МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ОСНОВИ ПРОГРАМУВАННЯ НА С++

Навчальний посібник
для студентів спеціальностей 113 – Прикладна математика
та 122 – Комп`ютерні науки

Рекомендовано
редакційно-видавничою
радою університету,
протокол № 3 від 30.10.2020р.

Харків
НТУ «ХПІ»
2021

УДК 004.43
О 25

Рецензенти:

О.О. Стрельнікова, д-р. техн. наук, професор, Інститут проблем машинобудування ім. А.М. Підгорного НАН України;

Д.М. Крицький, канд. техн. наук, доцент, Національний аерокосмічний університет ім. М.С. Жуковського «Харківський авіаційний інститут»

О 25 Основи програмування на С++: Навчальний посібник для студентів спеціальностей 113 – Прикладна математика та 122 – Комп`ютерні науки: навч. посіб./ Водка О.О., Дашкевич А.О., Іванченко К.В., Розова Л.В., Сенько А.В. – Харків: НТУ «ХПІ», 2021. – 112 с.

ISBN

Посібник містить опис мови програмування С++. Наведено необхідний теоретичний матеріал з основ об`єктно-орієнтованого програмування за темами: класи, успадкування, поліморфізм, виключення, перевантаження операцій та інші. Кожен з розглянутих розділів забезпечено значною кількістю прикладів програм, та завданнями для самостійного виконання.

Призначено для студентів спеціальностей 113 – Прикладна математика та 122 – Комп`ютерні науки та інших технічних спеціальностей.

Іл. 27. Табл. 1. Бібліогр. 22 назв.

УДК 004.43

ISBN

ЗМІСТ

ВСТУП

Мова С ++ – універсальна мова програмування середнього рівня, яка може бути застосована і початківцями-програмістами, при вивченні програмування, і досвідченими фахівцями для створення серйозних програмних продуктів. Об`єктно-орієнтована реалізація мови С++, що відповідає стандартам ANSI (American national standards institute) і Міжнародної організації стандартів (International Standards Organization – *ISO*), підтримує основні концепції об`єктно-орієнтованого програмування: абстракція, інкапсуляція, успадкування, поліморфізм [1, 2]. Такий підхід у програмуванні передбачає, що програміст у процесі розробки і створення може розділяти програму на частини, такі що легко піддаються контролю, та такі, що використовують так звану абстракцію даних. Інформація при цьому міститься в деяких об`єктах типів, визначених користувачем. Такі об`єкти прості та надійні у використанні в тих ситуаціях, коли їх тип не можна встановити на стадії компіляції. Об`єктна орієнтованість С++ означає, що він підтримує стиль програмування, що спрощує кодування великомасштабних програм і забезпечує їх розширюваність. В теперішній час С++ є досить популярною мовою прораммування, використовуваним для розробки додатків будь-якого типу [3]. Компілятори мови С++ підтримують єдині стандарти і постійно доповнюються новими можливостями.

У цьому навчальному посібнику подано необхідний теоретичний матеріал з основ об`єктно-орієнтованого програмування за темами: класи, успадкування, поліморфізм, виключення, перевантаження операцій та інші, а також розглянуто роботу зі структурами даних. Кожну з розглянутих тем забезпечено значною кількістю прикладів програм мовою С++. Наведені варіанти завдань для самостійного виконання. Цей посібник призначено для студентів спеціальностей 113 – Прикладна математика і 122 – Комп`ютерні науки та інших технічних спеціальностей.

1. СТРУКТУРИ ДАНИХ. РОБОТА З БІНАРНИМИ ФАЙЛАМИ

1.1. Поняття структури даних

Ранні версії мов програмування містили тільки прості вбудовані типи даних – цілі, дійсні, логічні і та ін. (так звані прості або атомарні типи). Ці дані можна було організовувати в масиви. Для обчислювальних задач цього було достатньо, однак поступово комп`ютери стали використовувати для обробки текстів, графічних зображень, ведення баз даних і та ін.

Таким чином, різноманітність оброблюваної інформації привела до необхідності створення програмістом власних складових типів даних – структур. Структури дозволяють поєднувати різнорідні дані – числові дані, масиви, рядки, самі структури і та ін. З структур можна утворювати масиви [4].

Структура – це складовий тип даних, побудований із використанням інших типів. Структура складається з полів. Поля (елементи структури) – змінні або масиви стандартного типу (*int, char*) або інші, раніше описані структури. Оголошення структури здійснюється за допомогою ключового слова struct, за яким йде її ім`я і далі список елементів, укладених у фігурні дужки. Наприклад структура, що описує дату, може бути записана так:

```
struct date {
 int day;
 int month;
 int year;
};
```

Синтаксис оголошення змінних-структур такий самий, як і змінних інших типів. Наприклад,

```
date days;
```

Для звернення до полів структури використовується оператор **.** (точка). Наприклад, щоб записати дату 20 листопада 2019 року в змінну ***days***, слід використовувати такий підхід:

```
days.day = 20;
days.month = 11;
days.year = 2019;
```

або можливо задати початкові значення відразу при оголошенні змінної:

```
date days = {20, 11, 2019};
```

Створення масиву структур має такий самий синтаксис, як і масиву елементів атомарного типу:

```
date days_array[20]; // масив типу date з 20 елементів.
date* days_dyn_array = new date[20]; // динамічний масив.
```

Робота з покажчиком на структуру має деякі особливості: формально, для того, щоб звернутися до поля структури через покажчик, його необхідно розіменувати за допомогою операції *, а потім скористатися оператором «.» (точка) [5, 6]. Наприклад:

```
date days;
date* pdays = &days; // покажчик на структуру.
(*pdays).day = 6;
```

Звернення до поля структури шляхом використання оператора розіменування (*) і точки можна спростити, застосовуючи оператор -> (стрілка, вона складається з символу «мінус» і «більше»). Оператор стрілка включає в себе операції розіменування і точку. Таким чином, останній приклад може бути перетворений:

```
date days;
date* pdays = &days; // покажчик на структуру.
pdays->day = 6;
```

1.2.Функції бібліотеки *fstream* для роботи з бінарними файлами

Для роботи з бінарними файлами в бібліотеці *fstream* реалізовані дві функції: для *read* і *write* [2,5]:

```
ostream& write( const char * s, streamsize n );
istream& read( char * s, streamsize n );
```

Перший параметр *char * s* є покажчиком на масив даних, який необхідно записати / зчитати з файлу. Другий параметр *streamsize n* – розмір масиву в байтах, який необхідно зчитати / записати.

Так само слід зазначити, що для коректної роботи цих функцій слід відкривати файл в режимі *ios :: binary*.

Роботу зі структурами та бінарними файлами будемо розглядати за допомогою практичного прикладу, програми **«Співробітник»**, в якій:

1) користувач має можливість зчитувати і записувати з бінарного файлу інформацію:
   a) ПІБ співробітника;
   b) рік народження.
2) користувач має можливість додавати та видаляти інформацію.

```
#include <iostream>
#include <fstream>
#include <cstring>
#include <windows.h>

using namespace std;

const int maxlen = 255;
#pragma pack(push, 1) //директиви компілятору для вирівнювання полів структур.
struct sworker {
    char fio[maxlen];
    int age;
};
#pragma pack(pop)

sworker arr[maxlen];

int worker_index = 0;

int menu();
void readFromFile(const char* fileName);
```

```
void saveToFile(const char* fileName);
void addNew();
void del();

int main()
{
    setlocale(LC_ALL, "Russian");
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
    while (1) {
        switch (menu()) {
        case 1:
            readFromFile("file.dat");
            break;
        case 2:
            saveToFile("file.dat");
            break;
        case 3:
            addNew();
            break;
        case 4:
            del();
            break;
        case 5:
            return 0;
        default:
            cout << "Невірний вибір" << endl;
        }
    }
}
int menu()
{
    cout << "\n";
    int ans;
    cout << "Оберіть\n";
    cout << "1-для зчитування з файла\n";
    cout << "2-для запису в файл\n";
    cout << "3-для додавання запису\n";
    cout << "4-для видалення запису\n";
    cout << "5-для виходу\n";
    cout << "\n";
    cout << "Ваш вибір  ";
    cin >> ans;
    return ans;
}
void saveToFile(const char* fileName)
{
    ofstream f;
    f.open(fileName, ios::binary);
    f.write((char*)arr, sizeof(sworker) * worker_index);
    f.close();
    cout << "Введені дані збережено в файлі\n";
}
void readFromFile(const char* fileName)
{
```

```
    ifstream f;
    f.open(fileName, ios::binary);
    if (!f) {
        cout << "Файла не існує";
    }
    else {
        sworker worker;
        worker_index = 0;
        while (1) {
            f.read((char*)&worker, sizeof(worker));
            if (f.eof())
                break;
            arr[worker_index] = worker;
            worker_index++;
        }
        f.close();
        cout << "Дані зчитано з файлу\n";
        for (int i = 0; i < worker_index; i++) {
            cout << i + 1 << "\t" << arr[i].fio << "\t" <<
arr[i].age << endl;
        }
    }
}
void addNew()
{
    cout << "Додавання нового запису\n\n";
    cout << "Запис номер " << worker_index + 1 << "\n";
    cin.ignore();
    cout << "Введіть ПІБ  ";
    cin.getline(arr[worker_index].fio, maxlen);
    cout << "Введіть вік  ";
    cin >> arr[worker_index].age;
    worker_index++;
    cout << "\n";
    for (int i = 0; i < worker_index; i++) {
        cout << i + 1 << "\t" << arr[i].fio << "\t" << arr[i].age
<< endl;
    }
    cout << "\n";
}

void del()
{
    int d;
    cout << "Оберіть номер запису, який необхідно видалити ";
    cin >> d;
    for (int i = d - 1; i < worker_index; i++) {
        arr[i] = arr[i + 1];
    }
        worker_index--;
    cout << "\n";
    for (int i = 0; i < worker_index; i++) {
        cout << i + 1 << "\t" << arr[i].fio << "\t" << arr[i].age
<< endl;
    }
```

```
    cout << "\n";
}
```

Приклад виконання програми наведено на рис. 1.

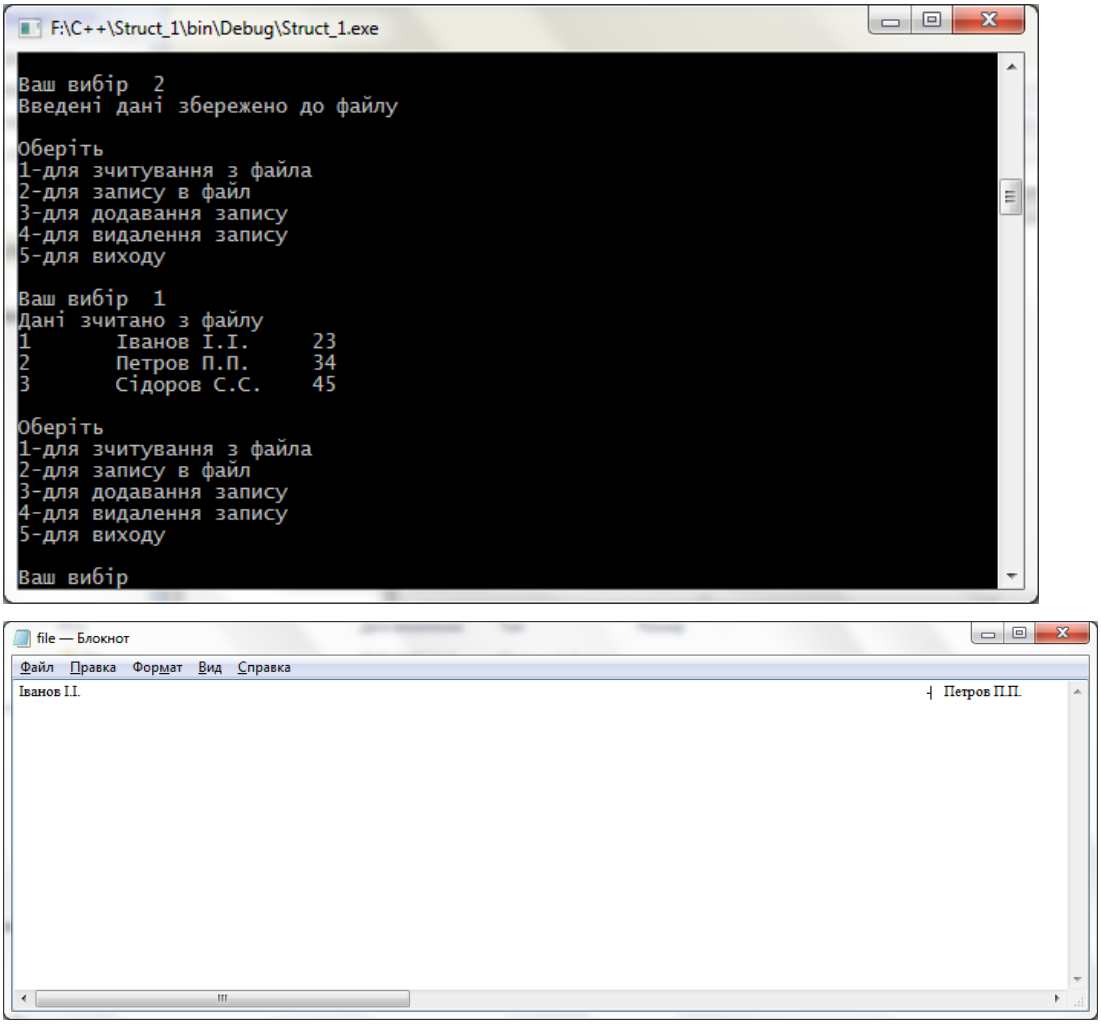


Рис. 1. Приклад роботи програми

1.3. Завдання для самостійного виконання

Написати програму, в якій:

1) користувач має можливість записувати та зчитувати з бінарного файлу інформацію про заданий об`єкт;
2) користувач має можливість додавати, видаляти, змінювати та сортувати по пункту а) інформацію;
3) для виконання усіх дій, що описані вище, використовуються спеціально написані для цього функції;
4) після виконання операції інформація про неї протоколюється (як на екран, так і в файл) таким чином, щоб користувач мав можливість бачити всю

історію виконання операцій.

Варіанти завдань:

1) службовець: а) ім`я, б) прізвище, в) зарплатня, г) зміна.
2) потяг: а) номер, б) час відправлення з місця формування, в) час прибуття в пункт призначення, г) тип вагонів .
3) товар: а) найменування, б) ціна, в) кількість, г) термін поставки.
4) процесор: а) виробник, б) тактова частота в гігагерцях, в) номінальне напруження, г) тип.
5) книга: а) автори, б) видавництво, в) кількість томів, г) бібліотечний шифр.
6) деталь: а) вид деталі, б) матеріал, в) вартість, г) вага.
7) автомобіль: а) виробник, б) марка, в) рік випуску, г) пробіг.
8) багаж авіапасажира: а) прізвище власника, б) вага, в) кількість місць, г) рейс.
9) місто: а) назва, б) країна, в) регіон (область), г) кількість мешканців.

1.4. Контрольні запитання

1) Поясніть, що таке структури даних? Для чого вони застосовуються?
2) Якими типами даних можуть бути поля структури?
3) Що таке складова структура?
4) Як організована робота у мові С++ з бінарними файлами? Дайте визначення бінарного файлу.
5) Чи відрізняється робота з текстовими файлами від роботи з бінарними файлами у мові С++?

2. РОБОТА З БІНАРНИМИ ФАЙЛАМИ. СТВОРЕННЯ ЗОБРАЖЕННЯ У ФОРМАТІ BMP

2.1. Файл у форматі BMP. Особливості роботи

BMP (від англ. Bitmap Picture) – формат зберігання растрових зображень.

З форматом *BMP* працює величезна кількість програм, оскільки його підтримка інтегрована в операційні системи *Windows* і *OS/2*. Файли формату ***BMP*** можуть мати розширення *.bmp, .dib* і *.rle*. Крім того, дані цього формату включаються в двійкові файли ресурсів *RES* і в *PE*-файли [7, 8].

Глибина кольору в цьому форматі може бути 1, 2, 4, 8, 16, 24, 32, 48 біт на піксель, максимальні розміри зображення 65535×65535 пікселів. Однак глибина 2 біт офіційно не підтримується.

У форматі ***BMP*** є підтримка стиснення за алгоритмом *RLE*, однак тепер існують формати з більш сильним стисненням, і через великий розмір ***BMP*** рідко використовується в Інтернеті, де для стиснення без втрат використовуються *PNG* і *GIF*.

BMP-файл складається з чотирьох частин:

1) заголовка файлу (***BITMAPFILEHEADER***);
2) заголовка зображення (***BITMAPINFOHEADER***, може бути відсутнім);
3) палітри (може бути відсутня);
4) зображення.

Структура ***BITMAPFILEHEADER*** містить інформацію про тип, розмір і представлення даних у файлі:

```
struct BitmapFileHeader {
  WORD      bfType;  // тип файла, символи «BM» (0x424d),
                     //  тип WORD - unsigned short
          // тип DWORD - unsigned long
  DWORD     bfSize;  // розмір всього файла в байтах.
  WORD      bfReserved1;  // зарезервовані.
  WORD      bfReserved2;  // повинні містити нулі.
  DWORD     bfOffBits;  // містить зміщення в байтах від початку
//структури BITMAPFILEHEADER до безпосередньо бітів зображення.
};
```

Найбільш простий варіант заголовка зображення *BITMAPINFOHEADER*:

```
struct BitmapInfoHeader {
  DWORD    biSize;        // розмір структури в байтах.
  LONG     biWidth;  // ширина зображення в пікселях.
  LONG     biHeight;// висота зображення в пікселях.
  WORD     biPlanes;// містить одиницю.
  WORD     biBitCount;// число бітів на піксель.
  DWORD    biCompression;// тип стиснення (0— без стиснення).
  DWORD    biSizeImage;       // розмір зображення в байтах.
  LONG     biXPelsPerMeter;  // горизонтальна роздільна здатність в у пікселях
// на метр для цільового пристрою.
// тип LONG – long int.
  LONG     biYPelsPerMeter;  // вертикальна роздільна здатність у пікселях
// на метр для цільового пристрою.
  DWORD    biClrUsed;  //кількість застосованих кольорових індексів у палітрі
  DWORD    biClrImportant;  //кількість індексів, для відображення зображення
};
```

2.2. Утворення кольорів у файлі формату *BMP*

Кольори в ***.bmp***-файлі задаються у форматі *RGB* (*Red-Green-Blue*). Формат *RGB* − це набір правил, за допомогою яких будь-який колір можна представити у вигляді певного коду цифр і букв.

Формат *RGB* − це вказівка комп`ютеру, як змішувати червоний, зелений і синій кольори у різних поєднаннях, щоб отримати всі кольори веселки (рис. 2). Кожен колір − червоний, зелений або синій − характеризується його інтенсивністю або насиченістю.

Кількість кожного кольору може бути в діапазоні від 0 до 255.

Абсолютно червоний колір матиме форму запису (255,0,0). Це означає, що кількість червоного кольору 255, зеленого 0 (тобто зеленої складової немає), синього 0 (синьої складової немає).

Абсолютно зелений колір (0,255,0) і синій (0,0,255).

При різних комбінаціях починають уже утворюватися різні кольори веселки: яскраво-фіолетовий − (255,0,255), чорний − (0,0,0).

Це форма запису (255,0,255) у вигляді десяткових чисел. Але колір *RGB* можна також представити у вигляді шістнадцяткової системи. Такими числами легше оперувати комп`ютеру.

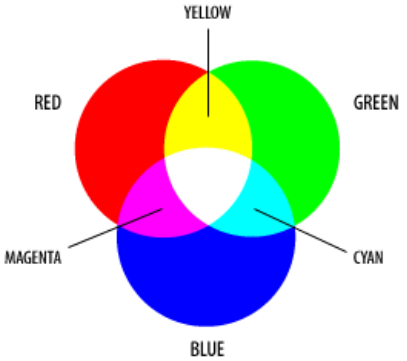


Рис. 2. Утворення кольорів у BMP-файлі

Якщо перетворити по черзі кожне з чисел, яке відповідає певному кольору, в шістнадцяткову систему, то ми отримаємо іншу форму запису кольору.

Наприклад, *FFFFFF* (255,255,255) – білий колір, де *FF* – число 255 в шістнадцятковій системі числення. Або, 000000 (0,0,0) – чорний колір. Тобто колір у форматі *RGB* можна представити як в шістнадцятковій, так і в десятковій системі числення (рис. 3). Знак # повідомляє про те, що використовується саме шістнадцяткова система.

