Роздрукувати числа 2^{10} , 2^{100} , $100!$.

4. Створити клас: рядок символів. Рядок символів зберігається в

пам'яті за допомогою однонапрявленого зв'язного списку.

Створити конструктори й деструктор.

Визначити операції:

- "+" – зчеплення двох рядків;
- "==", "!=", "<", ">", "<=", ">=" – операції порівняння двох рядків;
- "<<" – операція друку рядка.

Визначити функції:

- копіювання рядка;
- знищення **N** символів рядка, починаючи з позиції **Poz**;
- вставка рядка **st1** у рядок **st2**, починаючи з позиції **Poz**;
- визначення довжини рядка;
- виявлення першого входження в рядок **st1** підрядка **st2**.
Результат – номер позиції.

5. Створити клас: множина точок на площині. $P(x, y)$, $x \in Z$, $y \in Z$.

Для збереження множини використати лінійний однонаправлений список. Створити конструктори й деструктор.

Визначити операції:

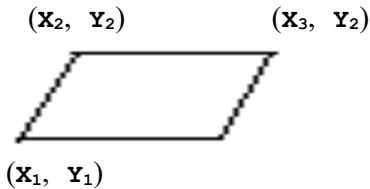
- "*", "+", "-" – перетин, об'єднання, різниця двох множин;
- "==", "!=", ">=", "<=" – операції відношення множин;
- "<<" – операція друку множини.

Визначити функції:

- створення копії множини;
- визначення кількості елементів у множині;
- визначення належності точки до множини;
- порівняння з порожньою множиною;
- впорядкування множини по відстанях від точок до початку координат;
- впорядкування множини по куту повороту відносно осі ОХ.

6. Визначити клас: прямий паралелограм на площині.

Паралелограм визначається числами ($X_1, Y_1, X_2, Y_2, X_3, Y_3$), де $X_i \in \mathbf{N}, Y_i \in \mathbf{N}$. Створити конструктор і деструктор.



Визначити операції:

"+" , "*" , "-" – площа об'єднання, перетину і різниці паралелограмів;

"==" – порівняння паралелограмів на збіг при переносі.

Визначити функції:

- друку інформації про паралелограм;
- знаходження площі та периметра об'єкта;
- графічного зображення об'єкта;
- знаходження гострого кута при вершині;
- зсуву об'єкта на 10 пікселів управо.

7. Створити клас: множина цілих чисел. Для опису множини використати бінарний вектор. Створити конструктори й деструктор.

Визначити операції:

"*" – перетин двох множин;

"+" – об'єднання множин;

"-" – різниця двох множин;

"~" – доповнення множини;

"%" – симетрична різниця множин;

"==", "!=" , ">=" , "<=" – операції відношення множин;

"<<" , ">>" – операції введення і виведення інформації про множини.

Визначити функції:

- створення копії множини;
- визначення кількості елементів у множині;
- визначення, чи належить число до множини;

- порівняння з порожньою множиною;
- знаходження мінімального елемента множини.

8. Визначити клас многочленів з комплексними коефіцієнтами $P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$, $x \in \mathbb{C}$, $a_k = u + iv$, $u \in \mathbb{Z}$, $v \in \mathbb{Z}$. Для збереження коефіцієнтів у пам'яті використати лінійний однонаправлений зв'язний список. Створити конструктори й деструктор.

Визначити операції:

- "+", "-", "*" – додавання, віднімання, множення многочленів;
 "==" – операція порівняння многочленів.

Визначити функції:

- друку многочлена;
- створення копії многочлена;
- обчислення значення многочлена в точці b ;
- друку многочлена $P(x + b)$ для заданого b .

9. Визначити клас многочленів довільного степеня з цілими коефіцієнтами $P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$. Для збереження коефіцієнтів у пам'яті використати лінійний однонаправлений зв'язний список. Створити конструктори й деструктор.

Визначити операції:

- "+", "-", "*" – додавання, віднімання, множення многочленів;
 "<<", ">>" – операції введення і виведення многочленів.

Визначити функції:

- перевірки рівності двох многочленів;
- створення копії многочлена;
- знаходження значення многочлена в точці;
- обчислення многочленів $P(x+b)$, $(x - b)P(x)$, $P'(x)$ для заданого b ;
- графічного зображення функції $y=P(x)$.

10. Визначити клас многочленів з раціональними коефіцієнтами

$P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$, $x \in \mathbb{Q}$, $a_k = u_k / v_k$, $u_k \in \mathbb{Z}$, $v_k \in \mathbb{N}$. Для збереження коефіцієнтів використати один двовимірний або два одновимірні масиви. Створити конструктори й деструктор.

Визначити операції:

- "+" – додавання двох многочленів;
- "-" – віднімання двох многочленів;
- "*" – множення двох многочленів;
- "<<" – друк многочлена;
- "==" – порівняння двох многочленів.

