МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

С. М. Порошин, А. М. Носик

ПРОГРАМУВАННЯ
Частина 2

Навчальний посібник
для студентів усіх форм навчання за спеціальністю
123 «Комп’ютерна інженерія»

Затверджено
редакційно-видавничою
радою НТУ «ХПІ»
протокол № 1 від 15.02.24

Харків
НТУ «ХПІ»
2024

УДК 0004.928
П59

Рецензенти:
М .А. Мірошник, доктор техн. наук, професор,
Харківський національний університет імені В.Н. Каразіна
Ю.Ф. Кучеренко, кандидат техн. наук, старший науковий співробітник,
Харківський національний університет Повітряних Сил імені Івана Кожедуба

Автори:
С .М. Порошин, д-р техн. наук, проф.;
А. М. Носик, к.т.н., с.н.с., доц.

Порошин С.М. Програмування. Частина 2: начальний посібник / С. М. Порошин,
П59 А. М. Носик. – Харків: НТУ «ХПІ», 2024. – 447 с.

ISBN 978-617-05-0470-8

У навчальному посібнику викладено основні принципи з побудови обчислювальних алгоритмів, вивчення основ мови програмування високого рівня С та розробки комп'ютерних програм, ознайомлення з методами розв'язання обчислювальних задач та формування вмінь з їх практичного використання. Поданий матеріал призначений для опрацювання в інтегрованому середовищі розробки Code::Blocks. Розглянуті основні принципи з використання структур, об'єднань та перелічених типів у програмах на С, основних функцій для роботи зі списком і деком а також при дослідженні принципів сортування списків. Досліджені особливості роботи з функціями динамічного виділення пам'яті, керування окремими бітами при виконанні логічних операцій та операцій зсуву, засобів препроцесорної обробки, а також основних можливостей роботи з графічною бібліотекою WinBGIm та її функціями.

Призначено для студентів спеціальності 123 «Комп'ютерна інженерія» та інших технічних спеціальностей.

УДК 004.4(75)

Іл. 109. Табл. 10. Бібліогр. 10 назв.

ISBN 978-617-05-0470-8

ЗМІСТ

ВСТУП 11
1. СТРУКТУРИ ТА ІНШІ ФОРМИ ОРГАНІЗАЦІЇ ДАНИХ 12
1.1. Структури 12
1.1.1. Оголошення структури 15
1.1.2. Визначення змінної типу «структура» 16
1.1.3. Ініціалізація структури 18
1.1.4. Доступ до членів структури 18
1.1.5. Ініціалізатори для структур 19
1.1.6. Масиви структур 20
1.1.7. Оголошення масиву структур 22
1.1.8. Ідентифікація членів в масиві структур 24
1.1.9. Аналіз програми 25
1.1.10. Вкладені структури 26
1.1.11. Вказівники на структури 29
1.1.12. Оголошення та ініціалізація вказівника на структуру 31
1.1.13. Доступ до членів структури за допомогою вказівника 32
1.1.14. Повідомлення функціям про структури 33
1.1.15. Передавання членів структури 33
1.1.16. Використання адреси структури 35
1.2. Можливості структур 36
1.2.1. Передавання структури в якості аргументу 36
1.2.2. Додаткові можливості структур 38
1.2.3. Символьні масиви або вказівники на char в структурах 44

1.2.4. Складені літерали та структури (C99) 46
1.2.5. Члени з типами гнучких масивів 48
1.2.6. Анонімні структури 52
1.2.7. Функції, що використовують масив структур 53
1.2.8. Зберігання вмісту структур у файлі 55
1.2.9. Приклад зберігання структури 57
1.2.10. Аналіз програми 59
1.3. Об’єднання, перелічувані типи та бітові поля 61
1.3.1. Об’єднання 61
1.3.2. Перелічувані типи 65
1.3.3. Простір імен, який спільно використовується 70
1.3.4. Засіб typedef 70
1.3.5. Бітові поля 74
1.4. Маніпулювання бітами 81
1.4.1. Побітові операції 82
1.4.2. Побітові операції зсуву 89
1.4.3. Бітові поля та побітові операції 96
1.4.4. Засоби вирівнювання (C11) 100
Контрольні запитання та завдання 104
Завдання для самостійного розв’язання 105
2. КЛАСИ ЗБЕРІГАННЯ ТА КЕРУВАННЯ ПАМ’ЯТТЮ 107
2.1. Класи зберігання 107
2.1.1. Класи зберігання даних 107
2.1.2. Область видимості 109

2.1.3. Одиниці та файли трансляції 112
2.1.4. Зв’язування 113
2.1.5. Формальні та неформальні терміни 113
2.1.6. Тривалість зберігання 114
2.1.7. Автоматичні змінні 117
2.1.8. Блоки без фігурних дужок 121
2.1.9. Ініціалізація автоматичних змінних 123
2.1.10. Регістрові змінні 123
2.1.11. Статичні змінні з областю видимості в межах блоку 124
2.1.12. Статичні змінні з зовнішнім зв’язуванням 127
2.1.13. Ініціалізація зовнішніх змінних 130
2.1.14. Використання зовнішньої змінної 130
2.1.15. Зовнішні імена 132
2.1.16. Визначення та оголошення 132
2.1.17. Статичні змінні з внутрішнім зв’язуванням 134
2.1.18. Використання декількох файлів 135
2.2. Специфікатори класів зберігання. Виділення пам’яті 135
2.2.1. Специфікатори класів зберігання 135
2.2.2. Узагальнені відомості про класи зберігання 137
2.2.3. Класи зберігання та функції 140
2.2.4. Вибір класу зберігання 141
2.2.5. Функція генерації випадкових чисел і статична змінна 142
2.2.6. Виділення пам’яті: функції malloc( ) і free( ) 145
2.2.7. Важливість функції free() 152

2.2.8. Функція calloc() 153
2.2.9. Динамічний розподіл пам’яті та масиви змінної довжини 154
2.2.10. Класи зберігання та динамічний розподіл пам’яті 156
2.2.11. Кваліфікатори типів ANSI C 158
2.2.12. Кваліфікатор типу const 159
2.2.13. Використання const з оголошеннями вказівників і параметрів 159
2.2.14. Використання const з глобальними даними 161
2.2.15. Кваліфікатор типу volatile 163
2.2.16. Кваліфікатор типу restrict 164
2.2.17. Кваліфікатор типу _Atomic 166
2.2.18. Нові місця для старих ключових слів 167
2.2.19. Ключові поняття керування пам’яттю 168
Контрольні запитання та завдання 171
Завдання для самостійного розв’язання 172
3. РОЗШИРЕНЕ ПРЕДСТАВЛЕННЯ ДАНИХ 173
3.1. Зв’язні списки 173
3.1.1. Дослідження представлення даних 174
3.1.2. Від масиву до зв’язного списку 179
3.1.3. Використання зв’язного списку 184
3.1.4. Відображення списку 187
3.1.5. Створення списку 187
3.1.6. Звільнення пам’яті, яка була зайнята списком 189
3.2. Абстракті типи даних 190
3.2.1. Отримання абстракції 192

3.2.2. Розробка інтерфейсу 194
3.2.3. Використання інтерфейсу 200
3.2.4. Реалізація інтерфейсу 203
3.2.5. Висновки щодо використання ADT 210
3.3. Черги 213
3.3.1. Створення черги за допомогою ADT 213
3.3.2. Визначення абстрактного типу даних для представлення черги 213
3.3.3. Визначення інтерфейсу 214
3.3.4. Реалізація представлення даних інтерфейсу 215
3.3.5. Реалізація функцій інтерфейсу 221
3.3.6. Тестування черги 227
3.3.7. Моделювання реальної черги 229
3.4. Стек 239
3.4.1 Формування стеку фіксованого розміру на основі масиву 241
3.4.2. Формування динамічно зростаючого стеку на основі масиву 247
3.4.3. Формування динамічно зростаючого стеку на основі зв’язного списку 255
3.5. Дек. Реалізація деку за допомогою списку 261
3.5.1. Дек (двостороння черга) 261
3.5.2. Реалізація деку за допомогою списку 262
3.5.3. Допоміжні функції для роботи з деком 272
3.5.4. Робота з деком 273
3.6. Дек. Реалізація деку на основі масиву 281
3.6.1. Принцип додавання елементів деку до масиву 282
3.6.2. Структура програми 283

3.6.3. Типи Item і Deq 284
3.6.4. Створення деку 285
3.6.5. Операції додавання елементів до деку 285
3.6.6. Операції видалення елементів з деку 286
3.6.7. Отримання інформації про стан деку 288
3.6.8. Отримання крайніх елементів без видалення їх з деку 288
3.6.9. Перебір елементів деку за допомогою ітератора 288
3.6.10. Тестування розроблених функцій 292
3.6.11. Зміна розміру деку 299
3.6.12. Тестування функції resize() 301
3.7. Двійкове дерево пошуку 307
3.7.1. Створення абстрактного типу даних для двійкового дерева 310
3.7.2. Інтерфейс двійкового дерева пошуку 310
3.7.3. Реалізація двійкового дерева 314
3.7.4. Видалення вузла 325
3.7.5. Обхід дерева 326
3.7.6. Спустошення дерева 327
3.7.7. Тестування пакету для деревовидного представлення 333
Контрольні запитання та завдання 340
Завдання для самостійного розв’язання 341
4. ПРЕПРОЦЕСОР І БІБЛІОТЕКА С 343
4.1. Препроцесор мови С 343
4.1.1. Перші кроки в трансляції програми 343
4.1.2. Символічні константи: директива #define 344

4.1.3. Лексеми препроцесора С 350
4.1.4. Перевизначення констант 350
4.1.5. Використання аргументів у директиві #define 351
4.1.6. Створення рядків з аргументів макросу: операція # 355
4.1.7. Засіб препроцесору «злиття»: операція ## 357
4.1.8. Макроси зі змінним числом аргументів: ... і __VA_ARGS__ 358
4.1.9. Вибір між макросом і функцією 360
4.1.10. Включення файлів: директива #include 361
4.1.11. Випадки застосування файлів заголовку 365
4.1.12. Інші директиви препроцесора 367
4.1.13. Наперед визначені макроси 375
4.1.14. Узагальнений вибір (C11) 379
4.1.15. Вбудовані функції (C99) 381
4.1.16. Специфікатор функції _Noreturn (C11) 385
4.2. Бібліотека С 385
4.2.1. Отримання доступу до бібліотеки С 386
4.2.2. Використання описів бібліотеки 387
4.2.3. Бібліотека математичних функцій 388
4.2.4. Бібліотека tgmath.h (C99) 394
4.2.5. Бібліотека утиліт загального призначення 395
4.2.6. Бібліотека тверджень 405
4.2.7. Функції memcpy() і memmove() з бібліотеки string.h 409
4.2.8. Змінна кількість аргументів: файл stdarg.h 412
Контрольні запитання та завдання 416

Завдання для самостійного розв’язання 416
5. ВІЗУАЛІЗАЦІЯ ІНФОРМАЦІЇ 418
5.1. Графічна бібліотека WinBGIm 418
5.2. Графічний курсор 422
5.3.Рисування точок і ліній 424
5.4. Рисування прямокутників 426
5.5. Рисування окружностей та еліпсів 428
5.6. Вивід тексту в графічному режимі 430
5.7. Побудова графіків функцій 433
5.8. Побудова кривих Гільберта 438
Контрольні запитання та завдання 444
Завдання для самостійного розв’язання 445
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 446

ВСТУП

Програмування має дуже багато сфер застосування, де інформаційні технології (ІТ) набувають розвитку дуже швидкими темпами. Розробники програмних продуктів потрібні в багатьох сферах, які навіть не завжди пов'язані тільки з використанням ІТ.

Метою вивчення програмування є отримання навичок побудови основних обчислювальних алгоритмів, вивчення основ мови програмування високого рівня С та розробки комп'ютерних програм, ознайомлення з методами розв'язання обчислювальних задач та формування вмінь з їх практичного використання.

Мобільність і продуктивність мови С є одними з найбільш затребуваних її характеристик при реалізації таких операційних систем як Linux, Microsoft Windows, а також Google Android. Мова *С* є однією з найпопулярніших мов для розробки вбудованих систем, від яких зазвичай вимагається висока швидкість виконання та низьке витрачання пам'яті, а також для систем реального часу та зв'язку.

Дисципліна «Програмування» є базовою для вивчення інших дисциплін, пов'язаних з питаннями алгоритмізації та програмування.

Представлені матеріали навчального посібника напралені на опрацювання питань, пов'язаних з роботою в інтегрованому середовищі розробки Code::Blocks. В посібнику розглянуті основні принципи з використання структур, об'єднань та перелічених типів, основних функцій роботи зі списком і деком, а також принципів сортування списків. Наведені особливості роботи з функціями динамічного виділення пам'яті, керування окремими бітами при виконанні логічних операцій та операцій зсуву, засобів препроцесорної обробки, а також основні можливості роботи з графічною бібліотекою WinBGIm та її функціями.

Навчальний посібник присвячений світлій пам'яті доцента кафедри мультмедійних та інформаційних технологій і систем В.В. Онищенко, чиї ідеї лягли в основу цього посібника.

1. СТРУКТУРИ ТА ІНШІ ФОРМИ ОРГАНІЗАЦІЇ ДАНИХ

1.1. Структури

Під час проектування програми одним з найбільш важливих кроків є вибір відповідного способу представлення даних. У багатьох випадках вибір звичайної змінної або навіть масиву може виявитися недостатнім. Мова `C` дозволяє розширити можливості представлення даних за допомогою змінних типу **«структура»**. У своїй базовій формі структура `C` є достатньо гнучким засобом для надання великої різноманітності даних, що дозволяє створювати нові форми.

Розглянемо конкретний приклад зі створення каталогу книг, який дозволить зрозуміти, чому саме потрібно використовувати структури, і продемонструє можливості їх створення та застосування.

При створенні каталогу необхідно мати різноманітну інформацію стосовно кожної книги: назва, автор, видавництво, дата реєстрації авторського права, кількість сторінок і вартість книги. Деякі з цих елементів даних, такі як назви, можуть зберігатися в масивах рядків. Інші елементи потребують масиву значень типу `int` або `float`.

За наявності семи різних масивів, відстеження усіх даних може виявитися незручним, особливо якщо врахувати, що необхідно здійснювати сортування за назвою, автором, ціною та інше. Найкраще рішення передбачає використання одного масиву, кожен елемент якого містить вичерпні відомості про одну книгу.

Крім цього знадобиться форма даних, яка може містити рядки та числа, але певним чином поділяє цю інформацію. Структура `C` відповідає таким вимогам. Для спрощення задачі зробимо два обмеження. По-перше, будемо включати тільки назву книги, її автора та поточну вартість. По-друге, обмежимо каталог однією книгою. Однак не слід звертати на це увагу, оскільки пізніше можливості програми будуть розширені.

Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
 char *s_gets(char *st, int n);

#define MAXTITL 41        // максимальна довжина назви + 1
#define MAXAUTL 31        // максимальна довжина імені автора + 1

struct Book               // шаблон структури Book
{
     char  title[MAXTITL];
     char  author[MAXAUTL];
     float value;
};                        // кінець шаблону структури

int main(void)
{
     // Оголошення змінної library типу Book
     struct Book library;

     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);
     printf("Введіть назву книги:\n");
     // Доступ до розділу назви книги
     s_gets(library.title, MAXTITL);
     printf("\nВведіть прізвище, ім'я та по батькові автора:\n");
     s_gets(library.author, MAXAUTL);
     printf("\nВведіть ціну книги:\n");
     scanf("%f", &library.value);
     printf("\n\n%s, автор - %s: $%.2f\n", library.title,
     library.author, library.value);
     printf("%s: \"%s\" ($%.2f)\n", library.author,
     library.title, library.value);
     printf("\n");
     return 0;
}
 char *s_gets(char *st, int n)
{
     char *ret_val;
     char *find;
```

```
      ret_val = fgets(st, n, stdin);
      if(ret_val)
      {
           find = strchr(st, '\n'); // Пошук нового рядка
if(find)                    // Якщо адреса не дорівнює NULL,
    *find = '\x0';          // помістити туди нульовий символ
else
      while(getchar() != '\n')
continue;          // Відкинути залишок рядка
      }
      return ret_val;
}
```

Результат роботи програми наведено на рис. 1.1.

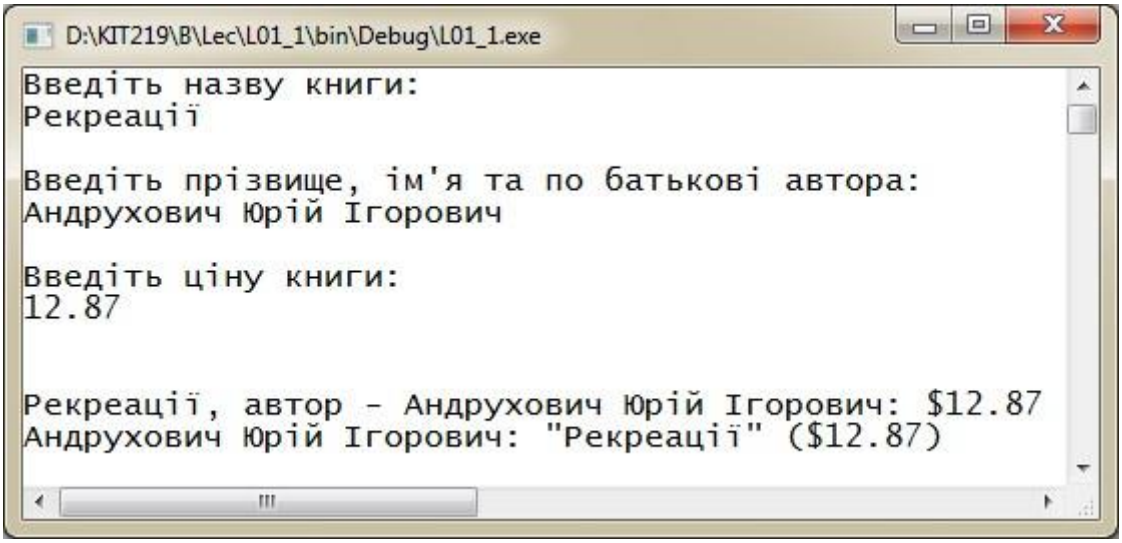


Рисунок 1.1 – Результат роботи програми зі структурою

Структура, яка була створена у попередній програмі, має три частини для зберігання назви книги, імені автора та ціни. Ці частини називаються членами або полями. Необхідно оволодіти наступними трьома навичками:

- налаштування формату або схеми для структури;

- оголошення змінної, яка відповідає цій схемі;

- забезпечення доступу до індивідуальних компонентів змінної типу «структура».

1.1.1. Оголошення структури

Оголошення структури являє собою генеральний план, що описує спосіб формування структури. Оголошення структури виглядає наступним чином:

```
struct Book
{  char    title[MAXTITL];
     char
     author[MAXAUTL];
     float  value;
};
```

Це оголошення описує структуру, яка створюється з двох символьних масивів і однієї змінної типу **float**. Воно не створює реальний об'єкт даних, але визначає, з чого складається такий об'єкт. Іноді доведеться посилатися на оголошення структури як на шаблон, тому що воно показує, яким чином будуть зберігатися дані. Давайте розглянемо оголошення більш детально. Першим йде ключове слово **struct**. Воно вказує на те, що за ним знаходиться структура. Далі може міститися необов'язковий дескриптор – слово **Book** – скорочена позначка, яку можна застосовувати для посилання на цю структуру. Таким чином, пізніше ми маємо наступне оголошення:

```
struct Book library;
```

Воно оголошує **library** як змінну типу «структура», яка використовує схему структури **Book**.

Після цього в оголошенні структури зазначений список членів, які вкладені в фігурні дужки. Кожен член описується власним оголошенням, яке закінчується крапкою з комою. Наприклад, назва книги (**title**) являє собою масив, що містить **MAXTITL** елементів типу **char**. Членом структури може бути будь-який тип даних **C**, в тому числі й інша структура.

Крапка з комою після закривальної фігурної дужки завершує визначення шаблону структури. Це оголошення можна розмістити за межами будь-якої функції (зробити його зовнішнім), як було зроблено в розглянутій програмі, або

всередині визначення функції. Якщо оголошення структури знаходиться всередині функції, її дескриптор може застосовуватися тільки в рамках цієї функції. Якщо оголошення є зовнішнім, воно буде доступним для усіх функцій, які йдуть за цим оголошенням у файлі. Наприклад, в другій функції можна було б визначити `struct Book dickens;` і в цій функції з'явилася б змінна **dickens**, що має ту ж саму форму, що й
структура **Book**.

Ім'я дескриптора вказувати не обов'язково, але воно повинно використовуватися, коли шаблон структури визначається в одному місці, а фактичні змінні – в інших місцях. Ми повернемося до цього питання пізніше, після того як розглянемо визначення змінних типу «структура».

1.1.2. Визначення змінної типу «структура»

Поняття «структура» застосовується в двох сенсах. Одним з них є «схема структури» – те, що ми нещодавно з вами обговорювали. Схема структури повідомляє компілятору, яким чином треба представляти дані, але вона не призводить до виділення пам'яті для них. Наступний крок полягає у створенні змінної типу «структура». Саме в цьому є другий сенс поняття «структура». Рядок програми, що створює змінну типу «структура», має вигляд:

```
struct Book library;
```

Зустрівши цю інструкцію, компілятор створює змінну **library**. Застосовуючи шаблон **Book**, компілятор надає пам'ять для масиву з **MAXTITL** елементів типу **char**, для масиву з **MAXAUTL** елементів типу **char** і для змінної **float**. Ця пам'ять об'єднана в єдину конструкцію під спільним ім'ям **library**, як показано на рис. 1.2 на прикладі `struct Stuff`.

