МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
Національний авіаційний університет

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Лабораторний практикум

для студентів
напряму 0915 "Комп'ютерна інженерія"
спеціальності 6.091500 "Системне програмування"

Київ 2011

УДК 004.415(076.5)
ББК З 973.20 – 018.2 я7
О 13

Укладачі: О.В. Коба, С.В. Пустова

Рецензенти:
Ю.К. Зіатдінов, М.Ю. Кузнєцов, Л.А. Журавльова

Затверджено науково методично-редакційною комісією факультету комп'ютерних систем
(протокол № _____ від _____ р.).

О13 Об'єктно-орієнтоване програмування: лабораторний практикум. Уклад.: О.В. Коба, С.В. Пустова. – К.: НАУ, 2011. – 76 с.

Лабораторний практикум містить завдання до лабораторних робіт, методичні вказівки до їх виконання, Контрольні завдання та запитання, вимоги до звітів, а також приклади, які супроводжуються теоретичними відомостями.

Призначено для студентів факультету комп'ютерних систем напряму 0915 "Комп'ютерна інженерія" спеціальності 6.091500 "Системне програмування".

ВСТУП

Об'єктно-орієнтоване програмування (ООП) — один із сучасних підходів до створення програм. На ньому ґрунтуються такі сучасні мови програмування як C++, Java, PHP, JavaScript, ObjectPascal тощо. Тому вивчення основ ООП є важливим для системного програміста.

Лабораторний практикум містить вступ, загальні методичні вказівки до виконання лабораторних робіт, мету та завдання до лабораторних робіт, основні теоретичні відомості, контрольні завдання та запитання.

Мета даного лабораторного практикуму — навчити студента самостійно створювати грамотні і по можливості професійні програми мовою C++.

Першу частину лабораторних робіт з курсу „Об'єктно-орієнтоване програмування" присвячено вивченню класів і об'єктів, масивів об'єктів, указівників на об'єкти, перевантаження функцій, перевантаження унарних і бінарних операторів, одиночного і множинного успадкування; другу частину —віртуальним функціям, родовим функціям, родовим класам, винятковим ситуаціям, системі введення-виведення.

Основи і базові концепції ООП, які вивчаються у лабораторному практикумі, дозволять студентам створювати такі програми, які будуть зручними для супроводження, модифікації та рефакторінгу програмного коду, зрозуміти, як саме використати ті чи інші принципи ООП, з яких компонентів – модулів, функцій, класів — складати програму, як розподіляються взаємозв'язки між цими компонентами, які критерії має задовольняти програмний проект.

Засоби C++, що розглядаються у цьому практикумі, відповідають стандарту ANSI ISO/IEC 14882 (1998). З розповсюджених компіляторів йому найбільш за все відповідають, наприклад, Microsoft Visual C++ (ОС Windows) та gcc (GNU).

Лабораторний практикум відповідає вимогам кредитно-модульної системи ESTC.

Загальні методичні вказівки до виконання лабораторних робіт

Головною метою типових практичних завдань лабораторних робіт практикуму є формування і набуття навичок з написання об'єктно-орієнтованих програм, а тому експериментами, виконуваними у кожній лабораторній роботі, є відлагодження та запуск на виконання програмного коду, написаного мовою C++.

1. Обладнання, прилади і матеріали

Лабораторні роботи з дисципліни "Об'єктно-орієнтоване програмування" ґрунтуються на написанні програмного коду мовою C++ на комп'ютері. Прилади і матеріали й інше обладнання, окрім комп'ютера, не використовуються. Тому під час виконання лабораторних робіт необхідно дотримуватися таких системних вимог до комп'ютера:

1. Персональний комп'ютер або ноутбук типу PC (або Mac) із частотою процесора близько 1,5 ГГц (залежить від типу операційної системи), оперативною пам'яттю від 256 Мбайт (залежить від програмного забезпечення).
2. Операційна система Windows 2000, XP або 2007 (або *nix-системи, FreeBSD, MacOS зі встановленними віртуальними машинами типу VirtualBox чи емуляторами Windows-програм).
3. Програмне середовище (одне на вибір): TC, Borland C++ Builder, MS Visual Studio, QtCreator, IDE NetBeans, KDevelop.

2. Заходи безпеки під час виконання лабораторних робіт

Під час виконання лабораторних робіт згідно із завданнями, наведеними у практикумі, потрібно дотримуватись правил охорони праці під час експлуатації електронно-обчислювальних машин (ЕОМ). Зокрема, для зниження й попередження шкідливого впливу ЕОМ слід:

— розміщувати екран дисплея на відстані не ближче 550 – 700 мм від очей;

— установити частоту регенерації дисплея не менше 72 Гц;

— використовувати технологічні перерви;

— не класти на блоки ЕОМ папір, книги, документи й інші предмети;

— переконатися у справності сполучних проводів, штепсельних рознімів, у надійності кріплення захисних кожухів і кришок блоків ЕОМ;

— не допускати забивання пилом і сторонніми предметами вентиляційних отворів для відведення тепла з блоків ЕОМ;

— відрегулювати положення екрана монітора відносно свого поля зору;

— умикаючи персональну ЕОМ і освітлення в електромережу, братися тільки за ізольовані частини штепсельних колодок;

— щоб уникнути розрядів статичної електрики, не торкатися до екрана дисплея;

— безперервна тривалість роботи перед екраном не повинна перевищувати 1 год з наступними регламентованими перервами по 10 хв для відпочинку;

— забороняється у разі невимкненого електроживлення ЕОМ: перемикати сполучні шнури блоків ЕОМ; змінювати встановлену конфігурацію робочого місця, переставляти блоки ЕОМ; робити вологе прибирання поверхонь комп'ютера; приймати їжу безпосередньо за клавіатурою комп'ютера;

— після закінчення роботи завершити працюючі програми, підготувати комп'ютер до вимикання.

3. Порядок і рекомендації щодо виконання робіт

Лабораторні роботи слід виконувати в такій послідовності:

1. Увійти до програмного середовища (наприклад, MS Visual Studio).
2. Створити новий файл проекту (меню File->New, Ctrl+O). Дати назву проекту.
3. Отримати завдання від викладача, подібне до викладених у практикумі.

4. Під час виконання завдання з лабораторного практикуму студент повинен виконати такі дії:
   4.1. Прочитати основні теоретичні відомості. У разі потреби — перечитати конспект лекцій, передивитись додатково рекомендовану літературу.
   4.2. Прочитати і зрозуміти завдання конкретної лабораторної роботи.
   4.3. Розробити алгоритм програми згідно з поставленим завданням.
   4.4. Закодувати алгоритм мовою C++ у програмному середовищі (тобто написати програму), обов'язково використавши принципи ООП, зазначені у меті лабораторної роботи.
   4.5. Зберегти файли з програмним кодом на диску та на зовнішньому флеш-пристрої (Ctrl+S).
   4.6. Відлагодити програмний код (відкомпілювати програму). За наявності синтаксичних помилок — виправити їх. Проводити ітерації доти, доки програмний код не міститиме помилок, про що буде свідчити повідомлення компілятора: "Compilation successful!" (компіляція відбулася успішно).
   4.7. Зберегти оновлені файли з програмним кодом (Ctrl+S).
   4.8. Запустити програму на виконання і отримати певний результат залежно від того, що вимагається у завданні.
   4.9. Проаналізувати результат. Підготувати відповіді викладачу про те, яким чином було використано той чи інший принцип ООП.
   4.10. Зробити висновки.
5. Підготувати звіт з лабораторної роботи.
6. Захистити лабораторну роботу.

Написання тексту програми слід починати з визначення змінних, необхідних для функціонування алгоритму. Якщо схему алгоритму розроблено досконало досить чітко, написання кодової частини програми зводиться до запису кожного елемента схеми операторами мови програмування. Необхідно уникати використання оператора *goto*. Під час написання

операторів виведення особливу увагу зосереджувати на тому, щоб результати мали нагляду форму, яку можна проаналізувати.

У процесі написання програм слід дотримуватись такого стилю програмування, щоб код кожного класу розміщувався у двох файлах: інтерфейс — у заголовному (.h), реалізація — у вихідному (.cpp), тому всі програми потрібно реалізовувати як багатофайлові проекти, у яких головна функція *main()* теж займає окремий файл. Імена всіх класів, а також імена методів повинні починатися з великої літери, імена полів (змінних) — із малої.

4. Оформлення результатів

Після виконання кожної лабораторної роботи необхідно оформляти звіт на аркушах формату А4 білого кольору, шрифтом, не меншим 10 pt. Звіт має містити титульний аркуш, назву і мету роботи, текст розроблених програм з детальними коментарями, висновок. Тексти програм у звіті слід виконати шрифтом Courier New. У додатку наведено зразок оформлення титульного аркушу звіту.

5. Оброблення даних

Оброблення експериментальних результатів полягає у збереженні отриманих у результаті виконання програми даних у вигляді скріншотів або текстових файлів (якщо це передбачено у програмі).

6. Аналіз результатів та основних висновків

Під час виконання лабораторної роботи студенти повинні проаналізувати результати та зробити основні висновки (у порядку виконання робіт — пункти 4.9—4.10). При цьому слід звернути особливу увагу на мету і основні завдання лабораторної роботи, а також на те, яким чином об’єктно-орієнтовані принципи втілені у програмному коді студента. У висновках слід відобразити, які саме навички ООП були набуті студентом у результаті виконання лабораторної роботи.

МОДУЛЬ 1

ЛАБОРАТОРНА РОБОТА 1.1

Проектування класів і створення об'єктів. Реалізація програми

Мета роботи — засвоєння поняття класів та об'єктів; набуття практичних навичок їх оголошення та використання.

Основні завдання роботи

1. Реалізувати програмно клас, який реалізує роботу стека.
2. Розробити і реалізувати програму, що використовує клас стека для моделювання Т-подібного сортувального вузла на залізній дорозі:
   - 2.1. У програмі слід реалізувати поділ поїзда, що складається з вагонів двох типів, на два напрямки (на кожний напрямок формується поїзд з вагонів одного типу).
   - 2.2. Реалізувати у програмі можливість формування залізничного состава користувачем з клавіатури та випадковим чином до попередньої програми.

Основні теоретичні відомості

1. Класи і об'єкти

Механізм, який об'єднує дані та код, що маніпулює цими даними, і захищає те й інше від зовнішнього втручання або неправильного використання, називається **інкапсуляцією** (incapsulation). У С++ інкапсуляція підтримується, зокрема, завдяки використанню класів і об'єктів.

Клас (class) — це складний тип даних, що визначається користувачем. Він може складатися як з елементів-даних (змінних), так і функціональних даних (функцій, методів), які реалізують операції з ними.

Функції та змінні, оголошені всередині класу, є членами (members) цього класу. Функції, оголошені всередині класу, називають функціями-членами (member functions) або методами.

Члени класу можуть бути **закритими** (private) або **відкритими** (public). Для оголошення закритих членів класу використовується ключове слово *private*. Закриті члени класу доступні для інших членів цього класу, але не досяжні поза цим класом. Для оголошення відкритих членів класу використовується ключове слово *public*. До відкритих членів класу мають доступ як члени цього класу, так і будь-яка інша частина програми. За замовчуванням (by default) члени класу є закритими.

Синтаксис описання класу (за замовчуванням члени класу є закритими, а тому ключове слово *private* є необов'язковим в оголошенні класу; *списку_об'єктів* теж може не бути або він складатиметься з одного чи кількох об'єктів):

```
class ім'я_класу
{
  private:
  ... //оголошення закритих функцій та змінних класу
  public:
  ... //оголошення відкритих функцій та змінних класу
 } список_об'єктів;
```

Тут *ім'я_класу* визначається згідно з правилами написання ідентифікаторів у C++: ім'я може починатись із символів A..Z, a..z або зі знака підкреслювання і містити будь-які літери та знак підкреслювання.

Крапка з комою є обов'язковою в кінці оголошення класу.

Наприклад, оголосимо клас *myclass*:

```
class myclass {
  int x, y; //оголошення закритих змінних класу
  public: //оголошення відкритих функцій класу
  void show();
  int getx();
  int gety();
 }; //кінець оголошення класу
```

Для визначення функцій-членів класу слід зв'язати ім'я цього класу з іменем цього методу (функція може бути без параметрів або мати один чи кілька параметрів):

```
тип_даних_що_повертається_функцією  ім'я_класу::
                        ім'я_функції (параметри_функції)
{
   ... // тіло функції
}
```

Наприклад,

```
void myclass:: show(){
   cout << "x=" << x <<", y=" << y <<'\n';
}
```

Подвійною двокрапкою визначається операція розширення діапазону видимості (scope resolution operator).