У мові С ++ для подання шістнадцяткових чисел # замінюється на ***0x.***

Named	Numeric	Color name	Hex rgb	Decimal
		black	#000000	0,0,0
		silver	#C0C0C0	192,192,192
		gray	#808080	128,128,128
		white	#FFFFFF	255,255,255
		maroon	#800000	128,0,0
		red	#FF0000	255,0,0
		purple	#800080	128,0,128
		fuchsia	#FF00FF	255,0,255
		green	#008000	0,128,0
		lime	#00FF00	0,255,0
		olive	#808000	128,128,0
		yellow	#FFFF00	255,255,0
		navy	#000080	0,0,128
		blue	#0000FF	0,0,255
		teal	#008080	0,128,128
		aqua	#00FFFF	0,255,255

Рис. 3. Деякі стандартні кольори, якими може оперувати комп`ютер

Розглянемо роботу з файлом, формату ***bmp***, на прикладі консольного додатку, в якому користувач має можливість:

a) задати ширину і висоту зображення;

b) залити зображення червоним кольором;

c) зберегти зображення у вигляді *bmp*-файлу.

```
#include <fstream>
#include <iostream>

using namespace std;

typedef uint16_t WORD;
typedef uint32_t DWORD;
typedef int32_t LONG;
//щоб уникнути округлення розміру структури до максимального 4-байтного розміру.
#pragma pack(push,1)
struct BitmapFileHeader {
  WORD  bfType;
  DWORD bfSize;
  WORD  bfReserved1;
  WORD  bfReserved2;
  DWORD bfOffBits;
};
#pragma pack(push,1)
struct BitmapInfoHeader {
  DWORD biSize;
  LONG  biWidth;
  LONG  biHeight;
  WORD  biPlanes;
  WORD  biBitCount;
  DWORD biCompression;
  DWORD biSizeImage;
  LONG  biXPelsPerMeter;
  LONG  biYPelsPerMeter;
  DWORD biClrUsed;
  DWORD biClrImportant;
};

/Сствоерення .bmp файла,
//задання параметрів полям заголовка .bmp-файла і зображення.
void CreateBmp(const char *fileName, unsigned int **colors , int
height, int width)
{
    BitmapFileHeader bfh = {0};
    BitmapInfoHeader bih = {0};

    bfh.bfType = 0x4D42; //символи `BM`
    bfh.bfOffBits = sizeof(bfh) + sizeof(bih); //
    bfh.bfSize =bfh.bfOffBits + sizeof(colors[0][0])*width*height;
     // розмір кінцевого файла.
```

```
    bih.biSize = sizeof(bih); // розмір структури.
    bih.biBitCount = 32;      // 32 біт (4 байта) на піксель.
    bih.biHeight = -height;   // висота.
    bih.biWidth = width;      // ширина.
    bih.biPlanes = 1;         // містить 1, інші поля дорівнюють нулю.

    ofstream f;
    f.open(fileName, ios::binary); // відкриваємо файл для запису.

    f.write((char*)&bfh, sizeof(bfh)); // записуємо заголовки.
    f.write((char*)&bih, sizeof(bih));

    // Записуємо зображення.
    for (int i = 0; i < height; i++)
        f.write((char*)colors[i], sizeof(colors[0][0]) * width);

    f.close(); // закриваємо файл.
}
#pragma pack (pop)

int main(int argc, char* argv[])
{
    int w = 0, h = 0;
    cout << "Input height in px" << endl;
    cin >> h;
    cout << "Input width in px" << endl;
    cin >> w;

    unsigned int** color = new unsigned int*[h];
    for (int i = 0; i < h; i++)
        color[i] = new unsigned int[w];

    for (int i = 0; i < h; i++)
      for (int j = 0; j < w; j++)
        color[i][j] = 0xFF0000; //задання червоного кольору.

    CreateBmp("test.bmp",color, h, w);

    for (int i = 0; i < h; i++)
        delete []color[i];
    delete []color;
    cout << "test.bmp has been successfully created; Press Enter
to exit";
    cin.ignore(2);
    return 0;
}
```

Приклад виконання програми наведено на рис. 4.

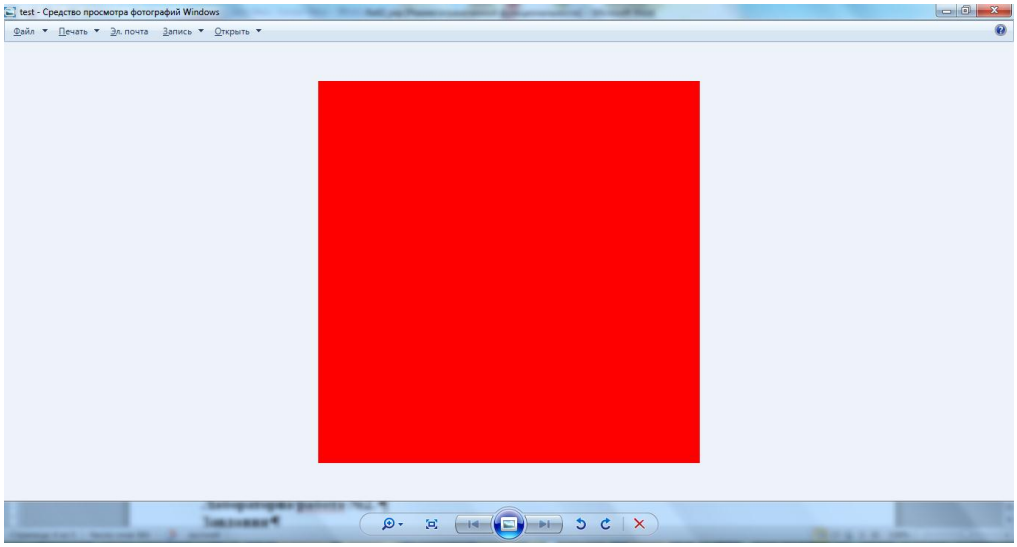


Рис. 4. Збережене зображення в *.bmp*-файлі

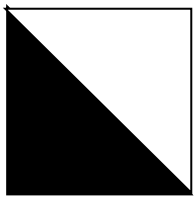
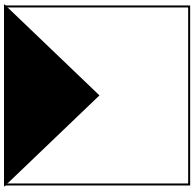
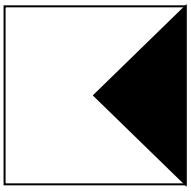
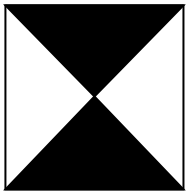
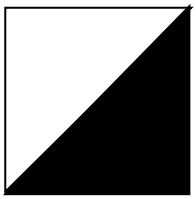
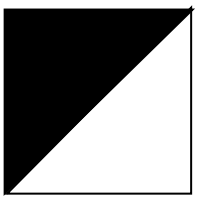
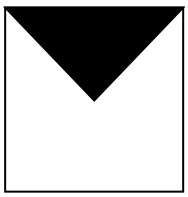
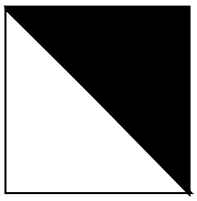
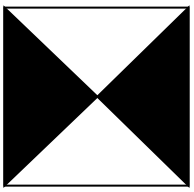
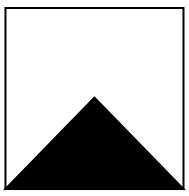
2.3. Завдання для самостійного виконання

Хід виконання завдання.

Написати консольний додаток, в якому:

1) користувач має можливість задати розмір зображення;
2) зображення формується згідно рисунку, наведеному нижче для кожного варіанту. Колір світлої та темної частини задає користувач;
3) сформоване зображення зберігається у вигляд *.bmp* файлу.

Варіанти завдань

Варіант № 1	Варіант № 2	Варіант № 3	Варіант № 4	Варіант № 5
				
Варіант № 6	Варіант № 7	Варіант № 8	Варіант № 9	Варіант № 10
				

2.4. Контрольні запитання

1) З яких частин складається файл формату *bmp*?
2) Які структури визначають заголовок і зображення *bmp*-файлу?
3) Який формат кольорів застосовується в *bmp*-файлі? Як утворюються кольори у цьому форматі?
4) Поясніть, яким чином можна організувати роботу з файлом формату bmp засобами мови програмування C++.

3. ВВЕДЕННЯ В ОБ`ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

3.1. Поняття класу та об`єкта. Абстракція та інкапсуляція

Об`єктно-орієнтований підхід у програмуванні дозволяє розкласти завдання на складові частини так, що кожна складова буде являти собою самостійний об`єкт, який містить власні інструкції і дані [9, 10]. Оскільки будь-який програмний об`єкт мовою C++ має певний тип, то таку задачу можна розв`язати, лише створюючи нові, призначені для користувача абстрактні типи даних, класи. *Клас* – це складний тип даних, в якому об`єднані елементи даних: поля і методи, що обробляють ці дані. *Поле класу* – це елемент класу, що описує дані. *Метод класу* – це процедура або функція, включена в опис класу. Метод класу викликається конкретним екземпляром класу і прив`язаний до опису та структури класу. Поля і методи класу є елементами класу.

Для моделювання нового користувацького типу даних, класу, що визначає яку-небудь предметну область, в об`єктно-орієнтованому програмуванні застовуються підходи *абстракції* (*abstraction*) та *інкапсуляції* (*encapsulation*). При *абстракції* визначаються мінімально необхідні поняття (*атрибути*) предметної області для подальшого аналізу. А поняття *інкапсуляції* визначає, що в якості одиниці цілого розглядається об`єднання певної групи даних (*атрибутів*) і деякої групи функцій (*методів*) для обробки цих даних [11].

Клас – це лише логічна абстракція, недоступна для прямого використання в програмі. Конкретні величини типу даних «клас» називаються екземплярами класу, або *об`єктами*. *Об`єкт* (або екземпляр класу) – це представник класу, побудований згідно зі створеним у класі описом. Кожному *класу* може належати одночасно декілька об`єктів, кожен з яких має унікальне ім`я. *Об`єкт* характеризується фізичним існуванням та є конкретним екземпляром *класу*. Клас являє собою логічну абстракцію і задає структуру деякої предметної області, а об`єкт — це конкретний екземпляр, що реалізує таку структуру.

Класи в C++ оголошуються з використанням ключового слова ***class***, що визначає появу нового типу, який поєднуватиме дані та методи між собою. Оголошення класу в C++ схоже на оголошення структури:

```
class ім`я_класу {
```

```
  private: // специфікатор доступу
 //приватні (приховані) атрибути (дані) та методи класу;
  public:
   //публічні атрибути та методи класу;
  protected:
   //захищені атрибути та методи класу;
} список_об`єктів_класу;
```

Специфікатори доступу можуть чергуватися у довільному порядку, область дії одного специфікатору простягається до наступного або до кінця оголошення класу.

Для моделювання *класу* необхідно:

1) застосувати метод *абстракції* та визначити необхідні поняття (атрибути) для предметної області, що розглядається;

2) визначити, які методи необхідні наданому класу для коректної роботи;

3) визначити, які з атрибутів і методів будуть суто внутрішніми (приватними – private) для цього класу, а які будуть визначати взаємодію класу із зовнішнім середовищем (іншими класами, програмою взагалі тощо) – публічними (public). Зазвичай усі атрибути класу роблять внутрішніми, а для доступу до них та їх зміни створюють спеціальні методи: *гетери та сетери*, які є доступними з інших місць програми. Методи класу, навпаки, робляться доступними іншим компонентам програми, винятком можуть бути специфічні методи, необхідні для розрахунків внутрішніх або допоміжних змінних;

4) визначити типи даних для атрибутів і типи значень, що прийматимуть і повертатимуть методи.

Розглянемо ці етапи на прикладі класу «**Співробітник компанії**».

1) Етап абстракції. Основними атрибутами класу «**Співробітник**» з точки зору роботодавця, наприклад, можуть бути такі:

- ім`я співробітника;
- вік співробітника;
- посада;
- заробітна платня співробітника;

- кар`єрна історія співробітника в цій компанії з назвами посад і датами початку роботи на посадах.

2) Необхідними методами можуть бути такі:

a) гетери та сетери для вказаних у пункті 1 атрибутів;

b) метод відображення кар`єрної історії;

c) метод розрахунку та гетер стажу співробітника в компанії;

d) метод відображення дати.

3) Усі атрибути встановлюємо як внутрішні без зовнішнього доступу. Усі гетери, сетери, методи розрахунку стажу та відображення кар`єрної історії встановлюємо публічними, методи показу дати та розрахунку стажу – приватними.

4) Визначимо типи даних для атрибутів і методів:

- ім`я співробітника: **рядок**;
- вік співробітника: **ціле** число;
- посада: **рядок**;
- заробітна платня співробітника: **дійсне** число;
- стаж роботи в компанії: **ціле** число;
- кар`єрна історія: **масив рядків**;
- дати початку роботи на посадах: двовимірний **масив цілих**;
- гетери повертатимуть змінну відповідного типу і не прийматимуть жодних параметрів, окрім гетерів ім`я та посади, що прийматимуть посилання на відповідну змінну;
- сетер посади додатково прийматиме рік, місяць і день зарахування на посаду, при цьому автоматично викликатиметься метод розрахунку стажу;
- метод відображення кар`єрної історії виводить усі елементи масиву посад з відповідними датами;
- метод розрахунку стажу перевіряє, чи не дорівнює поточна дата даті останньої посади, якщо вони різні, то перераховується значення стажу, яке потім повертається з методу;
- усі сетери не повертатимуть нічого, прийматимуть змінну відповідного

типу.

3.2. Основи *UML* для подання класів

Одним зі способів опису та подання об`єктно-орієнтованих програм взагалі та класів зокрема є використання *UML*-нотації.

UML (*unified modeling language*) – узагальнена мова опису предметних областей, зокрема, програмних систем, об`єктно-орієнтованого аналізу та проектування. Використовується для візуалізації, специфікації, конструювання та документування програмних систем.

У контексті об`єктно-орієнтованого програмування та аналізу в *UML* містяться такі базові компоненти:

1) сутності, які являють собою абстракції, що становлять модель;
2) зв`язки, що пов`язують сутності між собою.
3) діаграми для статичного подання структури, що згруповують набори сутностей у поняття.

Структурна сутність *клас* являє собою опис набору об`єктів з однаковими властивостями (*атрибутами*), операціями (*методами*), зв`язками та поведінкою. Графічно зображується у вигляді прямокутника, що розділений на 3 блоки горизонтальними лініями, порядок блоків (зверху-униз) (рис. 5):

1) ім`я класу;
2) атрибути (властивості) класу;
3) операції (методи) класу.

Додатково, для атрибутів та операцій може вказуватись один з трьох типів видимості:

- **public** позначається знаком «+» (плюс) перед назвою атрибута або методу;
- **protected** позначається знаком «#» (решітка) перед назвою атрибута або методу;
- **private** позначається знаком «–» (мінус) перед назвою атрибута або методу.

Кожен клас повинен мати унікальну назву, що відрізнятиме його від

інших класів. Ім`я класу – текстовий рядок, який може складатись із будь-якої кількості букв, цифр та інших знаків, за винятком двокрапки та крапки, а також може складатись із декількох рядків. Кращою практикою є короткі (за можливості) імена класів, що складаються з назв понять системи, що моделюється. Одним із підходів до іменування класів є так званий *CamelCase* (верблюже письмо), згідно з яким кожне слово в назві класу пишеться з великої літери, наприклад: *Sensor, TemperatureSensor* або *VelocityVectorField* тощо [12].

Назви абстрактних класів записуються курсивом.

Атрибут — іменована властивість класу, що задає діапазон можливих значень для екземпляра атрибута, цю властивість матимуть усі екземпляри заданого класу. Кожен клас може містити будь-яку кількість атрибутів або не містити жодного. В останньому випадку блок атрибутів залишається порожнім. Імена атрибутів також складаються з назв понять предметної області, але записуються у варіації *lowerCamelCase* (нижнє верблюже письмо), за якою усі слова в назві атрибута, окрім першого, починаються з великої літери, наприклад: *value, rangeMaximum* або *pointPositionX*.

Для атрибутів можна вказувати тип, кількість елементів (якщо атрибут є масивом), початкове значення.

Метод — іменована реалізація деякої функціональної операції класу. Клас може мати будь-яку кількість методів, або не мати жодного, в такому разі блок операцій залишається порожнім. Для іменування методів також може використовуватись *lowerCamelCase,* але як перше слово найчастіше використовується дієслово, наприклад: *getValue, isEqual, saveFileAndExit*. Для методу можливо вказувати його сигнатуру, що включає імена, типи та значення за замовчанням всіх параметрів і тип значення, що повертається.

В процесі зображення класу необов`язково відображати усі його атрибути та методи, для конкретного подання обираються найважливіші з них, але відповідний список атрибутів або методів у такому разі повинен закінчуватись трьома крапками. Для наочності довгі списки атрибутів або методів можливо згруповувати з використанням *стереотипів* — назв груп атрибутів або методів, які з них випливають (рис. 6). Стереотипи вказуються в кутових дужках.

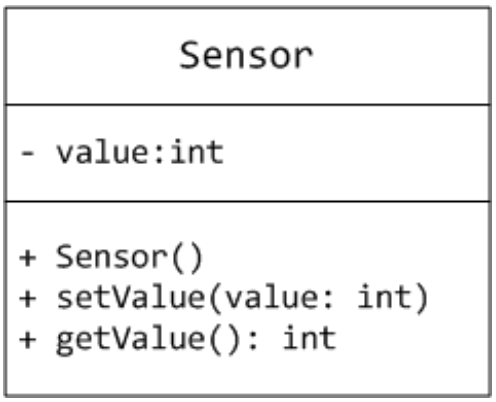


Рис. 5.

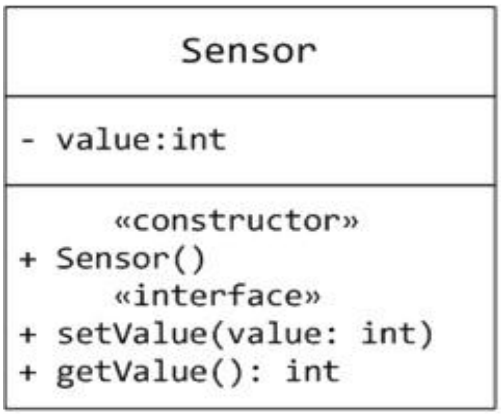


Рис. 6

Представлення класу «*Empoyee*» в *UML*-поданні наведено на рис. 7:

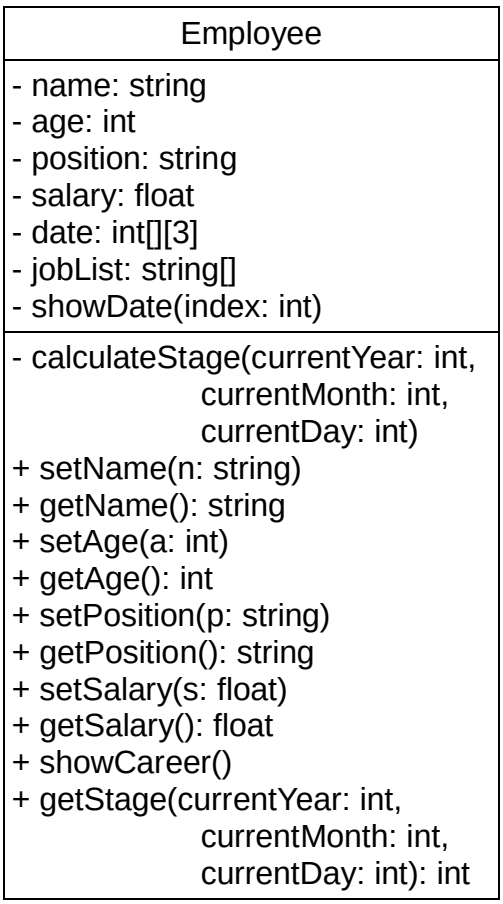


Рис. 7. Приклад зображення класу «*Empoyee*» в *UML*

Наведемо приклад коду оголошення та реалізації класу **«Співробітник»**.

Для зручності подальшої підтримки та сприйняття об`єктно-орієнтованого коду програми слід розділяти оголошення класу та його реалізацію, для цього оголошення класу слід надавати в заголовкових файлах (*.h, .hpp*), а реалізацію — у файлах вихідного коду (*.cpp*).