```
struct Stuff
{ int   number;
  char  code[4];
  float cost;
};
```

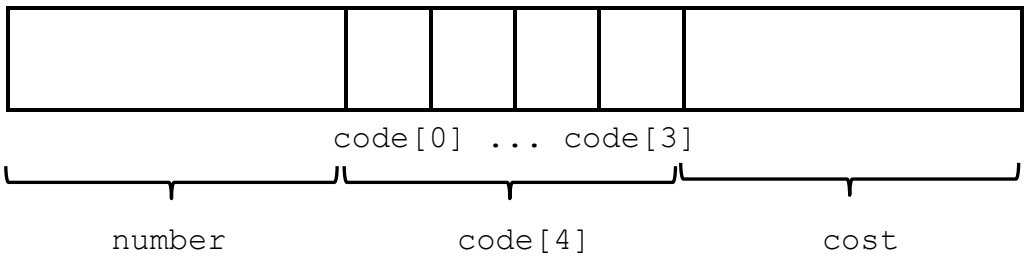


Рисунок 1.2 – Виділення пам’яті під структуру **Stuff**

В оголошенні змінної типу «структура» конструкція **struct** Book відіграє ту ж саму роль, що й ключові слова **int** або **float** в більш простих оголошеннях. Наприклад, можна було б оголосити дві змінні типу **struct** Book, і навіть вказівник на структуру такого виду:

```
struct Book doyle, panshin, *ptbook;
```

Кожна змінна **doyle** і **panshin** типу «структура» буде мати три частини: **title**, **author** і value. Вказівник **ptbook** може вказувати на змінні **doyle**, **panshin** або на будь-яку іншу структуру **Book**. По суті оголошення структури **Book** створює новий тип на ім’я **struct** Book. З точки зору комп’ютера оголошення

struct Book library; є скороченням для

```
struct Book
{  char    title[MAXTITL];
     char
     author[MAXAUTL];
     float  value;
} library;
```

Іншими словами, процес оголошення структури і процес визначення змінної типу «структура» можна об’єднати в одну дію. Комбінація оголошення та визначень змінних робить зайвим застосування дескриптора

```
struct          // дескриптор відсутній
{  char    title[MAXTITL];
     char
```

```
        author[MAXAUTL];
        float  value;
} library;
```

Тим не менш, дескриптор знадобиться, якщо шаблон структури повинен використовуватися декілька разів. Можна також застосувати альтернативний варіант з **typedef**.

1.1.3. Ініціалізація структури

Для ініціалізації структури застосовується синтаксис, аналогічний до того, що використовувався при ініціалізації масивів:

```
struct Book library =
          { "Рекреації", "Андрухович Юрій Ігорович", 1.95 };
```

Як можна помітити, використовується розташований у фігурних дужках список ініціалізаторів, які розділяються комами. Тип кожного ініціалізатора повинен відповідати типу члена структури, який він ініціалізує. Відповідно, член **title** можна ініціалізувати рядком, а член **value** – числом. Щоб зробити ці зв'язки більш очевидними, краще кожен член структури розташовувати на окремому рядку ініціалізації, але компілятору достатньо відокремлювати ініціалізатори один від одного комами, як це зроблено в розглянутому прикладі.

1.1.4. Доступ до членів структури

Структура схожа на «супермасив», в якому один елемент може мати тип **char**, інший – **float**, а наступний – масив елементів типу **int**. Звертатися до окремих елементів масиву можна за допомогою індексу.

Яким же чином можна отримати доступ до індивідуальних членів структури? Для цього існує операція членства в структурі – крапка (**'.'**).

Наприклад, **library.value** – це компонент **value** структури **library**. Конструкцію **library.value** можна використовувати як будь-яку іншу змінну типу **float**. Аналогічно, **library.title** можна застосовувати як масив типу

char. З цієї причини в наведеній вище програмі використовуються такі вирази, як

```
s_gets(library.title, MAXTITL);
```

і

```
scanf("%f", &library.value);
```

По суті **.title**, **.author** і **.value** відіграють роль індексів для структури **Book**.

Зверніть увагу, що хоча **library** – це структура, **library.value** має тип **float** і використовується подібно до будь-якої змінної типу **float**. Наприклад, **scanf("%f", ...)** потребує адреси комірки зі значенням **float**, а саме такою адресою є **&library.float**. В даному випадку операція «крапка» має більш високий пріоритет, ніж операція **&**, тому цей вираз буде еквівалентним виразу **&(library.float)**.

За наявності другої змінної типу «структура», що має той самий шаблон, як і перша, можна скористатися тим самим методом, що й раніше:

```
struct Book bill, newt;
s_gets(bill.title, MAXTITL);
s_gets(newt.title, MAXTITL);
```

Конструкція **.title** відноситься до першого члена структури **Book**. Слід зазначити, що програма виводить вміст структури **library** в двох різних форматах. Це підкреслює свободу, яку має програміст при роботі з членами структури.

1.1.5. Ініціалізатори для структур

Стандарти **C99** і **C11** надають призначені ініціалізатори для структур. Синтаксис схожий на синтаксис призначених ініціалізаторів для масивів. Однак призначені ініціалізатори для структур при ідентифікації конкретних членів використовують операцію «крапка» та імена членів, а не квадратні дужки та

індекси. Наприклад, щоб ініціалізувати лише член **value** структури **Book**, можна зробити це наступним чином:

```
struct Book surprise = { .value = 10.99 };
```

Призначені ініціалізатори можна вказувати в будь-якому порядку:

```
struct Book gift = { .value  = 25.99,
                     .author = "James Broadfool",
                     .title  = "Rue for the Toad"

                   };
```

Як і у випадку масивів, звичайний ініціалізатор, що йде за призначеним, присвоює значення члену, який йде за членом, що вказаний у призначеному ініціалізаторі. Крім того, член отримує значення, яке було надано останнім. Наприклад, погляньте на таке оголошення:

```
struct Book gift = { .value = 18.90,
                     .author = "Philionna Pestle",
                     0.25

                   };
```

Значення **0.25** присвоюється члену **value**, оскільки він знаходиться безпосередньо після члена **author** в оголошенні структури. Нове значення **0.25** замінює собою вказане раніше значення **18.90**. Тепер, маючи базові знання, ви готові розширити свій світогляд і ознайомитися з декількома типами, в яких задіяні структури: масиви структур, структури структур, вказівники на структури та функції, які обробляють структури.

1.1.6. Масиви структур

Розширимо програму каталогу книг для підтримки більшої кількості книг. Очевидно, що кожна книга може бути описана однією змінною типу **Book**. Щоб описати дві книги, необхідні дві такі змінні і т. д. Для підтримки декількох книг знадобиться масив структур подібного роду. Для цього створимо програму, текст якої має наступний вигляд:

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

char *s_gets(char *st, int n);

#define MAXTITL   40
#define MAXAUTL   40
#define MAXBOOKS 100                  // Максимальна кількість книг

struct Book                           // Встановлення шаблону Book
{
char        title[MAXTITL];
char       author[MAXAUTL];
float value;
};

int main(void)
{
   struct Book library[MAXBOOKS]; // Масив структур типу Book
   int count = 0;
   int index;

   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);

printf("Натисніть [Enter] на початку рядка, "
           "щоб завершити роботу.\n");
printf("=================================================\n");
printf("Книга №%d\n", count+1);

printf("=================================================\n");
printf("Введіть назву книги: ");

    while(count < MAXBOOKS && s_gets(library[count].title,
          MAXTITL) != NULL && library[count].title[0] != '\0')
    {
        printf("Введіть ПІБ автора:  ");
s_gets(library[count].author, MAXAUTL);
printf("Введіть ціну:        ");
scanf("%f", &library[count++].value);
while(getchar() != '\n')
            continue;                 // Очистити вхідний рядок
        if(count < MAXBOOKS)
        {
            printf("\nНатисніть [Enter] на початку рядка, "
```

```
                "щоб завершити роботу.\n");

printf("=========================================\n");
printf("Книга №%d\n", count+1);
printf("=========================================\n");
printf("Введіть назву книги: ");
        }

}
    if(count > 0)
    {
     printf("\n\nКаталог ваших книг:\n\n");
     for(index = 0; index < count; index++)
     printf("%s, автор - %s: $%.2f\n", library[index].title,
            library[index].author, library[index].value);
    }
else
        printf("Зовсім немає книг? Це дуже погано.\n");
return 0;
}

char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;

ret_val = fgets(st, n, stdin);
if(ret_val)
    {
  find = strchr(st, '\n');   // Пошук нового рядка
  if(find)                   // Якщо адреса не дорівнює NULL,
        *find = '\0';          // помістити туди нульовий символ
  else
            while(getchar() != '\n')
                continue;          // Відкинути залишок рядка
    }
    return ret_val;
}
```

Результат роботи програми наведено рис. 1.3.

1.1.7. Оголошення масиву структур

Оголошення масиву структур подібне до оголошення будь-якого іншого виду масиву, наприклад:

```
struct Book library[MAXBOOKS];
```

У попередньому рядку **library** оголошується як масив, що містить **MAXBOOKS** елементів. Кожен елемент цього масиву є структурою типу **Book**.

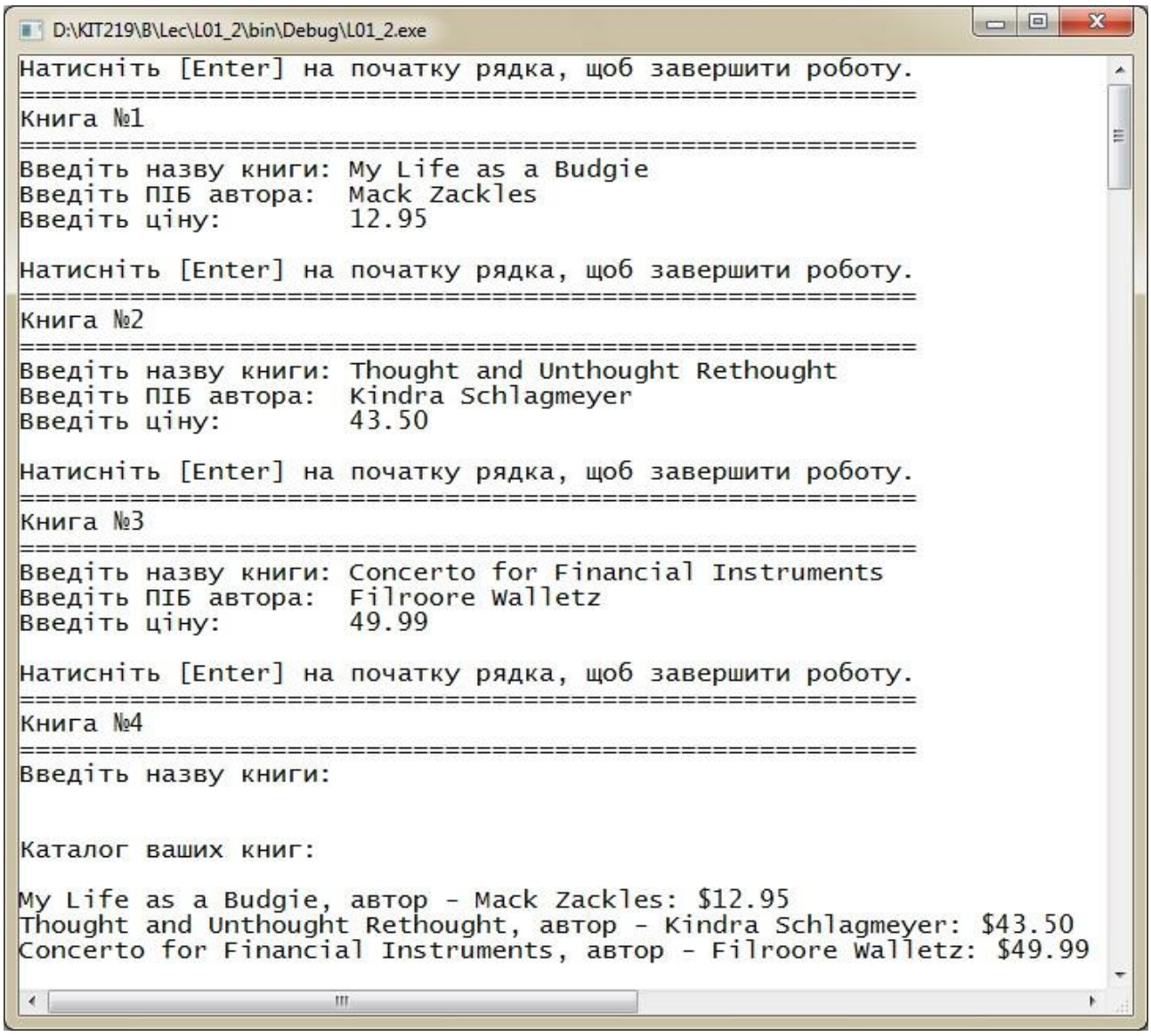


Рисунок 1.3 – Результат виконання програми з використанням масиву структур

Таким чином, **library[0]** – одна структура типа **Book**, **library[1]** – друга структура типа **Book** і т. д.

На рис. 1.4 наведена схема розміщення масиву структур, яка дозволяє краще зрозуміти сказане. Саме ім'я **library** не є іменем структури. Воно являє собою ім'я масиву, елементами якого є структури типу **struct Book**.

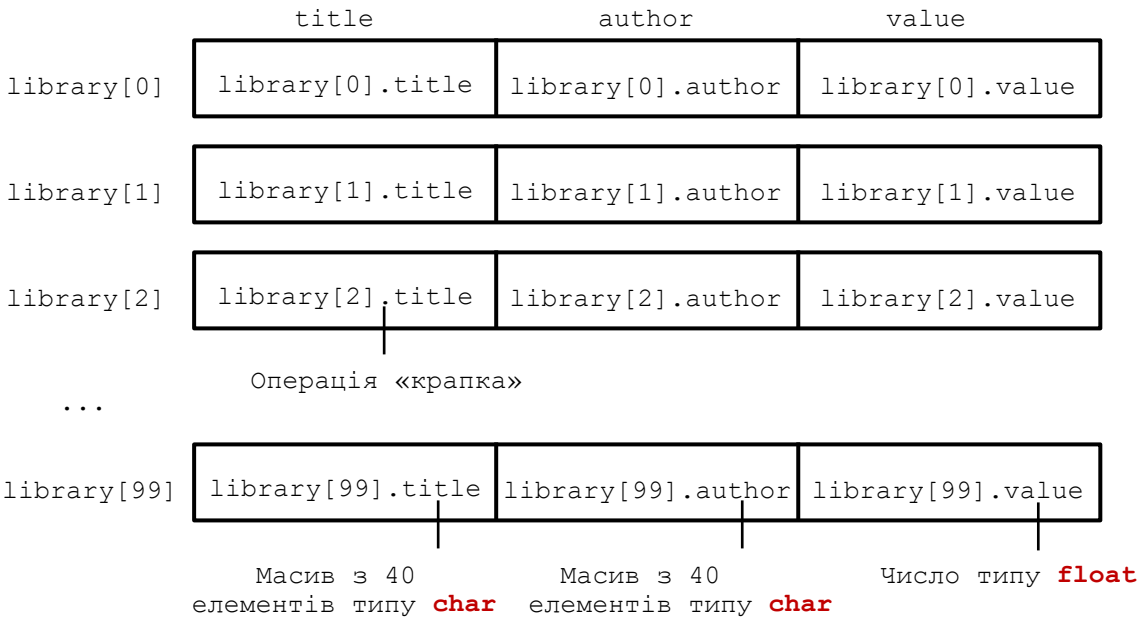


Рисунок 1.4 – Масив **library** структур типу **Book**, який містить **100** елементів

1.1.8. Ідентифікація членів в масиві структур

Для ідентифікації членів в масиві структур застосовуються ті ж самі правила, що й у випадку індивідуальних структур: за іменем структури повинна йти операція «крапка», а потім ім’я члена.

Наприклад:

```
library[0].value   // значення value, що асоціюється з першим
                   // елементом масиву
library[4].title   // значення title, що асоціюється з п’ятим
                   // елементом масиву
```

Зверніть увагу, що індекс масиву вказаний після **library**, а не після імені члена:

```
library.value[2] // неправильно
library[2].value // правильно
```

Причина використання конструкції **library[2].value** полягає в тому, що **library[2]** є іменем змінної типу «структура», так само, як і **library[1]** – це ім’я ще однієї змінної типу «структура».

Конструкція **library[2].title[4]** являє собою п’ятий символ у назві книги (частина **title[4]**), яку описує третя структура (частина **library[2]**). Даний приклад показує, що індекси, які знаходяться праворуч від операції «крапка», застосовується до індивідуальних членів, а індекси, що розташовані ліворуч від операції «крапка», стосуються масиву структур.

У підсумку припустимі наступні оператори:

```
library                  // масив структур типу Book
library[2]               // елемент масиву, тобто структура Book
library[2].title         // символьний масив (член title
                         // елементу library[2])
library[2].title[4]      // символ в масиві члену title
```

1.1.9. Аналіз програми

Основна відмінність попередньої програми від першої полягає в тому, що у неї є цикл для читання множини записів. Цикл починається з наступної умови **while**:

```
while(count < MAXBOOKS && s_gets(library[count].title,
      MAXTITL) != NULL && library[count].title[0] != '\0')
```

Вираз **s_gets(**library[count].title, **MAXTITL)** читає рядок для назви книги. Цей вираз приймає значення **NULL**, якщо функція **s_gets()** намагається прочитати символ, що йде за кінцем файлу.

Вираз library[count].title[0] != '\0' перевіряє, чи не є перший символ рядка нульовим (тобто чи не є він порожнім рядком). Якщо користувач натискає клавішу **<Enter>** на початку рядка, передається порожній рядок і цикл завершується. В програмі також передбачена перевірка, яка не дозволяє вводити більше записів для книг, ніж дозволяє розмір масиву.

Далі в програмі йдуть наступні рядки:

```
while(getchar() != '\n') continue;                    //
     Очистити вхідний рядок
```

Цей код компенсує ігнорування функцією **scanf()** пробілів і символів нового рядка. Коли ви маєте запит на те, щоб вказати ціну книги, ви набираєте щось на зразок: **12.50 [enter]**.

Це призводить до передавання наступної послідовності символів: **12.50\n**.