Об'єкт — це змінна типу "клас". Оголошення класу визначає лише тип об'єкта, який буде створено під час його фактичного оголошення. Для створення об'єкта використовується ім'я класу як специфікатор типу даних:

```
ім'я_класу  список_об'єктів;
```

Тут *список_об'єктів* може складатися як з одного імені об'єкта, так і з кількох, розділених комами.

Наприклад, для класу *myclass*, об'єкти *ob1, ob2* будуть оголошені так:

```
myclass ob1, ob2;
```

Оголошення класу є логічною абстракцією, яка задає новий тип даних для об'єкта, а оголошення об'єкта створює фізичну сутність об'єкта такого типу (тобто в разі оголошення об'єкта виділяється пам'ять, а в разі оголошення класу — не виділяється).

Доступ до відкритих членів класу можна отримати лише через об'єкт цього класу. Для доступу до відкритих членів класу використовують операцію доступу "крапка" (.):

```
ім'я_об'єкта.ім'я_функції (параметри_функції);
```

Для ініціалізації об'єктів класу використовують конструктори.

2. Конструктори і деструктори

Конструктор (constructor function) — функція-член класу, яка включається в описання класу і має те саме ім'я, що і клас. Конструктор створює значення типу "клас" (тобто об'єкт класу). Цей процес включає ініціалізацію змінних класу і часто розподіл вільної пам'яті. Конструктор класу викликається автоматично щоразу під час створення об'єкта. Конструктор не може мати значення, що повертається.

Оголошення конструктора:

```
class ім'я_класу
{   ... // оголошення закритих членів класу
  public:
    ... // оголошення відкритих членів класу
    ім'я_класу(список_параметрів); //конструктор
 };
```

Для глобальних об'єктів конструктор викликається на початку виконання програми. Для локальних об'єктів конструктор викликається щоразу під час оголошення об'єкта.

Функція, обернена до конструктора, є **деструктором** (destructor). Ця функція викликається у разі знищення об'єктів. Локальні об'єкти знищуються тоді, коли вони виходять з діапазону видимості. Глобальні об'єкти знищуються по завершенню програми. Деструктор задається символом ~ (тильда) з таким іменем класу:

```
class ім'я_класу
{     ... // оголошення закритих членів класу
    public:
```

```
    ... // оголошення відкритих членів класу
    ~ім'я_класу(); //деструктор
};
```

Неможливо отримати вказівники на конструктор та деструктор.

Конструктору можна передавати аргументи. Для цього просто додаються необхідні параметри в оголошення та визначення конструктора. Потім під час оголошення об'єкта конструктору слід передати аргументи:

```
ім'я_класу  ім'я_об'єкта ( список_аргументів );
```

Фактично синтаксис передачі аргументів конструктору з параметрами є скороченою формою запису більш довгого виразу:

```
ім'я_класу ім'я_об'єкта = ім'я_класу (список_параметрів);
```

На відміну від конструктора деструктор не може мати параметрів. Зміст цього зрозуміти досить просто: немає механізму передавання аргументів об'єкту, що знищується.

Порядок виконання роботи

1. Скласти і реалізувати програмно опис класу стека, як організованої структури даних, що працює за принципом FCLS (first come last served — першим прийшов останнім пішов).

2. Елементами стека будуть об'єкти класу вагону. У класі вагону слід відобразити тип вагону та його порядковий номер у закритій частині класу. Задати конструктори і деструктори цього класу. Задати функції-члени, які будуть повертати значення закритих елементів.

Передбачити у класі стека функції-члени, які забезпечують:

— формування початкового елемента стека;
— формування наступних за початковим елементів стека, ураховуючи зв'язки між елементами;
— додавання нового елемента в кінець стека;

— знищення елемента з кінця стека; якщо елемент, що знищується, є єдиним, передбачити існування порожнього стека;
— перевірку меж заповнення елементами стека;
— підрахування кількості елементів у стеку.

3. Передбачити у програмі функцію формування початкового об'єкта стека, що являє собою імітацію початкового залізничного состава, що складається з вагонів двох типів, з якого потім будуть братись елементи для сортування на состави з вагонів двох різних типів. Реалізувати можливість присвоєння вагонам типів випадковим чином та за допомогою клавіатури (користувачем).

4. Для моделювання роботи Т-подібного сортувального вузла, написати у програмі функцію, яка використовує початковий об'єкт класу стека для формування двох об'єктів типу стек, у кожний з яких заносяться вагони лише одного типу.

5. Передбачити у програмі функцію виведення елементів стека на екран.

6. Програма повинна містити меню, яке дозволить перевірити основні функції, створені у програмі.

Контрольні завдання та запитання

1. Наведіть визначення класу.
2. Наведіть визначення об'єкта.
3. Назвіть принципи ООП.
4. Що таке інкапсуляція?
5. Яка відмінність між закритими та відкритими членами класу?
6. Напишіть синтаксис описання класу.
7. Який синтаксис описання функції-члена класу?
8. Як оголошуються об'єкти? Яка відмінність між класом та об'єктом?
9. Яким чином організовано доступ до членів класу?
10. Нехай маємо клас *wigit*. Які імена його конструктора та деструктора?

```
class widgit {  int x, y;
  public:
```

```
    /*...впишіть конструктори та деструктори*/
};
```

ЛАБОРАТОРНА РОБОТА 1.2

Проектування і реалізація програми з масивами об'єктів, вказівниками та посиланнями на них

Мета роботи — засвоєння поняття масиву об'єктів, вказівників та посилань на об'єкти класів; набуття практичних навичок їх оголошення та використання.

Основні завдання роботи

Розробити і реалізувати програмний додаток нарахування заробітної плати співробітникам фірми. Для збереження даних про співробітників використати масиви об'єктів, а для доступу до функцій — вказівники. Для цього:

1. Розробити і реалізувати програмно клас "Особа".
2. Розробити і реалізувати програмно функції, які враховують такі вимоги:
   - 2.1. Кожному співробітнику заробітна плата нараховується, виходячи з тарифної сітки, у якій закладено такі позиції: "Спеціаліст", "Спеціаліст 1 категорії", "Спеціаліст 2 категорії", "Спеціаліст вищої категорії", "Начальник відділу", "Директор".
   - 2.2. Кожному співробітнику призначається додатково до ставки премія за принципом: якщо заробітна плата менша від середньої заробітної плати усіх співробітників, то розмір премії становить 50% від заробітної плати, в інших випадках — 30% від заробітної плати.
   - 2.3. Реалізувати програмно функцію, яка визначає суму, що видається на руки (заробітна плата плюс премія).

3. Продемонструвати роботу створеного класу і його функцій в основній частині програми.

Основні теоретичні відомості

1. Масиви об'єктів

Об'єкти — це змінні, які мають такі самі можливості, що й змінні інших типів. Тому допустимим є вкладення об'єктів у масив. Синтаксис оголошення масиву об'єктів аналогічний тому, який використовується для оголошення масиву змінних будь-якого іншого типу. Доступ до масивів об'єктів аналогічний доступу до масивів змінних будь-якого іншого типу.

Масиви об'єктів створюються таким чином:

```
ім'я_класу  ім'я_масиву [кількість_елементів_масиву];
```

Якщо клас містить конструктор, масив об'єктів може бути ініціалізований. Наприклад, для об'єкта *ob* класу *myclass* (конструктор якого ініціалізує лише одну змінну типу *int*) масив об'єктів можна ініціалізувати так:

```
myclass ob[4] = { -1, -2, -3, -4 };
//тобто присвоюємо ob[0].x=-1; …; ob[3].x=-4
```

Якщо ж конструктор ініціалізує більше однієї змінної, тоді під час ініціалізації масиву об'єктів слід передавати об'єкти класу. Наприклад:

```
class myclass {
    int x,y;
 public:
    myclass(int a, int b) {x = a; y= b;}; //конструктор
};
…
myclass ob[2] = { myclass(10, 2), myclass(-3, -4) };
//масив об'єктів, присвоюємо значення
//ob[0].x=10; ob[0].y=2; ob[1].x=-3; ob[1].y=-4;
```

Можна також оголосити двовимірний масив об'єктів.

Наприклад, щоб оголосити масив об’єктів попереднього класу розмірності 4×2, необхідно написати у програмі таке:

```
myclass obj[4][2] = { myclass (1, 2), myclass (3, 4),
                      myclass (5, 6), myclass (7, 8),
                      myclass (9, 10), myclass (11, 12),
                      myclass (13, 14), myclass (15, 16)};
```

2. Указівники на об’єкти

Доступ до об’єкта можна отримати також через вказівник на цей об’єкт. Для цього використовують оператор “стрілка” (->). Вказівник на об’єкт оголошується так само, як і вказівник на змінну будь-якого іншого типу. Для цього треба задати ім’я класу цього об’єкта, а потім ім’я змінної із зірочкою перед ним.

Для отримання адреси об’єкта перед його іменем (або ідентифікатором) потрібно поставити оператор “&” так само, як це робиться для отримання адреси змінної іншого типу.

Арифметика вказівників на об’єкт аналогічна арифметиці вказівників на дані будь-якого іншого типу: її використовуєть стосовно об’єкта. Як і вказівник будь-якого типу, під час інкрементації на об’єкт вказівник буде вказувати на наступний об’єкт такого самого типу. Якщо вказівник на об’єкт декрементується, то він починає вказувати на попередній об’єкт.

Приклад. Указівники на об’єкти класу:

```
#include <iostream>
class myclass {   int a;
  public:
    myclass(int x); // конструктор
    int get();
};
myclass::myclass(int x) {  a = x; }
int myclass::get() { return a; }
int main() {
  myclass ob(120);  // оголошення об’єкта
  myclass *p;  // оголошення вказівнка на об’єкт
  p = &ob; // запам’ятовування адреси ob у p
  cout << "Доступ через об’єкт: " << ob.get();
  cout << "\n";
```

```
  cout << "Доступ через вказівник: " << p->get();
  return 0;
}
```

3. Вказівник *this*

У C++ є спеціальний вказівник *this*. Це — вказівник, який автоматично передається будь-якій функції-члену класу при її виклику і вказує на об’єкт, що генерує цей виклик. Наприклад:

```
ob.f(); // нехай ob - це об’єкт
```

Функції *f()* автоматично передається вказівник на об’єкт *ob*. Цей вказівник називається *this*. Вказівник *this* передається лише функціям-членам класу.

Порядок виконання роботи

1. Для реалізації програмного коду оголосити та визначити клас *особа*. У цьому класі слід оголосити закриті змінні, що зберігають такі дані: прізвище та ініціали співробітника, заробітну плату, премію, ставку, позицію з тарифної сітки.

2. Додати у клас відкриті конструктори. Один з конструкторів оголосити без параметрів, інший з параметрами для ініціалізації об'єктів класу. Конструктор не повинен ініціалізувати значення ставки, натомість він має викликати функцію, яка її визначає.

3. Додати у клас деструктор.

4. Додати у клас функцію визначення ставки відповідно до позиції у тарифній сітці.

5. Додати у клас функцію призначення премій. Врахувати, що премія призначається за принципом: якщо заробітна плата менша, ніж середня зарплата усіх співробітників, то розмір премії становить 50% від заробітної плати, в інших випадках — 30% від заробітної плати.

6. Додати функцію визначення заробітної плати.

9. Додати функцію виведення даних про співробітника.

7. В основній частині програми створити масив об’єктів. Доступ до елементів масиву показати через вказівник.

8. Програма повинна містити меню, яке дозволить перевірити основні функції, створені у програмі.

Контрольні завдання та запитання

1. Як оголосити вказівник на об'єкт?
2. Що таке вказівник *this*?
3. Як звернутися у програмі до елементів масиву об'єктів?
4. Як отримати доступ елементу масива об'єктів до відкритих членів класу?
5. Якими способами можна виконати ініціалізацію масиву об'єктів?
6. Як оголосити двовимірний масив об'єктів?
7. Дано прототип конструктора:

```
myclass(int, char, char*);
```

Оголосіть масив із чотирьох об'єктів.

ЛАБОРАТОРНА РОБОТА 1.3

Проектування і реалізація програми з перевантаженням функцій

Мета роботи — засвоєння поняття статичного поліморфізму через перевантаження функцій; набуття навичок використання практичних прийомів перевантаження функцій та аргументів за замовчуванням.

Основні завдання роботи

Розробити та реалізувати програмний додаток, який оперує над множенням матриць та векторів. Для цього:

1. Розробити та реалізувати програмно клас *матриця*.
2. Розробити та реалізувати програмно клас *вектор*.
3. Реалізувати програмно перевантаження функції множення вектора на матрицю, матриці на вектор,

матриці на матрицю, множення числа на матрицю, матриці на число.

4. Продемонструвати роботу створених класів та їх функцій в основній частині програми.