- заголовковий файл «Employee.h»

```
#ifndef EMPLOYEE_H //умовна компіляція, ці директиви компілятору призначені для
#define EMPLOYEE_H // захисту від багатократного включення файлу  при наявності
 // складної ієрархії включення файлів (якщо такий файл вже було включено раніше)
#include <cstring>
#include <string>
#include <iostream>


class Employee // клас Співробітник.
{
  private:     // приватні дані та методи класу.
    char name[25];
    int age;
    char position[25];
    float salary;
    int stage;
    static const int maxlen = 255;
    int date[maxlen][3];
    std::string jobList[maxlen];
    int job_index = 0;
    void showDate(int index);
    void calculateStage(int currentYear, int currentMonth, int
currentDay);
    public: // публічні методи класу.
        void setName(char *n);
        void getName(char *n);
        void setAge(int s);
        int getAge();
        void setPosition(char *p, int year, int month, int day);
        void getPosition(char *p);
        void setSalary(float s);
        float getSalary();
        void showCareer();
        int getStage(int currentYear, int currentMonth, int
currentDay);
};

#endif // EMPLOYEE_H - закриття умовної компіляції.
```

- файл з реалізацією класу «Employee.cpp»

```
#include "Employee.h"

void Employee::setName(char* n)
{
    strcpy(name, n);
}
void Employee::getName(char* n)
{
    strcpy(n, name);
}
void Employee::setAge(int s)
{
    age = s;
}
int Employee::getAge()
{
    return age;
}
void Employee::setPosition(char* p, int year, int month, int day)
{
    strcpy(position, p);
    date[job_index][0] = year;
    date[job_index][1] = month;
    date[job_index][2] = day;
    jobList[job_index] = position;
    calculateStage(year, month, day);
    job_index++;
}
void Employee::getPosition(char* p)
{
    strcpy(p, position);
}
float Employee::getSalary()
{
    return salary;
}
void Employee::setSalary(float s)
```

```
{
    salary = s;
}
void Employee::showCareer()
{
    for(int i=0; i<job_index; i++)
    {
        std::cout << "Job " << i+1 << ": " << jobList[i] << ",
started from: ";
        showDate(i);
    }
}
void Employee::showDate(int index)
{
    std::cout << date[index][0] << '.' << date[index][1] << '.' <<
date[index][2] << std::endl;
}
void Employee::calculateStage(int currentYear, int currentMonth,
int currentDay)
{
    if(job_index == 0)
        stage = 0;
    else
    {
        int daysDelta = currentDay - date[0][2];
        if(daysDelta < 0)
            currentMonth -= 1;
        int monthsDelta = currentMonth - date[0][1];
        if(monthsDelta < 0)
            currentYear -= 1;
        stage = currentYear - date[0][0];
    }
}
int Employee::getStage(int currentYear, int currentMonth, int
currentDay)
{
    if((currentYear != date[job_index][0]) || (currentMonth !=
date[job_index][1]) || (currentDay != date[job_index][2]))
        calculateStage(currentYear, currentMonth, currentDay);
```

```
    return stage;
}
```

- файл «main.cpp» з прикладом створення і використання класу

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

using namespace std;

int main()
{
    Employee ivan;
    char name[25] = {"Ivan Ivanov"};
    ivan.setName(name);
    ivan.setSalary(10000);
    ivan.setPosition((char *)"Manager", 2004, 3, 1);
    char new_name[25];
    ivan.getName(new_name);
    cout << new_name << endl;
    cout << ivan.getSalary() << endl;
    ivan.setPosition((char *)"Director", 2010, 8, 21);
    ivan.showCareer();
    cout << ivan.getStage(2020, 2, 18) << endl;
    return 0;
}
```

Результат роботи програми наведено на рис. 8.

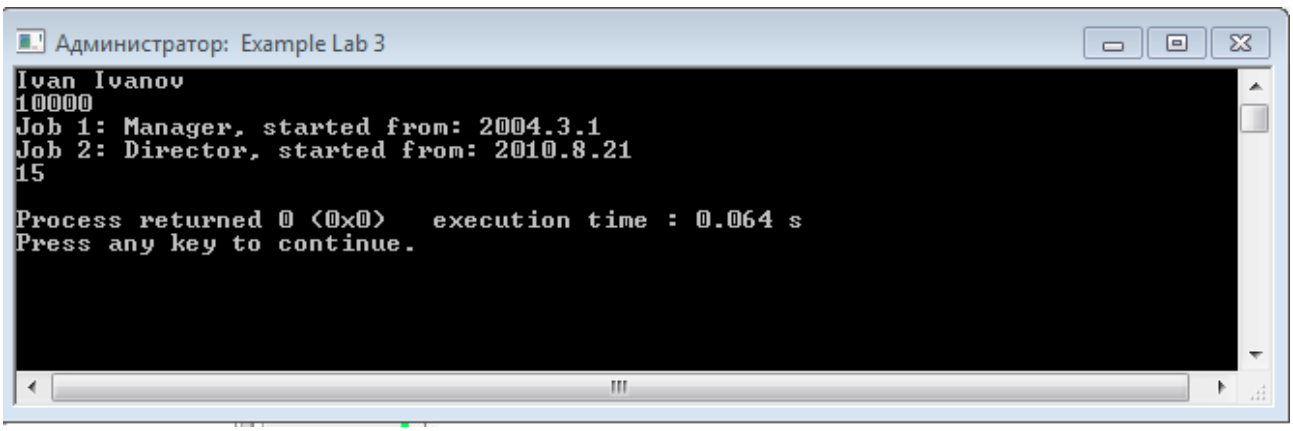


Рис. 8. Приклад виконання програми

3.3. Завдання для самостійного виконання

Хід виконання завдання:

1) Згідно свого варіанту завдання змоделювати клас для заданого поняття. Для цього застосувати метод абстракції: визначити необхідні атрибути (дані) класу, визначити необхідні методи для коректної роботи класу, а також режими доступу до елементів розглядаємого класу.
2) Зобразити *UML*-діаграму класу.
3) Реалізувати клас в вигляді програмного коду.
4) Написати програму, в якій користувач матиме можливість проводити маніпуляції зі створеним класом.

Варіанти завдань

1) Вектор в просторі.
2) Прямокутник.
3) Літак.
4) Дата.
5) Конус.
6) Пряма.
7) Ромб.
8) Відрізок.
9) Товар.
10) Студент.
11) Маршрут.
12) Трапеція.
13) Матриця.
14) Трикутник.
15) Час.
16) Циліндр.
17) Багатокутник.
18) Сфера.
19) Поїзд.
20) Ламана.
21) Вектор на площині.
22) Книга
23) Натуральне число.

3.4. Контрольні запитання

1) Чим подібні і чим відрізняються у С++ поняття структури та класу?
2) Що таке інкапсуляція? У чому вона полягає?
3) Що називається класом? Що таке атрибути класу?
4) Як можна визначити метод класу поза класом?
5) Як співвідносяться поміж собою поняття об`єкта та класу?
6) Які режими доступу до елементів класу Вам відомі? Для чого вони застосовуються?
7) Для чого потрібні гетери та сетери?
8) Що є елементом масиву об`єктів класу?

4. СТВОРЕННЯ КОНСТРУКТОРІВ І ДЕСТРУКТОРА КЛАСУ

4.1. Конструктор класу, його типи

Конструктор – це метод, який виконується автоматично в момент створення об`єкта. Він може не лише ініціалізовувати змінні класу, але й виконувати будь-які інші завдання ініціалізації, потрібні для підготовки об`єкта до використання. Для створення екземпляра класу потрібно, щоб конструктор був методом типу *public* [3, 11, 12].

Основні властивості конструктора.

- ✓ **Ім`я конструктора** – це **ім`я класу**. Отже, компілятор відрізняє конструктор від інших методів класу.
- ✓ Конструктор **не повертає жодного значення**, навіть типу *void*. Неможна здобути і покажчик на конструктор. Конструктор автоматично викликається системою, а отже, не існує програми чи функції, яка його викликає, та якій конструктор міг би повернути значення.
- ✓ **Клас** може мати **кілька конструкторів** із різними параметрами для різних видів ініціалізації, при цьому використовується механізм перевантаження.
- ✓ Конструктор може прийняти яку завгодно кількість аргументів (включаючи нульову).
- ✓ Параметри конструктора можуть бути якого завгодно типу, окрім цього класу. Можна задавати значення параметрів за замовчуванням. Їх може містити лише один з конструкторів.
- ✓ Конструктори не успадковуються.
- ✓ Конструктори не можна оголошувати з модифікаторами *const*, *virtual* та *static*.

Види конструкторів [11]:

- конструктор з параметрами;
- конструктор зі списком ініціалізації;
- конструктор за замовчуванням;
- конструктор копіювання;
- делегуючий конструктор.

4.1.1. *Конструктор за замовчуванням*

Конструктор без параметрів називають конструктором за замовчуванням. Якщо конструктор для будь-якого класу не визначено, то компілятор сам генерує конструктор за замовчуванням. Такі конструктори не присвоюють початкові значення полям класу, який визначає об`єкт класу без передавання параметрів, наприклад:

```
AnyClass ()
{cout<<”Конструктор за замовчуванням”;}
AnyClass obj1, obj2;
```

Кожен клас може включати лише один конструктор за замовчуванням.

Якщо у класі визначено будь-який конструктор, то компілятор не створить конструктора за замовчуванням.

4.1.2. *Конструктор із параметрами*

Конструктор може ініціалізувати поля класу через параметри, наприклад:

```
AnyClass(int ll, int nn, int mm)
{ l = ll; n =nn; m = mm;}
```

У цьому випадку створення об`єкта може мати вигляд:

```
AnyClass obj(5, 20, 13);
```

При створені об`єкта класу викликається конструктор, до якого передаються відповідні значення параметрів. При виділенні динамічної пам’яті під екземпляр класу можна передавати значення параметрів конструктору також:

```
AnyClass *Pobj = new AnyClass(5, 20, 13);
```

Конструктор може мати параметри за замовчуванням, які передаються автоматично , наприклад полям класу, при створенні екземплярів класу. Якщо, при створенні екземпляра класу значення параметрів конструктора не вказуються, об`єкт ініціалізується зі значеннями параметрів за замовчуванням:

```
AnyClass (int ll=1, int nn=1, int mm=1)
{ l = ll; n =nn; m = mm;}
AnyClass obj;
```

4.1.3. *Конструктор зі списком ініціалізації*

Конструктор з параметрами не може бути використаним при ініціалізації полів-констант чи полів-посилань, оскільки їм не можуть бути присвоєні значення. Для цього передбачено спеціальну властивість конструктора, що називається списком ініціалізації, який дозволяє ініціалізувати одну чи більше змінних і не надавати їм значення. Список розташовується так [11]:

```
AnyClass(): l(0), n(10), m(15) {}
```

При ініціалізації полів-об`єктів поля можна ініціалізувати за допомогою списку ініціалізаторів, у якому значення можуть бути виразами, наприклад:

```
AnyClass(int ll,int nn, int mm):l(ll),n(nn),m(mm){}
```

У першому випадку конструктор викликається при створюванні об`єкта, наприклад, при визначенні:

```
AnyClass obj1, obj2;
```

У другому випадку при створюванні об`єкта ***obj*** буде викликано конструктор із відповідними, зазначеними у дужках параметрами, наприклад:

```
AnyClass obj(10,15,20);
```

4.1.4. *Делегуючий конструктор*

У пізніших версіях компілятору C++ з`явилась можливість створення делегуючих конструкторів класу. Коли конструктор при створенні об`єкта класу викликає інший конструктор. Наприклад:

```
AnyClass(int N)
{n=N;}
AnyClass(int M): AnyClass(int N)
 {m = M;}
```

Таким чином здійснюється ієрархія виклику конструкторів. Ми делегуємо деяку операцію першому конструктору. Це дуже часто застосовується при створенні класів.

`AnyClass obj(40);`//викликається перший конструктор *AnyClass*(*int N*).

`AnyClass obj(40,60);`//викликається спочатку перший конструктор, потім другий *AnyClass*(*int M*).

4.1.5. *Конструктор копіювання*

Існує ще один спосіб ініціалізації об`єкта, який використовує значення полів уже існуючого об`єкта. Такий конструктор не потрібно самим створювати, він надається компілятором для кожного створюваного класу і називається *конструктором копіювання за замовчуванням*. Він має єдиний аргумент, який є об`єктом того ж самого класу [11, 14].

Наприклад, створимо три об`єкти ***t1, t2, t3*** у різні способи.

```
void main()
{ Interval t1(2, 30);
  Interval t2(t1);
  Interval t3 = t1;
  cout << "\nt1 = "; t1.showinterval();
  cout << "\nt2 = "; t2.showinterval();
  cout << "\nt3 = "; t3.showinterval();
  cout << endl;
}
```

Результат роботи програми має вигляд:

```
t1 = 2 год. 30 хв.
t2 = 2 год. 30 хв.
t3 = 2 год. 30 хв.
```

Якщо ж клас містить покажчики чи посилання, виклик конструктора копіювання призведе до того, що й копія, й оригінал вказуватимуть на одну й ту саму ділянку пам`яті. В такому разі конструктор копіювання має бути створений програмістом і мати вигляд

`T :: T(const T&)` // T – ім`я класу.

Наприклад:

```
AnyClass :: AnyClass (const AnyClass &D) { . . . }
```

4.2. Деструктор класу

Деструктор – це особлива форма методу, який застосовується для звільнення пам`яті, зайнятої об`єктом. Деструктор за суттю є антиподом

конструктора. Він викликається автоматично, коли об`єкт виходить з видимості [11, 14].

Ім`я деструктора розпочинається з тільди (~), безпосередньо за якою йде ім`я класу. Деструктор має такий формат:

```
~ <ім`я класу> () {}
```

Деструктор не має аргументів і значення, яке повертається. Але він може виконувати деякі дії, наприклад, виведення остаточних значень елементів даних класу, що буває зручно при налаштуванні програми.

Якщо деструктор явно не визначено, компілятор автоматично створює порожній деструктор.

Розміщувати деструктор у класі явно треба у випадку, коли об`єкт містить покажчики на пам`ять, яка виділяється динамічно. Інакше, при знищенні об`єкта, пам`ять, на яку посилались його поля-покажчики, не буде позначено як звільнену. Покажчик на деструктор визначити не можна. Деструктор не успадковується.

Розглянемо застосування конструкторів та деструктора на прикладі класу «Співробітник компанії» (див. практичний приклад розділу 3). Добавимо в методи класу ***Employee***:

- конструктор за замовчуванням;
- конструктор із параметрами, у тому числі з параметрами за замовчуванням;
- конструктор зі списком ініціалізації, або делегуючий конструктор;
- конструктор копіювання;
- деструктор.

Ці методи знаходяться в розділі *public*.

Наведемо зображення розроблюваного класу в *UML* (див. рис. 9).

Наведемо приклад програми об`явлення та реалізації класу «Співробітник»:

- заголовковий файл «Employee.h»

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
```

```
#include <iostream>
#include <string>

using namespace std;
class Employee
{
    string name;
    int age;
    string position;
    double salary;

public:
    Employee();//конструктором за замовчуванням
    Employee (string n, int a, double s);//конструктором з параметрами
    Employee (string n, int a=18);//конструктором з параметрами, у тому числі
// з параметрами за замовчуванням
    Employee (const Employee &src);//конструктор копіювання
    ~Employee();// Деструктор
    void setName(string n);
    string getName();
    void setAge(int s);
    int getAge();
    void setSalary(double s);
    double getSalary();
    void showEmployee();
    };
    #endif // EMPLOYEE_H
```

- файл із реалізацією класу «Employee.cpp»

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

 using namespace std;

//конструктором за замовчуванням
  Employee::Employee():name("N/A"),age(0),salary(0.0)
    {
```

```
      cout<<"Екземпляр класу створено конструкт за замовчув\n";
    }
//конструктором з параметрами
  Employee::Employee (string n, int a, double s)
    {
      name=n; age=a;
      if (s< 50000.0)
      salary = s;
      else // Непрпустимий оклад
      salary = 0.0;
      cout<<"Екземпляр класу створено конструктором з
            параметрами\n";
    }
//конструктором з параметрами, зі списком ініціалізації
  Employee::Employee (string n, int a):name(n),age(a)
    {
        do  {
        cout << "Введіть оклад для " << name << "меньше $50000: ";
        cin >> salary;
    }
        while (salary >= 50000.0);
        cout<<"Екземпляр класу створено конструктором з парам\n";
    }
  Employee::Employee (const Employee &src)//Конструктор копіювання
    {
       cout<<"Конструктор копіювання\n";
        name=src.name;
        age=src.age;
        salary=src.salary;
    }
  Employee::~Employee()// деструктор
    {
        cout<<"Працює деструктор\n";
    }
  void Employee::setName(string n)
    {
     name=n;
    }
```

```
  string Employee::getName()
    {
     return name;
    }
  void Employee::setAge(int s)
    {
     age=s;
    }
  int Employee::getAge()
    {
     return age;
    }
  void Employee::setSalary(double s)
    {
     salary = s;
    }
  double Employee::getSalary()
    {
     return salary;
    }
  void Employee::showEmployee()
    {
     cout<<"Employee: "<<name<<"\t"<<"Age: "<<age<<"\t"<<"Salary:"
<<salary<<endl;
    }
```

- файл «main.cpp» із прикладом створення і використання класу

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

using namespace std;

int main()
{
    setlocale(LC_ALL,"Russian");
    Employee emp1;
    emp1.showEmployee();
    Employee emp2("Ivan", 35,100000.0);
```

```
    emp2.showEmployee();
    Employee emp3("Andrew",18);
    emp3.showEmployee();
    Employee department[5]={Employee("A",30), Employee("B",25),
    Employee("C",61)};
     for (int i=0;i<5;i++)
     {
         department[i].showEmployee();
     }
     Employee emp4=emp3;
     emp4.showEmployee();
    return 0;
}
```

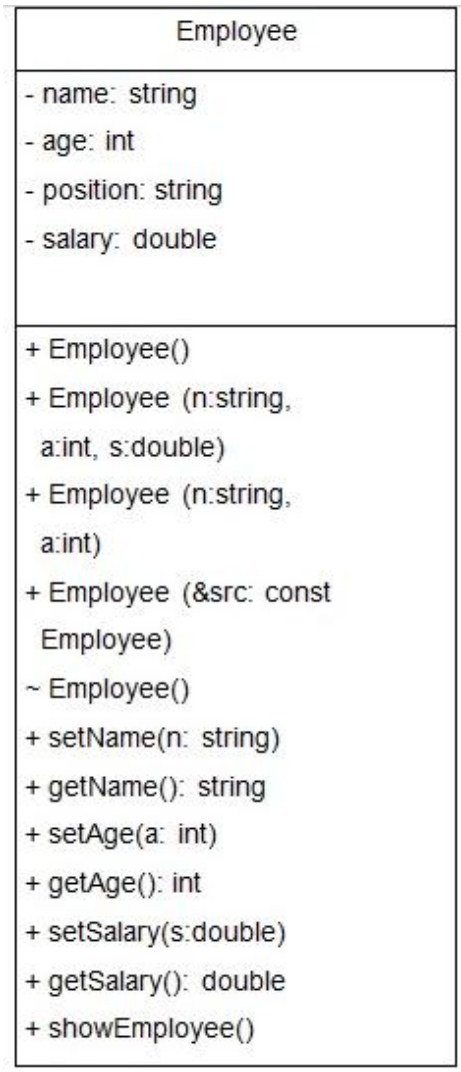


Рис. 9. Диаграма класу «*Employee*» в *UML*

На рис. 10 наведено приклад виконання роботи.

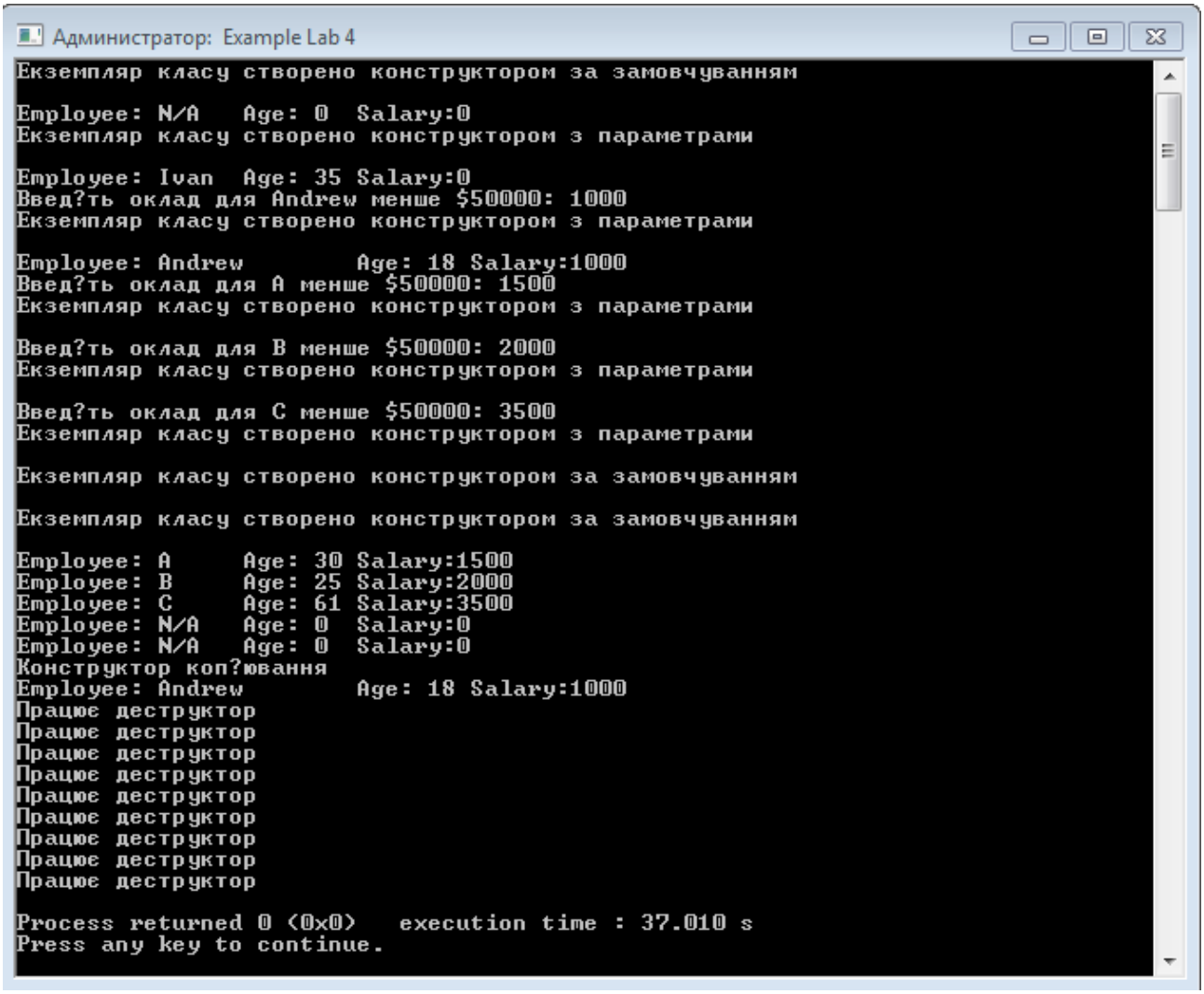


Рис.10. Приклад виконання програми

4.3. Завдання для самостійного виконання

Хід виконання завдання.

1) Змоделювати клас для заданого поняття (згідно варіанту):
   а) визначити атрибути класу та їх типи даних;
   б) визначити методи класу.
   в) до методів класу додати:
      - конструктори за замовчуванням, з параметрами, зі списком ініціалізації, конструктор копіювання;
      - деструктор.
2) Описати клас в *UML*-нотації.
3) Написати програму, в якій користувач матиме можливість проводити маніпуляції зі створеним класом. Організувати роботу з масивом

екземплярів класу.

Варіанти завдань

1) Сфера.
2) Дата.
3) Конус.
4) Пряма.
5) Ромб.
6) Відрізок.
7) Товар.
8) Студент.
9) Маршрут.
10) Трапеція.
11) Матриця.
12) Трикутник.
13) Час.
14) Циліндр.
15) Багатокутник.
16) Поїзд.
17) Ламана.
18) Автомобіль.
19) Книга.
20) Вектор в просторі.
21) Прямокутник.
22) Літак.
23) Натуральне число.

4.3. Контрольні запитання

1) Що називається класом? Що таке атрибути класу
2) Як співвідносяться поміж собою поняття об`єкта та класу?
3) Які режими доступу до елементів класу Вам відомі? Для чого вони застосовуються?
4) Що таке конструктор та коли він викликається? Чи повертає він значення?
5) Які типи конструкторів Ви знаєте? Чи може клас мати декілька конструкторів?
6) Що таке деструктор і коли він викликається? Чи може в класі бути декілька деструкторів?
7) Чи можна передати початкові значення полям класу за допомогою конструктора?