Функція **scanf()** отримує символи **'1'**, **'2'**, **'.'**, **'5'** і **'0'**, але залишає символ **'\n'** в очікуванні, що їм буде займатися наступний оператор читання. Якщо б код очищення вхідного рядка був відсутній, то наступний оператор читання, **s_gets(**library[count].title, **MAXTITL)**, прочитав би залишений символ нового рядка і вирішив, що введено порожній рядок, а це є сигналом до припинення вводу. Доданий нами код читає символи до тих пір, поки не виявить символ нового рядка і потім позбувається від нього. Він нічого не робить з цими символами, а лише видаляє їх з вхідної черги. Це дозволяє функції **s_gets()** коректно розпочати читання наступних вхідних даних.

1.1.10. Вкладені структури

Іноді зручно, щоб одна структура містила іншу структуру, яка називається вкладеною. Наприклад, Шейла створює структуру з інформацією про своїх друзів. Одним з членів структури є ім'я друга. Однак ім'я саме може бути представлено за допомогою структури, з окремими записами для імені та прізвища.

Розглянемо приклад використання вкладеної структури. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#define  LEN  20

const char *msgs[6] =
{
          "    Дякую Вам за чудово проведений вечір, ",
          "Ви однозначно продемонстрували, що ",
          "являє собою особливий тип людини.",
```

```
            "    Ми обов'язково повинні зустрітися з Вами ",
            "за чудовою вечерею з ",
            " і весело провести час"
};

struct Names                    // перша структура
{
     char   first[LEN];
     char   last[LEN];
};

struct Guy                      // друга структура
{
     struct Names handle;       // вкладена структура
     char   favfood[LEN];
     char   job[LEN];
     float  income;
};

int main(void)
{
     struct Guy fellow =
     {
          { "БІЛЛІ", "БОНС" },
          "запеченими омарами",
          "персональний тренер",
          68112.00
     };

     SetConsoleOutputCP(1251);
printf("Шановний %s, \n\n", fellow.handle.first);
printf("%s%s.\n", msgs[0], fellow.handle.first);
printf("%s%s\n", msgs[1], fellow.job);
printf("%s\n", msgs[2]);
printf("%s\n", msgs[3]);
printf("%s%s%s", msgs[4], fellow.favfood, msgs[5]);
     if(fellow.income > 150000.0)
     puts("!!!");
     else
     {
          if(fellow.income > 75000.0)
               puts("!");
          else
               puts(".");
     }
     printf("\n%40s%s\n", " ", "До скорої зустрічі,");
     printf("%40s%s\n", " ", "Шейла");
```

```
    return 0;
}
```

Результат роботи програми наведено на рис. 1.5.

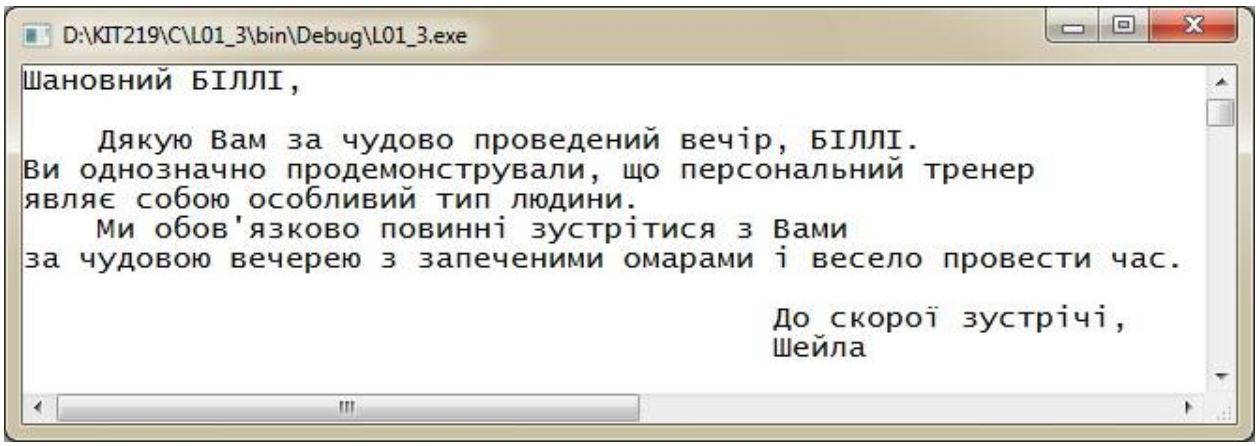


Рисунок 1.5 – Результат виконання програми з використанням вкладеної структури

По-перше, зверніть увагу на те, яким чином вкладена структура розміщується в оголошенні структури. Вона просто оголошується, майже як звичайна змінна типу **int**:

```
struct Names handle;
```

Таке оголошення говорить про те, що **handle** є змінною типу **struct** names. Зрозуміло, що файл повинен також містити оголошення структури **names**.

По-друге, подивіться, як отримувати доступ до члену вкладеної структури. Треба лише два рази скористатися операцією «крапка»:

```
printf("Шановний %s,\n\n", fellow.handle.first);
```

Якщо розглядати цю конструкцію зліва направо, вона інтерпретується наступним чином:

```
(fellow.handle).first
```

Тобто необхідно знайти структуру **fellow**, потім член **handle** структури **fellow** і, нарешті, член **first** структури типу **names**.

1.1.11. Вказівники на структури

Існують, щонайменше, чотири причини, що зумовлюють необхідність у вказівниках на структури. По-перше, з вказівниками на структури працювати переважним чином набагато легше, ніж з самими структурами. По-друге, в деяких застарілих реалізаціях структура не може бути передана як аргумент функції, але вказівник на структуру – може. По-третє, незважаючи на те, що структуру можна передавати як аргумент, передавання вказівника частіше є більш ефективним. Почетверте, багато інших видів представлення даних застосовують структури, що містять вказівники на інші структури.

Наступний короткий приклад демонструє визначення вказівника на структуру та його використання для доступу до членів структури. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#define LEN 20

struct Names
{
     char first[LEN];
     char last[LEN];
};

struct Guy
{
     struct Names handle;
     char favfood[LEN];
     char job[LEN];
     float income;
};

int main(void)
{
     struct Guy fellow[2]  =
     {
          { { "Біллі", "Бонс" },
              "запеченими омарами",
              "персональний тренер",
              68112.00
          },
```

```
        { { "Джим", "Хокінс" },
            "рибним фрикасе",
            "редактор таблоїду",
            232400.00
        }
    };

    SetConsoleOutputCP(1251);

    struct Guy *him; // вказівник на структуру
    printf("Адреса №1:    %p\nАдреса №2:    %p\n",
        &fellow[0], &fellow[1]);
     him = &fellow[0]; // повідомляє вказівник, на що
                       // вказувати
    printf("\nВказівник №1: %p\nВказівник №2: %p\n",
him, him + 1);
    printf("\nhim->income      дорівнює $%.2f\n"
        "(*him).income    дорівнює $%.2f\n",
him->income, (*him).income);
 him++;   // вказівник на наступну структуру
printf("him->favfood     дорівнює %s\n"
        "him->handle.last дорівнює %s\n",
        him->favfood, him->handle.last);
    return 0;
}
```

Результат роботи програми наведено на рис. 1.6.

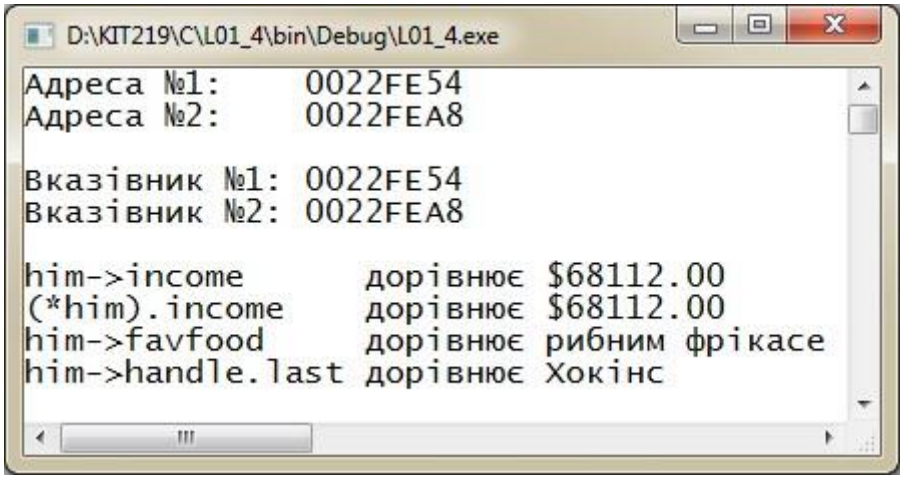


Рисунок 1.6 – Результат виконання програми з використанням вказівників на структури

1.1.12. Оголошення та ініціалізація вказівника на структуру

Оголосити вказівник на структуру можна наступним чином: `struct Guy *him;`

Першим йде ключове слово **struct**, потім дескриптор структури **Guy**, зірочка (*) і, нарешті, ім'я вказівника. Це такий самий синтаксис, який використовується для оголошення інших вказівників, як було показано раніше.

Оголошення не призводить до створення нової структури, але вказівник **him** тепер може посилатися на будь-яку існуючу структуру типу **Guy**. Наприклад, якщо **barney** – структура типу **Guy**, то можна написати наступний оператор:

```
him = &barney;
```

На відміну від масивів, ім'я структури не є її адресою – ви повинні застосовувати операцію **&**.

В нашому прикладі **fellow** – це масив структур, тобто **fellow[0]** являє собою структуру, тому код ініціалізує **him**, зробивши його таким, що вказує на **fellow[0]**:

```
him = &fellow[0];
```

Перші чотири рядки виводу (рис. 1.6) показують, що присвоювання пройшло успішно. Порівнюючи їх, можна побачити, що **him** вказує на **fellow[0]**, а **him+1** – на **fellow[1]**. Зверніть увагу, що додавання **1** до **him** призводить до додавання значення **84** до адреси. В шістнадцятковій формі запису **A8 - 54 = 54** (шістнадцяткове) = **84** (десяткове), тобто кожна структура **Guy** займає **84** байта пам'яті: під **names.first** відводиться **20** байтів, під **names.last** – **20** байтів, під **favfood** – **20** байтів, під **job** – **20** байтів і під **income** – **4** байти (розмір типу **float** в нашій системі). В деяких системах розмір структури може бути більше суми розмірів її частин. Причина полягає в тому, що вимоги до вирівнювання даних системи можуть призвести до виникнення зазорів. Наприклад, можливо, що система повинна розміщувати кожен член

структури за парною адресою або за адресою, кратною **4**. Такі структури можуть містити у собі ділянки пам'яті, що не будуть використовуватися.

1.1.13. Доступ до членів структури за допомогою вказівника

Вказівник **him** вказує на структуру **fellow[0]**. Яким чином можна за допомогою **him** отримати значення для члену **fellow[0]**? В п'ятому и шостому рядках виводу демонструються два методи.

Перший і найбільш поширений метод передбачає застосування нової операції **'->'**. Знак цієї операції формується з дефісу (**-**) і символу "більше" (**>**).

Ми маємо наступні залежності:

```
him->income дорівнює barney.income, якщо him == &barney him-
>income дорівнює fellow[0].income, якщо him == &fellow[0]
```

Іншими словами, вказівник на структуру, за яким йде операція **'->'**, працює таким самим чином, як ім'я структури з подальшою операцією «крапка» (**.**). Ви не можете використовувати просто **him.income**, тому що **him** не є іменем структури.

Важливо зазначити, що **him** – вказівник, але **him->income** – це член структури, на яку він вказує. Це означає, що в даному випадку **him->income** являє собою змінну типу **float**.

Другий метод для того, щоб вказати значення члена структури відповідає наступній послідовності тверджень:

якщо **him == &fellow[0]**, то ***him == fellow[0]**, оскільки операції **'&'** і **'*'** є оберненими. Відповідно, після підстановки отримаємо такий вираз:

```
fellow[0].income == (*him).income
```

Круглі дужки в ньому є обов'язковими, оскільки операція **'.'** має більш високий пріоритет, ніж **'*'**. Якщо **him** – це вказівник на структуру типу **Guy** на ім'я **barney**, то наступні вирази є еквівалентними:

```
//  вважаючи, що him == &barney, маємо
barney.income == (*him).income == him->income
```

1.1.14. Повідомлення функціям про структури

Згадайте, що аргументи функції передають їй значення. Кожне значення є числом – будь-то, **int**, **float**, **ASCII**-код символу або адреса.

Структура є дещо складнішою ніж одиночне значення, тому не слід дивуватися, що ранні реалізації **C** не дозволяють застосовувати структуру в якості аргументу для функції. В більш нових реалізаціях це обмеження було знято, і **ANSI C** дозволяє використовувати структури в аргументах функцій. Таким чином, сучасні реалізації **C** пропонують можливість вибору між передаванням в якості аргументів самих структур і вказівників на ці структури, або, якщо вас цікавить тільки частина структури – передаванням в аргументах членів структури. Розглянемо усі три методи, почавши з передачі членів структури у якості аргументів.

1.1.15. Передавання членів структури

До тих пір, поки член структури має тип даних з єдиним значенням (тобто **int** або один з його похідних типів, **char**, **float**, **double** або вказівник), його можна передавати в якості аргументу функції, яка приймає цей конкретний тип. Програма фінансового аналізу, яка обчислює суму на звичайному банківському рахунку клієнта, а також суму на його ощадному рахунку, ілюструє це твердження. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#define FUNDLEN 50


struct
Funds {
     char       bank[FUNDLEN];
     double     bankfund;
     char       save[FUNDLEN];
     double     savefund;
};


double sum(double, double);
int main(void)
```

```
{
      struct Funds stan =
      {
            "Garlic-Melon Bank",
            4032.27,
            "Lucky1s Savings and Loan",
            8543.94
      };
     SetConsoleOutputCP(1251);
     printf("Загальна сума на рахунках у Стена складає $%.2f\n",
               sum(stan.bankfund, stan.savefund));
      return 0;
}
//  Визначення суми двох чисел типу double double sum(double
x, double y)
{
      return (x + y);
}
{
```

Результат роботи програми наведено на рис. 1.7.

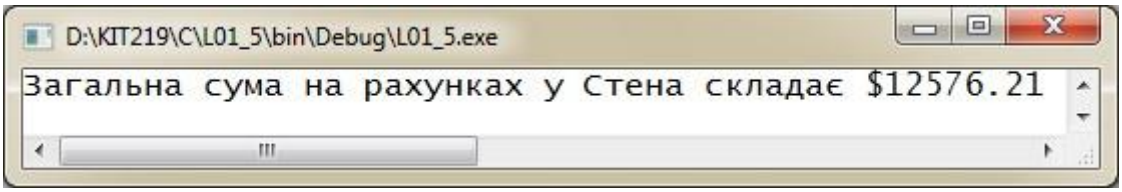


Рисунок 1.7 – Результат виконання програми з передаванням членів структури в якості параметрів

Зверніть увагу на те, що функція `sum()` не знає, чи є фактичні аргументи членами структури. Вона тільки вимагає, щоб вони мали тип `double`.

Звичайно, якщо треба, щоб функція, яка викликається, впливала на значення члена в функції, яка викликає, то можна передавати адресу цього члену:

```
modify(&stan.bankfund);
```

Це могла б бути функція, що змінює суму на банківському рахунку. Наступний підхід до того, щоб повідомити функцію про структуру передбачає повідомлення про те, що функція має справу зі структурою.

1.1.16. Використання адреси структури

Будемо вирішувати ту ж саму задачу, але на цей раз в якості аргументу застосуємо адресу структури. Оскільки функція буде працювати зі структурою **Funds**, вона також повинна використовувати оголошення **Funds**. Код програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#define FUNDLEN 50
 struct
Funds {
     char       bank[FUNDLEN];
     double     bankfund;
     char       save[FUNDLEN];
     double     savefund;
};

// аргумент є вказівником
double sum(const struct Funds
*);
 int main(void)
{
     struct Funds stan =
     {
          "Garlic-Melon Bank",
          4032.27,
          "Lucky1s Savings and Loan",
          8543.94
     };
     SetConsoleOutputCP(1251);
printf("Загальна сума на рахунках у Стена складає $%.2f\n",
       sum(&stan));
     return 0;
}
 double sum(const struct Funds *money)
{
     return(money->bankfund + money->savefund); }
```

Результат роботи програми наведено на рис. 1.8.

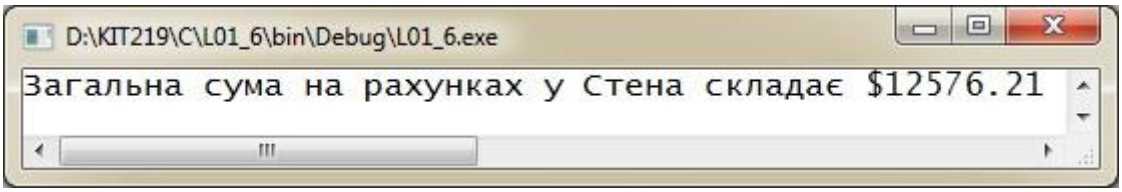


Рисунок 1.8 – Результат виконання програми з передаванням в якості параметру адреси структури

Функція **sum()** приймає вказівник (***money**) на структуру **Funds** в своєму єдиному аргументі. Передавання адреси (**&stan**) функції призводить до того, що тепер вказівник **money** вказує на структуру **stan**. Потім за допомогою операції **'->'** отримуємо значення **stan.bankfund** і **stan.savefund**. Оскільки функція не змінює значення, на яке посилається вказівник, **money** оголошується як вказівник на **const**.

Ця функція також має доступ до членів, які є назвами установ, хоча й не користується ними. Зверніть увагу, що для отримання адреси структури повинна застосовуватися операція **'&'**. На відміну від імені масиву ім'я структури не є синонімом її адреси.

1.2. Можливості структур

1.2.1. Передавання структури в якості аргументу

Для компіляторів, які дозволяють передавати структури в якості аргументів, код програми п. 1.1.16 можна переписати наступним чином:

```
#include <stdio.h>
#include <windows.h>
#define FUNDLEN 50
struct Funds
```

```
{
  char     bank[FUNDLEN];
  double   bankfund;
  char     save[FUNDLEN];
  double   savefund;
};

double sum(struct Funds moolah); // аргумент є структурою

int main(void)
{
    struct Funds stan =
    {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

SetConsoleOutputCP(1251);
printf("Загальна сума на рахунках у Стена складає $%.2f\n",
sum(stan));
return 0;
}
 double sum(struct Funds moolah)
{
    return(moolah.bankfund + moolah.savefund);
}
```

Результат роботи програми наведено на рис. 1.9.

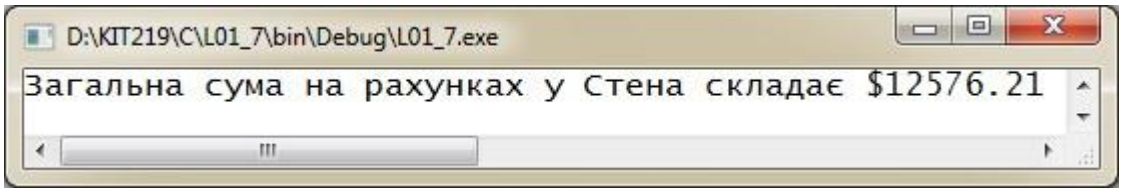


Рисунок 1.9 – Результат виконання програми з передаванням в якості параметра самої структури

Вказівник на **struct Funds money** був змінений на змінну типу **struct Funds moolah**. Під час виклику функції **sum()** створюється автоматична змінна **moolah**, що узгоджена з шаблоном **Funds**.

Потім члени цієї структури ініціалізуються копіями значень відповідних членів структури **stan**. З цієї причини обчислення виконуються за участю копії початкової структури, тоді як у попередній програмі (в якій використовувався вказівник) була задіяна сама початкова структура.

Оскільки **moolah** є структурою, то в програмі застосовується **moolah.bankfund**, а не **moolah->bankfund**.