Основні теоретичні відомості

1. Поняття поліморфізму

Поліморфізм — це процес, завдяки якому загальний інтерфейс застосовується до двох або більше схожих (але технічно різних) ситуацій, тобто реалізується філософія "один інтерфейс, багато методів".

2. Перевантаження функцій

Після класів важливою можливістю C++ є перевантаження функцій (function overloading). Це той механізм, завдяки якому в C++ досягається один з видів поліморфізму.

У C++ дві або більше функцій є **перевантаженими**, якщо вони мають одне й те саме ім'я, що відрізняються або типом, або кількістю аргументів, або тим і тим.

Щоб перевантажити функцію, треба оголосити та задати всі варіанти, які можуть знадобитися. Компілятор автоматично вибере правильний варіант виклику на основі кількості та/або типу аргументів, що використовуються в функції.

Наведемо приклади перевантаження функцій. В першому з них перевантажені функції відрізняються типом аргументів, у другому — їх кількістю.

Приклад 1. Функція *date( )* перевантажується для отримання дати або у вигляді рядка, або у вигляді трьох цілих чисел. У двох випадках функція виводить на екран передані їй значення:

```
#include <iostream. h>
void date(char *date); // дата у вигляді рядка
void date(int day, int month, int year);
                          // дата у вигляді чисел
main( )
{ data("23/5/95");
  date(23,5,95);
  return 0; }
```

```
//дата у вигляді рядка
void date (char *date)
{  cout <<"Дата:"<<date<<"\n"; }
//дата у вигляді чисел
void date(int day, int month, int year)
{  cout<< "Дата:" << day << "/" << month << "/" <<
          year <<"\n"; }
```

Приклад 2. Перевантажені функції можуть відрізнятися кількістю аргументів:

```
#include <iostream.h>
void f1(int a);
void f1(int a, int b);
main( ) {
    f(10);
    f(10, 20);
    return 0;
}
void f1(int a) {  cout <<"B f1(int a)\n"; }
void f1(int a, int b) { cout<< "Bf1 (int a, int b)\n"; }
```

3. Перевантаження конструкторів

Конструктор перевантажувати можна, деструктор — не можна. Кожному способу оголошення об'єкта класу має відповідати своя версія конструктора класу.

Найчастіше перевантаження конструктора використовується для забезпечення вибору: ініціалізувати об'єкт чи не ініціалізувати. Наведемо приклад:

```
class myclass { int x;
  public:
// перевантаження конструктора двома способами
    myclass( ) {x=0;} //немає ініціалізації
    myclass(int n) {x=n;} //є ініціалізація
    int getx( ) {return x;}
};
```

Перевантаження конструкторів також традиційно застосовується для підтримання масивів. Для співіснування в

програмі неініціалізованих масивів об’єктів поряд з ініціалізованими використовується конструктор, який підтримує ініціалізацію та конструктор, який її не підтримує. Ці ж конструктори можна використовувати для ініціалізації, або неініціалізації об’єктів. Наприклад, для класу *myclass* з попереднього прикладу правильні ці два оголошення:

```
myclass ob(10);
myclass ob[5];
```

4. Конструктор копіювання

Однією з найважливіших форм перевантаженого конструктора є конструктор копій (copy constructor).

Коли об'єкт передається у функцію, виконується порозрядна (тобто точна) копія цього об'єкта та передається тому параметру функції, який отримує об'єкт. Однак бувають ситуації, у яких така точна копія об'єкта небажана. Наприклад, якщо об'єкт містить вказівник на виділену ділянку пам'яті, то в копії вказівник буде посилатися на ту саму ділянку пам'яті, на яку посилається вихідний вказівник. Отже, якщо копія змінює вміст ділянки пам'яті, то ці зміни торкнуться також і вихідного об'єкта. Крім того, коли виконання функції завершується, копія видаляється, що приводить до виклику деструктора цієї копії. Подібна ситуація характерна і для випадку, коли об'єкт є типом значення, яке повертає функція.

Через визначення конструктора копіювання можна повністю контролювати весь процес створення копії об'єкта.

Важливо розуміти, що в C++ точно виділяють два типи ситуацій, у яких значення одного об'єкта передається другому. Перша ситуація — це присвоєння, друга — ініціалізація, яка може виконуватися у трьох випадках:

1) коли в інструкції оголошення об'єкта один об'єкт використовується для ініціалізації другого;

2) коли об'єкт передається у функцію як параметр;

3) коли як значення, що повертається функцією, створюється тимчасовий об'єкт.

Конструктор копій використовується лише для ініціалізації, але не для присвоєння. Після визначення конструктора копій,

він викликається завжди під час ініціалізації одного об'єкта другим.

Основна форма конструктора копій:

```
ім'я_класу (const ім'я_класу &obj) {
  ... // тіло конструктора
}
```

Тут *&obj* — посилання на об'єкт *obj*, що використовується для ініціалізації іншого об'єкта. Наприклад, нехай є клас *myclass*, а *y* —об'єкт типу *myclass*, тоді такі інструкції могли б викликати конструктора копій:

```
myclass x = y; // y явно ініціалізує x
fund (y); // y передається як параметр
y = func2(); // y отримує об’єкт, що повертається
```

У двох перших випадках конструктору копій можна було б передати посилання на об'єкт *y*, у третьому — конструктору копій передається посилання на об'єкт, що повертається функцією *func*().

5. Використання аргументів за замовчуванням

Можливість використання аргументів за замовчуванням (default arguments) пов’язана з перевантаженням функцій та дозволяє при виклику функції відповідний аргумент не задавати, а надати параметру значення за замовчуванням.

Щоб передати параметру аргумент за замовчуванням, треба поставити після параметра знак рівності і те значення, яке треба передати. Тоді, якщо при виклику функції відповідний аргумент не задається, то за замовчуванням функції буде передано задане значення. Наприклад, у наведеній функції двом параметрам за замовчуванням присвоєно значення 0:

```
void f (int a=0, int b=0);
```

Тепер функцію *f( )* можна викликати трьома різними способами: 1) з двома заданими аргументами; 2) тільки з першим заданим аргументом (тоді *b* за замовчуванням

дорівнюватиме нулю); 3) без будь-яких аргументів (тоді *a* та *b* за замовчуванням дорівнюватимуть нулю). Таким чином, всі такі виклики *f( )* правильні:

```
f( ); // a=0 і b=0 за замовчуванням
f(10); // a=10, b=0 за замовчуванням
f(10, 99); // a=10, b=99
```

Якщо створюються функції, які мають один або більше аргументів, що передаються за замовчуванням, ці аргументи потрібно задавати тільки один раз: або у визначенні функції, або в її прототипі.

Усі параметри, що задаються за замовчуванням, мають розміщуватися правіше від параметрів, що передаються звичайним шляхом. Більше того, якщо почали задавати параметри за замовчуванням, то вже не можна передавати параметри звичайним способом.

Аргументи за замовчуванням мають бути або константами, або глобальними параметрами.

Ще два зауваження щодо використання аргументів за замовчуванням:

а) можна передавати аргументи за замовчуванням конструкторам;

б) прикладом використання аргументів за замовчуванням є випадки, коли треба вибрати варіант.

Порядок виконання роботи

1. Для реалізації програмного коду створити два класи: *матриця* і *вектор*. У кожному з класів задати конструктори і деструктори.

2. Як закриті елементи класів створити динамічні масиви, що зберігають відповідно матрицю або вектор та їх розмірність. Динамічний масив матриці має бути двовимірним, вектора — одновимірним.

3. Додати дружні функції множення вектора на матрицю і навпаки для цих двох класів. Як параметри передати об’єкти створених класів.

4. Перевантажити створену функцію для множення вектора на число і навпаки.

5. Перевантажити створену функцію для множення матриці на число і навпаки.

6. Перевантажити створену функцію для множення двох матриць (як об'єктів класу *матриця*).

7. Перевантажити створену функцію для множення двох векторів (як об'єктів класу *вектор*).

8. Під час реалізації перевантажених функцій перевірити можливість такого множення (не всі вектори і матриці можна перемножати).

9. В основній частині програми продемонструвати роботу перевантажених функцій.

Контрольні завдання та запитання

1. Що таке поліморфізм?

2. Які функції називають перевантаженими?

3. Для чого введено перевантаження функцій?

4. Які переваги використання перевантаження конструктора класу?

5. Покажіть, як перевантажити конструктор для наступного класу так, щоб можна було створити неініціалізовані об'єкти. Створюючи неініціалізовані об'єкти, присвойте змінним *x* та *y* значення 0:

```
class myclass { int x,y;
   public:
     myclass(int i, int j) {
       x = i;
       y = j;
     }
   ... // інші функції класу
};
```

6. Поясніть, що таке аргумент за замовчуванням.

7. Що неправильно у наступних прототипах, які використовують аргументи, що передаються за замовчуванням:

```
int f(int count, int max = count);
void func(int x = 4, int y);
```

ЛАБОРАТОРНА РОБОТА 1.4

Проектування і реалізація програми з перевантаженням операторів (бінарних і унарних)

Мета роботи — засвоєння поняття статичного поліморфізму через перевантаження операторів (бінарних і унарних); набуття практичних навичок їх оголошення та використання.

Основні завдання роботи

Реалізувати програмно клас, що оперує з комплексними числами. Для цього виконати такі завдання:

1. Розробити і реалізувати програмно клас комплексного числа.
2. Реалізувати перевантаження операцій:
   — додавання (+);
   — віднімання (-);
   — множення (*);
   — множення на скалярне число (*);
   — спряжене число (унарний -);
   — порівняння двох комплексних чисел (==, !=);
   — присвоєння комплексних чисел (=).

Перевантажте (на вибір) частину унарних і бінарних операторів як функції, що належать класу, а другу частину — як дружні функції.

3. Продемонструвати роботу створеного класу та його функцій в основній частині програми.

Основні теоретичні відомості

1. Основи перевантаження операторів

Перевантаження операцій (множення, ділення, віднімання, додавання та інші операції) — одна з важливих особливостей C++. Ця властивість дозволяє визначати зміст тих операторів

C++, які зв'язані із заданими класами. Перевантаженням зв'язаних з класами операторів можна легко задавати в програмі нові типи даних. Перевантаження операторів подібне до перевантаження функцій. Більше того, перевантаження операторів є фактично одним з видів перевантаження функцій. Проте при цьому вводяться деякі додаткові правила. Наприклад, оператор, що перевантажується, завжди зв'язаний з класом.

Коли оператор перевантажується, він не втрачає свого вихідного змісту, а навпаки отримує додаткові властивості, пов'язані з класом, для якого він задається.

Для перевантаження оператора задається **оператор-функція** (operation function). Оператор-функція може бути членом класу або дружньою функцією класу, для якого її задано.

Загальна форма оператора-функції члена класу має вигляд:

```
тип_даних_що_повертається_функцією
          ім'я_класу:: operator#(список_аргументів)
{
  ... //дії, що виконуються; тіло оператора-функції
}
```

Частіше типом, що повертається оператором-функцією, є клас, для якого її задано. Проте оператор-функція може повертати будь-який тип. Замість "#" ставиться знак оператора, що перевантажується. Наприклад, якщо перевантажується операція додавання "+", то у функції повинне бути ім'я "*operator*+". Зміст *списку_аргументів* сильно залежить від того, як реалізована оператор-функція і від типу оператора, що перевантажується.

Треба пам'ятати два важливі обмеження на перевантаження операторів: 1) не можна змінювати пріоритет операторів; 2) не можна змінювати кількість операндів оператора. Наприклад, не можна перевантажувати оператор "/" так, щоб у ньому використовувався тільки один операнд.

Більшість операторів C++ перевантажується. Неперевантажуваними операторами є

. :: .* ?

Не можна перевантажувати оператори препроцесора.

За винятком оператора “=”, оператори-функції успадковуються похідним класом. Проте і для похідного класу можна перевантажити будь-який вибраний оператор, включаючи оператори, що вже перевантажені в базовому класі.

Хоча допустимо мати оператора-функцію для реалізації будь-якої дії, краще, якщо дії перевантажених операторів залишаються у сфері їх традиційного використання.

Проте іноді може виникнути потреба використати будь-який оператор нетрадиційним методом. Прикладом цього є оператори “<<” та “>>”, які перевантажуються для введення-виведення.

Оператор-функція не може мати параметрів, що передаються за замовчуванням.

2. Перевантаження бінарних операторів

Бінарними називаються оператори, які оперують з двома операндами (наприклад, арифметичні оператори: множення, ділення, додавання, віднімання).

Коли оператор-функція член класу перевантажує бінарний оператор, у функції буде тільки один параметр. Цей параметр отримає той операнд, що розміщений справа від знака оператора. Лівий операнд (об’єкт класу) викликає оператора-функцію і передається неявно через вказівник *this*.