5.ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ. ЗАГАЛЬНІ ПОЛОЖЕННЯ. СПОСОБИ ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

5.1. Ключовий покажчик *this*

Ключовий покажчик this – це покажчик на об`єкт, який викликає метод класу. При кожному виклику методу йому автоматично передається покажчик з ключовим словом *this,* на об`єкт, для якого викликається метод. Покажчик *this* – це неявний параметр, що приймається усіма методами класу. Відповідно, в будь-якому методі покажчик *this* можна використовувати для посилання на об`єкт, що його викликає [2, 3, 11, 14].

5.2. «Дружні» функції

В C++ є можливість дозволити доступ до закритих членів класу функціям, які не є членами цього класу. Для цього достатньо оголосити ці функції *«дружніми»* (або *«друзями»*) щодо класу, що розглядається. Для того, щоб зробити функцію *«другом»* класу, необхідно включити її прототип в *public*-розділ оголошення класу з ключовим словом *friend*. Функція може бути *«другом»* декількох класів [11, 14].

```
class someClass {
//...
public:
     friend void frnd (someClass obj);
//...
};
```

Ключове слово *friend* надає функції, що не є членом класу, доступ до його закритих членів.

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
```

```
    myclass(int i, int j) { a=i; b=j; }
    friend int sum(myclass x);
};
int sum(myclass x)
{
    return x.a + x.b;
}
int main ()
{
  myclass n(3, 4);
  cout << sum(n);
  return 0;
}
```

5.3. Перевантаження операторів

С++ дозволяє перевантажувати більшість операцій у такий спосіб, щоб стандартні операції можна було використовувати і для об`єктів створюваних користувачем класів [9, 11].

Перевантаження операцій надає можливість використовувати власні типи даних як стандартні, а складний та малозрозумілий текст програми перетворювати на інтуїтивно зрозумілий. Позначення власних операцій вводити не можна.

Можна перевантажувати будь-які операції, що існують у С++, за винятком:

. .* ? : :: # ## sizeof

Перевантаження операцій здійснюється за допомогою методів спеціальної форми (функцій-операцій) і підпорядковується таким правилам:

- при перевантаженні операцій зберігаються кількість аргументів, пріоритети операцій та правила асоціації (зліва направо чи справа наліво), які використовуються у стандартних типах даних);
- для стандартних типів даних перевизначати операції не можна;
- функції-операції не можуть мати аргументів за замовчуванням;
- функції-операції успадковуються (за винятком “=”);

- функції-операції не можуть визначатися як *static*.

Функцію-операцію можна визначити трьома способами:

- як метод класу;
- як «дружню» функцію класу;
- як звичайну функцію.

У двох останніх випадках функція повинна мати хоча б один аргумент, який має тип класу, покажчик або посилання на клас.

Функція-операція містить ключове слово *operator*, за яким йде знак операції, яку треба перевизначити:

```
<тип> operator <операція> (<список параметрів>)
{ <тіло функції> }
```

Такий синтаксис повідомляє компіляторові про те, що, якщо операнд належить до визначеного користувачем класу, треба викликати функцію з таким ім`ям, коли зустрічається в тексті програми ця операція.

5.3.1. *Перевантаження операторів з використанням методів класу*

Можливо перевантажувати як бінарні оператори (наприклад, «+», «=»), так і унарні (наприклад, «++», «– –») [9, 11].

При перевантаженні бінарної операції функція-оператор має один параметр. В усіх випадках об`єкт, що активує функцію-оператор, передається неявно за допомогою покажчика *this*. Об`єкт, що знаходиться справа від знака операції, передається методу як параметр.

```
class Point
{
    int x,y,z;
public:
    Point operator+ (Point op2);
};
Point Point::operator+( Point op2)
{
    Point temp;
    temp.x=x+op2.x;
```

```
    temp.y=y+op2.y;
    temp.z=z+op2.z;
    return temp;
}
```

Зверніть увагу, в рядку

```
temp.x=x+op2.x;
```

x відповідає *this->x*, , де *x* асоційовано з об`єктом, який викликає функцію-оператор.

При перевантаженні унарної операції функція-оператор не має параметрів, тобто жоден об`єкт не передається явно, а операція виконується над об`єктом, який генерує виклик цього методу через неявно переданий покажчик *this*.

Розглянемо приклад, у якому для об`єктів типу *Point* визначається операція інкременту.

```
class Point
{
    int x,y,z;
public:
    Point operator++();
};
  Point Point::operator++()
  {
      x++;
      y++;
      z++;
    return *this;
  }
```

Оператори інкременту та декременту мають як префіксну, так і постфіксну форми.

У наведеному вище прикладі функція ***operator***++ визначає префіксну форму оператора «++» для класу *Point*. Якщо необхідно перевантажити постфіксну форму оператора «++», це можна зробити так:

```
Point Point::operator++(int notused)
```

Параметр *notused* не використовується самою функцією. Він є індикатором для компілятора, який дозволяє відрізнити префіксну форму оператора інкремента від постфіксної. Так само він використовується для оператора декременту («– –»).

5.3.2. *Перевантаження операторів за допомогою «дружніх» функцій*

Оскільки «дружні» функції не є членами класу, вони не можуть мати неявний аргумент *this*. Тому при перевантаженні бінарної функції оператора обидва операнди передаються функції, а при перевантаженні унарних операторів передається один операнд [9, 12, 14].

Наступні оператори не можуть використовувати перевантаження за допомогою «дружніх» функцій:

= () [] ->.

```
class Point {
    int x, y, z;
 public:
    friend Point operator*( Point op1, Point op2);
};
Point operator*( Point op1, Point op2)
{
    Point temp;
    temp.x = op1.x * op2.x;
    temp.y = op1.y * op2.y;
    temp.z = op1.z * op2.z;
    return temp;
     }
```

Якщо необхідно перевантажити оператори інкременту або декременту з використанням «дружньої» функції, необхідно передати їх посилання на об`єкт.

Оскільки параметр у такому вигляді являє собою неявний покажчик на аргумент, то зміни, які будуть вноситися в параметр, впливатимуть і на аргумент. Тобто використання посилання як параметра функції дозволяє їй

успішно інкрементувати або декрементувати об`єкт, який передається як операнд.

Якщо для перевантаження операторів інкременту та декременту використовується функція-«друг», її префіксна форма приймає один параметр (який і є операндом), а постфіксна форма – два параметри (другим є цілочислове значення, яке не використовується).

```
friend Point operator++( Point &op1)
{
op1.x++;
op1.y++;
op1.z++;
return op1;
}
friend Point operator++( Point &op1, int notused)
{
Point temp = op1;
op1.x++;
op1.y++;
op1.z++;
return temp;
}
```

5.3.3. *Перевантаження операторів за допомогою звичайних функцій*

Оскільки принцип перевантаження операторів через звичайні та «дружні» функції практично ідентичний (вони мають різні рівні доступу до закритих членів класу), єдина відмінність – у випадку «дружньої» функції, її необхідно обов`язково оголосити в класі, на відміну від звичайної функції, яку достатньо визначити поза тілом класу, без вказування додаткового прототипу функції. Доступ до закритих членів класу, у випадку використання звичайної функції для перевантаження операторів, відбувається через гетери.

```
class Point
{
    int x,y,z;
```

```
public:
    int getX(){return x;}
    int gexY(){return y;}
    int getZ() {return z;}
};
Point operator+ (Point &op1, Point &op2)
{
return    Point    (op1.getX()+op2.getX(),    op1.gexY()+op2.gexY(),
op1.getZ()+op2.getZ());
}
```

Розглянемо перевантаження операторів на прикладі класу «*Point*». Перевантажимо:

а) оператори «+», «=» та префіксний і постфіксний декременти як методи класу;

б) оператори «*», префіксний і постфіксний інкременти як «дружні» функції;

в) оператор «–» як звичайну функцію.

- Заголовковий файл «Point.h»

```
#ifndef Point_H_INCLUDED
#define Point_H_INCLUDED
class Point
{
    int x,y,z;
public:
    Point (){x=y=z=0;}
    Point (int i,int j,int k) {x=i;y=j;z=k;}
    int getX(){return x;}
    int gexY(){return y;}
    int getZ() {return z;}
    Point operator+ (Point op2);
    Point operator= (Point op2);
    Point operator--();
    Point operator--(int notused);
    friend Point operator++( Point &op1);
    friend Point operator++( Point &op1, int notused);
```

```
    friend Point operator* (Point &op1, Point &op2);
    void show();
};
Point operator- (Point &op1, Point &op2);


#endif // Point_H_INCLUDED
```

- файл із реалізацією класу «Point.cpp»

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

using namespace std;
Point operator- (Point &op1, Point &op2) //перевантаження за допомогою
//зовнішньої функції.
    {
    return Point (op1.getX()-op2.getX(), op1.gexY()-op2.gexY(),
op1.getZ()-op2.getZ());
    }
Point Point::operator+( Point op2) //перевантаження методом класу.
  {
    Point temp;
    temp.x=x+op2.x;
    temp.y=y+op2.y;
    temp.z=z+op2.z;
    return temp;
  }
Point Point::operator= (Point Point op2) //перевантаження методом класу.
  {
      x=op2.x;
      y=op2.y;
      z=op2.z;
      return *this;
  }
Point Point::operator--()//перевантаження методом класу.
  {
      x--;
      y--;
      z--;
      return *this;
  }
Point Point::operator--(int notused) //перевантаження методом класу.
  {
      Point temp = *this;//збереження початкового значення.
      x--;
      y--;
      z--;
      return temp;
  }
Point operator* (Point &op1, Point &op2) //перевантаження за допомогою
//дружньої функції.
```

```
  {
      Point temp;
      temp.x = op1.x *op2.x;
      temp.y = op1.y * op2.y;
      temp.z = op1.z * op2.z;
      return temp;
  }
Point operator++( Point &op1) //перевантаження за допомогою дружньої функції.
  {
      op1.x++;
      op1.y++;
      op1.z++;
      return op1;
  }
Point operator++( Point &op1, int notused) //перевантаження за допомогою
//дружньої функції.
  {
      Point temp = op1;
      op1.x++;
      op1.y++;
      op1.z++;
      return temp;
  }
void Point::show()
  {
      cout<<x<<", "<<y<<", "<<z<<"\n";
  }
```

- файл «main.cpp» із прикладом використання перевантажених операторів

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

using namespace std;

int main(){
     setlocale(LC_ALL, "Russian");
     Point a(1,2,3), b(10,10,10), c;
     cout<<"c=a+b \n";
     c=a+b;
     a.show();  b.show();  c.show();
     cout<<"c=b=a \n";
     c=b=a;
     a.show();  b.show();  c.show();
     cout<<"a=--c \n";
     a=--c;  //префіксна форма декременту - об`єкт отримує значення с після його
//декрементування.
```

```
      a.show(); c.show();
      cout<<"a=c-- \n";
      a=c--;//постфіксна форма декременту - об`єкт отримує значення с до його
//декрементування.
      a.show(); c.show();
      cout<<"c=a*b \n";
       c=a*b;
       c.show();
       cout<<"a=++c; \n";
       a=++c; a.show(); c.show();
       cout<<"a=c++ \n";
       a=c++; a.show(); c.show();
       cout<<"c=a-b \n";
       c=a-b;   c.show();
  }
```

Діаграма *UML* класу та приклад виконання програми наведено на рис. 11 та 12 відповідно.

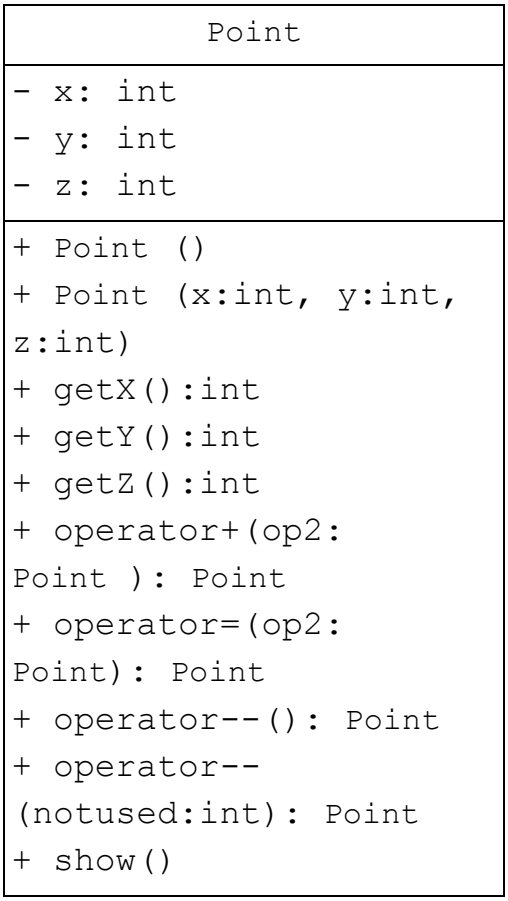


Рис. 11. Діаграма класу «*Point*» в *UML*

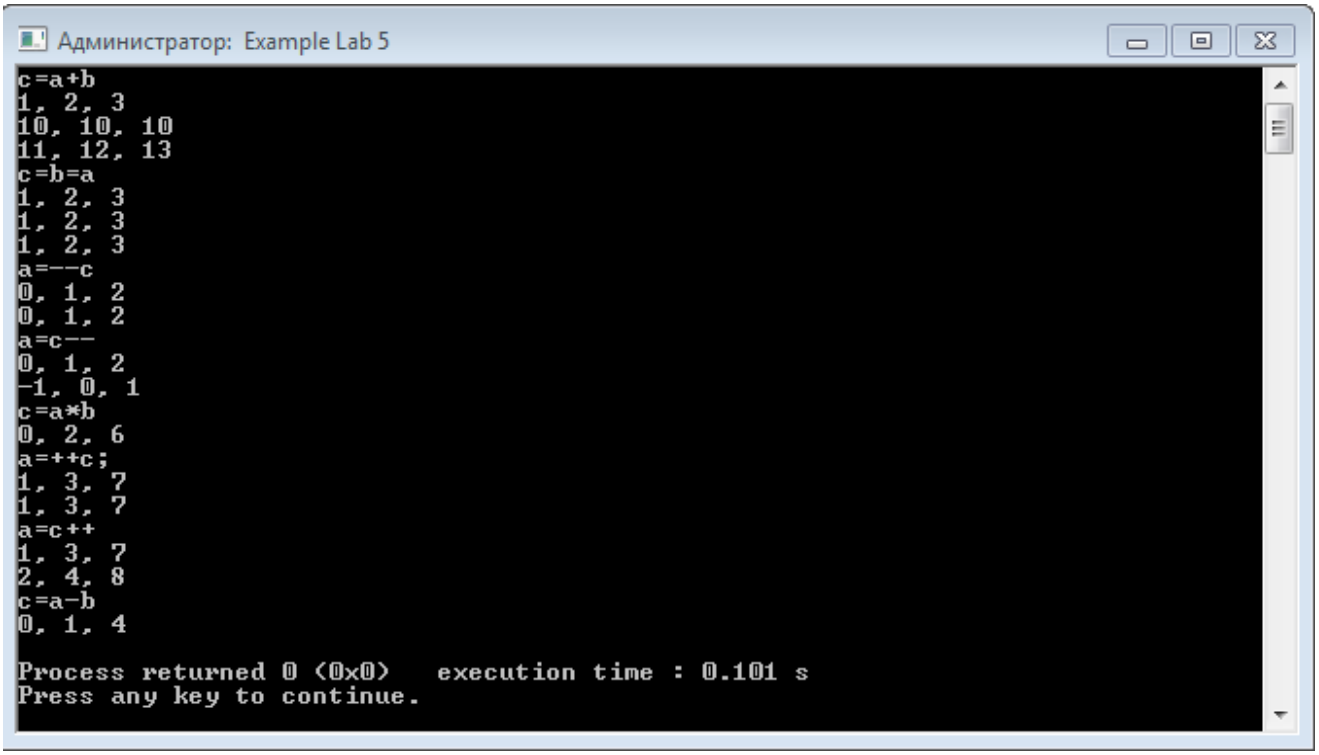


Рис. 12. Приклад виконання програми

5.3.4. *Завдання для самостійного виконання*

Хід виконання завдання.

1) Змоделювати клас для заданого поняття (згідно варіанту).
2) Перевантажити вказані у варіанті оператори – необхідно реалізувати кожен з трьох способів перевантаження: як метод класу, як «дружню» функцію та як звичайну функцію.
3) Написати програму, в якій користувач матиме можливість використовувати перевантажені оператори.

Варіанти завдань

1) Клас – комплексне число. Перевантажити оператори: віднімання (–), порівняння (>), декремент (– –), присвоювання (=).
2) Клас – комплексне число. Перевантажити оператори: додавання (+), порівняння (==), декремент (– –), присвоювання (=).
3) Клас – час. Перевантажити оператори: віднімання (–), порівняння (>,<), інкремент (++), присвоювання (=).
4) Клас – час. Перевантажити оператори: додавання (+), порівняння (==),

декремент (– –), присвоювання (=).

5) Клас – вектор. Перевантажити оператори: додавання (+), скалярний добуток (%), множення вектора на число (*=), присвоювання (=), декремент (– –).
6) Клас – вектор. Перевантажити оператори: віднімання (–), векторний добуток (*), інкремент (++), присвоювання (=).
7) Клас – дріб. Перевантажити оператори: віднімання (–), порівняння (>,<), ділення (/), інкремент (++), присвоювання (=).
8) Клас – дріб. Перевантажити оператори: додавання (+)порівняння (==, !=), добуток (*), декремент (– –), присвоювання (=).
9) Клас – пряма (задається через координати двох точок). Перевантажити оператори: визначення паралельності двох прямих (||), визначення кута між двома прямими (%), присвоювання (=), декремент (– –).
10) Клас – пряма (задається рівнянням $y = ax + b$). Перевантажити оператори: визначення паралельності двох прямих (||), визначення кута між двома прямими (%), присвоювання (=), інкремент (++).
11) Клас – матриця. Перевантажити оператори: присвоювання (=), додавання (+), добуток двох матриць (*), інкремент (++).
12) Клас – матриця. Перевантажити оператори: присвоювання (=), віднімання (–), добуток матриці на число (*=), декремент (– –).
13) Клас – парабола. $y = ax^2 + bx + c$. Перевантажити оператори: присвоювання (=), перевірка співпадіння парабол (||), перевірка перетину парабол (/).
14) Клас – вектор. Перевантажити оператори: інкремент(– –), визначення кута між двома векторами (/), присвоювання (=).
15) Клас – вектор. Перевантажити оператори: додавання (+), множення вектора на число (*=), присвоювання (=), інкремент (++).
16) Клас – комплексне число. Перевантажити оператори: порівняння (<), ділення (/), інкремент (++), присвоювання (=).
17) Клас – комплексне число. Перевантажити оператори: порівняння (!=), добуток (*), інкремент (++), присвоювання (=).

5.3.5. *Перевантаження операції* [ ]

Операція індексування [ ] перевантажується для того, щоб використовувати стандартний запис С++ для доступу до елементів членів класу. Операція [ ] перевантажується як бінарна операція [9, 15, 16].

Її можна перевантажувати тільки для класу і тільки за допомогою функцій-членів класу.

Однак оскільки ця операція зазвичай використовується ліворуч знака “=”, перевантажена функція має повертати власне значення за посиланням.

У наступному прикладі для класу ***Vector*** за допомогою перевантаження операції індексування повертається і-тий елемент масиву цілих чисел *beg*.

```
int Vector::operator [](int i)
{
    if(i<0) cout<<"index <0";
    if(i>=size) cout<<"index>size";
    return beg[i];
}
```

Параметр операторної функції *operator*[ ]( ) може мати будь-який тип даних: *символ, дійсне число, рядок.*

5.3.6. *Перевантаження оператора* ( )

С++ дозволяє перевантажувати *оператор виклику функції* ( ). При його перевантаженні створюється не новий засіб виклику функції, а операторна функція, якій можна передати довільну кількість параметрів [16, 17].

У загальному випадку при перевантаженні оператора ( ) визначаються параметри, які необхідно передати функції operator( )( ). А аргументи, які задаються при використанні оператора ( ) в програмі, копіюються у ці параметри. Об`єкт, який генерує виклик операторної функції, адресується покажчиком ***this***. Наприклад, для класу ***Vector***:

```
void Vector:: operator()(int n)
{ for(int i=0; i<size; i++)
   beg[i]=n*beg[i];}
```

5.3.7. *Перевантаження операторів введення даних >> та виведення <<*

У мові С++ передбачено засіб введення і виведення стандартних типів даних, з використанням операторів *помістити в потік* << і *взяти з потоку* >>. Ці оператори вже перевантажені в бібліотеці <*iosream*> для роботи з різними стандартними типами даних. Включаючи строки та адреси. Але ці оператори можна також перевантажувати для вводу та виведення типів даних, які визначені користувачем [11, 16, 17].

Функції перевантаження операторів помістити в потік << та взяти з потоку >> не можуть бути членами класу. Для того, щоб вони мали доступ до елементів класу, їх перевантажують як дружні функції.