1.2.2. Додаткові можливості структур

Сучасна мова **C** дозволяє присвоювати одну структуру іншій – тобто робити те, чого неможна було робити з масивами: якщо **new_data** і **old_data** – структури одного типу, то можна записати наступний код:

```
old_data = new_data;     // присвоювання однієї структури іншій
```

Це призводить до того, що кожному члену **new_data** присвоюється значення відповідного члена **old_data**. Це працює, навіть якщо член є масивом.

Крім того, структуру можна ініціалізувати іншою структурою того ж самого типу:

```
struct Names right_field = { "Джеймс", "Бонд" };
// ініціалізація структури іншою структурою
struct Names captain = right_field;
```

В сучасній мові **C**, включи **ANSI C**, структури не тільки можна передавати функції в якості аргументів, але також і повертати їх з функції. Застосування структур в аргументах функції дозволяє передавати їй інформацію про структуру. Використання функцій для повернення структур дає можливість передавати інформацію про структуру з функції, що викликається, до функції, що викликає. Вказівники на структури також припускають двосторонній обмін даними, тому ви часто будете застосовувати один з цих підходів при вирішенні різних задач.

Розглянемо ще декілька прикладів, які ілюструють ці два підходи. Щоб їх порівняти, напишемо просту програму, яка обробляє структури з використанням

вказівників, і потім переробимо її таким чином, щоб в ній виконувалося передавання та повернення структур. Сама програма запитує ім’я та прізвище і повідомляє загальну кількість букв в них. Цей проект навряд чи потребує застосування структур, але він пропонує просту інфраструктуру, яка дозволяє побачити, як вони працюють. Текст програми з вказівниками має наступний вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#define NLEN 30

struct Namect
{
char fname[NLEN];
char lname[NLEN];
int letters;
};

void getinfo(struct Namect *);
void makeinfo(struct Namect *);
void showinfo(const struct Namect *);
char *s_gets(char *st, int n);

int main(void)
{
    struct Namect person;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    return 0;
}
 void getinfo(struct Namect *pst)
{
    printf("Введіть своє ім'я:     ");
    s_gets(pst->fname, NLEN);
    printf("Введіть своє прізвище: ");
    s_gets(pst->lname, NLEN);
}
void makeinfo(struct Namect *pst)
{
```

```
  pst->letters = strlen(pst->fname) + strlen(pst->lname);
}

void showinfo(const struct Namect *pst)
{
    printf("\nСтудент: ");
    printf("%s %s\nВаше ім'я та прізвище містять %d букв\n",
pst->fname, pst->lname, pst->letters);
}
char *s_gets(char *st, int n)
{
   char *ret_val;
   char *find;

   ret_val = fgets(st, n, stdin);
   if(ret_val)
    {
     // пошук нового рядка
     find = strchr(st, '\n');
        if(find)            // якщо адреса не дорівнює NULL,
            *find ='\0';    // помістити туди нульовий символ
else
        while(getchar() != '\n')
        continue;     // відкинути залишок рядка
    }
return ret_val;
}
```

Результат роботи програми надано на рис. 1.10.

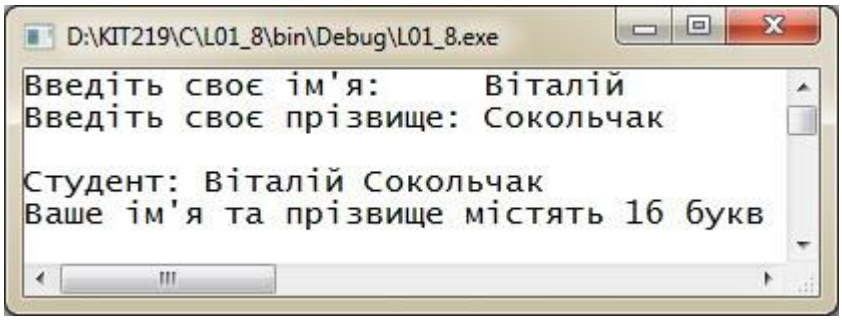


Рисунок 1.10– Результат виконання програми з використанням вказівників на структуру

Робота програми розподілена між трьома функціями, що викликаються в **main()**. В кожному випадку функції передається адреса структури **person**.

Функція **getinfo()** передає інформацію всередині себе в **main()**. До речі, вона отримує імена від користувача та поміщає їх до структури **person**, застосовуючи для доступу до неї вказівник **pst**. Згадайте, що **pst->lname** означає член **lname** структури, на яку вказує **pst**.

Це робить **pst->lname** еквівалентом імені масиву значень **char** і, відповідно, аргументом для функції **s_gets()**. Зверніть увагу на те, що хоча функція **getinfo()** і надає інформацію до головної програми, вона не використовує для цього механізм повернення, тому має тип **void**.

Функція **makeinfo()** виконує двостороннє передавання інформації. З застосуванням вказівника на **person** вона знаходить ім'я та прізвище, що зберігаються в цій структурі, та використовує функцію **strlen()** з бібліотеки **C** для підрахунку кількості букв в імені та прізвищі, а потім застосовує адресу структури **person** для зберігання отриманої суми. Ця функція також має тип **void**. Нарешті, функція **showinfo()** використовує вказівник для доступу до інформації, що призначена для виводу. Оскільки **showinfo()** не змінює вміст масиву, вказівник оголошений як **const**.

В усіх цих операціях приймала участь лише одна змінна **person** типу «структура», і кожна з функцій для доступу до структури застосовувала її адресу. Перша функція передавала інформацію зсередини себе програмі, що викликає. Друга функція приймала інформацію з програми, що викликається, всередину себе, а третя функція робила обидві ці дії.

Тепер поглянемо, яким чином запрограмувати рішення цієї задачі з використанням структур як аргументів і значень, що повертаються. По-перше, для передавання самої структури необхідно застосовувати аргумент **person**, а не **&person**. Тоді відповідний формальний аргумент оголошується з типом **struct** `Namect`, а не вказівником на цей тип. По-друге, щоб надати **main()** значення

структури, можна повернути саму структуру. Розглянемо версію програми без вказівників. Її текст має наступний вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#define NLEN 30

struct Namect
{
    char  fname[NLEN];
    char  lname[NLEN];
    int   letters;
};
struct Namect getinfo(void);
struct Namect makeinfo(struct Namect);
void showinfo(struct Namect);
char *s_gets(char *st, int n);
int main(void)
{
    struct Namect person;
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    person = getinfo();
    person = makeinfo(person);
    showinfo(person);
return 0;
}
struct Namect getinfo(void)
{
    struct Namect temp;

printf("Введіть своє ім'я:     ");
s_gets(temp.fname, NLEN);
printf("Введіть своє прізвище: ");
s_gets(temp.lname, NLEN);
return temp;
}

struct Namect makeinfo(struct Namect info)
{
    info.letters = strlen(info.fname) + strlen(info.lname);
return info;
}
 void showinfo(struct Namect info)
```

```
{
    printf("\nСтудент: ");
    printf("%s %s\nВаше ім'я та прізвище містять %d букв\n",
info.fname, info.lname, info.letters);
}
char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;

    ret_val = fgets(st, n, stdin);      if(ret_val)
    {
        // пошук нового рядка
        find = strchr(st, '\n');
        if(find)                // якщо адреса не дорівнює NULL,
            *find ='\0';        // помістити туди нульовий символ
else
        while(getchar() != '\n')
                continue;       // відкинути залишок рядка
    }
    return ret_val;
}
```

Результат роботи програми наведено на рис. 1.11.

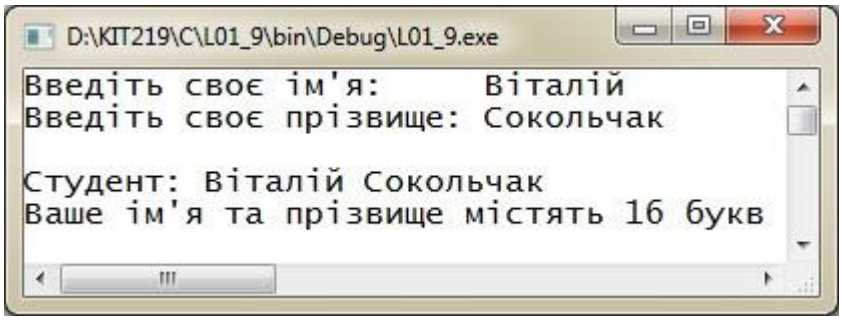


Рисунок 1.11 – Результат виконання програми без використання вказівників на структуру

Ця версія дає той самий результат, що й попередня, але працює по-іншому. Кожна з трьох функцій створює власну копію структури **person**, тому в програмі задіяні чотири різні структури, а не тільки одна.

Для прикладу розглянемо функцію **makeinfo()**. В першій програмі їй передавалася адреса структури **person**, і функція мала справу з дійсними

значеннями person. В другій версії програми створювалася нова структура info. Значення, що зберігалися в person, копіювалися в info, і функція працювала з копією. Відповідно, визначена кількість букв зберігалася в info, але не в person.

Проте, це виправляє механізм повернення. Рядок в makeinfo() return info; об'єднується з рядком в main() person = makeinfo(person); для копіювання значень, що зберігаються всередині info, в person. Зверніть увагу, що функція makeinfo() повинна бути оголошена з типом struct Namect, оскільки вона повертає структуру.

1.2.3. Символьні масиви або вказівники на char в структурах

В прикладах, які були розглянуті раніше, для зберігання рядків в структурі застосовувалися символьні масиви. Виникає питання, чи можна замість них використовувати вказівники на char? Наприклад, є наступне оголошення:

```
#define LEN 20
struct Names
{
 char first[LEN];
 char last[LEN];
};
```

Чи можна замість цього написати

```
struct Pnames
{ char *first;
  char *last;
};
```

Відповідь – так, це можливо, але можуть виникнути певні проблеми, якщо ви не поміркуєте про всі наслідки. Погляньте на код, що знаходиться нижче:

```
struct Names veep = { "Talia", "Summers" };
struct Pnames treas = { "Brad", "Fallingjaw" };
printf("%s i %s\n", veep.first, treas.first);
```

Цей код є припустимим, і він працює. Однак розглянемо, де зберігаються рядки. У випадку змінної veep типу struct Names рядки зберігаються всередині структури. Для зберігання двох імен структура виділяє усього 40 байтів. Проте,

у змінній **treas** типу **struct** Pnames рядки зберігаються там, де компілятор зберігає рядкові константи. Все, що містить ця структура – це дві адреси, які в системі в цілому займають **16** байтів. До речі, структура **struct** Pnames не виділяє пам'ять для зберігання рядків. Вона може застосовуватися тільки з рядками, для яких пам'ять була виділена десь в іншому місці, такими як рядкові константи або рядки в масивах. Таким чином, вказівники в структурі **struct** Pnames повинні використовуватися тільки для керування рядками, які були створені з наданням для них пам'яті в іншому місці програми.

Давайте розглянемо ситуацію, коли це обмеження перетворюється на проблему. Погляньте на наступний код:

```
struct Names accountant;
struct Pnames attorney;
puts("Введіть прізвище вашого бухгалтера:");
scanf("%s", accountant.last);
puts("Введіть прізвище вашого адвоката:");
scanf("%s", attorney.last);    // джерело небезпеки
```

З точки зору синтаксису цей код припустимий. Але куди зберігаються вхідні дані? Прізвище бухгалтера записується в останній член змінної **accountant**. Ця структура містить масив для зберігання рядка. У випадку прізвища адвоката функція **scanf()** отримує вказівку помістити рядок прізвища за адресою, що задається як **attorney.last**. Через те, що ця змінна не ініціалізована, адреса може мати довільне значення, і програма може спробувати помістити прізвище будь-куди. Якщо пощастить, програма буде працювати деякий час або одразу ж аварійно завершиться. Однак якщо програма працює, то вам насправді не пощастило, оскільки в ній присутня катастрофічна помилка, про яку вам ще невідомо.

Таким чином, якщо вам необхідна структура для зберігання рядків, то простіше застосовувати члени типу символьних масивів. Використання вказівників на **char** в окремих випадках допускається, але потенційно спряжено з суттєвими проблемами.

1.2.4. Складені літерали та структури (C99)

Використання складених літералів **C99** доступно для структур, а також для масивів. Це дуже зручно, якщо треба лише тимчасове значення структури. Наприклад, складені літерали можна застосовувати для створення структури, призначеної для використання в якості аргументу функції або для присвоювання іншій структурі. Синтаксис складеного літералу виглядає як поміщений у фігурні дужки список ініціалізаторів, якому передує ім'я типу в круглих дужках. Нижче надано складений літерал типу **struct** Book:

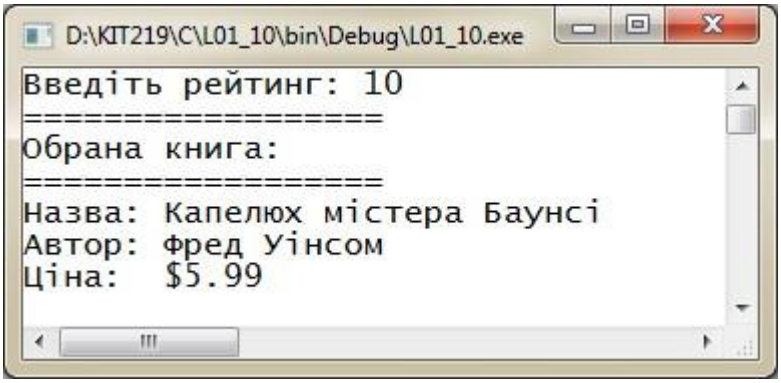
```
(struct Book) { "Ідіот", "Федір Достоєвський", 6.99 }
```

Розглянемо приклад застосування складених літералів для надання двох альтернативних значень змінній типу «структура». Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#define MAXTITL 41
#define MAXAUTL 31

struct Book    // шаблон структури: Book - дескриптор
{
   char   title[MAXTITL];
   char  author[MAXAUTL];
   float value;
};
int main(void)
{
 struct Book readfirst;
 int score;

SetConsoleOutputCP(1251);

printf("Введіть рейтинг: ");
scanf("%d",&score);
if(score >= 84)
    readfirst = (struct Book) { "Злочин і покарання",
                                "Федір Достоєвський",
```

```
                                11.25 };
else
    readfirst = (struct Book) { "Капелюх містера Баунсі",
                                "Фред Уінсом",
                                5.99 };
printf("==================\n");
printf("Обрана книга:     \n");
printf("==================\n");
printf("Назва: %s\nАвтор: %s\nЦіна:  $%.2f\n\n", readfirst.title,
         readfirst.author, readfirst.value);
return 0;
}
```

Результат роботи програми наведено на рис. 1.12.



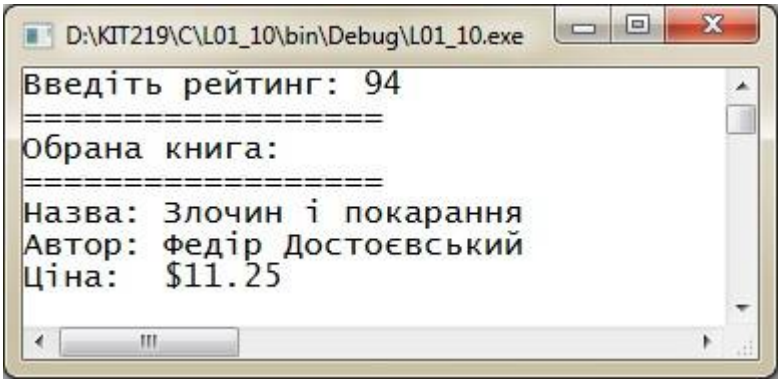


Рисунок 1.12 – Результат виконання програми, яка демонструє використання складених літералів для ініціалізації структури

Складені літерали можна використовувати також як аргументи функцій. Якщо функція очікує структуру, то їй можна передавати в якості фактичного аргументу складений літерал:

```
struct Rect { double x; double y; };
double rect_area(struct Rect r) { return r.x * r.y; }
...
double area;
area = rect_area( (struct Rect) { 10.5, 20.0 } );
```

Це призводить до того, що **area** присвоюється значення **210.0**.

Якщо функція очікує адресу, то їй можна передати адресу складеного літерала:

```
struct Rect { double x; double y; };
double rect_areap(struct Rect *rp) { return rp->x * rp->y; }
...
double area;
area = rect_areap( &(struct Rect) { 10.5, 20.0 } );
```

В результаті змінна **area** отримує значення **210.0**.

Складені літерали, які зустрічаються за межами будь-яких функцій, мають статичну тривалість зберігання, а ті, що знаходяться всередині блока – автоматичну тривалість зберігання. По відношенню до складених літералів діють ті ж самі синтаксичні правила, що й для звичайних списків ініціалізаторів. Це означає, наприклад, що в складених літералах можна застосовувати призначені ініціалізатори.

1.2.5. Члени з типами гнучких масивів

В стандарті **C99** пропонується новий засіб, що називається членом типу гнучкого масиву. Він дозволяє оголошувати структуру, останній член в якої є масивом зі спеціальними властивостями. Одна зі спеціальних властивостей полягає в тому, що такий масив не існує – у всякому разі, не з'являється миттєво. Друга спеціальна властивість полягає в тому, що при наявності коректного коду член типу гнучкого масиву можна використовувати, як ніби то він існує і має потрібну кількість елементів. Можливо, це звучить дещо своєрідно, тому давайте розглянемо кроки щодо створення та застосування структури з членом типу гнучкого масиву.

Правила, які регламентують створення члена типу гнучкого масиву:

- член типу гнучкого масиву повинен бути останнім у структурі;
- у структурі повинен бути присутнім, принаймні, ще один член іншого типу;

- гнучкий масив оголошується подібно звичайному масиву, але з порожніми квадратними дужками.

Ось приклад, що ілюструє ці правила:

```
struct Flex
{  int       count;
     double
     average;
    double scores[];      // член типу гнучкого масиву
};
```

Якщо ви оголосили змінну типу **struct** Flex, то не можна використовувати член **scores**, оскільки пам'ять для нього не зарезервована. Насправді, навіть не припускається, що ви будете оголошувати змінні типу **struct** Flex. Замість цього передбачається, що ви оголосите вказівник на тип **struct** Flex, а потім за допомогою функції **malloc()** виділите область пам'яті, яка буде достатньою для зберігання звичайного вмісту **struct** Flex, плюс додатковий простір, який необхідний для члена з типом гнучкого масиву. Наприклад, нехай ви бажаєте, щоб член **scores** являв собою масив з п'яти значень типу **double**. В цьому випадку треба зробити наступним чином:

```
struct Flex *pf; // оголошення вказівника
// запит на область пам'яті для розміщення структури та масиву
pf = malloc(sizeof(struct Flex) + 5 * sizeof(double));
```

Тепер ви маєте об'єм пам'яті, якого буде достатньо для зберігання **count**, **average** і масиву з п'яти значень типу **double**. Для доступу до цих членів можна застосовувати вказівник **pf**:

```
pf->count = 5;          // встановлення члена count
pf->scores[2] = 18.5;   // доступ до елементу члена типа масиву
```

В наступній програмі гнучкий масив отримує можливість представляти п'ять значень в першому випадку і дев'ять значень – у другому. В програмі також демонструється написання функції для обробки структури з членом типу гнучкого масиву.