Важливо розуміти, що для написання оператор-функцій є багато варіантів. Приклади, які наведено нижче, не є вичерпними, проте вони ілюструють декілька найбільш технічних прийомів.

Приклад 1. Програма, в якій перевантажено оператор “+” для класу *coord*. Цей клас описує координати точки (*x, y*) на площині:

```
#include <iostream.h>
class coord {
// змінні класу, що зберігають значення координат
    int x,y;
```

```
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
 //оголошення оператора-функції
    coord operator+(coord ob2);
 };
 // Перевантаження оператора + для класу coord.
 // Визначення оператора-функції
 coord coord::operator+(coord ob2) {
    coord temp;
    temp.x = x + ob2.x; // x = this->x
    temp.y = y + ob2.y; // y = this->y
    return temp;
 }
 int main() {
    coord o1(10, 10), o2(5, 3), o3; // оголошення об'єктів
    int x, y;
    // додавання двох об'єктів – виклик operator+()
    o3 = o1 + o2;  /* o1+ викличе оператора-функцію
   і o1 буде переданий вказівником this, o2 буде
   переданий параметру ob2 оператору-функції;
    функція operator+() поверне об'єкт, що зберігається у
    temp, і цей об'єкт буде присвоєно об'єкту o3*/
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";
    return 0;
 }
```

Проаналізуємо програму. Функція *operator+( )* повертає об'єкт типу *coord*, у якому сума координат *x* розміщена в *x*, а сума координат *y* — в *y*. Відзначимо, що тимчасовий об'єкт *temp* використовується всередині *operator+( )* для зберігання результату і є об'єктом, що повертається. Більше того, жоден з операндів не змінюється. Тобто оператор "+" був перевантажений способом, що є аналогічним своєму традиційному арифметичному використанню. Завдяки цьому можна використовувати вирази:

```
o3=o1+o2;
```

```
o3=o1+o2+o1+o3;
(o1+o2). get_xy(x, y);
```

3. Перевантаження операторів відношення та логічних операторів

Існує можливість перевантаження операторів відношення та логічних операторів. У разі перевантаження операторів відношення та логічних операторів оператори-функції будуть повертати ціле (integer), що інтерпретується як true або false.

4. Перевантаження унарних операторів

Унарними називають оператори, що працюють лише з одним операндом. Перевантаження унарних операторів виконується так само, як і перевантаження бінарних, за винятком того, що унарний оператор працює тільки з одним операндом. У разі перевантаження унарного оператора з використанням функції-члена в оператора-функції не буде параметрів. Оскільки є тільки один операнд, він і викликає оператора-функцію. Інші параметри не потрібні.

Приклад 2. Для класу *coord* перевантажується оператор інкрементації (++):

```
// Перевантаження ++ для класу coord.
#include <iostream.h>
class coord { int x, y; // значення координат
  public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    coord operator++();
};
// Перевантаження ++ для класу coord.
coord coord::operator++(){
    x++;  y++;
    return *this;
}
int main() {
    coord o1(10, 10);
    int x, y;
```

```
    ++o1; // інкрементація об'єкта
    o1.get_xy(x, y);
    cout << "(++o1) X: " << x << ", Y: " << y << "\n";
    return 0;
}
```

Оскільки оператор інкрементації задається для збільшення операнда на одиницю, перевантаження ++ змінює об'єкт з яким вона оперує. Це дозволяє використовувати оператор інкрементації як частину такого виразу:

```
o2=++o1;
```

5. Використання дружніх операторів-функцій

У дружню функцію не передається вказівник *this*. У випадку бінарного оператора це означає, що оператору-функції явно потрібно передати обидва операнди як аргументи. У разі перевантаження унарного оператора передається один операнд.

Використовуючи дружню функцію в операціях з об'єктами, можна застосовувати вбудовані типи даних, і при цьому вбудований тип може розміщуватись зліва від оператора (у разі перевантаженя оператора-функції члена класу лівий операнд може бути об'єктом, а правий — вбудованим типом, але не можна розміщувати вбудований тип зліва від оператора). Саме у випадку, коли один з операндів є вбудованим типом і розташування об'єкта відносно операції принципове, рекомендується використовувати в разі перевантаження дружню оператор-функцію.

Порядок виконання роботи

1. Комплексне число складається з двох частин: дійсної і уявної. Нехай $a = A + Bi$, $c = C + Di$. Операції, виконувані з комплексними числами:

— додавання $a + c = (A + C) + (B + D)i$;

— віднімання $a - c = (A - C) + (B - D)i$;

— множення $a * c = (A * C – B * D) + (A * D + B * C)i$;

— множення на число $X * c = (X * C) + (X * D)i$;

— спряжене число $\tilde{a} = A - Bi$;

2. Скласти опис класу *комплексне число* і реалізувати його програмно. Клас комплексного числа має у закритій частині містити змінні для збереження дійсної і уявної частин.

8.Клас комплексного числа має містити конструктор і деструктор, а також усі функції-оператори, необхідні для реалізації:

— операції додавання (операція “+”),

— віднімання (операція “-” та “унарний мінус”),

— множення комплексних чисел, у тому числі множення комплексного на звичайне число і множення звичайного числа на комплексне (операція “*”),

— порівняння двох комплексних чисел (операції “==”, “!=”),

— обчислення спряженого числа (оператор “унарний -”),

— присвоєння одного комплексного числа іншому (оператор “=”).

4. Програма має містити меню, яке дозволить перевірити основні функції, створені у програмі.

Зокрема, у програмі потрібно надати користувачу можливість введення комплексних чисел у текстові поля у форматі “-3+7i”, а також можливість вибрати операцію, яку може виконати користувач з комплексними числами. Результат операції слід виводити у текстове поле у вигляді “(-1 + 4i) + (7 - 6i) = 6 - 2i”.

5. Продемонструвати роботу створеної програми.

Контрольні завдання та запитання

1. Для чого введено перевантаження операторів?

2. Запишіть загальні форми подання оператора-функції члена класу та дружньої функції-оператора.

3. Чи губить оператор під час перевантаження своє вихідне призначення?

4. Чи можна змінити пріоритет перевантаженого оператора?

5. Чи можна змінювати кількість операндів у разі перевантаження операторів?

6. Чим дія дружньої функції-оператора відрізняються від дії оператора-функції члена класу?

ЛАБОРАТОРНА РОБОТА 1.5

Проектування і реалізація програми з ієрархією класів (одиночне і множинне успадкування)

Мета роботи — засвоєння поняття успадкування та його принципів; набуття практичних навичок з оголошення та використання ієрархій класів.

Основні завдання роботи

Розробити та реалізувати програмно ієрархію класів для роботи з рахунком вкладника банку. Для цього слід виконати такі завдання:

1. Розробити та реалізувати програмно базовий клас Rahunok.
2. Розробити та реалізувати програмно базовий клас Vkladnyk.
3. Розробити та реалізувати програмно похідний клас RahunokVkladnyka.
4. У головній частині програми продемонструвати роботу створеної ієрархії класів.

Основні теоретичні відомості

1. Поняття успадкування

Успадкування (inheritance) — це процес, завдяки якому об’єкт може набувати властивостей іншого об’єка з додаванням до них ознак, властивих тільки йому.

Успадкування — один з трьох базових принципів ООП. Завдяки успадкуванню підтримується концепція ієрархії класів. Застосування ієрархії класів робить керованими великі потоки інформації.

2. Доступ до елементів базового класу

Клас, властивості якого успадковуються, називається **базовим**, а клас, який успадковує властивості базового, —

похідним.

Для оголошення успадкування використовується така загальна форма:

```
class ім'я_похідного_класу : доступ
                                    ім'я_базового_класу
{
 ... // елементи похідного класу
}
```

Тут *доступ* — одне з трьох ключових слів: *public, private* або *protected.*

Специфікатор доступу визначає, як елементи базового класу успадковуються похідним класом. Якщо специфікатором доступу успадкованого базового класу є *public,* то всі відкриті члени базового класу стають відкритими і в похідному. Якщо специфікатором доступу успадкованого базового класу є *private*, то всі відкриті члени базового класу стають закритими в похідному. В обох випадках всі закриті члени базового класу залишаються закритими і недосяжними для похідного класу.

Якщо специфікатором доступу є *private*, то відкриті члени базового класу стають закритими в похідному, проте ці члени залишаються доступними для функцій-членів похідного класу.

Приклад 1. Доступ до членів базового класу всередині похідного:

```
class base {
     int x;
  public:
     void setx(int n) { x = n; }
     void showx() { cout << x << '\n'; }
};
// Успадкування базового класу як public
class derived : public base {
     int y;
  public:
    void sety(int n) {y=n;}
/* Зараз буде помилка. Закритий член x базового класу
     недосяжний всередині похідного */
```

```
    void show_sum(){cout << x+y << '\n'; } // Помилка!

    void showy(){cout << y << '\n'; }
    void show() {
      showx(); /* Відкритий член базового класу
                  досяжний всередині похідного */
      showy();
    }
};
```

У цьому прикладі в похідному класі зроблено спробу доступу до змінної *x*, яка є закритим членом *base*. Це помилка, оскільки закриті члени базового класу залишаються закритими, незалежно від того, як він успадковується.

3. Захищені члени класу

Іноді можлива ситуація, коли члени базового класу, залишаючись закритими, були досяжні для похідного класу. Для реалізації цієї ідеї в C++ уведено специфікатор доступу *protected* (захищений).

Специфікатор доступу *protected* еквівалентний *private* з єдиною різницею: захищені члени базового класу досяжні для членів усіх похідних класів цього базового класу. Поза базовим класом або похідними класами захищені члени недосяжні.

Специфікатор доступу *protected* може бути у будь-якому місці оголошення класу, хоча, як правило, його розміщують після оголошення закритих членів та перед оголошенням відкритих членів. Повна загальна форма оголошення класу має такий вигляд:

```
class ім'я_класу
{
   ... // закриті члени
  protected:
   ... // захищені члени
  public:
   ... // відкриті члени
};
```

Коли захищений член базового класу успадковується похідним класом як відкритий, він стає захищеним членом похідного класу. Якщо базовий клас успадковується як закритий, то захищений член базового класу стає закритим членом похідного класу.

Базовий клас може також успадковуватись похідним класом як захищений. У цьому випадку відкриті та захищені члени базового класу стають захищеними членами похідного класу. Закриті члени базового класу залишаються закритими, і вони не досяжні для похідного класу.

Особливості доступу до елементів базового класу при успадкуванні відображено у табл. 1.

Таблиця 1

Доступ під час успадкування

Специфікатор доступу елемента базового класу	Специфікатор доступу під час успадкування базового класу	Успадкування елементів базового класу похідним класом
public	public	public
protected		protected
private		private
public	protected	protected
protected		protected
private		private
public	private	protected
protected		private
private		private

4. Конструктори, деструктори та успадкування

Базовий клас, похідний клас або обидва можуть мати конструктори та/або деструктори. Якщо і у базового, і у похідного класів є конструктори та деструктори, то конструктори виконуються у порядку успадкування, а деструктори — у зворотному порядку. Таким чином, конструктор базового класу виконується раніше, ніж конструктор похідного класу. Для деструкторів правильний

зворотний порядок: деструктор похідного класу виконується раніше від деструктора базового класу.

5. Передача аргументів для конструкторів базового та похідного класів

Розглядаючи поняття успадкування, необхідно звернути увагу на особливості передачі аргументів для конструкторів похідного та базового класів. Коли ініціалізація виконується тільки в похідному класі, аргументи передаються звичайним шляхом. Проте, під час передавання аргумента конструктору базового класу виникають деякі складності. Передусім усі необхідні аргументи базового та похідного класів передаються конструктору похідного класу. Потім завдяки розширеній формі оголошення конструктора похідного класу відповідні аргументи передаються далі в базовий клас.

Синтаксис передавання аргументів з похідного в базовий клас такий:

```
ім'я_похідного_класу ( список_арг1 ) :
                    ім'я_базового_класу ( список_арг2 )
{
  ... //тіло конструктора похідного класу
}
```

Для базового та похідного класів допустимо використовувати одні й ті самі аргументи. Для похідного класу допустимим є також ігнорування всіх аргументів та передача їх напряму в базовий клас.

У більшості випадків конструктори базового та похідного класів не використовують один і той самий аргумент. Тоді у разі потреби передавання кожному конструктору класу одного або більше параметрів слід передати конструктору похідного класу всі аргументи, необхідні конструкторам цих двох класів. Потім конструктор похідного класу просто передає конструктору базового ті аргументи, які йому потрібні.