```
ostream &operator<<(ostream &output, const Vector &v)
{
    if(v.size==0) output<<"Empty\n";
    else
    {
        for (int i=0; i<v.size; i++)
            output<<v.beg[i]<<" ";
        output<<endl;
    }
    return output;
}
istream &operator >>(istream &input, Vector &v)
{
    for(int i=0; i<v.size; i++)
    {
        cout<<">";
        input>>v.beg[i];
    }
    return input;
}
```

Зверніть увагу, що згідно з оголошенням, ці функції повертають посилання на об`єкт типу *ostream* або *istream*. Це дозволяє об`єднати в одному складовому вираженні декілька операторів виведення.

Операторні функції у цьому випадку мають два параметри. Перший являє собою посилання на потік, який використовується у лівій частині оператору. Другий являє об`єкт, розташований у правій частині оператора. За необхідності другий параметр також може мати посилання на об`єкт. Саме тіло функції, для

розглянутого прикладу, складається з інструкцій виведення чи введення масиву класу ***Vector***.

Розглянемо застосування перевантаження операторів індексування [ ], присвоювання = , виклику функції ( ), введення >> та виведення << на прикладі програми роботи з класом ***Vector***.

Наведемо зображення розроблюваного класу в *UML* (рис. 13):

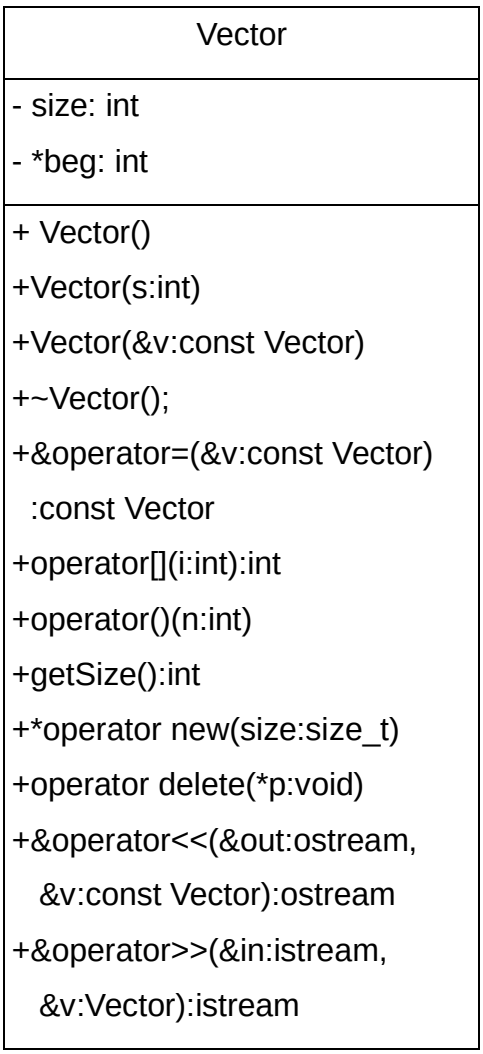


Рис.13. *UML*-діаграма класу «*Vector*»

- Заголовковий файл «Vector.h».

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR _H_INCLUDED
#include <iostream>
using namespace std;
const int MAX_SIZE=20;
class Vector
{
```

```
    int size;
    int *beg;
public:
    Vector()
    {
        size=0;
        beg=0;
    }
    Vector(int s);
    Vector(const Vector &v);
    ~Vector();
//Перевантаження оператора =.
    const Vector& operator=(const Vector&v);
//Перевантаження операції індексування.
    int& operator[](int i);
//Перевантаження оператора виклику функції.
    void operator()(int n);
    int getSize();
//Перевантаження операторів введення та виведення.
    friend ostream& operator<<(ostream &output,const Vector &v);
    friend istream& operator>>(istream &input, Vector &v);
};
#endif // VECTOR_H_INCLUDED
```

- файл із реалізацією класу «Vector.cpp».

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

using namespace std;
Vector::Vector(int s)
{
    if(s>MAX_SIZE)
        cout<<"Vector length more than MAXSIZE\n";

    size=s;
    beg=new int [s];
    for(int i=0; i<size; i++)
        beg[i]=i;
}
Vector::Vector(const Vector &v)
{ cout<<"Copy constructor"<<endl;
    size=v.size;
    beg=new int [size];
    for(int i=0; i<size; i++)
```

```
        beg[i]=v.beg[i];
}
Vector::~Vector()
{
    if (beg!=0)
        delete[]beg;
}
const Vector& Vector::operator =(const Vector &v)
{
    if(this==&v)
        return *this;
    if(beg!=0)
        delete []beg;
    size=v.size;
    beg=new int [size];
    for(int i=0; i<size; i++)
        beg[i]=v.beg[i];
    return*this;
}
ostream& operator<<(ostream &output, const Vector &v)
{
    if(v.size==0) output<<"Empty\n";
    else
    {
        for (int i=0; i<v.size; i++)
            output<<v.beg[i]<<" ";
        output<<endl;
    }
    return output;
}
istream& operator >>(istream &input, Vector &v)
{
    for(int i=0; i<v.size; i++)
    {
        cout<<">";
        input>>v.beg[i];
    }
    return input;
}
int& Vector::operator [](int i)
{
    if(i<0) cout<<"index <0";
    if(i>=size) cout<<"index>size";
    return beg[i];
}
int Vector::getSize ()
{
    return size;
}
void Vector:: operator()(int n)
{
    for(int i=0; i<size; i++)
        beg[i]=n*beg[i];
}
```

- файл «main.cpp» із прикладом використання перевантажених операторів.

```
#include "Vector.h"
#include <iostream>
using namespace std;
int main()
{    Vector x(10),y(x);//створення двох об`єктів.
     cout<<"Massiv x \n";
     cout<<x;//виклик перевантаженого оператору <<.
     cout<<"Nomer?\n";
     int i;
     cin>>i;
     cout<<"x["<<i<<"]= "<<x[i]<<endl; //виклик перевантаженого оператору []
     x[i]=7; //виклик перевантаженого оператору [].
     cout<<"Massiv x \n";
     cout<<x<<endl; //виклик перевантаженого оператору <<.
     cout<<"Massiv y \n";
     cout<<y<<endl; //виклик перевантаженого оператору <<.
     x(3); //виклик перевантаженого оператору ( ).
     cout<<"Massiv x*3 \n";
     cout<<x<<endl;
     Vector *p1=new Vector[3]; //виділення динамічної пам`яті під масив об`єктів
     *p1=y; //ініціалізація елементів масиву, виклик перевантаженого оператору =.
     *(p1+1)=x;
     for (int i=0; i<3; i++)
      {
        cout<<"\n p["<<i<<"]= "<<p1[i]<<endl; //перевантаж. оператор <<.
        cout<<"Size="<<(p1+i)->getSize()<<endl;
      }

       *(p1+2)=y;
       p1[2](2);//(*(p1+2))(2); //виклик перевантаженого оператору( ).
       cout<<"\n p1[2]= "<<p1[2]<<endl;
       delete [] p1;
    return 0;
}
```

Приклад виконання програми наведено на рис. 14.

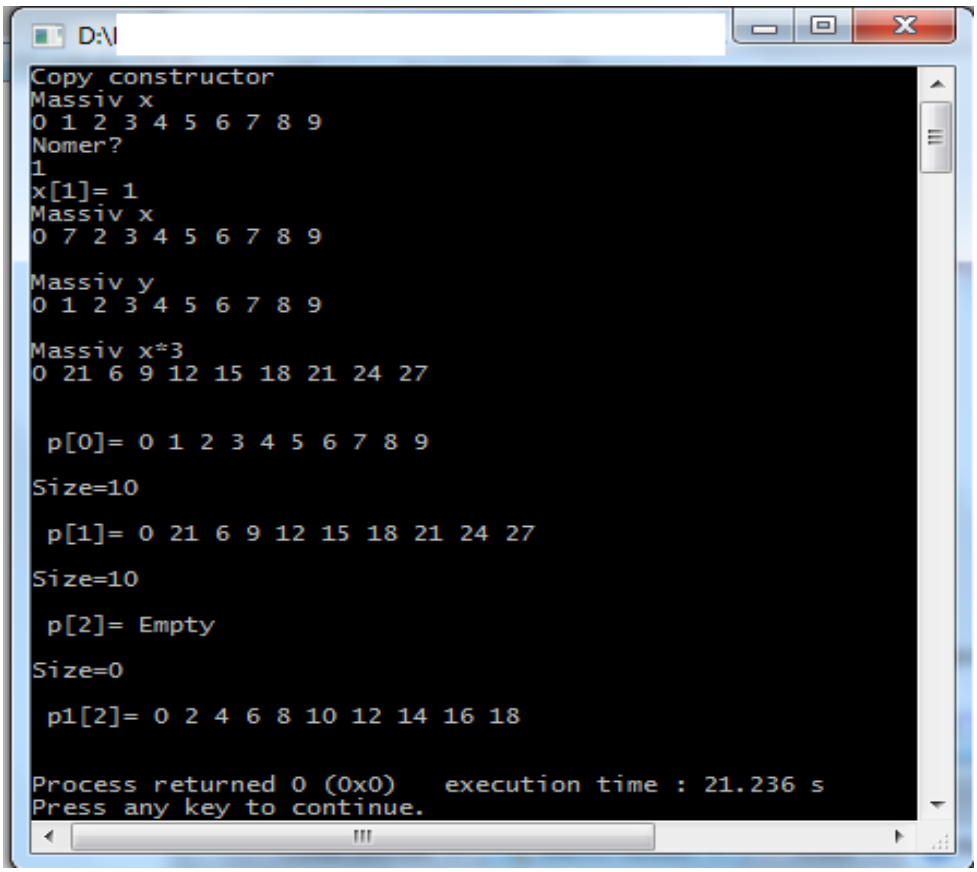


Рис. 14. Приклад виконання програми

5.3.8. *Завдання для самостійного виконання*

Хід виконання завдання:

1) Для заданого поняття (згідно з варіантом) змоделювати клас.

Клас повинен включати:

а) конструктори;

б) деструктор;

в) перевантаження операторів:

- індексування [ ];
- виклику функції ( );
- введення >> та виведення <<;

2) Описати клас в *UML*-нотації.

3) Написати програму, в якій користувач матиме можливість проводити маніпуляції зі створеним класом. Організувати роботу з масивом екземплярів класу та виділенням динамічної пам`яті під об`єкти класу.

Варіанти завдань.

1) Дата.
2) Багатокутник.
3) Циліндр.
4) Трикутник.
5) Матриця.
6) Час.
7) Трапеція.
8) Відрізок.
9) Конус.
10) Студент.
11) Комплексне число.
12) Ламана.
13) Правильний дріб.
14) Пряма.
15) Співробітник.
16) Прямокутник.
17) Вектор у просторі.
18) Призма.
19) Книга.
20) Натуральне число.
21) Прямокутний трикутник.
22) Квадратна матриця.
23) Рівнобедрений трикутник.

5.4. Контрольні запитання

1) Що таке перевантаження операцій?
2) Які операції не можуть бути перевантаженими?
3) Механізм перевантаження операторів.
4) Чим відрізняється дія перевантаженої операції ++ при її використанні у префіксній формі від її використання у постфіксній формі?

5) Скільки має бути аргументів у функції, яка перевантажує бінарну операцію за допомогою дружньої функції?

6) Покажчик *this*, яка його функція?

7) Чим відрізняється перевантаження операцій через звичайну функцію та через дружню?

8) У чому особливості перевантаження операції за допомогою методів класу.

9) Що так дружня функція? Її особливості.

10) Як перевантажується оператор індексування?

11) Для чого і як можна перевантажити оператори вводу та виведення?

12) Особливості перевантаження оператору виклику функції.

6. ГЛОБАЛЬНІ ЗМІННІ В ООП. СТАТИЧНІ ЧЛЕНИ КЛАСУ

Глобальні змінні — змінні, які було створено поза межами будь-якого блоку коду (*глобальна* або *файлова* ділянка видимості), вони є доступними з будь-якого місця програми і зберігаються в пам`яті до завершення роботи програми. Зазвичай, глобальні змінні оголошують після блоку *#include*, але вище будь-якого іншого коду, наприклад [3, 14, 18]:

```
#include <iostream>
using namespace std;
int g_x; // глобальна змінна g_x.
const int g_y(3); // глобальна константа g_y.

void doSomething()
{
	// Глобальні змінні можуть бути використані в будь-якому місці програми.
	g_x = 4;
	cout << g_y << "\n";
}

int main()
{
	doSomething();

	// Глобальні змінні можуть бути використані в будь-якому місці програми.
	g_x = 7;
	cout << g_y << "\n";
	return 0;
}
```

На відміну від глобальних, локальні змінні оголошуються всередині блока, обмеженого фігурними дужками, та є видимі тільки всередині цього блока [19]. Локальна змінна в деякому блоці коду завжди перекриває глобальну, яка має таке саме ім`я. Для того щоб примусово вказати, що в цьому місці блока має бути використана глобальна змінна, використовується оператор розширення видимості «::».

```
#include <iostream>
```

```
using namespace std;

int value(4); // глобальна змінна.

int main()
{
      int value = 8; // локальна змінна перекриває значення глобальної.
      // В цій точці коду, value=8.
      value++; // збільшується локальна змінна.
      ::value--; // зменшується глобальна змінна завдяки оператору ::
      cout << "Global value: " << ::value << "\n";
      cout << "Local value: " << value << "\n";
      return 0;
}
// Локальна змінна знищується, значення глобальної залишається  value=3.
```

Забезпечення глобального доступу є і перевагою, і недоліком глобальних змінних, оскільки їх використання може значно зменшити об`єм коду програми, але в той же час будь-які функції можуть змінювати значення глобальних змінних, тому у складних об`ємних проектах слід уникати використання таких змінних, тому що це може призвести до непрогнозованих змін значень таких змінних.

6.1. Статичні змінні

Компромісним рішенням для забезпечення глобального доступу до змінних є використання *статичних змінних*, які також можуть бути доступні глобально, але кожна така змінна наявна лише в одному екземплярі в пам`яті [9, 14].

Статичні змінні схожі до глобальних за механізмом розміщення в пам`яті, але на статичні додатково діють специфікатори доступу ***public*** та ***private***.

Статичні поля найчастіше використовуються в таких ситуаціях:

- необхідність контролю загальної кількості об`єктів класу;
- створення єдиної глобальної змінної, до якої мають доступ усі об`єкти

класу.

Статичні поля задаються ключовим словом ***static***, яке може бути використано як для атрибутів, так і для методів класу. Особливістю елементів, до яких додане ключове слово *static***,** є те, що вони належать класу, а не об`єкта цього класу, тому можуть бути використані навіть без створення об`єкта класу, і незалежно від кількості створених об`єктів цього класу, в пам`яті буде знаходитись лише одна копія елемента, який оголошено як статичний.

Таким чином, у пам`яті буде знаходитись завжди лише по одній копії кожної статичної змінної, а кількість копій нестатичних полів буде дорівнювати загальній кількості об`єктів класу.

Ключове слово *static* вказується перед зазначенням типу даних або методів:

static тип_змінної ім`яАтрибута;

static тип_значення_що_повертається ім`яМетоду(...);

Доступ до статичних змінних або функцій відбувається з використанням імені класу та оператора розширення області видимості «::»:

Ім`яКласу* :: *ім`яАтрибута;

Ім`яКласу* :: *ім`яМетоду(...);

Статичні атрибути класу оголошуються в оголошені класу (або в заголовковому файлі, якщо він є), а ініціалізуються поза блоком-оголошенням в глобальній ділянці видимості. ***Не можна*** ініціалізувати статичні змінні в тілі класу та в його методах. Такий спосіб ініціалізації потрібний для того, щоб уникнути повторної ініціалізації статичної змінної.

Розглянемо приклад використання статичної змінної для підрахунку загальної кількості створених об`єктів класу [20] (файл «*main.cpp*»):

```
#include <iostream>
#include <string>
using namespace std;
class X // оголошення класу.
{
     static int n; //змінна-лічильник створених екземплярів.
     static string ClassName;
```

```
public:
     static int getN() { return n; }
     static string getClass() { return ClassName; }
     X() { n++; } // конструктор.
};
int X::n = 0; // ініціалізація приватного атрибута поза тілом класу через оператор
// розширення видимості.
string X::ClassName = "My Class";

int main()
{
     X a, b, c; // створюємо 3 об`єкти класу X.
     cout << X::getN() << " objects of Class \"" << X::getClass()
<< "\""  << endl;
// звертання до методів класу також через оператор «::»
//cout << X::n <<  endl; призведе до помилки, спробі доступу до приватного члену
класу.
     return 0;
}
```

Приклад виконання програми наведено на рис. 15.

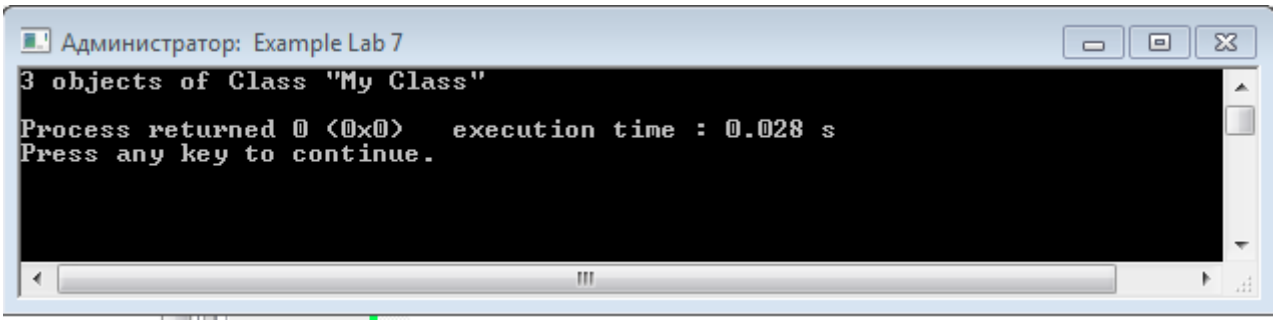


Рис. 15. Приклад виконання програми

6.2. Шаблон проектування «Одинак»

Шаблон «Одинак». Інший приклад управління створенням об`єктів класу з використанням статичних змінних та методів — так званий шаблон проектування «*Одинак*» (*Singleton*), при використанні якого можливо створити тільки один об`єкт такого класу:

```
class Singleton
{private: // єдиний екземпляр класу та базові конструктори оголошуються в приватній
     // секції класу.
     static Singleton* instance;
     Singleton() {}
     Singleton(const Singleton&);
     Singleton& operator=(const Singleton&);
public:
     static Singleton* getInstance()
     {
          if(!instance)//якщо нульове значення, тобто обїект не існує.
               instance = new Singleton();
          return instance;
     }
};

Singleton* Singleton::instance = 0; // ініціалізація статичної змінної.

int main()
{
     Singleton* s = Singleton::getInstance();
// отримуємо покажчик на єдиний екземпляр класу Singleton.
     return 0;
}
```

6.3. Завдання для самостійного виконання

Хід виконання завдання.

1) Організувати клас для предметної області, згідно варіанту. Для нього визначити та додати наступні поля:

а) глобальне;

б) статичне поле для підрахування кількості екземплярів зазначеного класу;

в) поле атрибут «***ім`яОб`єкта***» (не глобальне і не статичне).

2) Організувати додатково клас ***Logger*** на основі класу «*Одинак*»:

- задача цього класу — вносити поточні значення публічних полів об`єктів розробленого класу (згідно варіанту) до поля **log** з використанням

методу ***addRecord*()**, як рядки, наступного вигляду:

```
"object1Name: name
        object1Field1: field1Value
        object1Field1: field2Value"
```

- метод ***saveLog*()** викликатиметься в кінці роботи програми і зберігатиме текстовий файл за назвою «***log.txt***» (дата та час створення) наступного вигляду:

```
"ClassName: numberOfEntities
    time: YY.MM.DD HH:MM:SS
    object1Name: name
        object1Field1: field1Value
        object1Field2: field2Value
        ...
    time: YY.MM.DD HH:MM:SS
    object2Name: name
        object2Field1: field1Value
        object2Field2: field2Value
            ...":
```

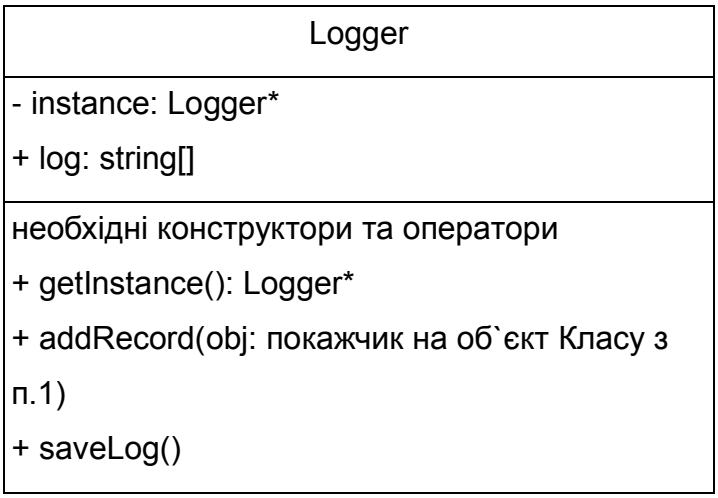


Рис. 16. *UML*-діаграма класу *Logger*

3) Написати програму, в якій користувач матиме можливість проводити маніпуляції з визначеним класом (згідно варіанту) і логувати їх (протоколювати в файл ***log.txt***) із використанням класу ***Logger***.

***Довідка: Log*-файли** призначаються для протоколювання інформації про виконання певних операцій.

Варіанти завдань

1) Вектор в просторі.
2) Прямокутник.
3) Літак.
4) Дата.
5) Конус.
6) Пряма.
7) Ромб.
8) Відрізок.
9) Товар.
10) Студент.
11) Маршрут.
12) Трапеція.
13) Матриця.
14) Трикутник.
15) Час.
16) Циліндр.
17) Багатокутник.
18) Сфера.
19) Поїзд.
20) Ламана.
21) Вектор на площині.
22) Книга.
23) Натуральне число.

6.4. Контрольні запитання

1) Що таке статичні поля класу? Чи належать вони об`єктам класу, у якому вони оголошені?
2) Для чого використовуються статичні змінні?
3) Як оголошуються статичні поля в класі?
4) Як можна звернутися до статичного елементу класу у програмі?
5) Поясніть призначення класу «*Singleton»*.

7. УСПАДКУВАННЯ

Успадкування – один із трьох фундаментальних принципів об`єктно-орієнтовного програмування, оскільки саме завдяки йому можливе створення ієрархічних класифікацій. Використовуючи успадкування, можливо створити загальний клас, який визначає характеристики, що властиві множині зв`язаних елементів. Цей клас може бути успадкований іншими, вузькоспеціалізованими класами з додаванням у кожен з них своїх, унікальних особливостей (змінення існуючих методів і долучення власних полів і методів) [8, 9, 11, 21].

Клас, який успадковується, називається *базовим* (або *предком*). Клас, який успадковує базовий клас, називається *похідним* (або *нащадком*). Похідний клас можна використовувати як базовий для іншого похідного класу. Таким чином вбудовується багаторівнева ієрархія класів.

Найважливішою властивістю успадкування є те, що воно дає можливість уникати повторень коду, адже спільний для множини подібних класів код може бути винесено у методи їх спільного предка.

Для створення похідного класу використовується ключове слово ***class*,** після якого слід записати ім`я нового класу, двокрапку, ключ доступу класу (*public, private, protected*), а потім ім`я базового класу:

class <*ім`я_похідного_класу*>:[public|protected|private]<*ім`я_базового_класу*>

{

<тіло класу>

};

У наступному прикладі клас ***Person*** є базовим, а клас ***Employee*** – похідним.