```c
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
 struct Flex
{
   size_t count;
   double average;
   double scores[];   // член з типом гнучкого масиву
};
 void showFlex(const struct Flex *p);

int main(void)
{
    struct Flex *pf1,
*pf2;     int n = 5;
int i;     int tot = 0;

    SetConsoleOutputCP(1251);

    // виділення пам'яті для структури та масиву
pf1 = malloc(sizeof(struct Flex) + n * sizeof(double));
pf1->count = n;
for(i = 0; i < n; i++)
    {
        pf1->scores[i] = 20.0 - i;
        tot += pf1->scores[i];
    }
pf1->average = tot / n;
showFlex(pf1);
     n = 9;
     tot = 0;
     pf2 = malloc(sizeof(struct Flex) + n * sizeof(double));
     pf2->count = n;
for(i = 0; i < n; i++)
{
     pf2->scores[i]  = 20.0 - i / 2.0;
     tot += pf2->scores[i];
}
   pf2->average = tot / n;
   showFlex(pf2);
   free(pf1);
   free(pf2);
   return 0;
}
 void showFlex(const struct Flex *p)
{
  int i;
  printf("Рейтинги: ");
  for(i = 0; i < p->count; i++)
```

```
    printf("%g ", p->scores[i]);
    printf("\nСереднє значення: %g\n", p->average);
}
```

Результат роботи програми наведено на рис. 1.13.

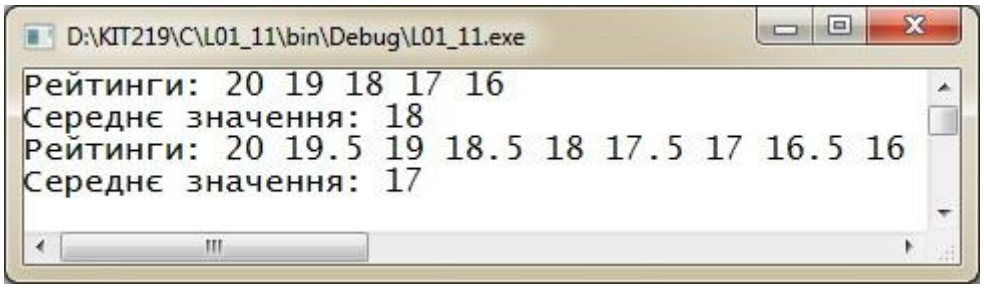


Рисунок 1.13 – Результат виконання програми, яка демонструє використання функції для обробки структури з членом типу гнучкого масиву

До обробки структур, що містять члени з типами гнучких масивів, пред'являється ряд спеціальних вимог. По-перше, не використовуйте присвоювання структур для копіювання:

```
struct Flex *pf1, *pf2;      // *pf1 і *pf2 є структурами
...
*pf2 = *pf1;                 // не робіть таким чином
```

Такий код призвів би до копіювання тільки членів структури, які не стосуються до типу гнучкого масиву. Замість цього застосовуйте функцію **memcpy()**.

По-друге, не використовуйте такі структури спільно з функціями, які передають структури за значенням. Причина такого обмеження та ж сама – передавання аргументу за значенням подібне до присвоювання. Замість цього застосовуйте функції, які передають адресу структури.

По-третє, не використовуйте структуру з членом типу гнучкого масиву в якості елемента масиву або члена іншої структури.

1.2.6. Анонімні структури

Анонімна структура – це член структури, який є неіменованою структурою. Щоб подивитися, як це працює, спочатку розглянемо наступне визначення для вкладеної структури:

```
struct Names
{
     char first[20];
     char last[20];
}
struct Person
{
     int    id;
     struct Names name;  // член, що являє собою вкладену
                         // структуру
};
struct Person ted = { 8483, ("Ted", "Grass"} };
```

В цьому прикладі член **name** – це вкладена структура, і для отримання доступу до "**Ted**" можна було б застосувати вираз на зразок **ted.name.first**:

```
puts(ted.name.first);
```

Стандарт **C11** дозволяє визначати структуру **Person**, використовуючи в якості члена вкладену неіменовану структуру:

```
struct Person
{
     int id;
     struct         // анонімна структура
     {
          char first [20];
          char last[20];
     };
};
```

Цю структуру можна було б ініціалізувати в тому ж самому стилі:

```
struct Person ted = { 8483, { "Ted", "Grass" } };
```

Але доступ до членів спрощується, оскільки для цього застосовуються імена членів на зразок **first**, ніби якщо б вони були членами **Person**:

```
puts(ted.first);
```

Розуміється, можна було б просто зробити **first** і **last** безпосередніми членами структури **Person**, позбувшись вкладеної структури. Засіб анонімності є більш корисним з вкладеними об'єднаннями.

1.2.7. Функції, що використовують масив структур

Припустимо, що є масив структур, який необхідно обробити за допомогою функції. Ім'я масиву – це синонім його адреси, відповідно його можна передавати функції. Додатково функція потребує доступу до шаблону структури. Щоб продемонструвати, як це працює, розглянемо програму фінансового аналізу, яка була розширена з метою обслуговування двох людей, тому в ній є присутнім масив з двох структур **Funds**.

```
  #include <stdio.h>
  #include <windows.h>
  #define FUNDLEN 50
  #define N        2
   struct
Funds
  {
    char
bank[FUNDLEN];
double  bankfund;
char    save[FUNDLEN];
double  savefund;
  };
  double sum(const struct Funds money[], int n);

  int main(void)
  {
    struct Funds jones[N] =
    {
      {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
      },
      {
        "Honest Jack's Bank",
3620.88,
        "Party Time Savings",
```

```
        3802.91
      }
    };

    SetConsoleOutputCP(1251);

    printf("Загальна сума на рахунках у Джонсів складає $%.2f\n",
sum(jones,N));
return 0;
}

  double sum(const struct Funds money[], int n)
  {
    double total;
    int    i;

    for(i = 0, total = 0; i < n; i++)
         total += money[i].bankfund + money[i].savefund;
    return(total);
  }
```

Результат роботи програми наведено на рис. 1.14.

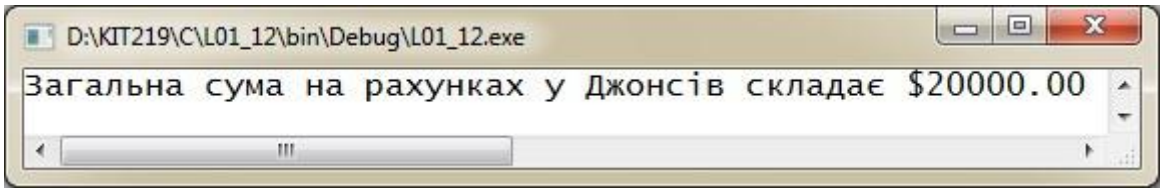


Рисунок 1.14 – Результат виконання програми, яка демонструє використання функції, що використовують масив структур

Ім’я масиву **jones** є його адресою. Зокрема, це адреса першого елемента масиву, який являє собою структуру **jones[0]**. Таким чином, спочатку вказівник **money** задається наступним виразом:

```
money = &jones[0];
```

Оскільки **money** вказує на перший елемент масиву **jones**, то **money[0]** – це ще одне ім’я першого елемента масиву. Аналогічно, **money[1]** – другий елемент

масиву. Кожен елемент є структурою **Funds**, тому для кожного з них можна застосовувати операцію «крапка» (.), щоб звертатися до членів структури.

Нижче перелічені основні аспекти:

- ім'я масиву можна використовувати для передавання до функції адреси першої структури масиву;
- для доступу до подальших структур масиву можна застосовувати запис з квадратними дужками. Зверніть увагу, що виклик функції `sum(&jones[0], N)`

призведе до таких самих результатів, як і у випадку зазначення імені масиву, оскільки **jones** і **&jones[0]** – це одна й та ж сама адреса. Використання імені масиву являє собою просто непрямий спосіб передавання адреси структури;

- через те, що функція **sum()** не повинна змінювати початкові дані, в ній застосовується кваліфікатор **const** з **ANSI C**.

1.2.8. Зберігання вмісту структур у файлі

Оскільки структури можуть містити різноманітну інформацію, вони є важливими інструментами для побудови баз даних. Наприклад, структуру можна використовувати для зберігання інформації про службовців компанії або про автомобільні запчастини. У підсумку неминуче виникає необхідність зберігати цю інформацію в файлі та отримувати її з файлу. Файл бази даних може містити довільну кількість таких об'єктів даних. Повний набір інформації, що зберігається в структурі, називається записом, а окремі члени структури – полями.

Ймовірно, що найбільш очевидний, але й найменш ефективний спосіб зберігання запису передбачає застосування функції **fprintf()**.

В якості прикладу згадаємо структуру **Book**:

```
#define MAXTITL 40
#define MAXAUTL 40
struct Book
{
```

```
    char  title[MAXTITL];
    char  author[MAXAUTL];
    float value;
};
```

Якщо **pbooks** ідентифікує файловий потік, то інформацію зі змінної **primer** типу **struct** Book можна було б зберегти за допомогою наступного оператора:

```
fprintf(pbooks, "&s %s %.2f\n", primer.title, primer.author,
primer.value);
```

Такий підхід стає громіздким для структур, які мають, скажімо, **30** членів. Крім того, виникає проблема отримання даних, оскільки програмі необхідно мати певний спосіб з'ясувати, де закінчується одно поле і де починається інше. Проблему можна вирішити, використовуючи формат з полями фіксованого розміру (наприклад, "%39s%39s%8.2f"), але громіздкість нікуди не зникає.

Більш прийнятне рішення полягає в застосуванні функцій **fread()** і **fwrite()** для читання та запису одиниць з розміром структури. Згадайте, що ці функції здійснюють читання та запис з використанням того ж самого двійкового представлення, як і програма. Наприклад, виклик

```
fwrite(&primer, sizeof(struct Book), 1, pbooks);
```

переходить до початкової адреси структури **primer** і копіює усі байти цієї структури до файлу, що асоціюється з **pbooks**. Вираз **sizeof(struct** Book) повідомляє функції розмір блоку, що підлягає копіюванню, а **1** означає, що повинен копіюватися тільки один блок. Функція **fread()** з тими ж самими аргументами копіює порцію даних розміром зі структуру з файлу до області пам'яті, на яку вказує **&primer**. Тобто ці функції читають і записують за один раз повний запис, а не поле.

Один з недоліків зберігання даних в двійковому представленні пов'язаний з тим, що в різних системах можуть застосовуватися різні двійкові представлення, тому файл даних може виявитися таким, що не може бути перенесений на інший комп'ютер. Навіть в одній і тій самій системі різні налаштування компілятора можуть в результаті призводити до отримання різних двійкових представлень.

1.2.9. Приклад зберігання структури

Щоб продемонструвати використання цих функцій, модифікуємо код однієї з попередніх програм, щоб відомості про книги зберігалися у файлі на ім'я **book.dat**. Якщо файл вже існує, програма відображає його поточний вміст і потім дозволяє додати до файлу нові дані. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <string.h>

#define MAXTITL  40
#define MAXAUTL  40
#define MAXBOOKS 10                    // максимальна кількість книг

char *s_gets(char *st, int n);

struct Book                            // визначення шаблону Book
{
  char  title[MAXTITL];
  char  author[MAXAUTL];
  float value;
};

int main(void)
{
   struct Book library[MAXBOOKS];      // масив структур
   int count = 0;
   int index, filecount;
   FILE *pbooks;

  SetConsoleCP(1251);
  SetConsoleOutputCP(1251);

  int size = sizeof(struct Book);
  if((pbooks = fopen("book.dat", "a+b")) == NULL)
  {
    fputs("Не вдається відкрити файл book.dat\n", stderr);
    exit(1);
  }
  rewind(pbooks);                      // перехід на початок файлу
while(count < MAXBOOKS && fread(&library[count],
     size, 1, pbooks) == 1)
```

```
  {
    if(count == 0)
      puts("Поточний вміст файлу book.dat:");
    printf("%s, автор: %s, $%.2f\n", library[count].title,
library[count].author, library[count].value);
count++;
}
  filecount = count;
  if(count == MAXBOOKS)
  {
    fputs("Файл book.dat заповнений", stderr);
    exit(2);
  }
  puts("Введіть назви нових книг");
  puts("Натисніть [enter] на початку рядка, щоб закінчити ввід.");
  while(count < MAXBOOKS && s_gets(library[count].title,
MAXTITL) != NULL && library[count].title[0] != '\0')
  {
    puts("Тепер введіть ім'я автора: ");
    s_gets(library[count].author, MAXAUTL);
    puts("Тепер введіть ціну книги:  ");
    scanf("%f", &library[count++].value);
    while(getchar() != '\n')
      continue;                    // очистити вхідний рядок
if(count < MAXBOOKS)
      puts("Введіть назву наступної книги: ");
}
if(count > 0)
{
    puts("Каталог ваших книг:");
for(index = 0; index < count; index++)
      printf("%s авторства %s:  $%.2f\n", library[index].title,
              library[index].author, library[index].value);
fwrite(&library[filecount], size, count - filecount, pbooks);
  }
else
    puts("Зовсім немає книг? Дуже погано.\n");
puts("Програма завершена.\n");
fclose(pbooks);
return 0;
}
char *s_gets(char *st, int n)
{
 char *ret_val;
 char *find;
  ret_val = fgets(st, n, stdin);
  if(ret_val)
  {
   find = strchr(st, '\n');  // пошук нового рядка
```

```
    if(find)                    // якщо адреса не дорівнює NULL,
      *find = '\0';             // помістіть туди нульовий символ
else
      while(getchar() != '\n')
        continue;               // відкинути залишок рядка
}

  return ret_val;
}
```

Результат роботи програми наведено на рис. 1.15.

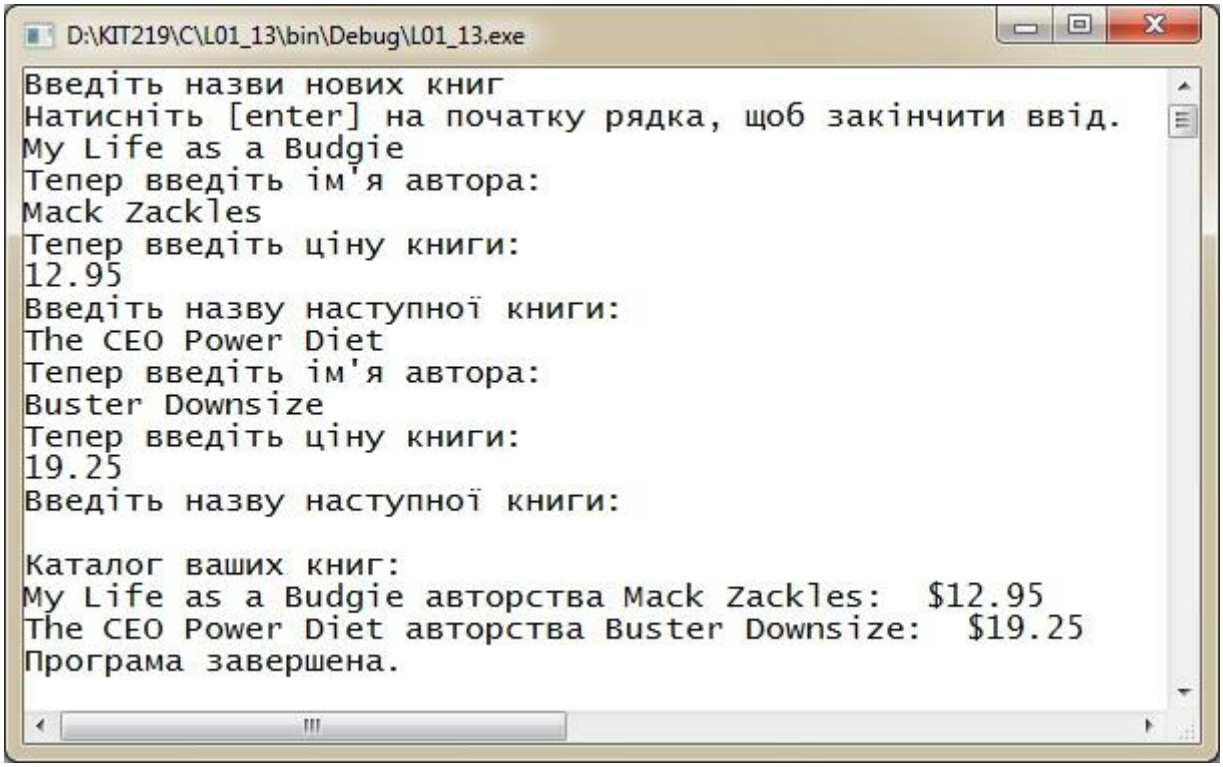


Рисунок 1.15 – Результат виконання програми, яка демонструє використання функцій роботи з файлами, що використовують масив структур

1.2.10. Аналіз програми

Спочатку файл відкривається в режимі "**a+b**". Частина **a+** дозволяє програмі читати весь файл і додавати дані в кінець файлу. Частина **b** – це спосіб, що прийнятий в **ANSI** для повідомлення про те, що програма буде застосовувати двійковий файловий формат.

Двійковий режим був обраний через те, що функції **fread()** і **fwrite()** призначені для роботи з двійковими файлами. Дійсно, деякий вміст структури є текстовим, однак член **value** – ні. Якщо ви скористаєтесь текстовим редактором для перегляду файлу **book.dat**, то текстова частина буде відображатися нормально, але числова частина виявиться нечитабельною і може навіть стати причиною отримання попереджень.

Виклик **rewind()** забезпечує встановлення вказівника позиції на початок файлу, приводячи його у стан готовності до першого читання.

Перший цикл **while** читає одну структуру за один раз до масиву структур, зупиняючись при заповненні цього масиву, або в ситуації, коли усі дані в файлі вичерпані. Змінна **filecount** відстежує кількість прочитаних структур.

Наступний цикл **while** запитує й отримує ввід користувача. Цей цикл припиняється, коли масив заповнений або користувач натиснув клавішу **<Enter>** на початку рядка. Зверніть увагу, що змінна **count** починається зі значення, яке вона отримала по закінченні попереднього циклу. Це призводить до додавання нових записів наприкінці масиву.

Потім в циклі **for** виводяться дані, які отримані з файлу та від користувача. Оскільки файл був відкритий в режимі додавання, нові записи приєднуються до існуючого вмісту.

Можна було б скористатися циклом і додавати структури в кінець файлу по одній кожного разу. Однак в програмі була використана здатність функції **fwrite()** записувати декілька блоків за один раз. Вираз **count - filecount** дає кількість доданих нових книг, а виклик **fwrite()** записує до файлу таку ж саму кількість блоків розміром зі структуру. Вираз **&library[filecount]** – це адреса першої нової структури в масиві, тому копіювання починається з цієї точки.

Ймовірно, що розглянутий приклад є найпростішим способом запису структур до файлу та їх отримання з файлу, але в ньому може даремно витрачатися простір, оскільки також зберігаються і частини структури, що не використовуються.

Розмір структури складає `2 * 40 * sizeof(char) + sizeof(float)`, що в системі дає в сумі **84** байти. Жоден з записів в дійсності не потребує всього цього простору. Проте, однаковий розмір усіх порцій даних спрощує отримання даних.

Інший підхід полягає в застосуванні записів змінних розмірів. Для полегшення зчитування таких записів з файлу кожен запис може починатися з числового поля, що вказує розмір запису. Зазвичай даний підхід передбачає використання зв'язаних структур.

1.3. Об'єднання, перелічувані типи та бітові поля

1.3.1. Об'єднання

Об'єднання – це тип, який дозволяє зберігати дані різних типів в одному й тому ж самому місці пам'яті (але не одночасно). Типовим видом об'єднання може служити таблиця, що призначена для зберігання суміші типів в певному порядку, який не є ні регулярним, ні відомим заздалегідь. Застосовуючи масив об'єднань, можна створити масив одиниць однакових розмірів, кожна з яких може містити дані різних типів.

Об'єднання формуються подібно структурам: існує шаблон об'єднання та змінна типу об'єднання. Вони можуть бути визначені за допомогою однієї або двох дій за рахунок використання дескриптора об'єднання. Нижче показаний приклад шаблону об'єднання з дескриптором:

```
union Hold
{
int         digit;
double      bigfl;
char     letter;
};
```

Структура зі схожим оголошенням може зберігати значення типів **int**, **double** та **char** одночасно, однак об'єднання може зберігати значення типу **int**

або **double** або **char**. Розглянемо приклад визначення трьох змінних об'єднання типу **Hold**:

```
union Hold fit;          // змінна об'єднання типу Hold
union Hold save[10];     // масив з 10 змінних об'єднання
union Hold *pu;          // вказівник на змінну типу Hold
```

Перше оголошення створить одиночну змінну **fit**. Компілятор виділяє ділянку пам'яті, яка буде достатньою для зберігання найбільшої з описаних можливостей. В даному випадку найбільшим варіантом з перерахованих є тип **double**, який потребує **8** байтів. Друге оголошення створює масив на ім'я **save** з **10** елементами, кожен з яких має розмір **8** байтів. Третє оголошення створює вказівник, який може містити адресу об'єднання **Hold**.