Визначити функції:

- створення копії многочлена;
- скорочення раціонального дробу;
- знаходження значення многочлена в точці;
- знаходження $P'(x)$, $P(x + b)$ (b – задане число).

11. Визначити клас: комплексне число. Створити конструктори й деструктор.

Визначити операції:

- "+", "-", "*", "/" – арифметичні операції;
- "=" – присвоєння;
- !" – отримання спряженого числа.

Створити функції:

- друку комплексного числа;
- створення копії;
- знаходження кореня n -го степеня з комплексного числа;
- знаходження $|z|$, $\text{Arg } z$, z^a (a – дійсне число);
- обчислення e^z , $\sin(z)$, $\cos(z)$, $\ln(1+z)$, $\arctg(z)$ з заданою точністю за допомогою рядів.

12. Створити клас матриць розмірності $n \times n$ з дійсними коефіцієнтами. Визначити конструктори й деструктор.

Визначити операції:

- "+", "-", "*" – для матриць;
- "<<" – друку матриці;
- ">>" – введення матриці;

"==" – порівняння матриць.

Визначити функції:

- знаходження визначника матриці;
- знаходження оберненої матриці;
- знаходження норми матриці;
- знаходження сліду матриці;
- функцію, яка визначає, чи є матриця симетричною.

13. Визначити клас векторів розмірності n . Створити конструктори й деструктор.

Визначити операції:

"+" , "-" – додавання і віднімання векторів;

"*" – скалярний добуток векторів;

">>" – введення вектора;

"^" – множення вектора на число.

Визначити функції:

- друку вектора;
- знаходження довжини вектора;
- створення копії вектора;
- сортування елементів вектора;
- нормування вектора.

14. Створити клас: рядок символів Рядок символів зберігається як у мові Паскаль. Для його збереження використовується масив символів. У нульовому елементі зберігається поточна довжина рядка. Створити конструктори й деструктор.

Визначити операції:

"+" – зчеплення двох рядків;

"%" – логічне додавання (додаються тільки ті символи з другого рядка, яких немає в першому рядку);

"==", "!=" , "<" , ">" , "<=" , ">=" – операції порівняння двох рядків.

Визначити функції:

- друку рядка;
- копіювання рядка;
- знищення N символів рядка, починаючи з позиції **Pos**;

- вставки рядка **st1** у рядок **st2**, починаючи з позиції **Poz**;
- визначення довжини рядка;
- виявлення першого входження в рядок **st1** підрядка **st2**.
Результат – номер позиції.

15. Визначити клас: рядок символів. Для збереження рядка використати масив, ознакою кінця рядка символів є символ `'\0'`. Створити конструктори й деструктор.

Визначити операції:

- "*" – логічне множення двох рядків (повертається рядок, в якому є тільки спільні символи);
- "+" – операція зчеплення двох рядків;
- "==", "!= ", "<", ">", "<=", ">=" – операції порівняння двох рядків.

Визначити функції:

- друку рядка;
- копіювання рядка;
- знищення **N** символів рядка, починаючи з позиції **Poz**;
- вставка рядка **st1** в рядок **st2**, починаючи з позиції **Poz**;
- визначення довжини рядка;
- виділення з рядка підрядка довжиною **N** символів, починаючи з позиції **Poz**;
- виявлення першого входження рядка в підрядок. (результат – номер позиції);
- перевірки, чи є рядок записом цілого додатного числа;
- перетворення рядка символів у ціле число, записом якого він є.

16. Визначити клас: круг на площині. Круг визначається трійкою чисел (**x**, **y**, **r**). Створити конструктори й деструктор.

Визначити операції:

- "+" – площа об'єднання двох кругів;
- "-" – площа різниці двох кругів;
- "*" – площа перетину двох кругів.

Визначити функції:

- друку інформації про коло;
- порівняння площ;
- знаходження площі круга;
- довжини дуги для заданого в градусах центрального кута;
- графічного зображення кола на екрані.

17. Визначити клас: прямокутник на площині. Прямокутник визначається координатами точки $A(x_1, y_1)$ – верхня ліва точка – і точки $B(x_2, y_2)$ – нижня права точка прямокутника ($x_1 \in \mathbb{N}$, $y_1 \in \mathbb{N}$, $x_2 \in \mathbb{N}$, $y_2 \in \mathbb{N}$). Створити конструктори й деструктор.

Визначити операції:

- "+" – площа об'єднання двох прямокутників;
- "-" – площа різниці двох прямокутників;
- "*" – площа перетину двох прямокутників.

Визначити функції:

- друку інформації про прямокутник;
- порівняння прямокутників за площами;
- порівняння прямокутників на збіг при переносі;
- знаходження площі та периметра прямокутника;
- графічного зображення прямокутника на екрані.

18. Визначити клас: довге ціле число. Для збереження числа використати лінійний однонапрямлений список. Створити конструктори й деструктор.