```
#include <iostream>
#include <string>

using namespace std;
class Person //базовий клас
{
public:
   string name;
    int age;
```

```
    void display()
    {
        cout << "Name: " << name << "\tAge: " << age <<endl;
    }
};
class Employee : public Person
{
public:
    string company;
};

int main()
{
    Person tom;
    tom.name = "Tom";
    tom.age = 23;
    tom.display();

    Employee bob;
    bob.name = "Bob";
    bob.age = 31;
    bob.company = "Microsoft";
    bob.display();

    return 0;
}
```

7.1. Керування доступом до членів базового класу

Якщо один клас успадковує інший, члени базового класу стають членами похідного. Статус доступу членів базового класу в похідному класі визначається специфікатором доступу, який використовується для успадкування базового класу [12, 16]. Специфікатор доступу базового класу виражається одним з ключових слів: *public*, *private* або *protected.*

Якщо специфікатор доступу не вказаний, то за замовчуванням використовується специфікатор *private.*

При оголошенні члена класу відкритим (з використанням ключового

слова *public*), до нього можна отримати доступ із будь-якої частини програми. Якщо член класу оголошується закритим (за допомогою специфікатора *private*), до нього можуть отримати доступ тільки члени того самого класу. До закритих членів базового класу не мають доступу похідні класи. Якщо член класу оголошується захищеним (*protected*-членом), до нього можуть отримати доступ лише члени цього або похідних класів. Таким чином, специфікатор *protected* дозволяє успадковувати члени, але залишає їх закритими в рамках ієрархії класів.

Якщо базовий клас успадковується з використанням ключового слова *public*, його *public*-члени стають *public*-членами похідного класу, а його *protected*-члени – *protected*-членами похідного класу.

Якщо базовий клас успадковується з використанням ключового слова *protected*, його *public*- і *protected*-члени стають *protected*- членами похідного класу.

Якщо базовий клас успадковується з використанням ключового слова *private*, його *public*- і *protected*-члени стають *private*- членами похідного класу.

В усіх випадках *private*-члени базового класу залишаються закритими в рамках цього класу і не успадковуються.

Керування доступом до елементів базового і похідного класу при різних ключах доступу під час успадкування наведено в таблиці 1.

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};
class B : public A
{
  // x – public
  // y – protected
```

```
  // z – не має доступу
};
class C : protected A
{
      // x – protected
      // y – protected
      // z – не має доступу
};
class D : private A
{
  // x – private
  // y – private
  // z – не має доступу
};
```

Таблиця 1 – Режими доступу до елементів при успадкуванні

Режим доступу до елемента в базовому класі	**Ключ доступу при успадкуванні класу**	**Режим доступу до елемента в похідному класі**
private	**public**	недоступний
protected		protected
public		public
private	**protected**	недоступний
protected		protected
public		protected
private	**private**	недоступний
protected		private
public		private

7.2. Конструктори та деструктори при успадкуванні

Оскільки конструктори не успадковуються, при створенні похідного класу, члени, які ним успадковуються, мають бути ініціалізовані конструктором базового класу. Конструктор базового класу викликається автоматично і виконується до конструктора похідного класу [19, 22].

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

class base {
```

```
public:
base() { cout <<"Створення base-об`єкта.\n"; }
~base() { cout <<"Знищення base-об`єкта.\n"; }
};
class derived: public base {
public:
derived() { cout <<" Створення derived-об`єкта.\n";}
~derived() { cout <<" Знищення derived-об`єкта.\n";}
};
int main()
{
derived ob;
return 0;
}
```

Результат виконання програми наведено на рис. 16.

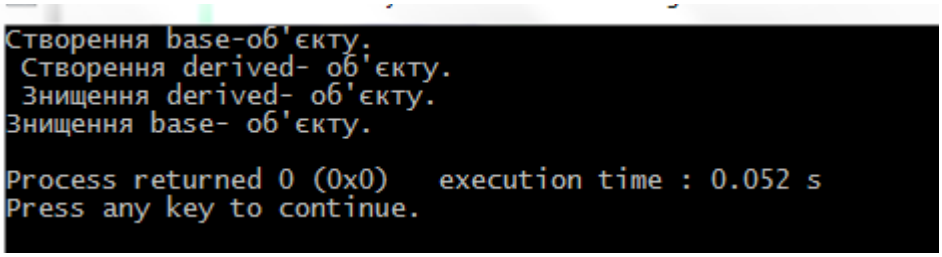


Рис. 16. Результат роботи програми

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout <<" Створення base-об`єкта.\n"; }
~base(){ cout <<" Знищення base-об`єкта.\n"; }
};
class derived1 : public base {
public:
derived1() { cout <<"Створення derived1-об`єкта.\n";}
~derived1(){ cout <<"Знищення derived1-об`єкта.\n";}
};
class derived2: public derived1 {
public:
derived2() { cout <<"Створення derived2-об`єкта\n";}
~derived2(){ cout <<"Знищення derived2-об`єкта.\n";}
```

```
};
int main()
{
derived2 ob;
return 0;
}
```

Результат виконання програми наведено на рис. 17.

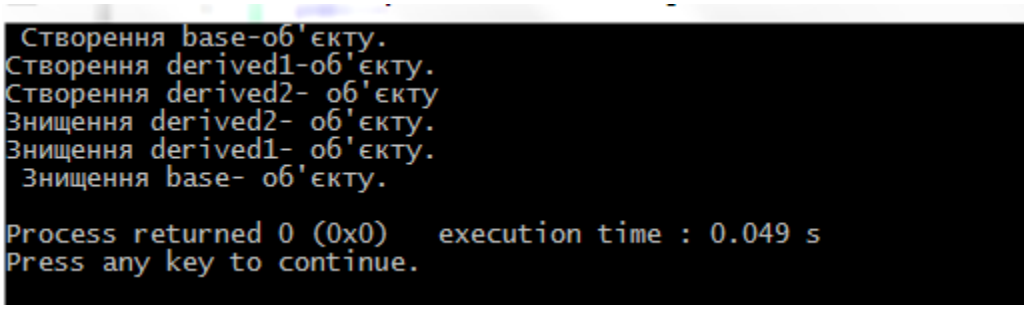


Рис. 17. Результат роботи програми

Таким чином, конструктори викликаються по черзі походження класів, а деструктори – в зворотному порядку.

Параметри конструктора базового класу вказуються при визначенні конструктора похідного класу. Таким чином, виконується передача аргументів від конструктора похідного класу конструктору базового класу.

У випадку, коли необхідно передати параметри конструктору базового класу, необхідно використовувати розширену форму оголошення конструктора похідного класу, в якій передбачена можливість передачі аргументів одному чи декільком конструкторам базового класу. Загальний формат такого розширеного оголошення:

Конструктор_похідного_класу (список аргументів)
: конструктор_базового_класу (список аргументів)
{
тіло конструктора похідного класу
}

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

```
{
protected:
    int i;
public:
    base (int x)
    {
    i = x;
    cout << "Створення base-об`єкта.\n";
    }
    ~base() {cout << "Знищення base- об`єкта.\n";}
};
class derived: public base
{
    int j;
public:
// Класс derived використовує параметр x, а параметр y передається конструктору класу base.
    derived(int x, int y): base(y)
    {
        j = x;
        cout << "Створення derived1-об`єкта.\n";
    }
    ~derived() { cout <<"Знищення derived1- об`єкта.\n";}
    void show() { cout << i << " " << j << "\n"; }
};
int main()
{
    derived ob(3, 4)
    ob.show(); //
    return 0;
}
```

Результат виконання програми наведено на рис. 16.

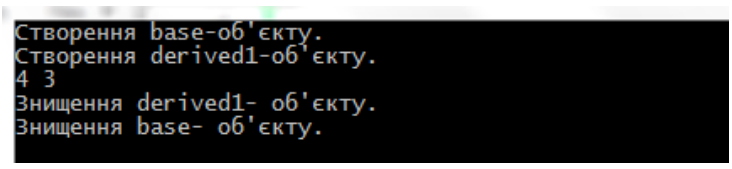


Рис. 18. Приклад виконання програми

Якщо базовий клас містить тільки конструктори з параметрами, то

похідний клас має викликати в своєму конструкторі один з конструкторів базового класу.

7.3. Принцип підстановки

Відкрите успадкування встановлює між класами ***відношення «є»:*** похідний клас є частиною базового класу. Це означає, що усюди, де може бути використаний об`єкт базового класу (при присвоюванні, при передачі параметрів та поверненні результату), замість нього дозволяється використовувати об`єкт похідного класу. Це положення має назву «принцип підстановки». При цьому зворотне невірно, тобто будь-який студент (похідний клас) є людиною (базовий клас), але не будь-яка людина є студентом [3].

Розглянемо основні елементи успадкування класів на наступному прикладі. Створимо базовий клас ***Car*** (машина), що характеризується торговою маркою, числом циліндрів, потужністю. Визначимо методи перепризначення і зміни потужності. Створимо похідний клас ***Lorry*** (вантажівка), додамо в нього характеристику вантажопідйомності кузова. Визначимо функції перепризначення марки і зміни вантажопідйомності. Реалізуємо функцію, яка одержує і повертає об`єкти базового класу. Застосуємо на практиці також на принцип підстановки.

UML-діаграма класів ***Car*** та ***Lorry*** наведено на рисунку 19.

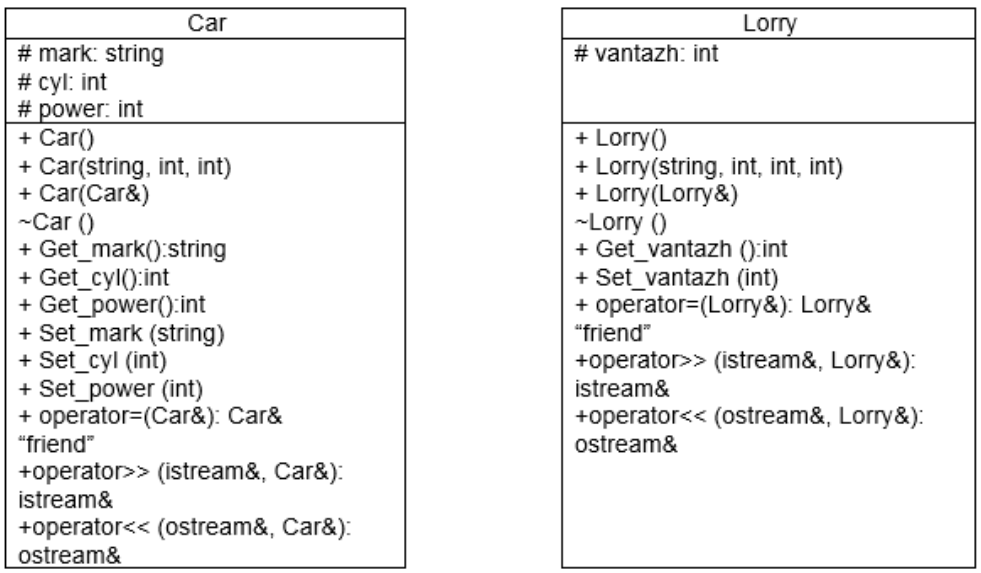


Рис. 19. UML-діаграма класів ***Car*** та ***Lorry***

- Заголовковий файл «Car.h»

```
#ifndef CAR_H_INCLUDED
#define CAR_H_INCLUDED
#include <string>
#include <iostream>

using namespace std;

class Car
{
public:
    Car();//конструктор без параметрів за замовчуванням.
    ~Car();//деструктор.
    Car(string,int,int);  //конструктор з параметрами.
    Car(const Car&);//конструктор копіювання.
    string Get_mark(){return mark;}
    int Get_cyl(){return cyl;}
    int Get_power(){return power;}
    void Set_mark(string);
    void Set_cyl(int);
    void Set_power(int);
    Car& operator=(const Car&);//перевантаження оператора  присвоювання.
//Перевантаження операторів введення-виведення.
friend istream& operator>>(istream&in,Car&c);
friend ostream& operator<<(ostream&out,const Car&c);

protected:
    string mark;
    int cyl;
    int power;
};

#endif // CAR_H_INCLUDED
```

- Файл із реалізацією класу «Car.cpp»

```
#include "Car.h"

Car::Car()
{
    mark="";
```

```
    cyl=0;
    power=0;
}
Car::~Car(){}
Car::Car(string M,int C,int P)
{
    mark=M;
    cyl=C;
    power=P;
}
Car::Car(const Car& car)
{
    mark=car.mark;
    cyl=car.cyl;
    power=car.power;
}
void Car::Set_cyl(int C)
{
    cyl=C;
}
void Car::Set_mark(string M)
{
    mark=M;
}
void Car::Set_power(int P)
{
    power=P;
}
Car& Car::operator=(const Car&c)
{
    if(&c==this)return *this;
    mark=c.mark;
    power=c.power;
    cyl=c.cyl;
    return *this;
}
istream& operator>>(istream&in,Car&c)
{
    cout<<"\nMark:";
    in>>c.mark;
    cout<<"\nPower:";
    in>>c.power;
    cout<<"\nCyl:";
    in>>c.cyl;
    return in;
}
ostream& operator<<(ostream&out,const Car&c)
{
    out<<"\nMARK : "<<c.mark;
    out<<"\nCYL : "<<c.cyl;
    out<<"\nPOWER : "<<c.power;
    out<<"\n";
    return out;
}
```

- Заголовковий файл «Lorry.h»

```
#ifndef LORRY_H_INCLUDED
#define LORRY_H_INCLUDED

#include "car.h"
//клас Lorry успадковується від класу Car з ключом доступу public.
class Lorry :public Car
{
public:
    Lorry();//конструктор без параметрів за замовчуванням.
    ~Lorry();//деструктор.
    Lorry(string,int,int,int);//конструктор з параметрами.
    Lorry(const Lorry & );//конструктор копіювання.
    int Get_vantazh (){return vantazh;}
    void Set_vantazh (int);
    Lorry& operator=(const Lorry&);//перевантаження оператора присвоювання.
    friend istream& operator>> (istream &in, Lorry&l);
//Перевантаження операторів введення та виведення.
    friend ostream& operator<<(ostream&out,const Lorry&l);
protected:
    int vantazh;//атрибут вантажності.
};
#endif // LORRY_H_INCLUDED
```

- Файл з реалізацією класу «Lorry.cpp»

```
#include "Lorry.h"
Lorry::Lorry():Car()
{
    vantazh =0;
}
Lorry::~Lorry(){}

Lorry::Lorry(string M, int C, int P, int G):Car(M,C,P)
{
    vantazh =G;
}
Lorry::Lorry(const Lorry &L)
{
    mark=L.mark;
    cyl=L.cyl;
    power=L.power;
    vantazh =L.vantazh;
}
void Lorry::Set_vantazh (int G)
{
    vantazh =G;
}
Lorry& Lorry::operator=(const Lorry&l)
{
    if(&l==this) return *this;
    mark=l.mark;
    power=l.power;
```

```
    cyl=l.cyl;
   vantazh =l.vantazh;
    return *this;
}
istream& operator>>(istream&in,Lorry&l)
{
    cout<<"\nMark:";
    in>>l.mark;
    cout<<"\nPower:";
    in>>l.power;
    cout<<"\nCyl:";
    in>>l.cyl;
    cout<< "\nVantazh:";
    in>>l.vantazh;
    return in;
}
ostream& operator<<(ostream&out,const Lorry&l)
{
    out<<"\nMARK : "<<l.mark;
    out<<"\nCYL : "<<l.cyl;
    out<<"\nPOWER : "<<l.power;
    out<<"\nVANTAZH: "<<l. vantazh;
    out<<"\n";
    return out;
}
```

- Файл «main.cpp» з прикладом використання розроблених класів

```
#include <iostream>
#include "Car.h"
#include "Lorry.h"

using namespace std;

void f1(Car&c)
{
    c.Set_mark("Opel");
    cout<<c;
}
Car f2()
{
Lorry l("Kia",1,2,3);
return l;
}

int main()
{
//Робота з класом Car.
Car a;
cin>>a;
cout<<a;
Car b("Ford",4,115);
cout<<b;
a=b;
cout<<a;
//Робота з класом Lorry.
```

```
Lorry c;
cin>>c;
cout<<c;
//Принцип підстановки.
f1(c);//передаємо об`єкт класу Lorry.
a=f2();//створюємо в функції об`єкт класу Lorry.
cout<<a;
}
```

Приклад виконання програми наведено на рис. 20.

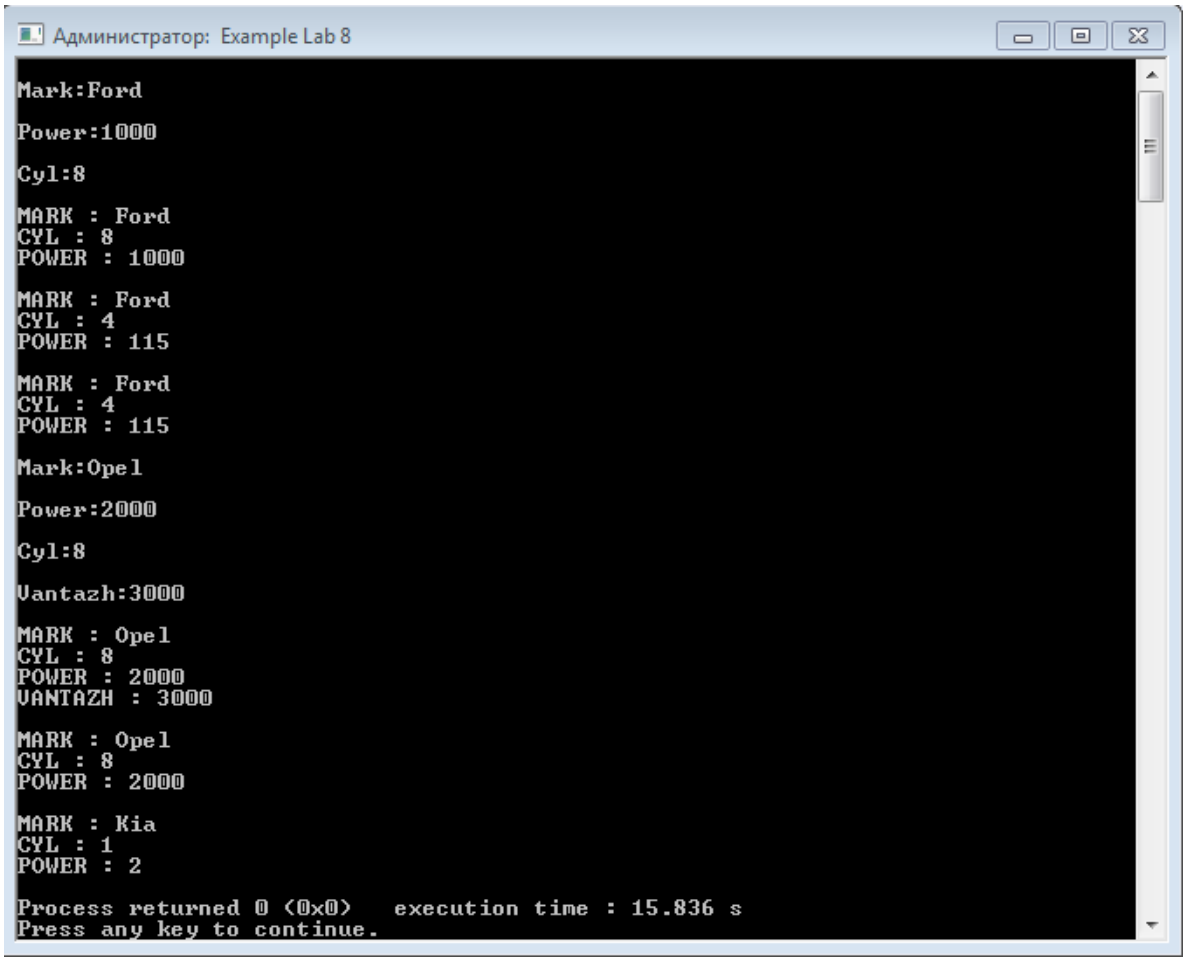


Рис. 20. Приклад виконання програми

7.4. Завдання для самостійного виконання

Хід виконання завдання:

1) Створити клас для заданого поняття (базовий клас) з відповідними полями та методами.
2) Створити похідний клас.
3) Реалізувати функцію, яка одержує та повертає об`єкти базового класу, продемонструвати принцип підстановки.