Ініціалізація об'єднань . Оскільки об'єднання зберігає тільки одне значення, правила його ініціалізації відрізняється від таких самих правил для структур. До речі, доступні три варіанти:

- ініціалізувати об'єднання іншим об'єднанням того ж самого типу;
- ініціалізувати перший елемент об'єднання;
- у випадку **C99** застосувати призначений ініціалізатор:

```
union Hold val_A;
val_A.letter = 'R';
// ініціалізація одного об'єднання іншим
union Hold val_B = val_A;
// ініціалізація члену digit об'єднання
union Hold val_C = { 88 };
// призначений ініціалізатор union
Hold val_D = { .bigfl = 118.2 };
```

Використання об'єднань. Нижче показано, як можна використовувати об'єднання:

```
// у змінній fit зберігається 23 (використовується 4 байти)
fit.digit = 23;
// 23 очищено, 2.0 збережено (використовується 8 байтів)
fit.bigfl = 2.0;
// 2.0 очищено, 'h' збережено (використовується 1 байт)
fit.letter = 'h';
```

Операція «крапка» показує, який тип даних застосовується в даний момент. За один раз зберігається тільки одне значення. Неможна одночасно зберігати значення **char** і **int**, незважаючи на те, що місяця для цього цілком достатньо. Відповідальність за відстеження в програмі типу даних, що зберігається в даний момент всередині об'єднання, накладається на програміста.

Можна використовувати операцію **"->"** з вказівниками на об'єднання в тому ж стилі, як це робилося з вказівниками на структури:

```
pu = &fit;
x = pu->digit;            // те ж саме, що й x = fit.digit
```

Нижче показано, як не слід робити:

```
fit.letter = 'A';
flnum = 3.02 * fit.bigfl;   // ПОМИЛКА!
```

Ця послідовність помилкова, оскільки збережено значення типу **char**, але в наступному рядку передбачається, що вміст **fit** має тип **double**.

Проте, іноді буває корисним використовувати один член для розташування значення в об'єднанні, а інший – для перегляду вмісту об'єднання.

Іншою ситуацією застосування об'єднань є структура, в якій інформація, що зберігається, залежить від значення одного з її членів. Припустимо, що ви маєте структуру «автомобіль». Якщо автомобіль належить користувачу, бажано, щоб член структури описував власника. Якщо автомобіль взятий напрокат, то необхідно, щоб член описував компанію з прокату. Тоді можна записати так:

```
struct Owner
{
    char socsecurity[12];
    ...
};

struct Leasecompany
{
    char name[40];
char headquarters[40];
    ...
};
 union
Data
{
```

```
    struct Owner owncar;
struct Leasecompany leasecar;
};

struct Car_data
{
    char make[15];
    int status;       // 0 - власник, 1 - взятий напрокат
union Data ownerinfo;
    ...
}
```

Нехай **flits** – структура **Car_data**, тоді якщо значення **flits.status == 0**, програма може використовувати **flits.ownerinfo.owncar.socsecurity**, а якщо значення **flits.status == 1**, – то **flits.ownerinfo.leasecar.name**.

Анонімні об'єднання (C11) працюють у багатьох випадках подібно до анонімних структур. Тобто анонімне об'єднання – це неіменоване об'єднання, яке є членом структури або об'єднання. Наприклад, структуру **Car_data** можна перевизначити наступним чином:

```
struct Owner
{
    char socsecurity[12];
    ...
};

struct Leasecompany
{
    char name[40];
char headquarters[40];
    ...
};

struct Car_data
{
    char make[15];
    int  status;       // 0 - власник, 1 - взятий напрокат
union
    {
        struct Owner owncar;
struct Leasecompany leasecar;
    };
};
```

Тепер, якщо **flits** – це структура **Car_data**, то можна застосовувати **flits.owncar.socsecurity** замість **flits.ownerinfo.owncar.socsecurity**.

1.3.2. Перелічувані типи

Перелічуваний тип можна використовувати для оголошення символічних імен, що являють собою цілочислові константи. Ключове слово **enum** дозволяє створити новий «тип» і вказати значення, які для нього допускаються. Насправді константи **enum** мають тип **int**, тому їх можна застосовувати усюди, де дозволено використовувати тип **int**. Метою перелічуваних типів є покращення читабельності програми. Їх синтаксис схожий на синтаксис, що застосовується для структур. Наприклад, можна записати наступні оголошення:

```
enum Spectrum { RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET };
enum Spectrum color;
```

Перше оголошення встановлює **Spectrum** як ім'я дескриптора, який дозволяє використовувати **enum Spectrum** в якості імені типу. Друге оголошення робить **color** змінною цього типу. Ідентифікатори всередині фігурних дужок перераховують можливі значення, які може мати змінна **Spectrum**. Таким чином, можливими значеннями **color** будуть **RED**, **ORANGE**, **YELLOW** і т. д. Ці символічні константи називаються нумераторами. Після цього допускається застосування операторів, які наведені нижче:

```
int c; color =
BLUE; if(color
== YELLOW)
    ...;
for(color = RED; color <= VIOLET; color++)
...;
```

Хоча нумератори на зразок **RED** і **BLUE** мають тип **int**, змінні перелічуваного типу не так жорстко прив'язані до цілочислового типу до тих пір, поки цей тип може містити перелічувані константи. Наприклад, перелічувані константи для **Spectrum** входять в діапазон **0-5**, тому для представлення змінної **color** компілятор міг би обрати тип **unsigned char**.

Деякі властивості перерахувань C не переносяться до C++. Наприклад, C дозволяє застосовувати до перелічуваної змінної операцію ++, але стандарт C++ цього не дозволяє. Таким чином, якщо ви припускаєте, що в майбутньому код може бути об'єднаний з програмою C++, то повинні оголосити змінну color в попередньому прикладі як такі, що відносяться до типу int. Тоді код буде працювати як в C, так і в C++.

Константи enum . Так що ж собою являють BLUE і RED? Формально вони є константами типу int. Наприклад, маючи попереднє оголошення перелічуваного типу, можна записати таким чином:

```
printf("red = %d, orange = %d\n", RED, ORANGE);
```

Нижче показаний результат виконання функції:

```
red = 0, orange = 1
```

Виявилося, що RED стала іменованою константою, яка являє собою цілочислове значення 0. Подібним чином інші ідентифікатори є іменованими константами, що являють собою цілі числа від 1 до 5. Перелічену константу можна використовувати всюди, де дозволяється застосування цілочислової константи. Наприклад, їх можна використовувати для зазначення розмірів в оголошеннях масивів або в якості міток в операторі switch.

Стандартні значення. За замовчуванням константам у списку перелічувань присвоюється цілочислові значення 0, 1, 2 і т. д. Відповідно, оголошення `enum Kids { NIPPY, SLATS, SKIPPY, NINA, LIZ };` призводить до того, що NINA має значення 3.

Присвоєні значення. За бажанням можна обрати цілочислові значення, які повинні мати константи. Для цього необхідно включити потрібні значення в оголошення:

```
enum Levels { LOW = 100, MEDIUM = 500, HIGH = 2000 };
```

Якщо значення присвоюється одній константі, але не тій, що йде за нею, то подальші константи отримають значення, які послідовно зростають на **1**.

Наприклад, погляньте на наступне оголошення:

```
enum Feline { CAT, LYNX = 10, PUMA, TIGER };
```

В цьому випадку **CAT** отримає стандартне значення **0**, а **LYNX**, **PUMA** і **TIGER** – відповідно, **10**, **11** і **12**.

Використання enum. Метою перелічуваних типів є покращення читабельності програми та спрощення її супроводу. Якщо ви маєте справу з кольорами, то застосування **RED** (червоний) і **BLUE** (синій) є набагато інформативним, ніж значення **0** і **1**. Зверніть увагу, що перелічувані типи призначені для внутрішнього використання. Якщо ви бажаєте ввести значення **ORANGE** для змінної **color**, то повинні вводити **1**, а не слово **ORANGE**, або ж можна прочитати рядок **"orange"** та змусити програму перетворити її на значення **ORANGE**.

Через те, що перелічуваний тип є цілочисловим, змінні **enum** можуть застосовуватися у виразах таким самим чином, як і цілочислові змінні. Вони являють собою зручні мітки для операторів **case**.

Розглянемо приклад використання **enum**, який розрахований на стандартну схему присвоювання значень. В результаті константа **RED** отримує значення **0**, яке робить її індексом для вказівника на рядок **"red"**. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>      // для strcmpn(), strchr()
#include <stdbool.h>     // засіб C99
#define LEN 30
 char *s_gets(char *st, int
n);
 enum Spectrum { RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET
};
 const char *colors[] = { "red",    "orange", "yellow",
                          "green", "blue",   "violet" };
 int main(void)
```

```
{
char choice[LEN];
enum Spectrum color;
bool color_is_found = false;

 SetConsoleOutputCP(1251);
 puts("Введіть колір (або порожній рядок для виходу):");
 while(s_gets(choice, LEN) != NULL && choice[0] != '\0')
    {
       for(color = RED; color <= VIOLET; color++)
       {
            if(strcmp(choice, colors[color]) == 0)
            {
                color_is_found = true;
                 break;
             }
        }
        if(color_is_found)
        {
            switch(color)
            {
                case RED:    puts("Троянди червоні.");
                             break;
                case ORANGE: puts("Маки оранжеві.");
                             break;
                case YELLOW: puts("Соняшники жовті.");
                             break;
                case GREEN:  puts("Трава зелена.");
                             break;
                case BLUE:   puts("Дзвоники сині.");
                             break;
                case VIOLET: puts("Фіалки фіолетові.");
                             break;
                default:     break;
             }
      }
      else
            printf("Колір %s не визначено.\n", choice);
  color_is_found = false;
  puts("Введіть наступний колір"
              " (або порожній рядок для виходу):");
    }
    puts("Програма завершена.");
  return 0;
}


  char * s_gets(char *st, int n)
```

```
{
    char *ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);

    if(ret_val)     // тобто ret_val  != NULL
    {
       while(st[i] != '\n' && st[i] != '\0')
             i++;
       if(st[i] == '\n')
          st[i] = '\0';
       else
            while(getchar() != '\n')
continue;
    }
    return ret_val;
}
```

Цикл **for** завершується, коли вхідний рядок збігається з одним з рядків, на які вказують елементи масиву **colors**. Якщо цикл знаходить відповідний колір, то значення перелічуваної змінної застосовується для зіставлення з перелічуваною константою, що використовується в якості мітки **case**.

Результати виконання програми наведено на рис. 1.16.

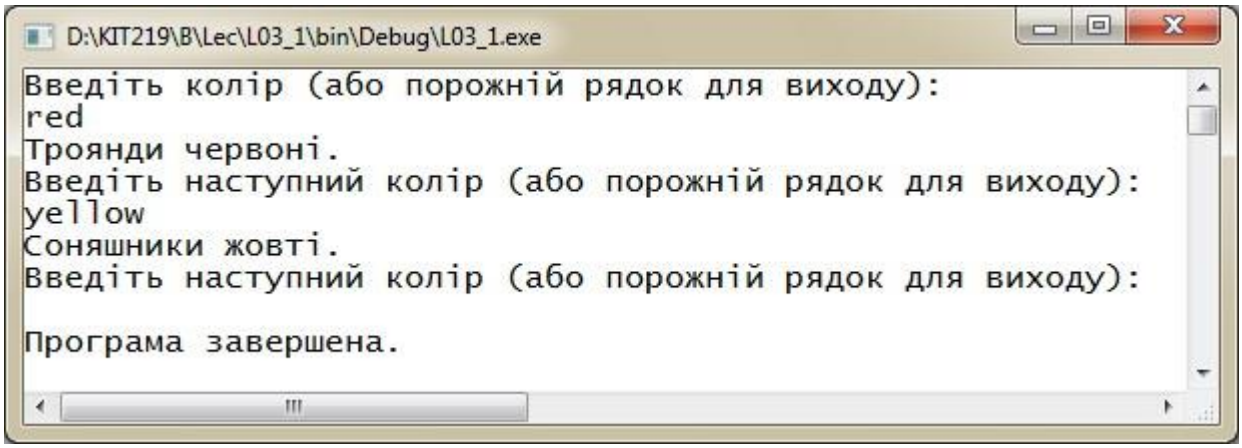


Рисунок 1.16 – Результат виконання програми для демонстрації роботи з **enum**

1.3.3. Простір імен, який спільно використовується

Термін простір імен в мові **C** застосовується для ідентифікації частин програми, в яких розпізнаються те або інше ім'я. Область видимості входить до складу цієї концепції: дві змінні, що мають одне й те ж саме ім'я, але різні області видимості, не конфліктують одна з одною, на відміну від двох змінних з однаковими іменами та однією й тією ж самою областю видимості. Існує також аспект категорії, що стосується простору імен. Дескриптори структур, дескриптори об'єднань і дескриптори перелічувань у визначеній області видимості спільно використовують один і той самий простір імен, і цей простір імен відрізняється від простору, який застосовується звичайними змінними. Це означає, що можна назначити однакові імена змінній і дескриптору в рамках однієї й тієї ж самої області видимості без виникнення помилки, але неможна оголошувати два дескриптори або дві змінні з одним і тим самим ім'ям в тій самій області видимості.

Наприклад, наступні оголошення не призводять до конфлікту імен в **C**:

```
struct Rect
{
double x;
double y;
};
int Rect;      // конфлікту в C не виникає
```

Проте, використання одного ідентифікатора двома різними шляхами може призвести до плутанини, до того ж це не дозволено в **C++**, оскільки там дескриптори та змінні поміщаються в той же самий простір імен.

1.3.4. Засіб typedef

Ключове слово **typedef** являє собою вдосконалений засіб маніпулювання даними, який дозволяє створювати власне ім'я для типу. В цьому сенсі він подібний директиві **#define**, але з трьома відмінностями:

1) На відміну від **#define**, засіб **typedef** обмежений призначенням символічних імен тільки типам, але не значенням.

2) Інтерпретація **typedef** виконується компілятором, а не препроцесором.

3) В рамках своїх обмежень засіб **typedef** є більш гнучким, ніж **#define**.

Давайте поглянемо, як працює **typedef**. Припустимо, що ви бажаєте використовувати елемент **BYTE** для позначення однобайтових чисел. Тоді ви просто оголошуєте **BYTE**, наче це змінна типу **unsigned char**, і попереду визначення записуєте ключове слово **typedef**:

```
typedef unsigned char BYTE;
```

Після цього **BYTE** можна застосовувати для визначення змінних:

```
BYTE x, y[10], *z;
```

Область видимості цього визначення залежить від місця розташування оператора **typedef**. Якщо визначення знаходиться всередині функції, область видимості буде локальною в межах цієї функції. Якщо визначення знаходиться поза межами функції, область видимості буде глобальною.

Для визначень **typedef** часто використовуються великі літери, щоб нагадувати користувачу про те, що ім'я типу в дійсності є символічним скороченням, але дозволені також і малі літери:

```
typedef unsigned char byte;
```

До імен в **typedef** застосовуються ті ж самі правила, які регламентують створення допустимих імен змінних.

Хоча створення імені для існуючого типу може здатися незначною можливістю, але часто воно може виявитися досить зручним. В попередньому прикладі позначення **BYTE** замість **unsigned char** допомагає документувати намір застосовувати змінні **BYTE** для представлення чисел, а не символьних кодів. Використання **typedef** також сприяє кращому перенесенню на іншу систему. Наприклад, раніше ми згадували про тип **size_t**, який являє собою тип, що повертається операцією **sizeof**, і про тип **time_t**, що являє собою тип, який

повертається функцією `time()`. Стандарт C стверджує, що `sizeof` і `time()` повертають цілочислові типи, але те, якими вони повинні бути, залишає на розсуд реалізації. Причина відсутності конкретики пояснюється тим, що в комітеті зі стандартів C дотримуються думки, що, скоріш за все, не існує єдиного вибору, який був би найкращим для усіх комп'ютерних платформ. Таким чином, було вирішено створити нове ім'я типу, подібне до `time_t`, і дозволити різним реалізаціям застосовувати `typedef` для встановлення цього імені в будь-який конкретний тип. Тоді з'являється можливість надати спільний прототип, такий як показано нижче:

```
time_t time(time_t *);
```

В одній системі `time_t` може бути `unsigned long`, а в іншій – `unsigned long long`. За умови, що підключено файл заголовку `time.h`. програма може отримувати доступ до потрібного визначення, і в коді можна оголошувати змінні типу `time_t`.

Деякі можливості `typedef` можна продублювати за допомогою `#define`.

Наприклад, вказівка `#define BYTE unsigned char` змушує препроцесор замінювати `BYTE` типом `unsigned char`. Нижче наведено приклад `typedef`, який неможливо відтворити за допомогою `#define`:

```
typedef char *STRING;
```

Без ключового слова `typedef` в цьому прикладі сама змінна `STRING` ідентифікувалася б як вказівник на `char`. Наявність `typedef` робить `STRING` ідентифікатором для вказівників на `char`. Таким чином

`STRING name,sign;` означає `char *name, *sign;`

Припустимо, що замість цього ви зробите таким чином:

```
#define STRING char *
```

Тоді

`STRING name, sign;` транслюється в `char *name, sign;`

В цьому випадку вказівником буде лише **name**.

Засіб **typedef** можна також використовувати зі структурами:

```
typedef struct Complex
{
   float real;
   float imag;
} COMPLEX;
```

Тепер для представлення комплексних чисел замість структури на ім'я **Complex** можна застосовувати тип **COMPLEX**. Одна з цілей використання **typedef** пов'язана зі створенням зручних імен, що легко впізнавати для типів, які зустрічаються найчастіше. Наприклад, багато хто з програмістів застосовують переважним чином ім'я типу **STRING** або його еквівалент, як у прикладі, що був розглянутий раніше. При використанні **typedef** для іменування типу «структура» дескриптор можна не вказувати:

```
typedef struct { double x; double y; } rect;
```

Припустимо, що визначений за допомогою **typedef** ідентифікатор застосовується так, як показано нижче;

```
rect r1 = { 3.0, 6.0 };
rect r2;
```

Цей код транслюється на наступні оператори:

```
struct { double x; double y; } r1 = { 3.0, 6.0 };
struct { double x; double y; } r2;
r2 = r1;
```

Якщо дві структури оголошені без дескриптора, але з ідентичними членами (збігаються імена членів і типів), то в **C** ці дві структури вважаються такими, що мають один і той самий тип, тому присвоювання **r1** змінній **r2** є допустимою операцією.

Друга причина використання **typedef** пов'язана з тим, що імена **typedef** часто застосовуються для складених типів. Наприклад, оголошення **typedef char (* FRPTC()) [5];** робить **FRPTC** ідентифікатором типу, який є функцією, що повертає вказівник на масив з **5** елементів **char**.

Під час використання засобу **typedef** майте на увазі, що він не створює нові типи: замість цього ви отримуєте усього лише зручні мітки. Це означає, наприклад, що змінні, що застосовують створений тип **STRING**, можуть передаватися в якості аргументів функціям, які очікують тип вказівника на **char**.

Завдяки структурам, об'єднанням і засобу **typedef**, мова **C** надає інструменти для ефективної обробки даних.

1.3.5. Бітові поля

Одним з методів маніпулювання бітами є застосування бітових полів, які являють собою набір сусідніх бітів всередині значення типу **signed int** або **unsigned int**. Стандарти **C99** і **C11** додатково дозволяють мати бітові поля типу **_Bool**.

Створення бітових полів. Бітове поле створюється шляхом оголошення структури, в якій зазначено кожне поле та визначено його розмір. Наприклад, наступне оголошення встановлює чотири однобітових поля:

```
struct
{
unsigned int a1 : 1;
unsigned int b1 : 1;
unsigned int c1 : 1;
unsigned int d1 : 1;
} prnt;
```

Таке визначення призводить до отримання структури **prnt**, яка містить чотири однобітових поля. Тепер для присвоювання значень окремим полям можна застосовувати звичайну операцію членства в структурі.