Визначити операції:

- "+", "-", "*" – додавання, віднімання, множення чисел;
- "/" – цілочислове ділення;
- "%" – залишок від ділення.

Визначити функції:

- друку числа;
- знаходження кількості цифр у числі;
- нормалізації числа (відкидання незначущих нулів);
- логічну функцію, яка визначає, чи дорівнює число нулю.

Визначити, чи є число $2^{100}+1$ простим.

19. Створити клас: правильний багатокутник. Об'єкт визначається через 5 цілих чисел x_1, y_1, R, N . Тут $O(x_1, y_1)$ – центр описаного кола, R – радіус описаного кола, N – кількість сторін, $A(x_1-R, y_1)$ – координати першої вершини. Створити конструктори й деструктор.

Визначити операцію:

“==”, “!=” – порівняння двох об'єктів;

“<<” – друк інформації про об'єкт.

Визначити функції:

- знаходження площі й периметра об'єкта;
- знаходження радіуса вписаного кола;
- знаходження кута при вершині;
- графічного зображення об'єкта на екрані;
- зсуву зображення на 20 пікселів уліво.

20. Визначити клас: площина у просторі. Площина задається загальним рівнянням A, B, C, D – коефіцієнти. Створити конструктори й деструктор, генератор копій.

Визначити операції:

“^” – кут між площинами;

“||” – відстань між площинами;

“==”, “!=” – порівняння двох площин.

Визначити функції:

- друку інформації про площину;
- перевірки на належність точки до площини;
- визначення коефіцієнтів нормального рівняння площини;
- перевірки на паралельність площини координатним площинам;
- перевірки паралельності й перпендикулярності двох площин;
- знаходження відстані між точкою і площиною.

21. Визначити клас: ламана на площині. Ламана задається координатами точок, які зберігаються в двовимірному або одновимірному масиві. Створити конструктори й

деструктор.

Визначити операції:

"+" – склеювання двох ламаних;

"<<", ">>" – виведення і введення ламаної (кількість вершин та їх координати).

Визначити функції:

- визначення, чи ламана замкнута;
- визначення, чи є самоперетин ламаної;
- визначення, чи є замкнута ламана опуклою;
- знаходження площі фігури, обмеженою опуклою ламаною;
- графічного зображення ламаної.

Для спрощення алгоритму всі вершини, крім, можливо, першої та останньої, вважати різними.

22. Визначити клас: пряма на площині. Пряма задається загальним рівнянням, **A**, **B**, **C** – коефіцієнти. Створити конструктори й деструктор, генератор копій.

Визначити операції:

"^" – кут між прямими;

"||" – відстань між прямими;

"==", "!=" – порівняння двох прямих.

Визначити функції:

- друку інформації про пряму;
- знаходження перетину двох прямих;
- перевірки на належність точки до прямої;
- визначення коефіцієнтів нормального рівняння прямої;
- перевірки на паралельність прямої координатним осям;
- перевірки паралельності й перпендикулярності двох прямих;
- знаходження відстані між точкою та прямою.

23. Визначити клас: точка на площині. Створити конструктори й деструктор, генератор копій.

Визначити операції:

"++", "--" – зсув точки вправо/вліво на 10 пікселів;

"||" – відстань між точками;

"<<", ">>" – виведення і введення точок.

"==", "!=" – порівняння двох точок.

Визначити функції:

- знаходження полярних координат точки;
- графічного зображення точки на координатній площині;
- визначення, в якій чверті координатної площини лежить точка;
- визначення, чи утворюють 4 дані точки коло;
- визначення належності 3 даних точок одній прямій.

24. Визначити клас: ламана у просторі. Ламана задається координатами точок, які зберігаються в масиві розміром **3×n**. Створити конструктори й деструктор, генератор копій.

Визначити операції:

"+" – склеювання двох ламаних;

"=" – присвоєння.

Визначити функції:

- друку інформації про ламану (кількість вершин та їх координати);
- визначення, чи ламана замкнута;
- визначення, чи належить вона одній площині;
- знаходження проекцій ламаної на координатні площини;
- графічного зображення проекції ламаної на **ХОУ**.

25. Визначити клас: точка у просторі. Створити конструктори, деструктор, генератор копій.

Визначити операції:

"==", "!=" – порівняння точок;

"||" – відстань між точками;

"=" – присвоєння;

"+", "-" – додавання, віднімання точок.

Створити функції:

- друку інформації про точку;
- знаходження сферичних координат;
- визначення належності точки координатним площинам;
- визначення рівняння прямої, що проходить через дві задані точки;

- визначення рівняння площини, що проходить через 3 задані точки.

26. Визначити клас: ціле число. Створити конструктор і деструктор.