Приклад 2. Передавання одного аргумента конструктору похідного класу, а другого — конструктору базового:

```
#include <iostream.h>
class base {
      int i;
    public:
      base(int n) {
          cout << ”Робота конструктора базового класу\n”
          i = n;
     }
  ~base() {cout<<”Робота деструктора базового
                                                  класу\n”;}
  void showi() {cout << i << ’\n’; }
}
class derived : public base  {
      int j;
    public:
      derived(int n, int m) : base(m)
      // Передача аргумента в базовий клас
      {
        cout << ”Робота конструктора похідного класу\n”;
        j = n;
       }
      ~derived() { cout << ”Робота деструктора похідного
                                                    класу\n”;}
       void showj() {cout << j << ‘\n’;}
};
void main()
{
   derived o(10,20);
   o.showi();
   o.showj();
}
```

Якщо похідному класу деякий аргумент не потрібен, він його ігнорує і просто передає в базовий клас.

6. Множинне успадкування

Є два способи, завдяки яким похідний клас може успадковувати більше, ніж один базовий клас. По-перше, похідний клас може використовуватися як базовий для іншого похідного класу, створюючи багаторівневу ієрархію класів. В

цьому випадку вихідний базовий клас є **непрямим** (inderect) **базовим класом** для іншого похідного класу. По-друге, похідний клас може прямо успадковувати більше, ніж один базовий клас. У такій ситуації комбінація двох або більше базових класів допомагає створенню похідного класу.

Коли клас використовується як базовий для похідного, який, у свою чергу, є базовим для іншого похідного класу, конструктори цих трьох класів викликаються в порядку успадкування. Деструктори викликаються у зворотному порядку.

Якщо похідний клас напряму успадковує множину базових класів, використовується таке розширене оголошення:

```
class ім'я_похідного_класу :
            спец_доступу1  ім'я_базового_класу1,
            спец_доступу2  ім'я_базового_класу2,
            . . .
            спец_доступуN  ім'я базового_класуN
 {
   ... // тіло класу
  }
```

Тут *ім'я_базового класу1, ... , ім'я_базового класуN* — імена базових класів, *спец_доступу* — специфікатор доступу, який може бути різним для кожного базового класу. Коли успадковується множина базових класів, конструктори використовуються зліва направо у порядку, що задається в оголошенні похідного класу. Деструктори виконуються у зворотному порядку.

Коли клас успадковує множину базових класів, конструкторам яких необхідні аргументи, похідний клас передає ці аргументи, використовуючи розширену форму оголошення конструктора похідного класу:

```
ім’я_похідного_класу ( список_арг ):
            ім’я_базового_класу1( список_арг1 ),
            ім’я_базового_класу2( список_арг2 ),
                        . . .
            ім’я_базового_класуN( список_аргN )
{
 ... // тіло конструктора похідного класу
}
```

Коли похідний клас успадковує ієрархію класів, кожний похідний клас повинен передавати попередньому базовому класу по ланцюжку необхідні аргументи.

7. Віртуальні базові класи

У разі багаторазового прямого успадкування похідним класом одного і того самого базового класу може виникнути проблема. Розглянемо такий варіант (рис. 1).

Тут базовий клас *Базовий* успадковується похідними класами *Похідний1* і *Похідний2*. Похідний клас *Похідний3* прямо успадковує ці класи. Це означає, що клас *Базовий* фактично успадковується класом *Похідний3* двічі — через класи *Похідний1* і *Похідний2***.** Однак, якщо член класу *Базовий* буде використовуватись у класі *Похідний3*, це зумовить неоднозначність, оскільки в ньому буде дві копії класу *Базовий***.** Для попередження такої ситуації в C++ включений механізм віртуального базового класу (virtual base class). У цьому випадку перед специфікатором доступу базового класу необхідно поставити ключове слово *virtual*, наприклад:

```
class derived2: virtual public base
```

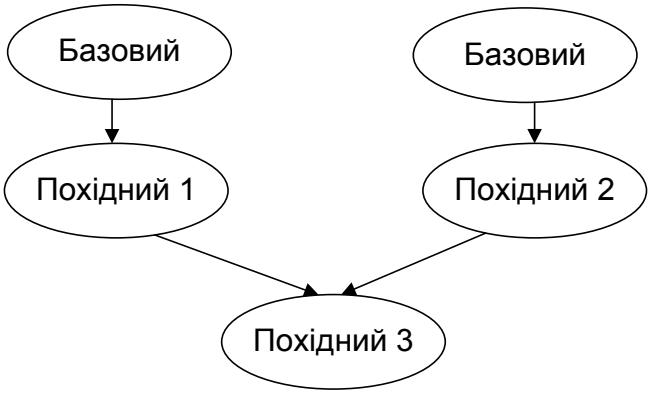

Рис. 1. — Багаторазове успадкування одного класу

Порядок виконання роботи

1. Створити у програмі клас *Rahunok*, який зберігатиме такі дані: унікальний номер рахунка, суму на рахунку (початкове значення задати при створенні рахунка), нарахування відсотків.

2. Додати до класу функції: конструктора з параметрами, відображення рахунка, встановлення додаткових відсотків, перерахування грошей з урахуванням відсотків.

3. Створити у програмі клас *Vkladnyk,* який зберігатиме такі дані: серію та номер паспорту, прізвище, ім'я та по батькові.

4. Додати до класу конструктора з параметрами, функцію відображення даних про вкладника.

5. Створити у програмі похідний клас *RahunokVkladnyka* від попередніх двох класів. Цей клас має містити дані про максимальну суму, яку можна зняти за добу, мінімальний залишок на рахунку після зняття грошей з рахунка, пароль, а також функції: конструктор з параметрами, функцію виведення інформації про рахунок вкладника, функцію зміни пароля, функцію вкладення грошей, функцію зняття грошей. Урахуйте необхідну політику конфіденційності та безпеки даних банку.

6. У головній частині програми продемонструйте роботу створеної ієрархії класів на прикладі такого об'єкта:

```
RahunokVkladnyka Geits(123456789, "Bill Geits",
                       "EC123456", 50000, 500, 500);
```

Контрольні завдання та запитання

1. Поясніть принцип ООП — успадкування. Наведіть приклади.

2. Які ви знаєте специфікатори доступу?

3. Що відбувається з відкритими та закритими членами базового класу, якщо базовий клас успадковується як відкритий похідним?

4. Що відбувається з закритими та відкритими членами базового класу, якщо базовий клас успадковується як закритий похідним?

5. Поясніть, навіщо потрібна категорія захищеності *protected*? Розгляньте два випадки: коли *protected* використовується всередині описання класу та під час успадкування.

6. Якщо один клас успадковується іншим, яким порядок виклику конструкторів та деструкторів? Наведіть приклади.

7. Що таке множинне успадкування? Які ви знаєте можливі варіанти множинного успадкування?

МОДУЛЬ 2

ЛАБОРАТОРНА РОБОТА 2.1

Проектування і опрацювання програми з віртуальними функціями

Мета роботи — засвоєння поняття віртуальної функції; набуття практичних навичок їх оголошення та використання.

Основні завдання роботи

Розробити та реалізувати програмно код, у якому демонструється робота віртуальної функції. Для цього виконати такі завдання:

1. Розробити і реалізувати програмно клас “Число”.
2. Розробити ієрархію класів: *число* (базовий), *матриця* (похідний).
3. Розробити та реалізувати програмно віртуальну функцію, яка обчислює факторіал числа, заданого в базовому класі, і використовується похідним класом, що містить масив цілих чисел, факторіали яких слід підрахувати та вивести у таблицю.

Основні теоретичні відомості

1. Вказівники на базові класи

Основою практичного застосування віртуальних функцій та динамічного поліморфізму є вказівники на базові класи.

Важливою особливістю вказівників у C++ є те, що вказівник, оголошений як вказівник на базовий клас, також може використовуватися як вказівник на будь-який клас, похідний від цього базового. У такій ситуації такі оператори є правильними:

```
base *p; // вказівник базового класу
base base_ob; // об’єкт базового класу
derived derived_ob; // об’єкт похідного класу
p = &base_ob; // p вказує на об’єкт базового класу
p = &derived_ob; // p вказує на об’єкт похідного класу
```

Указівник на похідний клас неможливо використати для доступу до об’єктів базового класу.

2. Віртуальні функції

Віртуальна функція (virtual function) є функцією-членом класу, оголошується всередині базового класу та перевизначається в похідному класі. Для створення віртуальної функції перед оголошенням функції ставиться ключове слово

virtual. Якщо клас, що містить віртуальну функцію, успадковується, то в похідному класі віртуальна функція перевизначається.

Для перевизначення віртуальної функції в похідному класі ключове слово *virtual* не потрібне.

Віртуальна функція може викликатися так само, як і будь-яка інша функція-член класу. Проте більш цікавим використанням віртуальної функції є її виклик через вказівник, завдяки чому підтримується динамічний поліморфізм. Якщо вказівник базового класу вказує на об'єкт похідного класу, який містить віртуальну функцію і для якого віртуальна функція викликається через цей вказівник, то компілятор визначає, яку версію віртуальної функцію викликати на основі типу об'єкта, на який вказує вказівник. Цей процес є реалізацією принципу динамічного поліморфізму. Клас, що містить віртуальну функцію, називають **поліморфним класом** (polymorphic class).

Приклад. Використання віртуальної функції:

```
#include <iostream.h>
class base {
  public:    int i;
     base(int x) {i=x;}
     virtual void func()
     {    cout<<"Використання базової версії func():";

       cout<<i<<'\n'; }
};
class derived1 : public base {
    public:
       derived1(int x) : base(x) { }
       void func() {
        cout<<"Використання версії func() похідного класу
                                                    derived1:";
        cout<<i*i<<'\n'; }
};
class derived2 : public base {
   public:
      derived2(int x) : base(x) { }
      void func() {
       cout<<"Використання версії func() похідного
```

```
        класу derived2:";
        cout<<i+i<<'\n'; }
};
int main()
{
  base *p;
  base ob(10);
  derived1 d_ob1(10);
  derived2 d_ob2(10);
  p=&ob;
  p->func(); // Використання базової версії func()
  p=&d_ob1;
  p->func(); // Використання версії func() похідного
             // класу derived1
  p=&d_ob2;
  p->func(); // Використання версії func() похідного
           // класу derived2
  return 0;
}
```

Ця програма виводить на екран такий запис:

```
Використання базової версії func( ):10
Використання версії func( ) похідного класу derived1:100
Використання версії func( ) похідного класу derived2:20
```

Віртуальні функції мають ієрархічний порядок успадкування. Окрім цього, якщо віртуальна функція не перевизначається в похідному класі, то використовується її реалізація, задана в базовому класі.

3. Чисто віртуальні функції

Іноді, коли віртуальна функція оголошується в базовому класі, вона не виконує ніяких значущих дій. Це звичайна ситуація, оскільки часто базовий клас не визначає закінченого типу. Замість цього він просто вміщує базовий набір функцій-членів та змінних, для яких похідний клас задає все, чого не вистачає. Коли у віртуальній функції базового класу не виконується значуща дія, у реалізації будь-якого похідного класу ця функція має обов'язково перевизначатись. Для

реалізації цього положення C++ підтримує **чисто віртуальні функції** (pure virtual function).

Чисто віртуальні функції не визначаються в базовому класі. В нього включаються тільки прототипи цих функцій. Для чисто віртуальної функції використовується така загальна форма:

```
virtual тип_даних_що_повертається_функцією
                    ім'я_функції (список_параметрів) = 0;
```

Прирівняння функції до нуля — це повідомлення компілятору, що в базовому класі не існує тіла функції. Якщо функція задається як чисто віртуальна, то вона має перевизначатися в кожному похідному класі. Якщо цього немає, то під час компіляції виникає помилка. Таким чином, створення чисто віртуальних функцій гарантує, що похідні класи забезпечать їх перевизначення.

Якщо клас має хоча б одну чисто віртуальну функцію, то це буде **абстрактний клас** (abstract class). Оскільки в абстрактному класі є хоча б одна функція, яка не має тіла, технічно цей клас є неповним і жодного об'єкта цього класу створити не можна. Таким чином абстрактні класи можуть бути тільки успадкованими. Можна створювати вказівники абстрактного класу, завдяки яким досягається динамічний поліморфізм.

Порядок виконання роботи

1. Створити у програмі клас *Chislo*. В цьому класі задати закриту змінну у вигляді цілого додатного числа (типу *long*).

Визначити конструктор і деструктор цього класу.

Визначити віртуальну функцію *factorial()*, яка має обчислювати і повертати факторіал числа, яке передається йому як параметр.

2. Створити похідний клас *Matrix*, який успадковує *Chislo*. Цей клас в розділі *public* має містити одномірний масив 100 цілих додатних чисел, а успадковуване ціле число (вводиться з клавіатури) від базового класу має сенс розмірності цього масиву.

Визначити конструктор і деструктор цього класу.

Віртуальну функцію у похідному класі не перевизначати. Проаналізуйте отриманий результат.