Варіанти завдань

1) Створити клас *Man* (чоловік), із полями: ім`я, вік, стать і вага. Визначити методи перепризначення імені, зміни віку і зміни ваги. Створити похідний клас *Employee*, що має поля – посада та оклад. Визначити методи зміни полів
2) Створити клас *Pair* (пара чисел); визначити методи зміни полів і порівняння пар: пара *p1* більша від пари *p2*, якщо (*pl.first* > *p2.first*) або (*pl.first* = *p2.first*) і (*p1.second* > *p2.second*). Визначити клас-спадкоємець *Fraction* із полями: ціла частина числа і дрібна частина числа. Визначити повний набір методів порівняння.
3) Створити клас *Liquid* (рідина), який має поля назви і щільності. Визначити методи перепризначення і зміни щільності. Створити похідний клас *Alcohol* (спирт), який має міцність. Визначити методи перепризначення і зміни міцності.
4) Створити клас *Pair* (пара чисел); визначити методи зміни полів і обчислення добутку чисел. Визначити похідний клас *Rectangle* (прямокутник) з полями-сторонами. Визначити методи обчислення периметра та площі прямокутника.
5) Створити клас *Man* (чоловік), із полями: ім`я, вік, стать і вага. Визначити методи перепризначення імені, зміни віку та зміни ваги. Створити похідний клас *Student*, що має поле року навчання. Визначити методи перепризначення і збільшення року навчання.
6) Створити клас *Triad* (трійка чисел); визначити методи зміни полів і обчислення суми чисел. Визначити похідний клас *Triangle* з полями-сторонами. Визначити методи обчислення кутів і площі трикутника.
7) Створити клас *Triangle* з полями-сторонами. Визначити методи зміни сторін, обчислення кутів, обчислення периметра. Створити похідний клас *Equilateral* (рівносторонній), що має поле площі. Визначити метод обчислення площі.
8) Створити клас *Triangle* з полями-сторонами. Визначити методи зміни

сторін, обчислення кутів, обчислення периметра. Створити похідний клас *RightAngled* (прямокутний), який має поле площі. Визначити метод обчислення площі.

9) Створити клас *Pair* (пара чисел); визначити методи зміни полів і обчислення добутку чисел. Визначити похідний клас *RightAngled* з полями-катетами. Визначити методи обчислення гіпотенузи і площі трикутника.
10) Створити клас *Triad* (трійка чисел); визначити метод порівняння тріад. Визначити похідний клас *Time* з полями: година, хвилина та секунда. Визначити повний набір методів порівняння моментів часу.
11) Створити клас *Triad* (трійка чисел); визначити методи збільшення полів на 1. Визначити клас-спадкоємець *Time* з полями: година, хвилина, секунда. Перевизначити методи збільшення полів на 1 і визначити методи збільшення на *n* секунд і хвилин.
12) Створити базовий клас *Pair* (пара цілих чисел) з операціями перевірки на рівність і перемноження полів. Реалізувати операцію віднімання пар за формулою $(a, b) - (c, d) = (a - b, c - d)$. Створити похідний клас *Rational*; визначити нові операції додавання $(a, b) + (c, d) = (ad + bc, bd)$ і ділення $(a, b) / (c, d) = (ad, bc)$; перевизначити операцію віднімання $(a, b) - (c, d) = (ad - bc, bd)$.
13) Створити клас *Pair* (пара чисел); визначити метод перемноження полів та операцію складання пар $(a, b) + (c, d) = (a + b, c + d)$. Визначити похідний клас *Complex* із полями: дійсна й уявна частини числа. Визначити методи множення $(a, b) * (c, d) = (ac - bd, ad + bc)$ і віднімання $(a, b) - (c, d) = (a - b, c - d)$.
14) Створити клас *Pair* (пара цілих чисел); визначити методи зміни полів та операцію складання пар $(a, b) + (c, d) = (a + b, c + d)$. Визначити клас-спадкоємець *Long* із полями: старша частина числа та молодша частина числа. Перевизначити операцію складання і визначити методи множення та віднімання.
15) Створити базовий клас *Triad* (трійка чисел) із операціями додавання

числа, множення на число, перевірки на рівність. Створити похідний клас *vector3D*, що задається трійкою координат. Повинні бути реалізовані: операція додавання векторів, скалярний добуток векторів.

16) Створити клас *Pair* (пара цілих чисел); визначити метод множення на число й операцію складання пар $(a, b) + (c, d) = (a + b, c + d)$. Визначити клас-спадкоємець *Money* з полями: гривні та копійки. Перевизначити операцію складання та визначити методи вирахування і розподілу грошових сум.
17) Створити клас *Man* (чоловік), із полями: ім`я, вік, стать. Визначити методи перепризначення імені, зміни віку. Створити похідний клас *Teacher*, що має поля: предмет і кількість годин. Визначити методи зміни полів, а також збільшення та зменшення годин.

7.5. Контрольні запитання

1) Поясніть поняття успадкування в контексті ООП? Для чого застосовується успадкування класів?
2) Які види успадкування Вам відомі?
3) Що таке ключ доступу при успадкуванні класів? На що він впливає?
4) Як викликаються конструктори при успадкуванні? Чи успадковуються конструктори?
5) Як можна передати параметри конструктору базового класу через похідний?
6) Як викликаються деструктори при успадкуванні? Чи успадковуються деструктори?
7) Поясніть принцип підстановки.

8. ПОЛІМОРФІЗМ В С++

Поняття *поліморфізму* дослівно означає можливість приймати різноманітні форми, зберігаючи суть, або одночасно приймати різноманітні форми. У програмуванні можна дати визначення *поліморфізму* таке, що одне і те ж ім`я може використовуватися для логічно пов`язаних, але різних цілей, тобто ім`я визначає клас дій, які в залежності від типу даних можуть істотно відрізнятися. Поліморфізм в програмуванні проявляється, наприклад, в перевантаженні функцій, операторів, операцій. В контексті *успадкування поліморфізм* можна розглядати, як можливість об`єктів різних класів, що пов`язані відносинами наслідування, реагувати по різному під час виклику одного й того-самого методу [9, 11, 14].

Наприклад, базовий клас *Quad* (Чотирикутник) та похідний від нього клас *Rectangle* (Прямокутник), обидва можуть містити методи розрахунку площі або периметра, але, при цьому, такі методи будуть розрізнятись між собою за реалізацією.

8.1. Віртуальні функції

Реалізація динамічного поліморфізму в С++ здійснюється завдяки поєднання успадкування і *віртуальних функцій* (методів) – функцій, що оголошуються в базовому класі з використанням ключового слова *virtual* і перевизначаються в одному або декількох похідних класах. Таким чином, кожний похідний клас може мати власну версію віртуальної функції.

По відношенню до всіх віртуальних методів компілятор застосовує стратегію *пізнього* або *динамічного зв`язування*. Це означає, що на етапі компіляції він не визначає, який з методів повинен бути викликаний, а передає відповідальність програмі, яка приймає рішення на етапі виконання, коли вже точно відомо, який тип об`єкта, на який вказує наш покажчик.

Важливим моментом забезпечення ідеї поліморфізму є те, що звернення до віртуальної функції відбувається через покажчик (або посилання) на базовий клас, у такому випадку компілятор С++ автоматично визначає, яку саме версію віртуальної функції потрібно викликати, по типу об`єкта, що адресується цим

покажчиком, такий вибір відбувається під час виконання програми. Ключове слово *virtual* також може вказуватись і перед назвами методів в похідних класах, але це не є обов`язковим.

Іншими словами, саме за типом об`єкта, що адресується, визначається, яка версія віртуального методу буде виконана.

Віртуальна функція, як говорилося вище, оголошується в базовому класі з використанням ключового слова *virtual*. При визначенні віртуальної функції в похідному класі ключове слово *virtual* повторювати не потрібно.

Якщо віртуальна функція перевизначається в похідному класі, її називають *перевизначеною.*

Поліморфний клас – клас, який включає віртуальну функцію.

Розглянемо використання віртуальної функції наступному прикладі:

```
class A{
public:
     virtual void who() { // оголошення віртуальної функції.
          cout << "Базовий клас.\n ";
     }
};
class A1 : public A {
public:
     void who() { // перевизначення функції who() для класу A1,
                  // ключове слово virtual не є обов`язковим.
          cout << "Перший похідний клас.\n ";
     }
};
class A2 : public A {
public:
     void who() { // ще одне перевизначення функції who() для класу A2.
          cout << "Другий похідний клас.\n ";
     }
};
int main() {
     A base_object;
     A *p;
     A1 a1_object;
```

```
    A2 a2_object;
    p = &base_object;  // встановлюємо покажчик на об`єкт базового класу.
    p->who();//викликається метод who() класу A.
    p = &a1_object;  // встановлюємо покажчик на об`єкт класу A1.
    p->who();//викликається метод who() класу A1.
    p = &a2_object;  // встановлюємо покажчик на об`єкт класу A2.
    p->who();//викликається метод who() класу A2.
    return 0;
}
```

Результат виконання програми наведено на рис. 21.

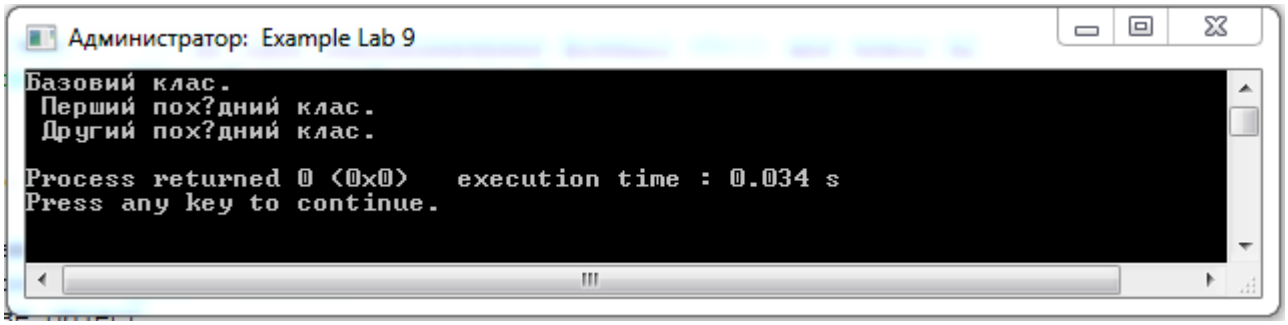


Рис. 21. Приклад виконання програми

На відміну від перевантаженої функції в похідному класі, кількість і тип параметрів віртуальних функцій у базовому та похідному класах повинні точно збігатися і мати однакові прототипи. Якщо прототипи таких функцій будуть відрізнятись, то функція в похідному класі буде вважатись компілятором просто перевантаженою, а не віртуальною. Крім того, віртуальна функція має бути членом класу, для якого вона визначається, а не його «другом», але віртуальна функція може бути дружньою для іншого класу [14, 15]. Деструктори в С++ можуть бути *віртуальними*, а конструктори – ні.

Наслідування віртуальних функцій. Якщо функція оголошена як віртуальна, то вона залишається такою незалежно від кількості рівнів похідних класів, у яких вона використана. Наприклад, якщо б клас ***A2*** було б наслідувано від ***A1***, а не від класу ***A***, то функція *who*() все одно б залишалась віртуальною і механізм поліморфізму працював би коректно:

```
class A2 : public A1 {
public:
```

```
     void who() {
          cout << "Другий похідний клас.\n";
     }
};
int main() {
     ...
     p = &a2_object;
     p->who();
     return 0;
}
```

Якщо похідний клас не перевизначає віртуальну функцію, то використовується варіант функції, визначений у базовому класі:

```
...
class A2 : public A { // функцію who() не визначено.
};
...
int main() {
     ...
     p = &a2_object;
     p->who();
     return 0;
}
```

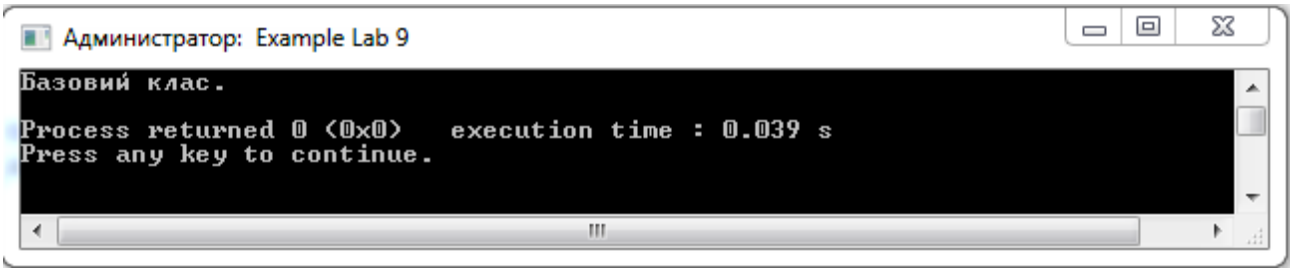


Рис. 22. Приклад виконання програми

8.2. Суто віртуальні функції та абстрактні класи

В багатьох випадках базовий клас може задавати тільки каркас для поняття, і в ньому не може бути реалізовано деякі функції.

Абстрактний клас – це клас, який висловлює якусь загальну концепцію, яка відображатиме основну ідею для використання в похідних класах. Абстрактний клас створюють тільки для того, щоб на його основі створювати інші класи. Створювати екземпляри об`єктів таких класів не можна, тому їх називають абстрактними.

В таких випадках у базовому класі оголошуються функції без чіткої реалізації, які повинні бути обов`язково реалізовані в похідних класах [19, 20]. Такі функції мають назву *суто віртуальні*, будь-який похідний клас від базового, що містить суто віртуальну функцію, має примусово реалізувати таку функцію. Формат оголошення суто віртуальної функції в базовому класі:

```
virtual тип ім`я_функції(список_параметрів) = 0;
```

Клас, який містить хоча б одну суто віртуальну функцію – *абстрактний клас*. В абстрактного класу не може бути об`єктів, спроба створення об`єктів такого класу призведе до помилки компіляції, винятком є оголошення покажчика на тип базового класу, який потім використовується для вказання об`єктів похідних класів (див. приклади вище):

```
#include <iostream>

using namespace std;

class A{ // оголошення абстрактного класу A.
public:
     virtual void who() = 0; // метод who() є суто віртуальним.
};

class A1 : public A {
public:
     void who() { // визначення функції who() для класу A1.
          cout << "Перший похідний клас.\n ";
     }
};

class A2 : public A { // функцію who() не визначено для класу A2.
};
```

```
int main() {
     // A base_object; // !помилка компіляції при спробі створення об`єкта
                       //абстрактного класу.

     A *p; // покажчик на об`єкт абстрактного класу є допустимим.

     A1 a1_object; // об`єкт буде створено, оскільки метод who() реалізовано.

     // A2 a2_object; // помилка компіляції при спробі створення об`єкта
                       //похідного класу без реалізованої віртуальної функції.
     p = &a1_object; // єдиний коректний варіант виклику методу who() .
                       // можливий тільки для класу A1.
     p->who();
     return 0;
}
```

Результат виконання програми наведено на рис. 23.

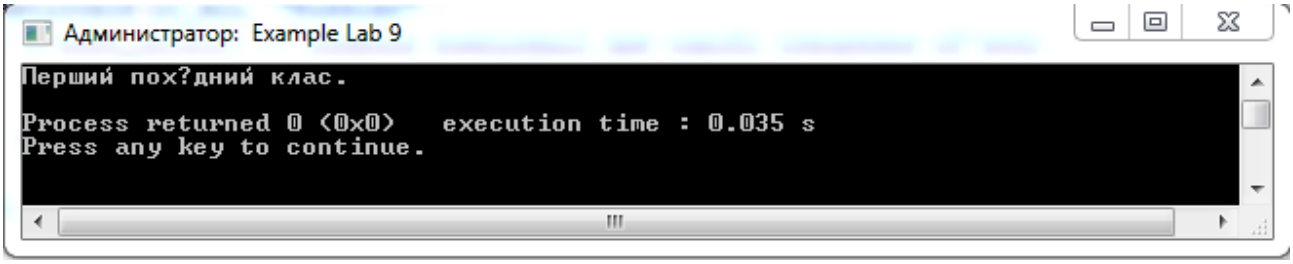


Рис. 23. Приклад виконання програми

Ще один приклад використання абстрактних класів та поліморфізму:

```
#include <iostream>
#include <string>

using namespace std;

class Animal { // Базовий клас
protected:
     int size;
public:
     void setSize(int s){
          size = s;
```

```
    };
    virtual string soundsLike() = 0;
};
class Cat : public Animal {
public:
    string  soundsLike(){
        return "Meow!";
    }
};
class Dog : public Animal {
public:
    string  soundsLike(){
        return "Woof!";
    }
};
class Tiger : public Cat {
public:
    string  soundsLike(){
        if(size < 50)
            return Cat::soundsLike();
        else
            return "RRRRR!";
    }
};
int main()
{
    Animal *a;
    Cat cat;
    Dog dog;
    Tiger tiger;
    tiger.setSize(60);
    a = &cat;
    cout << "A cat says " << a->soundsLike() << endl;
    a = &dog;
    cout << "A dog says " << a->soundsLike() << endl;
    a = &tiger;
    cout << "A tiger says " << a->soundsLike() << endl;
    return 0;
}
```

Результат виконання програми наведено на рис. 24.

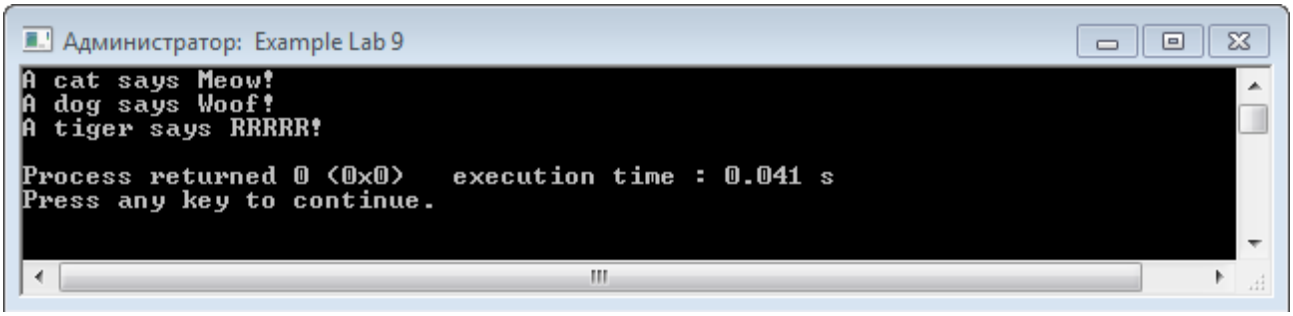


Рис.24. Приклад виконання програми

8.3. Завдання для самостійного виконання

Хід виконання завдання:

1) Створити абстрактний клас для заданого поняття з вказаною віртуальною функцією.
2) Створити два класи-нащадки з реалізацією віртуальної функції.
3) Написати програму, яка демонструє поліморфізм створених класів.

Варіанти завдань

1) Базовий абстрактний клас ***Polygon*** (Плоский багатокутник) містить суто віртуальну функцію *area*() (площа фігури). Похідні класи ***Triangle*** (Трикутник) та ***Rectangle*** (Прямокутник) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться відповідні площі.

2) Базовий абстрактний клас ***Solid*** (Тверде тіло) містить суто віртуальну функцію *area*() (повна площа поверхні тіла). Похідні класи ***Sphere*** (Сфера) та ***Cube*** (Куб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться площі їх поверхні.

3) Базовий абстрактний клас ***Number*** (Число) містить суто віртуальну функцію *toFloat*() (перетворення в дійсне число). Похідні класи ***Rational*** (Раціональне число) та ***Decimal*** (Десятковий дріб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться результати їх перетворення в дійсне число.

4) Базовий абстрактний клас ***Vector*** (Радіус-вектор) задається координатами точки та містить суто віртуальну функцію *add*() (додавання скаляра до усіх координат). Похідні класи ***Vector2D*** (Вектор на площині) та ***Vector3D*** (Вектор у просторі) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів, і через відповідні гетери виводяться їх координати перед та після операції додавання.

5) Базовий абстрактний клас ***Number*** (Число) містить суто віртуальну функцію *toStr*() (перетворення в рядок). Похідні класи ***Mixed*** (Мішаний дріб) та ***Complex*** (Комплексне число) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться їх рядкові представлення.

6) Базовий абстрактний клас ***Polygon*** (Плоский багатокутник) містить суто віртуальну функцію *area*() (площа фігури). Похідні класи ***Trapezoid*** (Трапеція) та ***Rectangle*** (Прямокутник) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться відповідні площі.

7) Базовий абстрактний клас ***Number*** (Число) містить суто віртуальну функцію *add*() (складення з іншим числом). Похідні класи ***Rational*** (Раціональне число) та ***Decimal*** (Десятковий дріб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться результати операції додавання аналогічного числа: *Rational+Rational* та *Decimal+Decimal*.

8) Базовий абстрактний клас ***Solid*** (Тверде тіло) містить суто віртуальну функцію *volume*() (об`єм тіла). Похідні класи ***Cone*** (Конус) та ***Cube*** (Куб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться їх об`єми.

9) Базовий абстрактний клас ***Polygon*** (Плоский багатокутник) містить суто віртуальну функцію *perimeter*() (периметр фігури). Похідні класи ***Trapezoid*** (Трапеція) та ***Square*** (Квадрат) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться відповідні периметри.

10) Базовий абстрактний клас ***Number*** (Число) містить суто віртуальну функцію *multiply*() (множення на інше число). Похідні класи ***Rational*** (Раціональне число) та ***Decimal*** (Десятковий дріб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться результати операції множення на аналогічне число: *Rational*Rational* та *Decimal*Decimal*.

11) Базовий абстрактний клас ***Polygon*** (Плоский багатокутник) містить суто віртуальну функцію *perimeter*() (периметр фігури). Похідні класи ***Triangle*** (Трикутник) та ***Trapezoid*** (Трапеція) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться відповідні периметри.

12) Базовий абстрактний клас ***Line*** (Лінія) задається списком координат її точок та містить суто віртуальну функцію *length*() (довжина лінії). Похідні класи ***Segment*** (Відрізок) та ***Polyline*** (Ламана) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів, і через відповідні гетери виводяться списки їх координат вершин і відповідні довжини.

13) Базовий абстрактний клас ***Number*** (Число) містить суто віртуальну функцію *toInt*() (перетворення в ціле число). Похідні класи ***Rational*** (Раціональне число) та ***Decimal*** (Десятковий дріб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться результати їх перетворення в ціле число.

14) Базовий абстрактний клас ***Solid*** (Тверде тіло) містить суто віртуальну функцію *volume*() (об`єм тіла). Похідні класи ***Cylinder*** (Циліндр) та ***Sphere*** (Сфера) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться їх об`єми.