Визначити операції:

- "++" - сума цифр числа;
- " – знищення останніх цифр числа, якщо вони дорівнюють нулю;
- !" - обертання числа (цифри – в оберненому порядку);
- "&" – ділення чисел (результат – дійсне число).

Створити функції:

- перевірки, чи є число простим;
- знаходження простих дільників числа;
- розклад числа на прості множники;
- знаходження найменшого спільного кратного двох чисел;
- друку числа в різних системах числення (від двійкової до десяткової).

27. Визначити клас: раціональне число $x=p/q$, де p – ціле, а q – натуральне числа. Створити конструктори й деструктор.

Визначити операції:

- "+", "-", "*", "/" – додавання, віднімання, множення, ділення дробів;
- "==", "!=" , ">" , "<" , "<=" , ">=" – порівняння;
- "<<" – виведення та ">>" – введення .

Визначити функції:

- піднесення до степеня: x^a , a – натуральне число;
- скорочення раціонального дробу;
- копії об'єкта.

28. Створити клас: куля у просторі. Куля задається координатами центру й радіусом. Створити конструктори, деструктор і генератор копій.

Визначити операції:

- "==" – порівняння радіусів
- "<<" – друк інформації про кулю;

- "<=" – визначення належності однієї кулі іншій;
- ">=" – визначення включення однієї кулі іншою;
- "*" – знаходження об'єму перетину куль.

Визначити функції:

- зсуву кулі на заданий вектор;
- стискання кулі в задану кількість разів;
- друку інформації про перетин з іншою кулею (одна точка та її координати, безліч точок перетину).

29. Створити клас: вектор на площині. Створити конструктори й деструктор, генератор копій.

Визначити операції:

- "+" – сума векторів;
- "-" – різниця векторів;
- "*" – скалярний добуток;
- "=" – присвоєння;
- "<" – кут між векторами.

Визначити функції:

- визначення, чи вектори колінеарні;
- визначення, чи вектори ортогональні;
- друку вектора;
- графічного зображення вектора;
- визначення довжини вектора.

30. Створити клас: вектор у просторі. Створити конструктори й деструктор, генератор копій.

Визначити операції:

- "&" – кут між векторами;
- "&&" – векторний добуток векторів;
- "*" – скалярний добуток;
- "=" – присвоєння;
- "<<" – друк вектора.

Визначити функції:

- визначення, чи два вектори колінеарні;
- визначення, чи три вектори компланарні;
- визначення мішаного добутку векторів;

- графічного зображення проекцій вектора на кожную координатну площину;
- визначення довжини вектора.

31. Створити клас: еліпс. Еліпс задається коефіцієнтами **a**, **b** канонічного рівняння. Створити конструктори, деструктор.

Визначити операції:

- "=" – присвоєння;
- "==" – порівняння;
- "<<" – друк інформації про еліпс;
- "<=", ">=" – включення одного еліпсу в інший та навпаки.

Визначити функції:

- визначення координат фокусів;
- визначення ексцентриситету;
- визначення фокального параметра;
- визначення рівнянь директрис;
- графічного зображення еліпса.

32. Створити клас: циліндр, нижня основа якого паралельна координатній площині **хоу**. Циліндр задається координатами центру нижньої основи, радіусом основи й висотою. Створити конструктори, деструктор.

Визначити операції:

- "=" – присвоєння;
- "==" – порівняння циліндрів за об'ємами;
- "<<" – друк інформації про циліндр;
- "<=" – визначення належності одного циліндру іншому;
- ">=" – визначення включення одним циліндром іншого.

Визначити функції:

- зсуву циліндра на заданий вектор;
- стискання розмірів циліндру в задану кількість разів;
- друку інформації про перетин з іншим циліндром (так, ні).

33. Створити клас: автомобіль. Автомобіль такими задається такими даними: номер, марка, ціна, колір, прізвище та ім'я власника, рік випуску, відмітка проходження мита. Створити конструктори, деструктор.

Визначити операції:

- "=" – присвоєння;
- "<", ">" – порівняння автомобілів за ціною;
- "<=", ">=" – порівняння автомобілів за роком випуску;
- "<<" – друк інформації про автомобіль;
- "+" – сумарна ціна автомобілів.

Визначити функції:

- розмитнення авто в залежності від року випуску;
- зміна кольору та власника;
- визначення, чи автомобіль розмитнений;
- визначення, чи номер автомобіля щасливий (є симетричним);
- визначення, скільки років автомобілю.

34. Створити клас: таймер. Таймер задається проміжком часу, через який буде викликана функція, та часом початку і кінця роботи. Створити конструктори, деструктор.

Визначити операції:

- "=" – присвоєння;
- "==" – порівняння двох таймерів;
- "<=", ">=" – порівняння проміжків часу двох таймерів;
- "<", ">" – порівняння часу початку роботи двох таймерів;
- "<<" – друк інформації про таймер;
- "-" – визначення різниці в часі між двома таймерами.