3. Числа масиву вводяться з клавіатури користувачем. У програмі написати функцію обчислення факторіала кожного числа масиву тієї розмірності, яка зберігається у змінній базового класу.

4. Продемонструвати роботу створених функцій в основній частині програми. Використати вказівники на базові класи.

Контрольні завдання та запитання

1. Що таке віртуальна функція?
2. Які функції не можуть бути віртуальними?
3. Яка відмінність між віртуальною та перевантаженою функціями?
4. Як віртуальні функції допомагають реалізувати динамічний поліморфізм?
5. Що таке чисто віртуальна функція? Які її відмінності від звичайної віртуальної функції?
6. Що таке абстрактний клас? Що таке поліморфний клас?
7. Чи успадковується віртуальність?
8. У чому перевага динамічного поліморфізму?
9. Динамічний поліморфізм досягається за допомогою _____ функцій та вказівників на _____ класу. Впишіть пропущені слова.

ЛАБОРАТОРНА РОБОТА 2.2

Проектування і опрацювання програми з родовими функціями і родовими класами

Мета роботи — засвоєння поняття родової функції та засобу її реалізації — шаблону функції; засвоєння поняття родового класу та засобу його реалізації — шаблону класу.

Основні завдання роботи

Розробити і реалізувати програмно шаблонний клас для подання розріджених одновимірних масивів. Розмір логічного масиву передавати через аргумент конструктора. Для цього виконати такі завдання:

1. Розробити і реалізувати програмно структуру класу розрідженого одновимірного масиву.
2. Реалізувати зберігання даних будь-якого типу *T* у створеному класі.
3. Передбачити у класі родовий конструктор за замовчуванням, родовий конструктор копіювання і операцію присвоєння.
4. Реалізувати у класі операцію індексування, що повертає посилання на знайдений елемент у масиві. Якщо елемент із заданим індексом не знайдений, то операція повинна створити новий елемент з цим індексом і розмістити його у масив.
5. В основній частині програми продемонструвати використання створеного класу.

Основні теоретичні відомості

1. Родові функції

Родова функція (generic function, template function) визначає базовий набір операцій, які будуть застосовуватися до різних типів даних. Родова функція оперує з тим типом даних, який вона отримує як параметр. За допомогою цього механізму одна й та сама процедура може застосовуватись до різних даних.

Родова функція створюється за допомогою ключового слова *template* (шаблон). Наведемо типову форму визначення родової функції:

```
template <class Ttype>тип_даних_що_повертає_функція
                          ім'я_функції(список_параметрів)
{
. . . // Тіло функції
}
```

Тут на місці *Ttype* вказується тип даних, що використовується функцією. Це — фіктивне ім'я, яке компілятор автоматично замінює іменем реального типу даних під час створення конкретної версії функції.

Іноді родову функцію називають функцією-шаблоном (template function), а конкретну версію цієї функції — **породженою функцією** (generated function), процес генерації породженої функції називають створенням екземпляра (instantiating) функції.

2. Родова функція декількох родових типів

За допомогою оператора *template* можна визначити більше одного родового типу даних, відокремлюючи їх комами, тобто:

```
template <class Ttype1, class Ttype2>
тип_даних_що_повертає_функція
                              ім'я_функції(список_параметрів)
{
. . . // Тіло функції
}
```

3. Родові класи

Родовий клас (template class, generic class) — це клас, у якому визначені всі алгоритми цього класу, а фактичні типи даних, що обробляються, задаються як параметри пізніше під час створення об'єктів цього класу.

Загальна форма оголошення родового класу така:

```
template <class Ttype> class ім'я_класу {
...//Тіло класу
}
```

Тут *Ttype* — це ім'я фіктивного типу даних, який буде задано під час створення екземпляра класу. У разі потреби можна визначити більше одного родового типу даних, що розділяються комами.

Після створення родового класу можна створити конкретний екземпляр цього класу за допомогою такої форми:

```
ім’я_класу <type> об’єкт;
```

Тут *type* — це ім’я фактичного типу даних, з якими буде оперувати клас.

Функції-члени родового класу автоматично стають родовими. Для них не обов’язково явно задавати ключове слово *template*.

Створенням родового (параметризованого) класу створюється ціла сім’я родинних класів, які можна застосувати до будь-якого типу даних.

4. Контейнерні класи

Найбільшого застосування шаблони класів набули для створення контейнерних класів.

Контейнерними класами (контейнерами) у загальному випадку називаються класи, в яких зберігаються організовані дані.

Найчастіше використовуються контейнери таких типів: обмежений (“захищений”) масив, черга, стек, зв’язаний список, бінарне дерево. Кожний з цих контейнерів виконує конкретні операції збереження та вилучення інформації, отримання запитів.

5. Приклади параметризованих контейнерних класів

Зв’язаний список. Зв’язаний список допускає гнучкі методи доступу, оскільки кожний елемент даних має посилання на наступний елемент даних у ланцюжку даних. Для зв’язаних списків операція вилучення елемента не призводить до його знищення зі списку та втрати його даних. Для фактичного знищення елемента зв’язаного списку треба визначити спеціальну операцію знищення.

Зв’язані списки можуть мати одиночні або подвійні зв’язки. У списку з одиночними зв’язками (single linked list) кожний інформаційний елемент має посилання на наступний елемент даних, у списку з подвійними зв’язками (double linked list) кожний елемент має посилання на попередній та наступний елементи.

Черга. Черга являє собою лінійний список, доступ до елементів якого здійснюється за принципом FIFO (first in, first out) таким чином, що першим у черзі знищується той елемент, який був розміщений там першим, потім — елемент, що був розміщений в черзі другим, і так далі. Для черги цей метод доступу і зберігання є єдиним. На відміну від масиву довільний доступ до вказаного елемента не допускається.

Принцип роботи з чергою оснований на роботі двох функцій: *qstore()* і *qretrive()*. Функція *qstore()* розміщує елемент у кінець черги, а функція *qretrive()* вилучає з черги перший елемент і повертає його значення. Результати виконання послідовності таких операцій наведено в табл. 2.

Таблиця 2

Операції з чергою

Операція	Уміст черги
qstore(A)	A
qstore(B)	AB
qstore(C)	ABC
qretrive(), повертає A	BC
qstore(D)	BCD
qretrive(), повертає B	CD
qretrive(), повертає C	D

Операція вилучення знищує елемент з черги, і дані, що зберігалися в цьому елементі, будуть зруйновані, якщо тільки він не буде збережений ще десь. Таким чином, якщо з черги вилучити всі елементи, вона виявиться порожньою.

Стек. Стек за змістом протилежний черзі, оскільки використовує протилежний за змістом метод доступу, що називається LIFO (last in, first out).

Стеки широко використовуються в системному програмному забезпеченні, у тому числі компілятори та інтерпретатори. Фактично компілятори C++ використовують стек для передачі аргументів функціям.

Дві базові операції зі стеком — розміщення та вилучення — традиційно називаються *push* і *pop* відповідно. Тому для

реалізації стеку, знадобляться дві функції: *push()* (яка розміщує елемент у стек) і *pop()* (яка вилучає елемент зі стека). Окрім цього, для реалізації стека знадобиться ділянка пам'яті, яка буде використовуватися як стек. Для цієї мети можна використати масив, а можна виділити пам'ять за допомогою *new*. Функція вилучення, як і у випадку з чергою, вилучає елемент зі списку і знищує його вміст (якщо тільки цей елемент не зберігається ще десь у пам'яті). Приклад роботи стеку наведено в табл. 3.

Таблиця 3

Операції зі стеком

Уміст стека	Операція
push(A)	A
push(B)	BA
push(C)	CBA
pop() вилучає C	BA
push(F)	FBA
pop() вилучає F	BA
pop() вилучає B	A
pop() вилучає A	Стек порожній

Порядок виконання роботи

Згідно з умовами завдання потрібно розробити дуже примітивний клас з мінімальною функціональністю. У реальних додатках такий клас буде містити додатково інші, складніші методи, наприклад, видалення з фізичного масиву елемента із заданим індексом.

1. За умовами завдання кожний елемент фізичного масиву повинен містити два поля: логічний індекс елемента і його значення. Тому написання програмного коду потрібно почати з розроблення класу для подання одного елемента фізичного масиву. Цей клас має бути шаблонним.

2. Необхідно вирішити, яку структуру обрати для збереження фізичного масиву. Зазвичай використовують лінійні списки, бінарні дерева або структури із хешуванням індексів.

Лінійний список має найгірші показники за часом пошуку інформації із заданим ключем (індексом), однак є найбільш простим для програмування.

Можна використати контейнерний клас *list* із бібліотеки шаблонів *STL,* який реалізований у вигляді двозв'язного списку, кожний елемент якого містить посилання на попередній та наступні елементи. Для використання цього класу слід підключити заголовний файл <*list*>.

У класі *list* є конструктор за замовчуванням, який створює клас нульової довжини. У кінець списку можна додати елемент за допомогою функції *push_back*(). Доступ до будь-якого елемента списку здійснюється через ітератор — змінну типу *list<T>::iterator*. Ітератор можна розглядати як вказівник на елемент списку. Він використовується для перегляду списку у прямому чи зворотному напрямках. У першому випадку використовується операція інкремента, у другому — декремента.

У класі *list* також є два методи, що допомагають переглянути список: *begin*() повертає вказівник на початковий елемент, *end*() повертає вказівник на елемент, що слідує за останнім. Поточне значення ітератора у циклі порівнюється із значенням, отриманим від *end*(), за допомогою операції "!=", оскільки за довільного розміщення в пам'яті сусідніх елементів операція "<" для адрес елементів втрачає сенс.

Приклад. Використання класу *list*:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
   list <char> v1;

   v1.push_back('A');
   v1.push_back('B');
   v1.push_back('C');

   list<char>:: iterator i = v1.begin();
   list <char>:: iterator n = v1.end();
```

```
    for (i; i != n; ++i)
      cout << *i << ' ';
            //вміст комірки пам'яті, на яку вказує i
      cout << endl;
    return 0;
}
```

3. Створити клас-шаблон для подання розрідженого масиву, передбачивши поле для зберігання фізичного масиву елементів типу *list*.

4. Додати у клас-шаблон перевантажену операцію індексування *operator*[]().

5. Додати функцію *show*(), яка виводитиме дані про елемент списку.

6. У головній частині програми створити об'єкти заданого класу для різних типів даних (наприклад, *double*, *list*).

7. Продемонструйте роботу створених функцій в основній частині програми для різних типів даних: цілочислового, з плаваючою крапкою, подвійної точності.

Контрольні завдання та запитання

1. Що таке родова функція?
2. Яка загальна форма родової функції?
3. Чи може родова функція мати кілька родових аргументів?
4. Яка загальна форма родової функції з кількома родовими аргументами?
5. Яка відмінність між родовою та перевантаженою функціями?
6. Яку версію генерує компілятор за наявності в програмі явно перевантаженої і родової функцій)?
7. Що таке родовий (параметризований) клас?
8. Яка загальна форма родового класу?
9. Яка загальна форма родового класу з кількома родовими типами даних?
10. Що таке параметричний поліморфізм? Завдяки чому він підтримується?
11. Що таке контейнери? Наведіть приклади.
12. Які переваги використання родових класів?

13. Яка структура зв'язного списка? Яке його застосування?

14. Як організовано чергу? Які застосування черги ви знаєте?

15. Як організовано стек? Які застосування стека ви знаєте?

ЛАБОРАТОРНА РОБОТА 2.3

Проектування і опрацювання програми з обробленням виняткових ситуацій

Мета роботи — засвоєння поняття виняткової ситуації та засобів її перехоплення; набуття практичних прийомів перехоплення виняткової ситауції.

Основні завдання роботи

1. Набрати та запустити на виконання програмний код. Проаналізувати його.