```
prnt.a1 = 0;
prnt.b1 = 1;
```

Оскільки кожне з цих полів – це просто один біт, присвоювати можна тільки значення **1** і **0**. Змінна **prnt** зберігається в комірці пам'яті розміром типу **int**, але в цьому прикладі використовуються тільки чотири біта.

Структури з бітовими полями є зручним засобом для відстеження налаштувань. Чисельні налаштування, такі як напівжирне або курсивне

начертання шрифту, зводяться до зазначення однієї з двох опцій: "увімкнути" або "вимкнути", "так" або "ні", "істинно" або "хибно". Коли потрібен одиночний біт, не має сенсу застосовувати цілу змінну. Структура з бітовими полями дозволяє зберігати множину налаштувань в одній конструкції.

Часом налаштування передбачає більш ніж дві опції, тому для представлення всіх варіантів одного біта виявляється недостатньо. Це не є проблемою, тому що розміри полів не обмежені одним бітом. Структуру можна визначити наступним чином:

```
struct
{
unsigned int code1 : 2;
unsigned int code2 : 2;
unsigned int code3 : 8; }
prcode;
```

Цей код створить два двобітових поля та одне восьмибітове. Тепер можливі наступні присвоювання:

```
prcode.code1 =   0;
prcode.code2 =   3;
prcode.code3 = 102;
```

Треба просто стежити, щоб значення не перевищувало розмірність поля.

Що ж відбудеться, якщо загальна кількість оголошених бітів перевищить розмір типу `unsigned int`? Тоді буде використовуватися наступна область для зберігання `unsigned int`. Окреме поле не повинно перекривати межу між двома суміжними областями `unsigned int`. Компілятор автоматично зсовує таке визначення поля, яке перекривається, щоб вирівняти його по межі `unsigned int`. Коли це відбувається, в першій області `unsigned int` залишається неіменований проміжок.

Структуру полів можна заповнити неіменованими проміжками з застосуванням ширини неіменованих полів. Використання неіменованого поля шириною `0` призводить до того, що наступне поле вирівнюється по наступній області цілочислового значення:

```
struct
{
```

```
unsigned int field1 : 1;
unsigned int        : 2;
unsigned int fieid2 : 1;
unsigned int        : 0;
unsigned int field3 : 1; }
stuff;
```

В даному фрагменті між полями **stuff.field1** і **stuff.field2** існує двобітовий проміжок, а поле **stuff.field3** зберігається в наступній області **int**. Важливою залежністю від системи є порядок, в якому поля поміщаються в область **int**. В одних системах підтримується порядок зліва направо, в інших – справа наліво. Крім того, системи різняться місцезнаходженням меж між полями. За цими причинами бітові поля не особливо можна переносити. Однак зазвичай вони застосовуються з метою, що не передбачає цього, такою як розміщення даних в точній формі, що виконується окремим апаратним пристроєм.

Використання бітових полів. Бітові поля часто застосовуються в якості більш компактного способу зберігання даних. Припустимо, що ви вирішили представити властивості вікна, що виводиться на екран. Давайте не будемо вдаватися до усіх складностей графіки і припустимо, що вікно має всі властивості, що зазначені нижче:

- вікно може бути прозорим або непрозорим;
- колір фону обирається з наступної палітри: чорний, червоний, зелений, жовтий, синій, пурпурний, блакитний і білий;
- рамка може бути прихована або відображатися;
- колір рамки обирається з тієї ж самої палітри, що й колір фону;
- для рамки застосовуються три стилі лінії: суцільна, пунктирна та штрихпунктирна.

Для кожної властивості можна було б використовувати окрему змінну або повнорозмірний член структури, але це призвело б до даремної витрати бітів. Наприклад, для зазначення прозорості або непрозорості вікна достатньо одного біта. Те ж саме можна сказати про властивість відображення або приховування рамки. Вісім можливих значень кольору можуть бути представлені 3-х бітовим

елементом, а 2-х бітового елемента більш ніж достатньо для представлення трьох можливих стилів рамки. Таким чином, для представлення усіх п'яти властивостей достатньо 10 бітів.

Один з варіантів представлення інформації передбачає застосування заповнювачів, щоб помістити в один байт інформацію, яка пов'язана з фоном вікна, а у другий байт – інформацію, що пов'язана з рамкою. Це реалізовано в наступному оголошенні **struct** Box_props:

```
struct Box_props
{
bool opaque                : 1;
unsigned int fill_color    : 3;
unsigned int               : 4;
bool show_border           : 1;
unsigned int border_color  : 3;
unsigned int border_style  : 2;
unsigned int               : 2;
};
```

В результаті використання заповнювачів розмір структури збільшується до 16 бітів. Без них було б достатньо 10 бітів. Однак майте на увазі, що в **C** для структур з бітовими полями в якості базової одиниці розміщення застосовується тип **unsigned int**. Тому, навіть якщо структура містить єдиний елемент, яким є однобітове поле, структура буде мати такий самий розмір, як у типу **unsigned int**, що складає 32 біти. Крім того, в цьому коді передбачається, що тип **_Bool** з **C99** доступний і в файлі заголовку **stdbool.h** йому призначений псевдонім **bool**.

Для члена **opaque** можна використовувати значення 1 для зазначення непрозорості вікна та значення 0 – для прозорості. Те ж саме можна застосовувати до члена **show_border**.

Для кольорів можна використовувати просте представлення **RGB** (червоний, зелений, синій). Це основні кольори для змішування спектру. В моніторі для відтворення різних кольорів застосовується змішане свічення червоних, зелених і синіх пікселів. В ранніх моделях моніторів кожен піксель міг мати тільки стан «ввімкнено» або «вимкнено», тому для представлення

інтенсивності кожній з трьох складових було достатньо одного біта. Зазвичай лівий біт являв собою інтенсивність синього, середній – інтенсивність зеленого, а правий – червоного кольору.

В табл. 1.1 показані вісім можливих комбінацій.

Вони можуть бути значеннями для членів **fill_color** і **border_color**. Нарешті, значення **0**, **1** і **2** можуть представляти суцільний, пунктирний і штрихпунктирний типи ліній, що визначається членом **border_style**.

Таблиця 1.1 – Представлення кольорів

Комбінація бітів (**RGB**)	Десятковий еквівалент	Колір
000	0	Чорний
001	1	Синій
010	2	Зелений
011	3	Блакитний
100	4	Червоний
101	5	Пурпурний
110	6	Жовтий
111	7	Білий

Розглянемо приклад використання бітових полів, в якому застосовується структура **Box_props**. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdbool.h>  // C99, визначення bool, true, false

// стилі лінії                                                    97
#define DOTTED                                                    97
#define DASHED                                                    97
// основні кольори #define BLUE                                   97


#define SOLID    0
#define GREEN    2
#define RED      1
```

```
// змішані кольори
#define BLACK    0
#define YELLOW   ( RED   | GREEN )
#define MAGENTA  ( RED   | BLUE  )
#define CYAN     ( GREEN | BLUE  )
#define WHITE    ( RED   | GREEN | BLUE )

const char *colors[8] =  {
    "чорний", "червоний",  "зелений",   "жовтий",
    "синій",  "пурпурний", "блакитний", "білий" };

struct Box_props
{
bool opaque               : 1; // або unsigned int (до C99)
unsigned int fill_color   : 3;
unsigned int              : 4;
bool show_border          : 1; // або unsigned int (до C99)
unsigned int border_color : 3;
unsigned int border_style : 2;
unsigned int              : 2;
};
void show_settings(const struct Box_props *pb);

int main(void)
{
struct Box_props box =
  { true, YELLOW, true, GREEN, DASHED };

SetConsoleOutputCP(1251);
printf("==============================\n");
printf("Початкові налаштування вікна: \n");
printf("==============================\n");
show_settings(&box);
box.opaque       = false;
box.fill_color   = WHITE;
box.border_color = MAGENTA;
box.border_style = SOLID;
printf("\n==============================\n");
printf("Змінені налаштування вікна:   \n");
printf("==============================\n");
show_settings(&box);
return 0;
}

void show_settings(const struct Box_props *pb)
{
printf("Вікно        %s.\n",
```

```
        pb->opaque == true ? "непрозоре" : "прозоре");

printf("Колір фону    %s.\n", colors[pb->fill_color]);
printf("Рамка         %s.\n",
        pb->show_border == true ? "відображається"
                                : "не відображається");
printf("Колір рамки   %s.\n", colors[pb->border_color]);
printf("Стиль рамки   ");     switch(pb->border_style)
    {
        case  SOLID: printf("суцільний.\n");
                     break;
        case DOTTED: printf("пунктирний.\n");
                     break;
        case DASHED: printf("штрихпунктирний.\n");
                     break;
            default: printf("невідомого типу\n");
                     break;
    }
}
```

Результат виконання програми наведено на рис. 1.17.

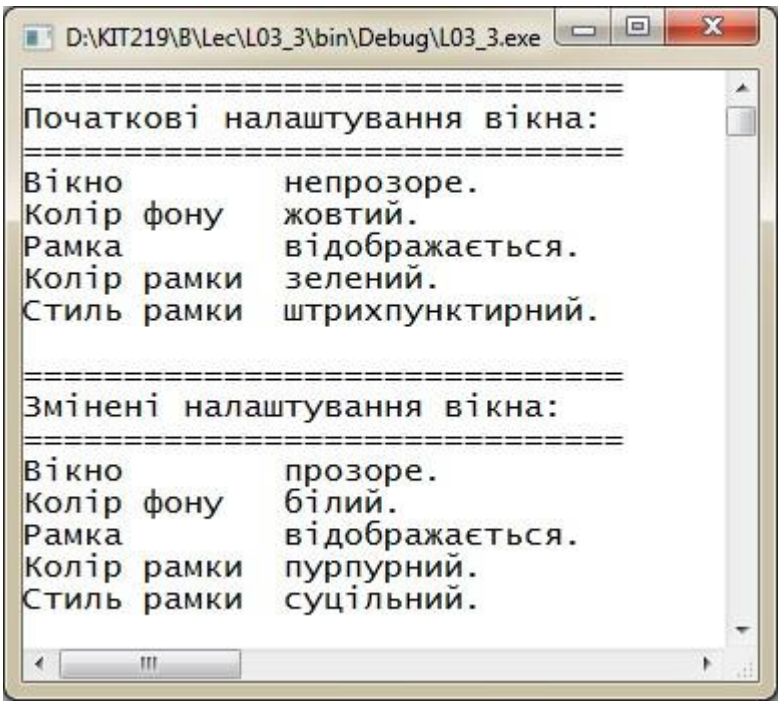


Рисунок 1.17 – Результат виконання програми для демонстрації можливостей роботи з бітовими полями

Відзначимо декілька моментів. Перш за все, структуру бітових полів можна ініціалізувати з використанням звичайного для структур синтаксису:

```
struct Box_props box = { YES, YELLOW, YES, GREEN, DASHED };
```

Аналогічно можна присвоювати значення елементам бітових полів:

```
box.fill_color = WHITE;
```

Крім того, член бітового поля може служити виразом в операторі **switch**. Він навіть може виступати в якості індексу масиву:

```
printf("Колір фону %s.\n", colors[pb->fill_color]);
```

Зверніть увагу, що масив **colors** був визначений так, щоб кожне значення індексу відповідало б рядковому представленню назви кольору, яка має значення індексу та збігається з числовим значенням кольору. Наприклад, індекс **1** відповідає рядку "червоний" і константа **RED** має значення **1**. Директиви **#define** застосовуються для створення символічних констант, які являють собою можливі значення членів. Зверніть увагу, що основні кольори представлені увімкненням єдиного біта. Решта кольорів можуть позначатися комбінаціями основних кольорів. Наприклад, пурпурний колір створюється увімкненням бітів синього та червоного кольорів, тому його можна записати як комбінацію **BLUE | RED**.

1.4. Маніпулювання бітами

Мова **C** дозволяє керувати індивідуальними бітами значення змінної. Може виникнути питання: для чого це потрібно? Прикладом може служити керування деяким фізичним пристроєм, що часто пов’язано з передачею декількох бітів, причому кожен з них має певний сенс. Крім того, інформація про файли в операційній системі зазвичай зберігається у вигляді певних бітів, що вказують на окремі елементи. Багато операцій стиснення та шифрування даних пов’язані з керуванням бітами. Мови високого рівня, як правило, не забезпечують такого рівня деталізації. Здатність поєднувати можливості мови високого рівня з операціями на рівні, який зазвичай залишається за мовою асемблера, робить **C** кращим вибором для написання драйверів пристроїв та вбудованого коду.

Ви вже знаєте, що являють собою біти та байти. Тепер давайте подивимося, що на мові **C** можна з ними робити. Існують два засоби, які допомагають маніпулювати бітами. Перший – це набір з шести побітових операцій, які впливають на біти. Другий – це форма полів даних, яка надає доступ до бітів всередині значення `int`.

1.4.1. Побітові операції

Мова С пропонує два види побітових операцій: **логічні операції** та **операції зсуву**. У наступних прикладах будемо записувати значення у двійковій системі, щоб ви могли бачити, що саме відбувається з бітами. Насправді в програмі ви будете застосовувати цілочислові змінні або константи в звичайних формах. Наприклад, замість `00011001` буде використовуватися запис `25`, `031` або `0x19`. У розглянутих прикладах ми будемо застосовувати `8`-ми бітові числа з нумерацією бітів зліва направо від `0` до `7`.

Побітові логічні операції. Чотири логічних побітових операції працюють з цілочисловими даними, включаючи тип `char`. Вони називаються побітовими тому, що виконуються над кожним бітом окремо незалежно від біта, що знаходиться ліворуч або праворуч. Не плутайте їх зі звичайними логічними операціями (`&&`, `||` і `!`), які мають справу з безпосередніми значеннями змінних.

Доповнення до одиниці або побітове заперечення `(~)`. Унарна операція (`~`) перетворює кожну одиницю на нуль, а кожен нуль на одиницю, як показано в наступному прикладі:

```
~(10011010)          // вираз
 (01100101)          // результат
```

Припустимо, що змінній `val` типу `unsigned char` присвоєно значення `2`. У двійковому вигляді `2` має вигляд `00000010`. Тоді `~val` буде мати значення `11111101`, або `253` у десятковій системі числення. Зверніть увагу, що операція не змінює значення змінної `val`, так само як не змінює значення `val` вираз `3*val`. Значенням `val` як і до того є `2`, але при цьому створюється нове значення, яке можна використовувати або присвоювати десь в іншому місці:

```
new_val = ~val;
printf("%d", ~val);
```

Якщо ви бажаєте змінити значення **val** на **~val**, застосовуйте наступний простий оператор присвоювання:

```
val = ~val;
```

Побітова операція "І" (&). Двійкова операція (**&**) створює нове значення за рахунок виконання побітового порівняння двох операндів. Для кожної позиції підсумковий біт буде дорівнювати **1**, тільки якщо обидва відповідні біти в операндах дорівнюють **1**. Таким чином, в результаті обчислення виразу

```
(10010011) & (00111101)          // вираз
```

отримаємо наступне значення:

```
(00010001)                       // результат
```

Як можна побачити, тільки нульовий і четвертий біти дорівнюють **1** в обох операндах. В **C** також є операція **"І"**, що об'єднана з присвоюванням: (**&=**).

Оператор

```
val &= 0377;
```

дає такий самий результат, як і наступний оператор:

```
val = val & 0377;
```

Побітова операція "АБО" (¦). Двійкова операція (**¦**) створює нове значення за рахунок виконання побітового порівняння двох операндів. Для кожної позиції біт буде дорівнювати **1**, якщо будь-який з відповідних бітів в операндах дорівнює **1**.

Таким чином, в результаті обчислення виразу

```
(10010011) ¦ (00111101)          // вираз
```

отримаємо наступне значення:

```
(10111111)                       // результат
```

Як можна побачити, біти в усіх позиціях крім **6** мають значення **1** в першому або в другому операнді (або в обох). В **C** також існує операція **"АБО"**,

що об'єднана з присвоюванням (`|=`). Оператор `val |= 0377;` дасть такий самий результат, як і наступний оператор:

```
val = val | 0377;
```

Побітова операція "виключне АБО" (^). Двійкова операція (`^`) виконує побітове порівняння двох операндів. Для кожної позиції підсумковий біт буде дорівнювати `1`, якщо один або інший (але не обидва) з відповідних бітів в операндах дорівнює `1`. Таким чином, в результаті обчислення виразу

```
(10010011) ^ (00111101)          // вираз
```

отримаємо наступне значення:

```
(10101110)                       // результат
```

Зверніть увагу, що оскільки нульовий біт дорівнює `1` в обох операндах, підсумковий нульовий біт отримує значення `0`.

У мові `C` також існує операція **"виключне АБО"**, що об'єднана з присвоюванням (`^=`). Оператор `val ^= 0377;` дає такий самий результат, що й наступний оператор:

```
val = val ^ 0377;
```

Використання побітових логічних операцій: маски. Побітова операція **"I"** часто використовується з маскою.

Маска – це комбінація бітів, в якій деякі біти дорівнюють `1`, а деякі `0`. Щоб зрозуміти, чому вона має таку назву, давайте поглянемо, що відбувається, коли ми об'єднуємо будь-яку величину та маску з використанням операції (`&`). Для прикладу припустимо, що ви визначили символічну константу `MASK` як `2` (тобто `00000010`), в якій ненульовим є тільки біт з номером `1`. Тоді оператор `flags = flags & MASK;` призведе до встановлення усіх бітів `flags` (крім першого) в `0`, оскільки будь-який біт, що об'єднується з `0` за допомогою операції **"I"**, дає `0`. Біт номер `1` змінної залишається незмінним. Такий процес називається **"використанням маски"**, оскільки нулі в масці приховують відповідні біти у змінній `flags`.

Поширюючи цю аналогію, біти з `0` в масці можна вважати непрозорими, а біти с `1` – прозорими. Вираз `flags & MASK` схожий на накривання маскою комбінації бітів `flags`. Видимими з-під маски будуть тільки ті біти, яким в `MASK` відповідають біти з `1` (рис. 1.18).

Для скорочення коду можна застосувати операцію `"I"`, яка об'єднана з присвоюванням:

```
flags &= MASK;
```

Нижче наведено один з поширених випадків використання цієї операції:

```
ch &= 0xff;              // або ch &= 0377;
```

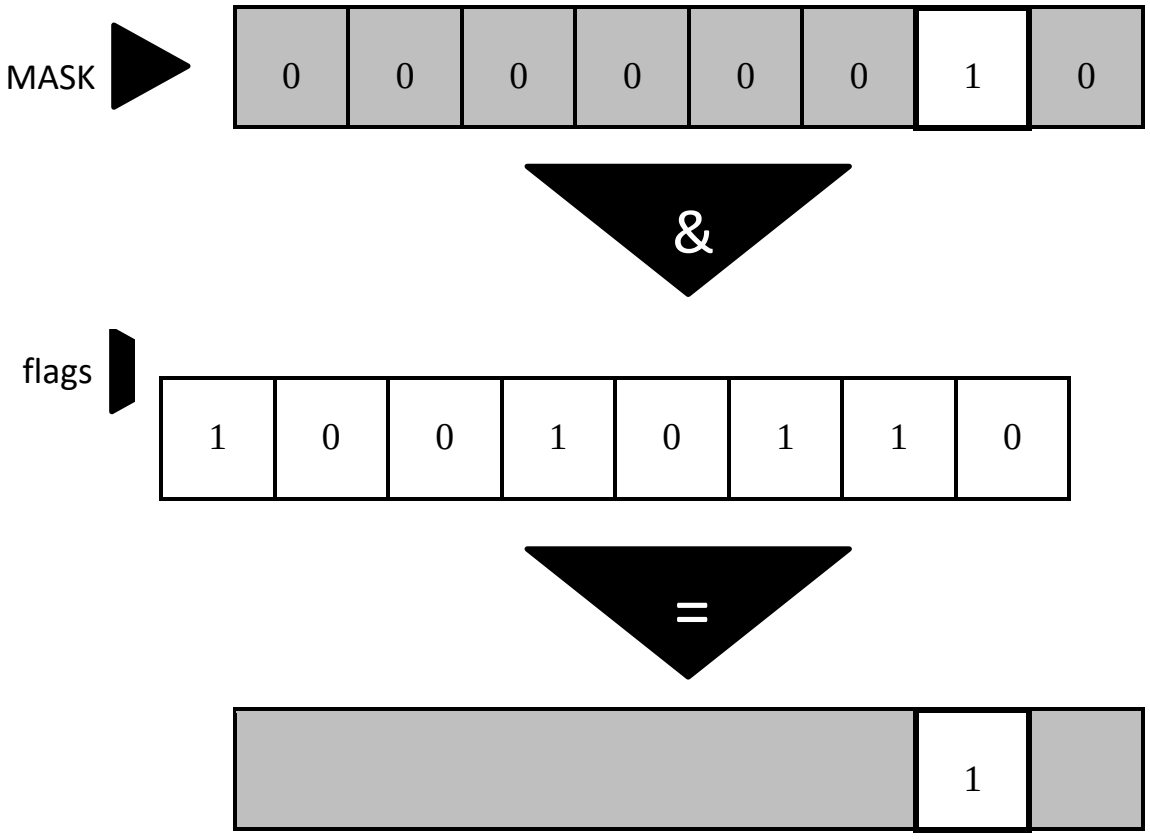


Рисунок 1.18 – Наочне представлення маски

Згадайте, що значення `0xff` записується як `11111111` у двійковому або як `0377` у вісімковому вигляді. Ця маска залишає останні вісім бітів в `ch` без змін, а

інші встановлює в 0. Незалежно від того, скільки бітів містить змінна ch – 8, 16 або більше, – фінальне значення усікається до величини, яка вміщується в один 8-ми бітовий байт. В даному випадку маска має ширину 8 бітів.