Визначити функції:

- графічного зображення зміни часу у вигляді відрізка, довжина якого збільшується з часом (функція повинна викликатися через заданий таймером проміжок часу);
- зміни проміжку часу;
- визначення кількості викликів таймером заданої функції;
- створення копії таймера;
- зміни часу початку роботи таймера.

Лабораторна робота № 5

**Тема: Створення класів з використанням існуючих класів.
Успадкування. Агрегація**

Завдання

Використовуючи розроблений у лабораторній роботі № 4 клас, утворити новий клас [17] одним із двох способів:

- утворити новий клас, в якому **успадкувати старий клас**, розширивши його можливості;
- утворити новий клас, використавши в ньому члени-дані – **вказівники на тип старого класу (агрегація)**.

У новому класі обов'язково додати новий член-даних, наприклад, колір об'єкта чи його ім'я. Визначити конструктори, реалізувавши їх через виклик конструкторів класу, розробленого у лабораторній роботі № 4. Інші нові функції повинні максимально використовувати можливості написаного раніше коду.

Можливі нові оператори – унарні постфіксний і префіксний декременти та інкременти, оператори індексування, виклику, перетворення типу.

Можливі нові функції-члени – графічне зображення об'єктів.

Можливі перевизначені функції – введення/виведення (враховуючи нові члени-дані).

Проект розбити на кілька файлів, які підключати за допомогою директив препроцесора.

Конкретну умову задачі скласти самостійно, узгодити її з викладачем і розмістити в електронному робочому зошиті.

Нижче наведено приклади умов задач для варіанта 1 (успадкування) і для варіанта 2 (агрегація), які відповідають першим двом умовам лабораторної роботи № 4.

Варіанти завдань

1. Створити клас: пряма призма, основи якої паралельні координатній площині **ХОУ**. Для цього успадкувати клас "трикутник на площині". Доповнити члени-дані успадкованого класу висотою і відстанню від нижньої основи до площини **ХОУ**. Створити конструктори, деструктор.

Визначити операції:

- "=" – присвоєння;
- "==" – порівняння призм за об'ємами;
- "!=" – порівняння призм на збіг при паралельному переносі;
- "<<" – друк інформації про призму (координати всіх вершин).

Визначити функції:

- зсуву призми на заданий вектор;
 - знаходження об'єму описаного навколо призми циліндра;
 - знаходження суми довжин усіх ребер призми;
 - знаходження площі поверхні призми.
2. Визначити клас: раціональна функція. Члени-дані – чисельник і знаменник, – це вказівники на об'єкти класу "многочлен степеня n з дійсними коефіцієнтами". Створити конструктори й деструктор.

Визначити операції:

- "+", "-", "*" – додавання, віднімання та множення раціональних функцій;
- "<<" – друк раціональної функції.

Визначити функції:

- піднесення до цілого степеня раціональної функції;
- створення копії раціональної функції;
- знаходження значення раціональної функції в точці;
- знаходження похідної раціональної функції.

Лабораторна робота № 6

Тема: Абстрактні класи

Завдання

Побудувати ієрархію класів [11]. Корінь ієрархії – абстрактний клас з вказаними в умові задачі віртуальними функціями. Утворити похідні реальні класи, в яких здійснити реалізацію всіх чисто віртуальних функцій пращура. Написати функції введення/виведення. Самостійно визначити, які членидані необхідні та які з них визначити в базовому класі, а які – в похідних. Самостійно визначити будь-яку поліморфну функцію для маніпулювання різнотиповими об'єктами створеної ієрархії.

Варіанти завдань

1. Створити абстрактний базовий клас **TPair** з віртуальними арифметичними операціями $+=$, $-=$, $*=$. Створити похідні класи **TComplex** (комплексне число) і **TRational** (раціональний дріб) з власною реалізацією арифметичних операцій.
2. Створити базовий клас – геометрична фігура із заданою основою (круг радіуса **r**) і висотою **h** та віртуальними членами-функціями обчислення площі поверхні фігури й об'єму. Створити похідні класи – **TCylinder**, **TCone**, **TTruncatedCone** (циліндр, конус, усічений конус) з власною реалізацією віртуальних функцій.
3. Створити абстрактний базовий клас **TFigure** з віртуальними членами-функціями обчислення площі фігури й довжини її контура. Створити похідні класи: **TRectangle** (прямокутник), **TCircle** (коло), **TTrapezium** (трапеція) зі своїми функціями для обчислення площі фігури й довжини контура.
4. Створити базовий клас – працівник із заданою ставкою і функцію нарахування зарплати. Створити похідні класи – службовець з погодинною ставкою, службовець у штаті з

фіксованою ставкою і службовець з відсотковою ставкою з власною реалізацією функції нарахування зарплати за введену кількість восьмигодинних робочих днів.