15) Базовий абстрактний клас ***Vector*** (Радіус-вектор) задається координатами точки та містить суто віртуальну функцію *module*() (модуль або довжина вектора). Похідні класи ***Vector2D*** (Вектор на площині) та ***Vector3D*** (Вектор у просторі) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів, і через відповідні гетери

виводяться списки їх координат і відповідні довжини.

16) Базовий абстрактний клас ***Vector*** (Радіус-вектор) задається координатами точки та містить суто віртуальну функцію *scale*() (масштабування або множення на скаляр усіх координат вектора). Похідні класи ***Vector2D*** (Вектор на площині) та ***Vector3D*** (Вектор у просторі) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів, і через відповідні гетери виводяться їх координати перед та після операції масштабування.

17) Базовий абстрактний клас ***Number*** (Число) містить суто віртуальну функцію *toStr*() (перетворення в рядок). Похідні класи ***Rational*** (Раціональне число) та ***Decimal*** (Десятковий дріб) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться результати їх рядкові представлення.

18) Базовий абстрактний клас ***Solid*** (Тверде тіло) містить суто віртуальну функцію *area*() (повна площа поверхні тіла). Похідні класи ***Cylinder*** (Циліндр) та ***Cone*** (Конус) мають реалізувати вказану віртуальну функцію. Написати програму, в якій створюються об`єкти похідних класів і виводяться площі їх поверхні.

8.4. Контрольні запитання

1) Поясність поняття поліморфізму в контексті ООП.

2) Для чого застосовуються віртуальні функції?

3) Що таке раннє та пізнє зв`язування?

4) Як відбувається процес пізнього динамічного зв`язування для віртуальних функцій?

5) Що таке пере визначення віртуальної функції? Їх особливості.

6) Поясність застосування суто віртуальних функцій і абстрактних класів в ООП.

9. ВИКЛЮЧНІ СИТУАЦІЇ

9.1. Типи помилок, які можуть виникати в програмах

У програмах на С++ можуть виникати помилки. Розрізняють три типи помилок, які можуть виникати у програмах [11, 19]:

- *синтаксичні*. Це помилки в синтаксисі мови С++. Виявляються компілятором;
- *логічні*. Це помилки побудови алгоритму, які важко виявити на етапі розробки програми. Ці помилки виявляються на етапі виконання під час тестування роботи програми;
- *помилки часу виконання*. Такі помилки виникають під час роботи програми. Помилки часу виконання можуть бути логічними помилками програміста, помилками зовнішніх подій (наприклад, недостача оперативної пам`яті), невірним уведенням даних користувачем тощо. У результаті виникнення помилки часу виконання програма призупиняє свою роботу. Тому важливо перехопити цю помилку і правильно обробити її для того, щоб програма продовжила свою роботу без зупинки.

Для обробки помилки часу виконання застосовується механізм обробки виключних ситуацій.

9.2. Виключна ситуація та виключення

Виключна ситуація – це подія, що призвела до збою в роботі програми. У результаті виникнення виключної ситуації програма не може коректно продовжити своє виконання [9, 11, 14].

Приклади дій у програмі, що можуть призвести до виникнення виключних ситуацій:

- ділення на нуль;
- недостача оперативної пам`яті при застосуванні оператора *new* для її виділення (або іншої функції);
- доступ до елемента масиву за його межами (помилковий індекс);
- переповнення значення для деякого типу;
- добування кореня з від`ємного числа;

- інші ситуації.

9.3. Обробка виключної ситуації

Мова програмування С++ дає можливість перехоплювати виключні ситуації та відповідним чином їх обробляти.

В С++ механізм обробки виключних ситуацій ґрунтується на трьох ключових словах: ***try, catch*** і ***throw***. Вони утворюють взаємопов`язану підсистему, в якій використання одного з них передбачає застосування іншого.

Програмні інструкції, які, на думку програміста, повинні бути проконтрольовані на предмет виникнення помилок (виключень), розміщуються в *try-блоці.* Якщо виключення виникає в цьому блоці, воно генерує певну інформацію за допомогою ключового слова *throw* (викид виключення). Цей виняток може бути перехоплено програмним шляхом за допомогою *catch-блока* й оброблено відповідним чином.

```
try {
      // try-блок (блок коду, що підлягає перевірці на наявність
  //помилок)
  // генерування виключення оператором throw
}
catch(type1 argument1)
{
      // catch-блок (обробник виключення типа type1
}
```

Щоб у блоці *try* згенерувати виключну ситуацію, потрібно використати оператор *throw.* Оператор *throw* може бути викликаний всередині блоку *try* або всередині функції, яка викликається з блоку *try*.

Загальна форма оператора *throw* така:

```
throw виключення;
```

У результаті виконання оператора *throw* генерується виключення деякого типу. Це виключення має бути оброблено в блоці *catch*.

Розглянемо принцип оброблення виключної ситуації на прикладі виникнення виключних ситуацій при діленні на нуль та визначенні кореню від`ємного числа.

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

void main()
{
  // обробка виразу sqrt(a)/sqrt(b)
  double a, b;
  cout << "a = ";
  cin >> a;
  cout << "b = ";
  cin >> b;
  double c;

  try {
    // початок блока try
    if (b == 0)
      throw 1;
    if (b < 0)
      throw 2;
    if (a < 0)
      throw 2;
    c = sqrt(a) / sqrt(b);
    cout << "c = " << c << endl;
  }
  catch (int e) // перехоплення помилки
  {
    if (e == 1)
      cout << "Division by 0." << endl;
    if (e == 2)
      cout << "Negative root." << endl;
  }
}
```

Результат виконання програми наведено на рис. 25.

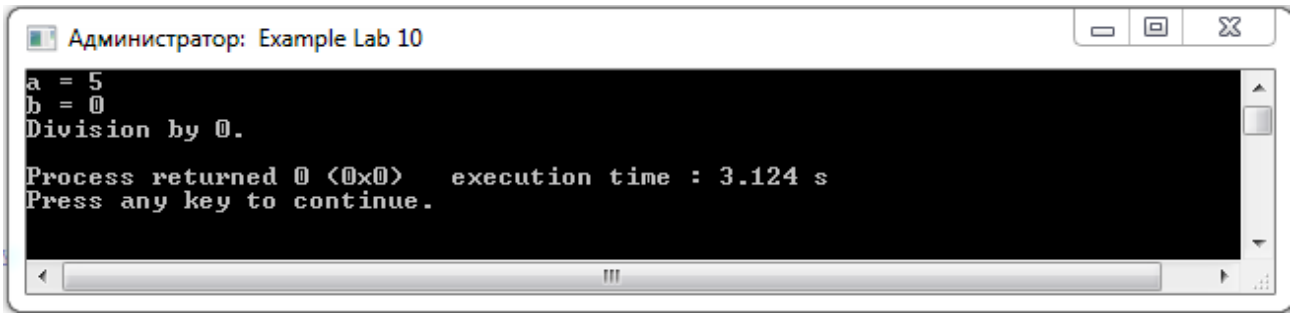


Рис.25. Приклад виконання програми

Якщо у блоці *try* виникне виключна ситуація, яка не передбачена блоком *catch*, то викликається стандартна функція *terminate*(), яка за замовчуванням викликає функцію *abort*(). Ця стандартна функція припиняє виконання програми.

Бувають випадки, коли потрібно перехопити усі виключні ситуації підряд. Для цього в C++ використовується блок *catch*(…), який має таку загальну форму.

```
catch(...)
{
   // Обробка усіх виключних ситуацій
   //  ...
}
```

Розглянемо наступний приклад, що реалізує роботу з класом *Vector* (рис.26). Розмір вектора обмежений значенням *MAX_SIZE* = 30.

Перевантажити для нього операції:

- ✓ доступ за індексом ([*int i*]),
- ✓ додавання елемента (+ *int*),
- ✓ видалення елемента (–).

Передбачити генерацію виняткових ситуацій.

Виняткові ситуації генеруються:

1) в конструкторі з параметром при спробі створити вектор більше максимального розміру;

2) в операції [ ], при спробі звернутися до елемента з номером менше 0 або більше поточного розміру вектора;
3) в операції + –, при спробі додати елемент з номером більше максимального розміру;
4) в операції, при спробі видалити елемент з пустого вектора.

Інформація про виняткові ситуації передається за допомогою призначеного для користувача класу.

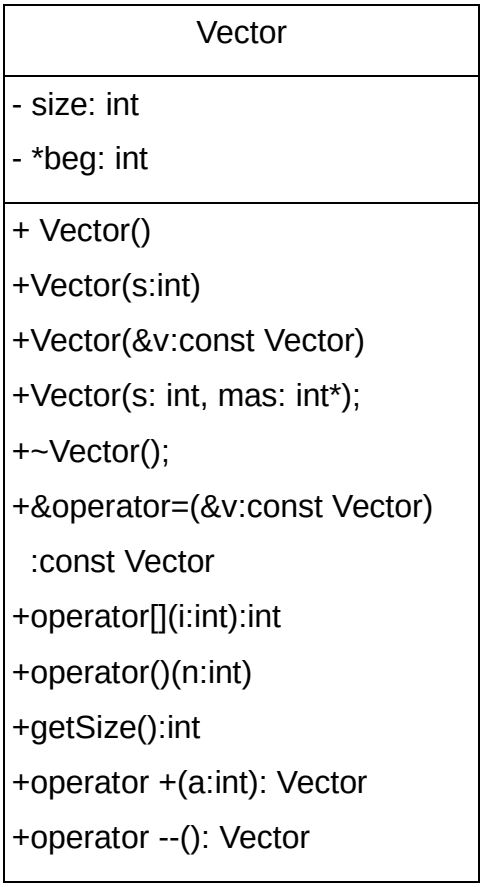


Рис. 26. *UML* діаграма класу *Vector*

- Заголовковий файл «Error.h»

```
#ifndef ERROR_H_INCLUDED
#define ERROR_H_INCLUDED
#include <string>
#include <iostream>

using namespace std;

class error //клас помилка.
```

```
{
    string str;
public:
//конструктор, ініціює атрибут str повідомленням про помилку.
    error(string s)
    {
        str=s;
    }
    void what()
    {
        cout<<str<<endl;  //виводить значення атрибута str.
    }
};

#endif
```

- Заголовковий файл «Vector.h»

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED

#include <iostream>
using namespace std;
const int MAX_SIZE=20;

class Vector
{
    int size;
    int *beg;
public:
    Vector()
    {
        size=0;
        beg=0;
    }
    Vector(int s);
    Vector(int s,int* mas);
```

```
    Vector(const Vector&v);
    ~Vector();
    const Vector& operator=(const Vector&v);
    int operator[](int i);
    Vector operator+(int a);
    Vector operator--();
    friend ostream& operator<<(ostream&out,const Vector&v);
    friend istream& operator>>(istream& in, Vector&v);
};
#endif // VECTOR_H_INCLUDED
```

- Файл з реалізацією класу «Vector.cpp»

```
#include "Vector.h"
#include "Error.h"
#include <iostream>
using namespace std;
Vector::Vector(int s)
{
    if(s>MAX_SIZE)    throw    error("Vector    length    more    than
MAXSIZE\n");
    size=s;
    beg=new int [s];
    for(int i=0; i<size; i++)
        beg[i]=i+5;
}
Vector::Vector(const Vector &v)
{
    size=v.size;
    beg=new int [size];
    for(int i=0; i<size; i++)
        beg[i]=v.beg[i];
}
Vector::~Vector()
{
    if (beg!=0) delete[]beg;
```

```
}
Vector::Vector(int s,int *mas)
{
    if(s>MAX_SIZE)    throw    error("Vector    length    more    than
MAXSIZE\n");
    size=s;
    beg=new int[size];
    for(int i=0; i<size; i++)
        beg[i]=mas[i];
}
const Vector& Vector::operator =(const Vector &v)
{
    if(this==&v)return *this;
    if(beg!=0) delete []beg;
    size=v.size;
    beg=new int [size];
    for(int i=0; i<size; i++)
        beg[i]=v.beg[i];
    return*this;
}
ostream& operator<<(ostream&out, const Vector&v)
{
    if(v.size==0) out<<"Empty\n";
    else
    {
        for (int i=0; i<v.size; i++)
            out<<v.beg[i]<<" ";
        out<<endl;
    }
    return out;
}
istream& operator >>(istream&in, Vector&v)
{
    for(int i=0; i<v.size; i++)
    {
        cout<<">";
        in>>v.beg[i];
```

```
    }
    return in;
}
int Vector::operator [](int i)
{
    if(i<0) throw error("index <0");
    if(i>=size) throw error("index>size");
    return beg[i];
}
Vector Vector::operator +(int a)
{
    if(size+1==MAX_SIZE) throw error("Vector is full");
    Vector temp(size+1,beg);
    temp.beg[size]=a;
    return temp;
}
Vector Vector::operator --()
{
    if(size==0) throw error("Vector is empty");
    if (size==1)
    {
        size=0;
        delete[]beg;
        beg=0;
        return *this;
    };
    Vector temp(size,beg);
    delete[]beg;
    size--;
    beg=new int[size];
    for(int i=0; i<size; i++)
        beg[i]=temp.beg[i];
    return*this;
}
```

- Файл «main.cpp»

```
#include "Vector.h"
#include "Error.h"
#include <iostream>
using namespace std;
int main()
{
    try
    {
        Vector x(2);
        Vector y;
        cout<<x;
        cout<<"Nomer?";
        int i;
        cin>>i;
        cout<<x[i]<<endl;
        y=x+3;
        cout<<y;
        --x;
        cout<<x;
        --x;
        cout<<x;
        --x;
    }
    catch(error &e)
    {
        e.what();
    }
    return 0;
}
```

Приклад виконання програми наведено на рис. 27.

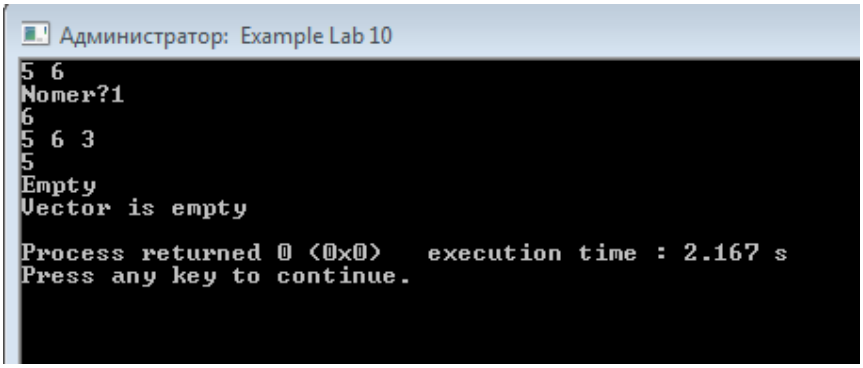


Рис. 27. Приклад виконання програми

9.4. Завдання для самостійного виконання

Хід виконання завдання:

1) Змоделювати клас (згідно варіанту).

2) Перевантажити вказані у варіанті операції.

3) Передбачити генерацію виняткових ситуацій. Інформація про виняткові ситуації передається за допомогою призначеного для користувача класу.

4) Написати програму, що створює об`єкти класу і дозволяє генерувати виняткові ситуації

Варіанти завдань

1) Клас-контейнер *Vector* з елементами типу *int*. Реалізувати операції: [] – доступу за індексом; ( ) – визначення розміру вектора; + число – додає константу до всіх елементів вектора, – *n* – видаляє *n* елементів з кінця вектора.

2) Клас-контейнер *Vector* з елементами типу *int.* Реалізувати операції: [] – доступу за індексом; *int* ()– визначення розміру вектора; – *n* – видаляє *n* елементів з кінця вектора; + *N* – додає n елементів у кінець вектора.

3) Клас-контейнер *Vector* з елементами типу *int*. Реалізувати операції: [] – доступу за індексом; + + додає елемент у вектор (постфіксна операція додає елемент у кінець, префіксна – в початок).

4) Клас-контейнер *Vector* з елементами типу *int*. Реалізувати операції: [] – доступу за індексом; ()– визначення розміру вектора; – – видаляє елемент із вектора (постфіксна операція видаляє елемент у кінці вектора, префіксна – на початку).

5) Клас-контейнер *Vector* з елементами типу *int*. Реалізувати операції: [] – доступу за індексом; *int* ()– визначення розміру вектора; * вектор – множення елементів векторів *a* [i] * *b* [i]; + *N* – перехід управо до елемента з номером *n* .

9.5. Контрольні запитання

1) Які типи помилок можуть виникати в програмах на С++?

2) Коли застосовують механізм обробки виключних ситуацій в програмах?

3) У чому полягає механізм обробки виключних ситуацій?

ВИСНОВОК

Усі сфери сучасного світу пронизано інформаційними технологіями, які впевнено закріпили позиції в нашому громадському житті.

Роботою інформаційних технологій керує різного роду програмне забезпечення, організовує і підтримує звичні та часто непомітні для нас процеси.

Будь то розважальна комп`ютерна гра або програма для графічного зображення об`єктів, або програма для моделювання об`єктів чи процесів фізичної природи, керування проводиться спеціально розробленими програмами, що взаємодіють між собою. Для розробки різного роду програмного забезпечення застосовуються різні мови програмування, в тому числі і достатньо широко мова С++, яка використовує основні концепції об`єктно-орієнтованого програмування. Основні складові такого підходу наведені у цьому навчальному посібнику і можуть бути використані для вивчення програмування.

СПИСОК ЛІТЕРАТУРИ

1. Страуструп Б. Дизайн и эволюция C++ [Текст] / Бьерн Страуструп; пер. с англ. – М. : ДМК Пресс; СПб. : Питер, 2006. – 448 с.

2. Страуструп Б. Язык программирования C++. Специальное издание [Текст] / Бьерн Страуструп; пер. с англ. – М. : ООО Бином-Пресс, 2007. –1104 с.

3. Шилдт Г. C++: базовый курс [Текст] / Герберт Шилдт. – 3-е изд. –М. : Вильямс, 2009. – 624 с.

4. Ковалюк Т. В. Основи програмування [Текст] / Т. В. Ковалюк. – К. : BHV, 2005. – 384 с.

5. Дейтел Х. М. Как программировать на C++ [Текст] / Х. М. Дейтел, П. Дж. Дейтел; пер. с англ. – 5-е изд. – М. : ООО Бином-Пресс, 2008. – 1456 с.

6. Павловская Т. А. C/C++. Структурное программирование: практикум [Текст] : учебник / Т. А. Павловская, Ю.А. Щупак. – СПб. : Питер, 2003. – 240 с.

7. Глушаков С. В. Программирование в среде C++ Builder 6 [Текст] /С. В. Глушаков, В. Н. Зорянский, С. Н. Хоменко. – Харьков : Фолио, 2003. – 508 с.

8. Шилдт Г. C для профессиональных программистов [Електронний ресурс] / Герберт Шилдт. – Режим доступу: http://www.codenet.ru/progr/cpp/5/ – Заголовок з екрана.

9. Алейников Д. В., Холодова Е. П., Силкович Ю. Н. Язык программирования C++ : введение в объектно-ориентированное программирование: учеб.-метод. пособ /Д. В. Алейников, Е. П. Холодова, Ю. Н. Силкович – Минск : Национальная библиотека Беларуси, 2016. – 85 с.

10. Крупник А. Б. Изучаем C++ [Текст] / А. Б. Крупник. – СПб.: Питер, 2004. – 251 с.

11. C++. Основи програмування. Теорія та практика : підручник / [О. Г. Трофименко, Ю. В. Прокоп, І. Г. Швайко та ін.]; за ред. О. Г. Трофименко. – Одеса : Фенікс, 2010. – 544 с.

12. Павловская Т. А. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2002. – 464 с.

13. Динман М. И. С++. Освой на примерах / М. И. Динман. – СПб.: БХВ-Петербург, 2006. – 384 с.

14. Павловская Т. А., Щупак Ю. А. С++. Объектно-ориентированное программирование : Практикум / Т. А. Павловская, Ю. А. Щупак – СПб.: Питер, 2006. – 265 с:

15. Страуструп Б. Справочное руководство по С++ [Електронний ресурс] / Бьерн Страуструп. – Режим доступу: http://wm-help.net/books-online/book/36152.html – Заголовок з екрана.

16. Шилдт Г. Справочник программиста по С/С++ [Текст] / Герберт Шилдт. – М. : Вильямс, 2006. – 432 с.

17. Глушаков С. В. Практикум по С++ [Текст] / С. В. Глушаков, С. В. Смирнов, А. В. Коваль. – Харьков : Фолио, 2006. – 526 с.

18. Глушаков С. В. Язык программирования С++ [Текст] : учебн. курс / С. В. Глушаков, С. В. Смирнов, А. В. Коваль. – Харьков : Фолио, 2001. – 500 с.

19. Прата Ст. Язык программирования С++. Лекции и упражнения : учебник / Стивен Прата; пер. с англ. – СПб. : ООО ДиаСофтЮП, 2005. – 1104 с.

20. Златопольский Д. М. Сборник задач по программированию [Текст] / Д. М. Златопольский. – 2-е изд. – СПб. : БХВ-Петербург, 2007. – 240 с.

21. Культин Н. Б. С/С++ в задачах и примерах [Текст] / Н. Б. Культин. – СПб. : БХВ-Петербург, 2001. – 288 с.

22. Лаптев В. В. С++. Экспресс-курс / В. В. Лаптев – СПб.: БХВ-Петербург, 2004. – 512 с.

Навчальне видання

ВОДКА Олексій Олександрович
ДАШКЕВИЧ Андрій Олександрович
ІВАНЧЕНКО Ксенія Вікторівна
РОЗОВА Людмила Вікторівна
СЕНЬКО Альона Володимирівна

ОСНОВИ ПРОГРАМУВАННЯ НА С++

Навчальний посібник
для студентів спеціальностей
113 – Прикладна математика та 122 – Комп’ютерні науки

Відповідальний за випуск доц. О.О.Водка

Роботу до видання рекомендував проф. Д.В.Бреславський

Редактор О.В.Козлюк

План _2020_ р., поз. _94_

Підп. до друку 20.04.2021р.
Гарнітура Times New Roman 14

Електронна версія