```
class Vect {
public:
     Vect(char);
     ~Vect() {
        delete [] p; }
     int& operator [] (int i) { return p[i]; }
     void Print();
  private:
     int* p;
     char size;
};

Vect::Vect(char n) : size(n)
{
  p = new int[size];
  if (!p) {
```

```
    cerr << "Error of Vect constructor" << endl;
    return ;
  }
  for(int i=0;i<size;++i) p[i]=int();
}

void Vect::Print()
{
  for (int i = 0; i < size; ++i)
    cout<< p[i]<< " ";
    cout<< endl;
}

int main() {
     Vect a(3);
     a[0]=0;
     a[1]=1;
     a[2]= 2;
     a.Print();
     Vect a1(200);
     a1[10] = 5; a1.Print();
     return 0;
}
```

2. Реалізувати програмно перехоплення виявлених помилок у коді.

3. Реалізувати програмно перехоплення перериванньь у конструкторі та деструкторі. Реалізуйте програмно оброблення виняткових ситуацій.

Основні теоретичні відомості

Виняткові ситуації — виникнення непередбачених помилкових умов (наприклад, ділення на нуль в операціях з плаваючою точкою). Як правило, ці умови завершають програму користувача із системним повідомленням про помилку. C++ дає програмісту можливість відновити програму за цих умов та продовжити її виконання. Це вбудований механізм, який називається **обробленням виняткових ситуацій** (*exception handling*).

Виняткові ситуації в С++ оброблюються за допомогою трьох ключових слів: *try, catch* та *throw*. Оператори програми, під час виконання яких потрібно забезпечити оброблення виняткових ситуацій, розміщуються в блоці *try.* Якщо виняткова ситуація (тобто помилка) виникає всередині блока *try*, вона генерується за допомогою оператора *throw.* Перехоплюється та обробляється виняткова ситуація за допомогою оператора *catch,* що розміщений безпосередньо за блоком *try.* З блоком *try* може бути зв'язано більше одного оператора *catch.* Блок *try* може вміщувати як декілька операторів всередині однієї функції, так і всі оператори функції *main().*

Загальна форма операторів *try* і *catch* така:

```
try
{
… //блок try
}
catch(type1 arg)
{
… //блок catch
}
catch(type2 arg)
{
… //блок catch
}
...
catch(typeN arg)
{
…//блок catch
}
```

Як видно, з блоком *try* може бути зв'язано більше одного оператора *catch.* Те, який конкретно оператор *catch* буде виконуватися, залежить від типу виняткової ситуації. Тобто, якщо тип даних, указаний в операторі *catch,* відповідає типу виняткової ситуації, то виконується поточний оператор *catch.* Усі інші оператори *catch* ігноруються. Якщо виняткова ситуація перехоплена, аргумент *arg* отримує її значення. Можна

перехоплювати будь-які типи даних, включаючи і створені класи.

Загальна форма оператора *throw* така:

```
throw виняткова_ситуація;
```

Оператор *throw* повинен виконуватися або всередині блока *try,* або в будь-якій функції, яку цей блок викликає (прямо чи непрямо). У загальній формі виняткова_ситуація – це виняткова ситуація, що генерується оператором.

Приклад 1. Оброблення виняткової ситуації типу *int*.

```
#include <iostream.h>
main()
{
  cout << "Початок\n";
  try
  { //Початок блока try
    cout << "Всередині блока try\n";
    throw 10; //Генерація помилки
    cout << "Це виконано не буде";
  }
  catch (int i)
  { // Перехоплення помилки
    cout << "Перехоплена помилка номер: ";
    cout << i << "\n";
  }
  cout << "Кінець";
  return 0;
}
```

Якщо в цьому прикладі замінити тип даних в операторі *catch* на *double,* то виняткова ситуація не буде перехоплена, і програма завершиться з помилкою (abnormal program termination).

Виняткову ситуацію можна викликати з оператора, що не входить у блок *try*, якщо сам цей оператор входить у функцію, яка викликається з блока *try*. Блок *try* можна також розміщувати

всередині функції. В цьому випадку з кожним викликом функції обробник виняткової ситуації встановлюється знову.

З блоком *try* можна зв'язати більше одного оператора *catch*. Як правило так і робиться. Проте кожний оператор *catch* призначається для перехоплення свого типу виняткової ситуації.

Приклад 2. Перехоплення двох виняткових ситуацій для цілих і однієї виняткової ситуації для рядка.

```
#include <iostream.h>
//Перехоплення різних типів виняткових ситуацій
void Xhandler(int test)
{
 try
 {
   if(test) throw test;
   else throw "Значення дорівнює нулю";
 }
 catch(int i)
 {
  cout << "Перехоплена помилка номер: " << i << '\n';
 }
 catch(char *str)
 {
   cout << "Перехоплений рядок: ";
   cout << str << '\n';
 }
}

main()
{
  cout << "Початок\n";
   handler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);
  cout << "Кінець";
  return 0;
}
```

У деяких випадках необхідно так налаштувати оброблення виняткових ситуацій, щоб можна було перехопити всі виняткові ситуації, незалежно від їх типу. Для цього використовується така форма оператора *catch* ("..." є обов'язковими в оголошенні *catch(...)*):

```
catch(…)
{
... //Обробка всіх виняткових ситуацій
}
```

Дуже зручно оператор *catch(...)* використовувати як останній в групі операторів *catch.* Він за замовчуванням стає оператором, який "перехоплює все".

Використання оператора *catch(…)* — це гарний спосіб перехоплювати ті виняткові ситуації, які явно не обробляються. Крім того, перехопленням усіх виняткових ситуацій попереджається аварійне завершення програми через необроблену виняткову ситуацію.

Порядок виконання роботи

У програмі із завдання реалізовано клас *Vect*, призначений для створення і використання одновимірних масивів типу *int* довільного розміру. Розмір масиву не перевищує число 256. Проект розроблявся для "заліза" з обмеженням ємності оперативної пам'яті. Для зберігання у класі *Vect* інформації про розмір масиву програміст використав поле *char size*, попокладаючись на те, що для типу *char* виділяється один байт, а діапазон можливих значень для восьмирозрядного двійкового числа становить 0...255.

Якщо протестувати програму, то на екрані з'явиться запис:

```
0 1 2
Error of Vect constructor
```

1. Виконати покрокове виконання програми. Воно покаже, що помилка з'являється при спробі виконати оператор *a1[10] = 5*. Повідомлення про помилку виводиться тоді, коли операція

new не могла виділити пам'ять і повернула нульове значення вказівника *p*.

Тип *char* – це скорочення типу *signed char*, у якому старший біт використовується для зображення знака числа. Тому діапазон поданих чисел для типу *char* становить 128...+127.

Як тоді інтерпретується десяткове число 200? Перетворивши його у шістнадцяткове, отримаємо число C8 у двійковому еквіваленті 11001000. Старший розряд цього числа дорівнює 1. Це означає, що комп'ютер сприйме його як -1001000.

Нагадаймо, що від'ємні значення зберігаються у вигляді додаткового коду, який отримується інверсією усіх розрядів з наступним додаванням 1. Тому після оберненого перетворення слід відняти 1, а потім проінвертувати усі розряди. У результаті отримаємо число -56.

Отже, помилка міститься у реалізації класу. Отримавши від'ємне число як ємність запитуваної пам'яті, операція *new* повертає 0.

2. Потім слід виправити помилку, не змінюючи тип. Для цього необхідно змінити код конструктора класу, а також основну частину програми.

3. Найважливіша вимога до деструктора: жодне із виключень, яке могло б з'явитись у процесі роботи деструктора, не повинне залишити його межі.

Щоб виконати цю вимогу, слід дотримуватись таких двох правил:

1) ніколи не генерувати виключення у тілі деструктора за допомогою *throw*;

2) якщо дії, що виконуються з деструктурованим об'єктом складні і пов'язані з викликом інших функцій, то рекомендується інкапсулювати усі ці дії в деякому методі, наприклад *Destroy*(), і викликати цей метод з використанням блока try/catch.

5. Додати оброблення виняткових ситуацій у деструктор заданого класу.

6. Продемонструвати роботу програми з оброблення виняткових ситуацій.

Контрольні завдання та запитання

1. Що таке виняткова ситуація?
2. Запишіть загальні форми операторів *try*, *catch* і *throw*. Опишіть їх функції.
3. Як сумісна робота операторів *try, catch* і *throw* забезпечує в C++ оброблення виняткових ситуацій?
4. Чи можна використовувати *throw*, якщо виконання програми не проходить через блок *try*?
5. Яка форма оператора *catch* буде обробляти всі типи виняткових ситуацій?
6. Яким буде результат генерації виняткової ситуації, для якої не задано відповідного оператора *catch*?

ЛАБОРАТОРНА РОБОТА 2.4

Проектування і опрацювання програми з власними маніпуляторами і власними функціями введення-виведення

Мета роботи — засвоєння загальних принципів і набуття практичних навичок роботи із системою введення-виведення; набуття практичних навичок створення власних функцій введення-виведення та власних маніпуляторів.

Основні завдання роботи

Реалізувати програмний додаток відображення графічних залежностей за допомогою даних, наведених у файлі. Для цього виконати такі завдання:

1. Розробіть та реалізуйте програмно клас функції.
2. Розробіть структуру і створіть файл, у якому зберігаються дані про математичну функцію, представлену значеннями по осі абсцис і ординат.

3. Реалізуйте у програмі знаходження та відображення на графіку мінімального і максимального значень серед усіх з файлу.
4. Реалізуйте програмно функцію запису значень зі створеного файлу до нового файлу у певному форматі, за якого функція замінює пробіли на табуляцію, кожне число записує в нотації з фіксованою точкою, з трьома знаками після коми, кожне числове значення займає 10 позицій (ширина поля), цілі числа мають бути виведені у вигляді “+10.000”.

Основні теоретичні відомості

1. Базові положення системи введення-виведення С++

Система введення-виведення С++ діє через потоки (streams).

Потік — це логічний пристрій, який видає та приймає інформацію. Потік зв’язаний з фізичним пристроєм за допомогою системи введення-виведення С++.

Коли починається програма мовою С++, автоматично відкриваються чотири потоки (інформацію про них наведено в табл. 4).

Таблиця 4

Стандартні потоки у С++

Потік	Значення	Пристрій за замовчуванням
cin	Стандартний ввод	Клавіатура
cout	Стандартний виведення	Екран
cerr	Стандартна помилка	Екран
clog	Буферизована версія cerr	Екран

У С++ система введення-виведення підтримується заголовним файлом iostream.h. У цьому файлі для підтримання введення-виведення задано ієрархію класів. Клас нижнього рівня введення-виведення називається streambuf. Цей клас забезпечує базові дії з уведення-виведення та використовується переважно як базовий для інших класів. Ще один клас в ієрархії — ios. Цей клас забезпечує форматування, контроль помилок та

інформацію про стан потоків уведення-виведення. Ios використовується як базовий для трьох класів: istream, ostream та iostream. Ці класи застосовуються для створення потоків, сумісних відповідно з уведенням, виведенням та введенням-виведенням.

Клас ios має багато функцій та змінних – членів класу, які керують та контролюють основну роботу потоку.

2. Форматне введення-виведення даних

Інформацію можна виводити в широкому діапазоні форм. Кожний потік C++ зв'язаний з набором прапорів формату, які задають формат відображення даних.

Для встановлення прапора формату користуються функцією *setf()*. Ця функція є членом класу *ios*. Її типова форма:

```
long setf(long прапори);
```

Функція *setf()* повертає попередні установлення прапорів формату і встановлює задані прапори. Наприклад, для встановлення прапора *showpos* можна скористатися оператором:

```
потік.setf(ios::showpos);
```

Тут потік — це той потік, на який можна впливати; *showpos* — це константа, що входить в *enumeration* всередині класу *ios*. Тому, щоб повідомити компілятору про це, необхідно поставити перед *showpos* ім'я класу і операцію розширення діапазону бачення.

Замість кількох викликів *setf(),* можна встановлювати більш одного прапора за один виклик. Для того щоб об'єднати необхідні прапори, використовується операція OR. Наприклад,

```
cout.setf(ios::showbase | ios::hex);
```

Для скинення одного або декількох прапорів формату, використовується функція *unsetf()*:

```
long unsetf(long  прапори);
```

В *ios* також внесено функцію-член *flags()*, яка повертає поточний стан прапорів формату в змінній типу *long*.

Приклад 1. Робота функцій *setf()* і *unsetf()*:

```
#include <iostream.h>
int main()
{
   cout.setf(ios::uppercase|ios::showbase|ios::hex);
   cout << 88 << '\n';
    cout.unsetf(ios::uppercase);
    cout << 88 << '\n';
    return 0;
}
```

У програмі спочатку встановлюються прапори *uppercase, showbase* і *hex*. Потім виводиться число 88 з використанням наукової нотації. В цьому випадку шістнадцяткове "x" виводиться у верхньому регістрі. Далі *unsetf()* скидає прапор *uppercase* і знову виводиться шістнадцяткове 88. Тепер "x" буде у нижньому регістрі.

3. Використання функцій width(), precision() і fill()

Крім прапорів формату, існують три функції-члени, що визначені в класі *ios*, які визначають параметри формату: ширину поля, точність і символ заповнення. Це, відповідно, функції *width(), precision()* і *fill()*.

За замовчуванням під час виведення будь-якого значення воно займає кількість позицій, що відповідає кількості символів, що виводяться. Проте, використовуючи функцію *width()*, можна задати мінімальну ширину поля. Прототип функції такий:

```
int width(int w);
```

Тут *w* — ширина поля, а функція повертає попередню ширину поля.

Після встановлення мінімальної ширини поля, якщо значення, що виводиться, потребує меншої ширини поля, то його решта заповнюється поточним символом заповнення (за замовчуванням пробілом). Проте, якщо розмір значення, що

виводиться, перевищує мінімальну ширину поля, буде зайнято стільки символів, скільки треба.

За замовчуванням при виведенні значень з плаваючою точкою, після десяткової точки ставиться шість цифр. Використовуючи функцію *precision()*, це значення можна змінити. Прототип функції такий:

```
int precision(int p);
```

Тут *p* – це точність (еількість цифр, що виводяться після коми), сама функція повертає попередню точність.

За замовчуванням при заповненні поля використовуються пробіли. Проте можна змінити символ заповнення, використовуючи функцію *fill()*. Її прототип такий:

```
char fill(char ch);
```

Після виклику функції *fill()* змінна *ch* становиться символом заповнення, а функція повертає попередній символ заповнення.

4. Маніпулятори введення-виведення

У C++ є інший спосіб форматування інформації з використанням системи введення-виведення C++. За цього способу застосовуються спеціальні функції – маніпулятори введення-виведення (i/o manipulators). У деяких випадках маніпулятори більш зручні, ніж прапори формату та функції класу *ios*.

Маніпулятори введення-виведення — це спеціальні функції формату введення-виведення, які можуть міститися в тілі оператора введення-виведення; більшість з них діють аналогічно функціям-членам класу *ios*.

Наприклад:

```
cout<<oct<<100<<hex<<100;
cout<<setw(10)<<100;
```

Перший оператор повідомляє *cout* про необхідність виведення цілих у вісімковій системі числення, потім

виводиться значення 100 у вісімковій системі числення. Далі він повідомляє потоку про необхідність виведення цілих у шістнадцятковій системі числення і далі виводиться число 100 у шістнадцятковому форматі. У другому операторі встановлюється ширина поля 10, і потім знову виводиться 100 у шістнадцятковому форматі.

Маніпулятор введення-виведення впливає лише на потік, який є частиною виразу введення-виведення, що має маніпулятор. Маніпулятори введення-виведення не впливають на всі потоки, відкриті в поточний момент часу для використання.

4. Файлове введення-виведення

Для реалізації файлового введення-виведення необхідно включити в програму заголовний файл fstream.h. У ньому визначено декілька класів, включаючи ifstream, ofstream і fstream. Ці класи є похідними класів istream і ostream.

У C++ файл відкривається за допомогою його зв'язування з потоком. Є три види потоків: вхідний, вихідний і вхідний-вихідний. Перед тим, як відкрити файл, передусім треба створити потік. Для створення вхідного потоку необхідно оголосити потік класу *ifstream*, для створення вихідного — оголосити потік класу *ofstream*. Потоки, які реалізують і введення, і виведення, повинні оголошуватися як об'єкти класу *fstream*. Наприклад:

```
ifstream in; // введення
ofstream on; // виведення
fstream io; // введення і виведення
```

Після створення потоку, одним зі способів зв'язати його з файлом є функція *open()*. Ця функція є членом кожного з трьох початкових класів. Її прототип такий:

```
void open(char *filename, int mode, int access);
```

Тут *filename* — ім'я файлу, в який можна включити шлях. Величина *mode* задає параметри відкриття файлу. Параметр

access задає права доступу до файлу. За замовчуванням значення *access* дорівнює *filebuf::openprot* і дорівнює 0×644 для середовища UNIX, що означає звичайний клас. У середовищі DOS/WINDOWS, *access* відповідає кодам атрибутів файлів DOS/WINDOWS.

Хоча використовувати функцію *open()* для відкриття файлу в цілому правильно, на практиці частіше це не робиться, оскільки у класів *ifstream, ofstream* і *fstream* є конструктори, які відкривають файл автоматично. Конструктори мають такі самі параметри, що задані за замовчуванням у функції *open()*.

Наприклад,

```
ifstream mystream(”myfile”); //Відкриття файлу введення
```

Для закриття файлу використовується функція-член *close()*. Наприклад, щоб закрити файл, зв’язаний з потоком *mystream*, використовується оператор:

```
mystream.close();
```

Функція *close()* не має параметрів та значення, що повертається.

Використовуючи функцію-член *eof()*, можна визначити, чи був досягнутий кінець файлу під час уведення.

Прототип функції такий:

```
int eof();
```

Вона повертає ненульове значення в тому випадку, якщо досягнуто кінець файлу; в противному випадку функція повертає нуль.

Після відкриття файлу для введення-виведення інформації використовуються операції “<<” і ” >>” і зв’язаний з файлом потік.

5. Створення власних функцій вставки

Однією з переваг використання операторів уведення-виведення C++ замість аналогічних функцій введення-

виведення С є можливість перевантаження операторів уведення-виведення для створених класів.

У мові С++ виведення іноді називається **вставкою** (insertion), а оператор "<<" — оператором вставки. Зміст цього терміна полягає в тому, що операція виведення вставляє (inserts) інформацію в потік. Коли перевантажується "<<" для виведення, створюється функція вставки (inserter function) або (inserter).

У всіх функцій вставки така загальна форма:

```
ostream &operator<<(ostream &stream, ім'я_класу ob)
{
  … //Тіло функції вставки
  return stream;
}
```

Перший параметр є посиланням на об'єкт типу *ostream*. Це означає, що *stream* має бути потоком виведення. Другий параметр отримує об'єкт для виведення, він також може бути посиланням, якщо це треба. Всередині функції вставки можна виконувати будь-яку процедуру, проте відповідно до хорошого стилю програмування рекомендується обмежувати роботу власної функції вставки тільки виведенням інформації в потік.

Функція вставки не може бути членом класу, для роботи з яким її задано (лівий операнд — потік, а не об'єкт класу). Тому, щоб мати доступ до закритих членів класу, функція вставки повинна бути дружньою класу.

Приклад 2. Створення дружньої функції вставки:

```
#include <iostream.h>
class coord {
   int x, y;
 public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
friend ostream &operator<<(ostream &stream, coord ob);
};

ostream &operator<<(ostream &stream, coord ob)
```

```
{
  stream << ob.x << ", " << ob.y << '\n';
  return stream;
}
int main()
{
   coord a(1, 1), b(10, 23);
   cout << a << b;
   return 0;
}
```

Варто звернути увагу на те, що оператор уведення-виведення всередині функції виводить значення *x* і *y* в *stream*, який є довільним потоком, який передається у функцію.

6. Створення власних функцій вилучення

Оператор уведення “>>” можна перевантажувати так само, як і оператор виведення “<<”. У C++ оператор уведення “>>” називають оператором вилучення (extraction operator). Зміст цього терміна в тому, що під час уведення інформації з потоку вилучаються дані.

Загальна форма функції введення користувача така:

```
istream &operator>>(istream &stream, ім’я_класу &ob)
{
  … //Тіло функції вилучення
  return stream;
}
```

Функція вилучення повертає посилання на *istream*, який є потоком уведення. Перший параметр повинен бути посиланням на потік уведення, другий — це посилання на об’єкт, що отримує інформацію.

Функція вилучення не може бути функцією-членом. Усередині функції вилучення може виконуватися будь-яка операція, проте доцільно обмежити її роботу введенням інформації.

Приклад 3. Створення дружньої функції вилучення:

```
#include <iostream.h>
class coord {
    int x, y;
  public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
friend ostream &operator<<(ostream &stream, coord ob);
friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}
istream &operator>>(istream &stream, coord &ob)
{
  cout << "Введіть координати: ";
  stream >> ob.x >> ob.y;
  return stream;
}
int main()
{
 coord a(1, 1), b(10, 23);
 cout << a << b;
 cin >> a;
 cout << a;
 return 0;
}
```

7. Створення власних маніпуляторів

Усі маніпулятори без параметрів для виведення мають таку форму:

```
ostream &ім'я_маніпулятора(osream &stream)
{
…//Код тіла
return stream;
};
```

Усі маніпулятори без параметрів для введення мають таку форму:

```
istream &ім'я_маніпулятора(isream &stream)
{
…//Код тіла
return stream;
};
```

Порядок виконання роботи

1. Створити у програмі клас *функція.* Як закриті змінні використайте динамічні масиви, які визначають значення математичної функції по осі абсцис і ординат відповідно. Додайте у клас конструктор і деструктор, а також функції вставки і вилучення, які повинні зчитувати дані з файлу і відображати дані на екрані відповідним чином.

2. Створити на диску файл, який містить три стовпчики цифр. Перший — значення по осі абсцис, два інші — по осі ординат для двох різних математичних функцій. Числа у стовпцях розділені пробілами.

3. Створити у програмі функцію вилучення. Вона повинна зчитувати дані з файлу, при цьому враховуючи, що числа відокремлені один від одного пробілами, а також те, що перший стовпчик цифр — це значення по осі абсцис (тобто їх слід зберегти у масиві для абсцис), другий (або третій) стовпчики — значення по осі ординат (їх слід зберігати в іншому масиві, призначеному для значень ординат). Оскільки у файлі зберігається інформація для двох функцій, то слід в основній частині програми створити два об'єкти класу *функція*: один для збереження даних про одну, інший — про другу функцію з файлу.

4. Додати до програми дружню функцію визначення максимального і мінімального значення з файлу та виведення цих значень на екран.

5. Додайте у програму функцію вставки. Вона повинна відобразити на екрані графік відповідної функції на осях координат.

6. Додайте у програму функцію, що зчитує дані з файлу і записує їх до іншого таким чином: замінює пробіли на табуляцію, кожне число записує в нотації з фіксованою краточкоюпкою, з трьома знаками після коми, кожне число займає 10 позицій (ширина поля), цілі числа мають бути виведені у вигляді “+10.000”.

7. Продемонструвати роботу усіх функцій створених класів в основній частині програми, для цього оголосивши попередньо необхідні об’єкти.

Контрольні завдання та запитання

1. Які основні положення системи уведення-виведення С++?
2. Що таке потік?
3. Які потоки відкриваються під час запуску програми мовою С++? Яке їх призначення?
4. Які прапори формату ви знаєте? Яке їх призначення? Наведіть приклади.
5. Які функції, що визначають параметри формату ви знаєте? Яке їх призначення? Наведіть приклади застосування.
6. Які особливості файлового введення-виведення в С++?
7. Які функції виконують оператори вставки та вилучення?
8. Як і для чого створюються функції вставки та вилучення?
9. Яка загальна форма функції вставки?
10. Яка загальна форма функції вилучення?
11. Чи можуть функції вставки та вилучення належати класу, для якого їх створюють?
12. Які типи параметрів та який тип значення, що повертається, у функції вставки та вилучення?
13. Яка мета створення власних маніпуляторів? Наведіть їх загальну форму.
14. Наведіть декілька переваг використання функцій введення-виведення С++ порівняно із системою введення-виведення мови С.

Список рекомендованої літератури

1. Шилдт Г. Самоучитель C++ / Г. Шилдт : Пер. с англ. — 3-е изд. — СПб. : BHV-Санкт-Петербург, 2006. — 688 с.

2. Шилдт Г. Полный справочник по C++ / Г. Шилдт. — 4.изд. — М. : СПб.; К. : Издательский дом "Вильямс", 2003. — 796с.

3. Дейтел Х.М. Как программировать на C++ / Х.М.Дейтел, П.Дж.Дейтел. — 5-е изд. — М. : Бином-Пресс, 2008. — 1456 с.

4. Прата С. Язык программирования C++ / С. Прата. — 5-е изд. — М. ; СПб. ; К. : Вильямс, 2007. — 1184 с.

5. Павловская Т.А. C/C++. Структурное программирование: практикум / Т.А. Павловская, Ю.А. Щупак — СПб. : Питер, 2005. — 239 с.

6. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч. — М.: Бином, 2000. — 560 с.

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних систем

Кафедра комп'ютеризованих систем управління

ЛАБОРАТОРНА РОБОТА 1

з дисципліни

"Об'єктно-рієнтоване програмування"

Класи і об'єкти. Конструктори і деструктори

Виконав студент 308 гр. ФКС

Левитська А.О.

Прийняв проф. І.М. Коваленко

Київ 2011

ЗМІСТ

Навчальне видання

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Лабораторний практикум
для студентів
напряму 0915 "Комп'ютерна інженерія"
спеціальності 6.091500 "Системне програмування"

Укладачі: КОБА Олена Вікторівна
ПУСТОВА Світлана Вікторівна

Редактор *Р.М. Шульженко*
Коректор
Технічний редактор

Підп. до друку _____ Формат 60x84/16. Папір офс.
Офс. друк. Ум.друк.арк.___ Обл.-вид. арк.____
Тираж 100 пр. Замовлення № ____Вид. №_____

Видавництво НАУ
03058, м. Київ-58, проспект Космонавта Комарова, 1.

Свідоцтво про внесення до Державного реєстру ДК
№ 977 від 05.07.2002 р.