5. Створити абстрактний базовий клас **TBody** (тіло) з віртуальними функціями обчислення площі поверхні й об'єму. Створити похідні класи **TParalelepiped** (паралелепіпед) і **TPiramida** (правильна n -кутна піраміда) зі своїми функціями площі поверхні та об'єму.

6. Створити абстрактний базовий клас **TCurrency** (валюта) для роботи з грошовими сумами. Визначити віртуальні функції переведення суми в гривні й виведення на екран. Створити похідні класи **TDollar** (долар) і **TEuro** (євро) зі своїми функціями переведення в гривні та друку.

7. Створити абстрактний базовий клас **TTriangle** для опису трикутника з віртуальними функціями обчислення площі й периметра. Як параметри класу використати дві сторони трикутника і кут між ними. Створити похідні класи – прямокутний трикутник, рівнобедрений трикутник і рівносторонній трикутник зі своїми функціями обчислення площі й периметра.

8. Створити абстрактний базовий клас **TRoot** (корінь) з віртуальними функціями обчислення коренів і друку результату. Визначити похідні класи **TLinear** (лінійне рівняння) і **TSquare** (квадратне рівняння) з власними функціями обчислення коренів і друку рівнянь.

9. Створити абстрактний базовий клас **TFunction** (функція) з віртуальними членами-функціями обчислення значення функції в заданій точці x і виведення результату на екран. Визначити похідні класи **TEllipse** (еліпс) і **THyperbola** (гіпербола) з власними функціями обчислення значення та друку рівнянь кривих.

10. Створити абстрактний базовий клас **TTriad** з віртуальними функціями збільшення на 1 кожного з трьох членів-даних і друку значення. Створити похідні класи **TDate** (поточна дата), **TTime** (поточний час) з власною реалізацією вищенаведених функцій.

11. Створити абстрактний клас **TSorting** (сортування) з ідентифікатором послідовності, віртуальними членами-функціями сортування, одержання суми та виведення на екран. На його основі реалізувати класи **TChoice** (метод вибору) і **TQuick** (швидке сортування).

12. Створити базовий клас **TList** список з віртуальними функціями додавання **push()** у список та видалення зі списку **pop()**. Реалізувати на базі списку стек і чергу з власною реалізацією вищенаведених функцій.

13. Створити абстрактний базовий клас **TInteger** (ціле число, представлене масивом своїх цифр) з віртуальними арифметичними операціями $+=$, $-=$, $*=$ і функцією виведення на екран. Створити похідні класи **THeximal** (шіснадцяткове число), **TBinary** (двійкове число) і перервантажити для них вищевказані функції.

14. Створити абстрактний клас **TLine** (відрізок). На основі класу **TLine** створити клас **TColoredLine** і класи **TPolyLine** та **TColoredPolyLine** (правильні многокутники із заданою кількістю вершин). Класи повинні мати віртуальні методи установки координат відрізка й отримання значень усіх координат фігури, а також зміни кольору та отримання поточного кольору.

15. Створити абстрактний базовий клас **TSeries** (прогресія) з віртуальними функціями обчислення вказаного члена прогресії та її суми. Визначити похідні класи **TLinear** (арифметична

прогресія) та **TExponential** (геометрична прогресія) з власною реалізацією вищенаведених функцій.

16. Створити абстрактний базовий клас **TNorm** з віртуальними функціями нормування й обчислення довжини вектора. Визначити похідні класи **TVector2D** (вектор на площині) і **TVector3D** (вектор у просторі) з власною реалізацією вищенаведених функцій. Для **TVector2D** норму обчислювати як корінь з суми квадратів, для **TVector3D** – як максимальне з модулів компонентів вектора.

17. Створити абстрактний базовий клас **TContainer** з віртуальними членами-функціями **sort()** і **norma()**. Створити похідні класи **TBubble** і **TChoice**. Для **TBubble** метод **sort()** має реалізовувати бульбашкове сортування, а **norma()** – обчислювати квадратний корінь із суми елементів. Для **TChoice** метод **sort()** має реалізовувати сортування методом вибору максимального елемента, а **norma()** – обчислювати середнє арифметичне всіх елементів.

18. Створити абстрактний базовий клас **TArray** з віртуальними членами-функціями знаходження суми елементів і поелементної обробки елементів масиву **foreach()**. Створити похідні класи **TUnionArray** і **XorArray**. Для **TUnionArray** знаходження суми елементів реалізується як побітове об'єднання, а **foreach()** – інвертування елементів масиву (зміна знака кожного елемента на протилежний). Для **XorArray** знаходження суми елементів реалізується як побітове виключаюче АБО для елементів масивів, а **foreach()** виконує нормування елементів масиву.

19. Створити абстрактний базовий клас базовий клас **TArray** (масив з різних цілих чисел) з віртуальними членами-функціями знаходження суми елементів і поелементної обробки елементів масиву **foreach()**. Створити похідні класи **TAndArray** і **TOrArray**. Для **TAndArray** знаходження суми елементів

реалізується як перетин множин, а **foreach()** – нормування елементів масиву. Для **TOrArray** знаходження суми елементів реалізується як об'єднання множин, а **foreach()** виконує швидке сортування.

20. Створити абстрактний клас **TFunction** (крива, що визначається коефіцієнтами **a**, **b**, **c**) із віртуальними членами-функціями обчислення значення функції **y(x)**, в точці **x0**, перевірки належності заданої точки (**x0**, **y0**) кривій і малювання кривої. Створити похідні класи **TSquareFunction** (функція **y=ax²+bx+c**) і **TLinearFunction** (функція **ax+by+c=0**) з власними реалізаціями віртуальних функцій.

21. Створити абстрактний клас **TMatrix** (квадратна матриця четвертого порядку) з віртуальними членами-функціями сортування **sort()** рядків/стовпців матриці і **norm()** – знаходження норми матриці. Створити похідні класи **TRowMatrix** (функцію сортування рядків **sort()** реалізувати за зростанням мінімального елемента рядка, функцію **norm()** реалізувати, використовуючи норму матриці, яка визначається квадратним коренем із суми квадратів її елементів) і **TColumnMatrix** (функцію сортування стовпців **sort()** реалізувати за зростанням максимального елемента стовпчика, функцію **norm()** реалізувати, використовуючи норму матриці, яка визначається максимальним модулем її елементів).

22. Створити абстрактний клас **TShape** із членами-даними дійсне число **r** і масив точок. Створити похідні класи **TCircle** (з функціями **count()** визначення кількості точок масиву, які належать колу радіусом **r** з центром у початку координат і **sort()** сортування масиву за відстанню до початку координат) і **TSquare** (з функціями **count()** визначення кількості точок масиву, які належать квадрату зі стороною **r** з центром у початку і сторонами, паралельними координатним осям, і **sort()** – сортування масиву точок за кутом повороту відносно

додатного напрямку осі абсцис) з власними реалізаціями віртуальних функцій.

23. Створити абстрактний клас **TMatrix** (квадратна матриця п'ятого порядку) з віртуальними членами-функціями **count()** – обчислення кількості рядків/стовпців матриці і **remake()** – перетворення матриці. Створити похідні класи **TRowMatrix** (функція **count()** повертає кількість рядків, які містять зростаючу послідовність елементів, функція **remake()** занулює всі від'ємні елементи матриці) і **TColumnMatrix** (функція **count** рядків повертає кількість стовпців, які містять тільки парні елементи, функція **remake()** змінює знаки елементів матриці на протилежні) з власними реалізаціями віртуальних функцій.

24. Створити абстрактний базовий клас **Row** (рядок символів) з віртуальними арифметичною операцією **+=** і функцією **change()**. Створити похідні класи **TNameRow** і **TColorRow** з власними реалізаціями віртуальних операцій і функцій. У класі **TNameRow** додати член-даний ім'я рядка (операція **+=** формує конкатенацію рядків з іменем першого рядка, а функція **change()** змінює ім'я рядка на задане). У класі **TColorRow** додати член-даний колір рядка (операція **+=** знаходить конкатенацію перегорнутих рядків з кольором першого рядка, а функція **change()** – змінює колір рядка).

25. Створити абстрактний базовий клас **TFunction** з віртуальними функціями **equation()** і **draw()**, клас-нащадок **TPolinom** (многочлен степеня **n** з коефіцієнтами, заданими у масиві) і похідні від нього класи **TPohidna** і **TPervisna** (похідна і первісна) з власними реалізаціями віртуальних функцій. У класах **TFunction**, **TPohidna** і **TPervisna** функція **equation()** табулює відповідно многочлен, його похідну і первісну на заданій рівномірній сітці. Функція **draw()** у кожному класі малює графік відповідного об'єкта на цій сітці.

СПИСОК ЛИТЕРАТУРЫ

1. Айра П. Объектно-ориентированное программирование на C++, 2-е издание / П. Айра. – М. : Бином, 2001. – 462 с.
3. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч. – М. : Бином, 2001. – 560 с.
4. Грегори К. Использование Visual C++ 6. Специальное издание / К. Грегори. – М.; СПб. К. : Вильямс, 1999. – 864 с.
5. Дерк Л. Си и C++ [справочник] / Л. Дерк. – М. : Бином, 1997. – 588 с.
6. Джефф Э. C++ : библиотека программиста / Э. Джефф. – М.; СПб : Питер, 2002. – 320 с.
7. Дьюхарст С. Программирование на C++ / С. Дьюхарст, К. Старк. – К. : ДиаСофт, 1993. – 271 с.
8. Вандевурд Д. Шаблоны C++: справочник разработчика = C++ Templates: The Complete Guide / Д. Вандевурд, М. Николай. – М. : Вильямс, 2003. – 544 с.
9. Лаптев В. В. Объектно-ориентированное программирование. Задачи и упражнения / В. В. Лаптев, А. В. Морозов, А. В. Бокова. – СПб. : Питер, 2007. – 288 с.
10. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению наших программ и проектов / С. Мейерс. – М. : ДМК Пресс, 2000. – 304 с.
11. Павловская Т. А. Программирование на языке высокого уровня / Т.А. Павловская. – СПб. : Питер, 2003. – 461 с.
12. Павловская Т. А. C++. Объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2005. – 265 с.
13. Рассохин Д. От Си к Си++ / Д. Рассохин. – М. : Эдель, 1993. – 128 с.
14. Ритчи Д. Язык программирования СИ / Д. Ритчи, Б. Керниган. – М. : Финансы и статистика, 1992. – 294 с.
15. Саттер Г. Решение сложных задач на C++. Серия C++ In-

- Depth M.; СПб. / Г. Саттер. – К. : Вильямс, 2002. – 400 с.
16. Сопронюк Т. М. Технології візуального й узагальненого програмування в C++Builder : навчальний посібник / Т. М. Сопронюк. – Чернівці : ЧНУ, 2009. – 80 с.
17. Сопронюк Т. М. Об'єкто-зорієнтоване програмування на C++ : Методичні рекомендації та завдання для лабораторних робіт / Т. М. Сопронюк, С. М. Тимку. – Чернівці : ЧНУ, 2004. – 51 с.
18. Страуструп Б. Дизайн и эволюция C++/ Б. Страуструп. пер. с англ. – М. : ДМК Пресс; СПб. : Питер, 2006. – 448 с.
19. Страуструп Б. Язык программирования C++ / Б. Страуструп. – К. : ДиаСофт, 1993. – 256 с.
20. Уолтер С. C++. Курс объектно-ориентированного программирования / С. Уолтер. – М.; СПб. ; К. : Вильямс, 2001. – 704 с.
21. Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.5. / Т. Фейсон. – К. : Диалектика, 1996. – 544 с.
22. Войтенко В.В., Морозов А.В. C/C++: Теория та практика: [Електрон. ресурс]. – Режим доступу: http://tanet.tneu.org/phocadownloadpap/osnovy_progrsmuvannja_C_C++.pdf
23. Підручники з C++: [Електрон. ресурс]. – Режим доступу: <http://bookwebmaster.narod.ru/cplusplus.html>
24. Список книг по C++ : [Електрон. ресурс]. – Режим доступа: <http://www.intuit.ru/departament/pl/ccpp/print.lit.html>
25. Павловская Т. А. C/C++. Программирование на языке высокого уровня. 2003: [Электрон. ресурс]. – Режим доступа: http://bulletinsite.net/index.php?id1=6&category=programmer&auth_or=pavlovskaya-ta&book=2003
26. Страуструп Б. Введение в язык C++. 1995: [Электрон. ресурс]. – Режим доступа: <http://khpi-iip.mipk.kharkiv.edu/library/pgm/cpp/index.html>
27. Online підручник по Builder C++: [Електрон. ресурс]. – Режим доступу: <http://articles.org.ru/docum/builder/3.php>
28. Инкапсуляция, полиморфизм, наследование: [Электрон. ресурс]. – Режим доступа: <http://www.codenet.ru/progr/cpp/ipn.php>

Навчальне видання

Сопронюк Тетяна Миколаївна

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА C++

Навчальний посібник

Відповідальний за випуск **Я. Й. Бізун**

Літературний редактор **О. В. Колодій**

Комп'ютерний набір

і програмування **Т. М. Сопронюк, А. Ю. Сопронюк**

SharpMinds is proud to support this beautiful and useful book to prepare yourself to become the SharpMind from tomorrow. SharpMinds is very interested to start discussions with you how to use your skills in the future in IT business. We have great apprenticeship places, provide IT and English lessons and have challenging projects. We invite you to experience the open and pleasant atmosphere in our office. Just send your CV to hrm@sharpminds.com and we invite you for an open discussion and a good cup of coffee.

Success with your study!

Edwin Zuijdendorp

hrm@sharpminds.nl

Director SharpMinds



Sharp
Minds

Формат. 60x84/16. Папір офсетний. Друк різнографічний.
Умов. друк. арк. 10,23. Обл.-вид. арк. 9,5. Зам. №6. Тираж 300 прим.
Видавництво Чернівецького національного університету.
58012, Чернівці, вул. Коцюбинського, 2
Свідцтво суб'єкта видавничої справи ДК № 891 від 08.04.2002.
Виготовник: ПБКФ «Технодрук»
Свідцтво суб'єкта видавничої справи ДК №1841 від 10.06.2004 р.
58000, м. Чернівці, вул. І. Франка, 20, оф.18, тел. (0372) 55-05-85