Встановлення бітів в одиничні значення. Іноді треба встановити окремі біти в одиничні значення, залишивши інші без змін. Наприклад, комп'ютер **IBM PC** керує обладнанням, відправляючи потрібні значення в порти. Для активізації, скажімо, динаміка, необхідно встановити в одиницю біт 1, а інші біти залишити незмінними. Це можна зробити за допомогою побітової операції **"АБО"**.

Наприклад, нехай існує константа MASK, в якій біт 1 встановлений в 1. Тоді оператор `flags = flags | MASK;` встановлює в одиницю біт номер 1 у змінній flags і залишає усі інші біти без змін. Це пояснюється тим, що будь-який біт, який об'єднаний з 0 за допомогою операції |, залишається без змін, а той, що об'єднується з 1 з використанням |, дорівнює 1. Наприклад, нехай flags дорівнює 00001111 і MASK – 10110110. Вираз `flags | MASK` стає

```
(00001111) | (10110110)        // вираз
```

і після обчислення дає наступний результат:

```
(10111111)                     // результат
```

Усі біти, що встановлені в 1 всередині MASK, також будуть встановлені у підсумку в 1. Усі біти у flags, які відповідають бітам 0 в MASK, залишаються незмінними. Для стислості можна використовувати побітову операцію **"АБО"**, що об'єднана з присвоюванням:

```
flags |= MASK;
```

Цей оператор встановить в 1 ті біти flags, яким відповідають одиничні біти в MASK, залишивши інші біти без змін.

Встановлення бітів в нульові значення. Дуже зручно мати можливість встановлювати окремі біти в нульові значення, тобто очищувати їх. Припустимо, що треба встановити в 0 біт номер 1 у змінній flags. Знову будемо

використовувати MASK, що має встановлений в одиницю тільки біт 1. Можна скористатися наступним оператором:

```
flags = flags & ~MASK;
```

Оскільки в MASK усі біти крім біта 1 дорівнюють нулю, вираз ~MASK дає значення, в якому усі біти крім біта 1 дорівнюють одиниці. Об'єднання 1 з будьяким бітом, використовуючи операцію (&), залишає цей біт без змін, тому оператор залишає усі біти крім біта 1 незмінними. Об'єднання 0 з будь-яким бітом за допомогою операції (&) дає 0 незалежно від значення цього біта.

Наприклад, нехай flags дорівнює 00001111 і MASK – 10110110. Вираз `flags & ~MASK` стає

```
(00001111) & ~(10110110)          // вираз
```

і після обчислення дає наступний результат:

```
(00001001)                        // результат
```

Усі біти, які встановлені в 1 всередині MASK, будуть встановлені у підсумку в 0. Усі біти у flags, які відповідають бітам 0 в MASK, залишаються незмінними.

Нижче представлена скорочена форма:

```
flags &= ~MASK;
```

Перемикання бітів. Перемикання біта означає зміну його значення на протилежне. Для перемикання бітів можна застосовувати побітову операцію **"виключного АБО"** (^). Ідея полягає в тому, що якщо b – це встановлений стан біту (1 або 0), то 1 ^ b дорівнює 0, коли b дорівнює 1, і 1, коли b дорівнює 0. Крім того, вираз 0 ^ b дає b незалежно від значення b. Відповідно, в результаті об'єднання значення та маски з використанням операції (^) біти, що відповідають 1 в масці, перемикаються, а біти, що відповідають 0 в масці, залишаться незмінними. Щоб перемикнути біт 1 змінної flags, можна виконати одну з наступних дій:

```
flags = flags ^ MASK;
flags ^= MASK;
```

Наприклад, нехай **flags** дорівнює **00001111** і **MASK** – **10110110**. Вираз

```
flags ^ MASK
```

стає

```
(00001111) ^ (10110110)        // вираз
```

і після обчислення дає наступний результат:

```
(10111001)                     // результат
```

Усі біти, що були встановлені в **1** всередині **MASK**, призводять до перемикання відповідних бітів у **flags**. Усі біти у **flags**, які відповідають бітам **0** в **MASK**, залишаються незмінними.

Перевірка значення біта. Ви вже бачили, як змінювати значення бітів. Припустимо, що замість цього треба перевірити значення будь-якого біта. Наприклад, чи встановлений в **1** біт **1** у **flags**? Просте порівняння **flags** і **MASK** тут не підійде:

```
if(flags == MASK)
puts("Збігається!");          // не працює
```

Навіть якщо біт **1** змінної **flags** встановлений в **1**, значення будь-якого іншого біта у **flags** може зробити результат порівняння недійсним. Щоб виконати порівняння тільки біта **1** у **flags** з **MASK**, необхідно спочатку замаскувати інші біти **flags**:

```
if( (flags & MASK) == MASK)
puts("Збігається!");
```

Побітові операції мають пріоритет нижче, ніж у операції (**==**), тому вираз **flags & MASK** повинен бути обмежений круглими дужками.

Для запобігання неповного охоплення інформації, бітова маска повинна мати ширину не менш, ніж у значення, що маскується.

1.4.2. Побітові операції зсуву

Тепер давайте поглянемо на операції зсуву в мові **C**. Побітові операції зсуву зсувають біти ліворуч або праворуч. Для більшої наочності будемо застосовувати двійковий запис чисел.

Операція "зсув вліво" (**<<**) зсуває біти значення лівого операнда вліво на кількість позицій, яка задається правим операндом. Позиції, що звільняються, заповнюються **0**, а біти, що виходять за межі значення лівого операнда, втрачаються. В наступному прикладі кожен біт зсувається на дві позиції вліво:

```
(10001010) << 2          // вираз
(00101000)               // результат
```

Ця операція отримує нове бітове значення, але не змінює операнди. Для прикладу припустимо, що змінна **stonk** має значення **1**. Вираз **stonk << 2** дає **4**, але значенням **stonk** як і раніше є **1**. Щоб змінити значення змінної, можна скористатися операцією зсуву вліво з присвоюванням (**<<=**). Ця операція зсуває біти змінної вліво на кількість позицій, яка вказана в правому операнді.

Приклад використання операції **"зсув вліво"**:

```
int stonk = 1;
int on;
on = stonk << 2;         // присвоює 4 змінній on stonk <<= 2;
// змінює значення stonk на 4
```

Операція **"зсув вправо"** (**>>**) робить зсув бітів значення лівого операнда вправо на кількість позицій, яка вказується в правому операнді. Біти, які виходять за праву межу лівого операнда, втрачаються. Для типів даних без знаку, позиції, які звільняються зліва, заповнюються **0**. Для типів даних зі знаком, результат залежить від системи. Позиції, що звільняються, можуть заповнюватися **0** або бітом знаку (самого лівого):

```
(10001010) >> 2          // вираз, значення зі знаком
(00100010)               // результат в одних системах
(10001010) >> 2          // вираз, значення зі знаком
(11100010)               // результат в інших системах
```

Для значення без знаку результат буде наступним:

```
(10001010) >> 2          // вираз, значення без знаку
(00100010)               // результат в усіх системах
```

Кожен біт переміщується на дві позиції вправо, а позиції, що звільняються, заповнюються **0**.

Операція зсуву вправо з присвоюванням (**>>=**) робить зсув вправо бітів лівого операнда на задану в правому операнді кількість позицій, наприклад:

```
int sweet = 16;
int sw;

sw = sweet >> 3;         // sw дорівнює 2, sweet як і раніше
16 sweet >>= 3;             // значення sweet змінилося на 2
```

Можливості побітових операцій зсуву. Побітові операції зсуву є зручним та ефективним (в залежності від обладнання) засобом виконання множення та ділення на степінь **2**:

```
number << n              // Помножує number на 2 в степені n
number >> n              // Ділить number на 2 в степені n,
// якщо значення number □ 0
```

Ці операції зсуву аналогічні до операції зміщення десяткової крапки під час множення або ділення на **10**.

Операції зсуву можуть також використовуватися для отримання груп бітів з більш великих конструкцій. Припустимо, що для представлення значений кольору застосовується змінна типу **unsigned long**, причому молодший байт містить інтенсивність червоної складової, наступний байт – інтенсивність зеленої складової, а третій байт – інтенсивність синьої складової кольору. Нехай необхідно зберегти інтенсивність кожної складової у змінній типу **unsigned char**. Для цього можна написати такий код:

```
#define BYTE_MASK 0xff

unsigned long color = 0x002a162f;
unsigned char blue, green, red;

red   =  color        & BYTE_MASK;
green = (color >>  8) & BYTE_MASK;
blue  = (color >> 16} & BYTE_MASK;
```

В розглянутому коді за допомогою операції **"зсуву вправо"** 8-ми бітове значення складової кольору переміщується до молодшого байту. Потім за допомогою маски значення молодшого байта присвоюється відповідній змінній.

Практичне застосування побітових операцій зсуву. Розглянемо програму для перетворення десяткових чисел у двійкове представлення за допомогою побітових операцій. Програма читає ціле число, яке вводиться з клавіатури, і передає його разом з адресою рядка до функції на ім'я **itobs()**. Функція **itobs()** формує для цілочислового значення рядок з двійковим представленням. Для визначення потрібної комбінації **0** і **1**, яка буде поміщатися в рядок, ця функція використовує побітові операції.

```
#include <stdio.h>
#include <windows.h>
#include <limits.h>  // для CHAR_BIT кількість бітів на символ

char *itobs(int, char *);
void show_bstr(const char *);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;

    SetConsoleOutputCP(1251);

    puts("Введіть ціле число.");
    puts("Якщо ввести не число, програма завершиться.\n");
    while(scanf("%d", &number) == 1)
    {
       itobs(number, bin_str);
       printf("Двійкове представлення: ");
       show_bstr(bin_str);
       putchar('\n');
    }
    puts("Програма завершена.");
    return 0;
}
char *itobs(int n, char *ps)
{     int  i;
      const static int size = CHAR_BIT * sizeof(int);
```

```
for(i = size - 1; i >= 0; i--, n >>= 1)
    // передбачається кодування ASCII або схоже
    ps[i] = (01 & n) + '0';
ps[size] = '\0';
return ps;
}

// відображення двійкового рядка блоками по 4
void show_bstr(const char *str)
{
    int i = 0;
    while(str[i])    // поки не буде отриманий нульовий символ
    {
        putchar(str[i]);
        if( ++i % 4 == 0 && str[i])
            putchar(' ');
    }
    printf("\n");
}
```

Результат виконання програми наведено на рис. 1.19.

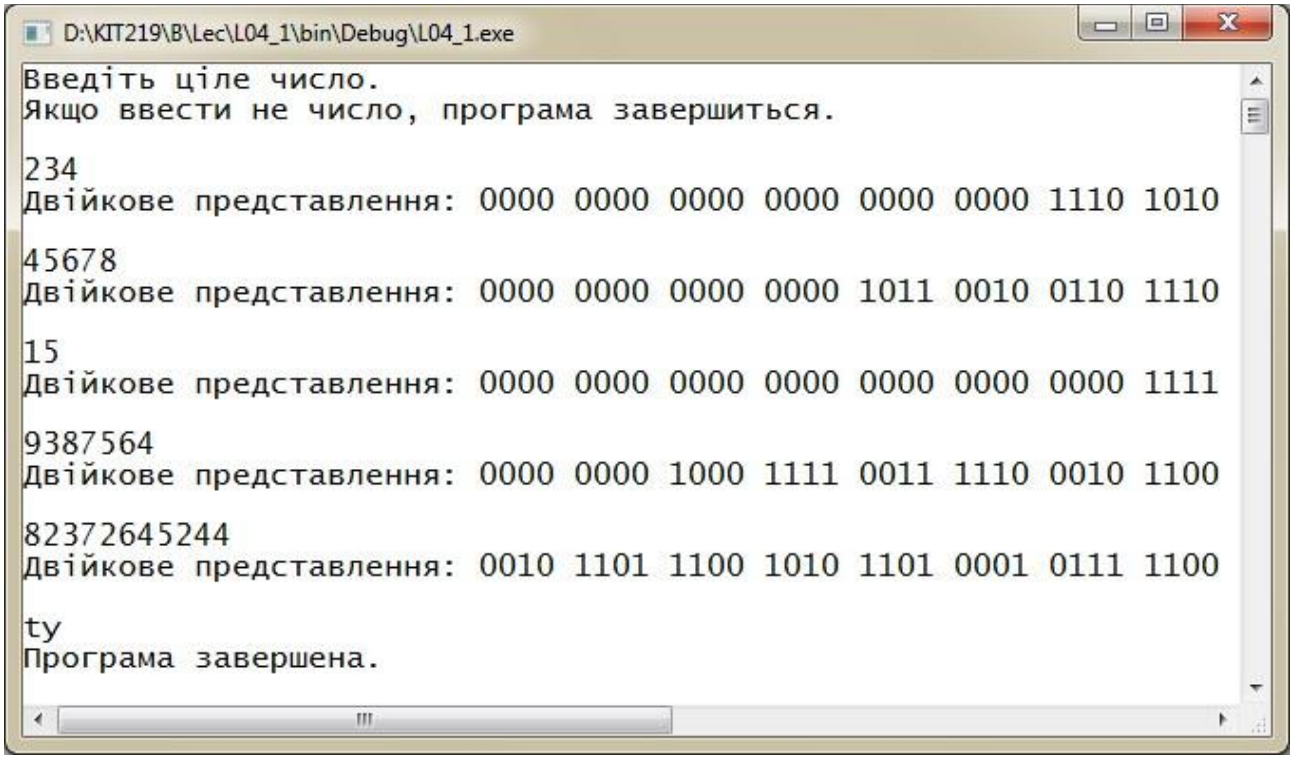


Рисунок 1.19 – Результат роботи програми перетворення чисел з десяткової на двійкову систему числення

У програмі застосовується макрос `CHAR_BIT` з файлу заголовку `limits.h`. Цей макрос представляє собою кількість бітів у типі `char`. Операція `sizeof` повертає розмір в термінах `char`, тому вираз `CHAR_BIT * sizeof(int)` дає кількість бітів у значенні `int`. Масив `bin_str` містить на один елемент більше цієї величини, щоб можна було додати до нього завершальний нульовий символ.

Функція `itobs()` повертає ту ж саму адресу, яка їй була передана, тому її виклик можна використовувати, наприклад, в якості аргументу функції `printf()`. На першій ітерації циклу `for` функція обчислює вираз `01 & n`. Операнд `01` – це вісімкове представлення маски, у якої усі біти крім нульового встановлені в `0`. Відповідно, результатом `01 & n` буде значення останнього біта в `n`. Значенням є `0` або `1`, але для масиву потрібен символ `'0'` або символ `'1'`. Перетворення відбувається додаванням коду для `'0'`. Результат поміщається в передостанній елемент масиву. Останній елемент зарезервований для нульового символу.

Замість виразу `01 & n` можна застосувати і `1 & n`. Використання вісімкового значення `1` замість десяткового виглядає більш стильним. З цієї точки зору варіант `0x1 & n`, є навіть кращим.

Потім в циклі виконуються оператори `(i--)` і `(n >>= 1)`. Перший оператор здійснює перехід до попереднього елементу масиву, а другий – зсуває біти в змінній `n` на одну позицію вправо. На наступній ітерації циклу код знайде значення нового самого правого біту. Після цього відповідний йому символ цифри поміщається в елемент, що передує останній цифрі. В подібній манері функція заповнює масив справа наліво.

Для відображення підсумкового рядка можна застосовувати `printf()` або `puts()`. Проте, у програмі визначена функція `show_bstr()`, яка поділяє послідовність бітів на групи по чотири, щоб полегшити сприйняття рядка.

Давайте розглянемо ще один приклад. На цей раз мета полягає в тому, щоб написати функцію, яка інвертує останні `n` бітів у значенні, приймаючи в якості аргументів `n` і саме значення.

Операція (~) інвертує біти, але робить це з усіма бітами в байті, а не тільки з обраними. Однак, як ви вже бачили, для перемикання окремих бітів можна використовувати операцію **"виключне АБО" (^)**. Припустимо, що створена маска, в якій останні **n** бітів встановлені в **1**, а інші – в **0**. Тоді застосування (^) до цієї маски і до значення, перемикає (інвертує) останні **n** бітів, залишаючи інші біти незмінними. Такий підхід реалізований в наступному фрагменті коду:

```
int invert_end(int num, int bits)
{
int mask   = 0;
int bitval = 1;
     while(bits-- > 0)
     {
       mask |= bitval;
       bitval <<= 1;
     }
     return num ^ mask; }
```

Маска створюється в циклі **while**. Спочатку в **mask** усі біти встановлені в **0**. На першій ітерації циклу біт **0** встановлюється в **1**, після чого значення **bitval** збільшується до **2**, тобто в ньому біт **0** встановлюється в **0**, а біт **1** – в **1**. На наступній ітерації біт **1** в **mask** встановлюється в **1** і т. д. У підсумку, операція **num ^ mask** дає бажаний результат.

Для тестування функції її можна вбудувати в попередню програму. Текст модифікованої програми має наступний вигляд:

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

char *itobs(int, char *); void
show_bstr(const char *); int
invert_end(int num, int bits);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;
    SetConsoleOutputCP(1251);
puts("Введіть ціле число.");
```

```
  puts("Якщо ввести не число, програма завершиться.\n");
      while(scanf("%d", &number) == 1)
      {
         itobs(number, bin_str);
         show_bstr(bin_str);
         printf("   Двійкове представлення");
         putchar('\n');
         number = invert_end(number, 4);
         show_bstr(itobs(number,bin_str));
         printf("   Інвертування останніх 4 бітів\n");
         putchar('\n');
      }
      puts("Програма завершена.");
      return 0; }
   char *itobs(int n, char *ps)
{     int i;
    const static int size = CHAR_BIT * sizeof(int);
    for(i = size - 1; i >= 0; i--, n >>= 1)
         ps[i] = (01 & n) + '0';
    ps[size] = '\0';
    return ps;
}
// відображення двійкового рядка блоками по 4
void show_bstr(const char *str)
{
     int i = 0;
     while(str[i])  // поки не буде отриманий нульовий символ
    {
        putchar(str[i]);
        if(++i % 4 == 0 && str[i])
          putchar(' ');
    }
}
 int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;
     while(bits-- > 0)
    {
        mask  ¦= bitval;
        bitval <<= 1;
    }
    return num ^ mask; }
```

Результат виконання програми наведено на рис. 1.20.

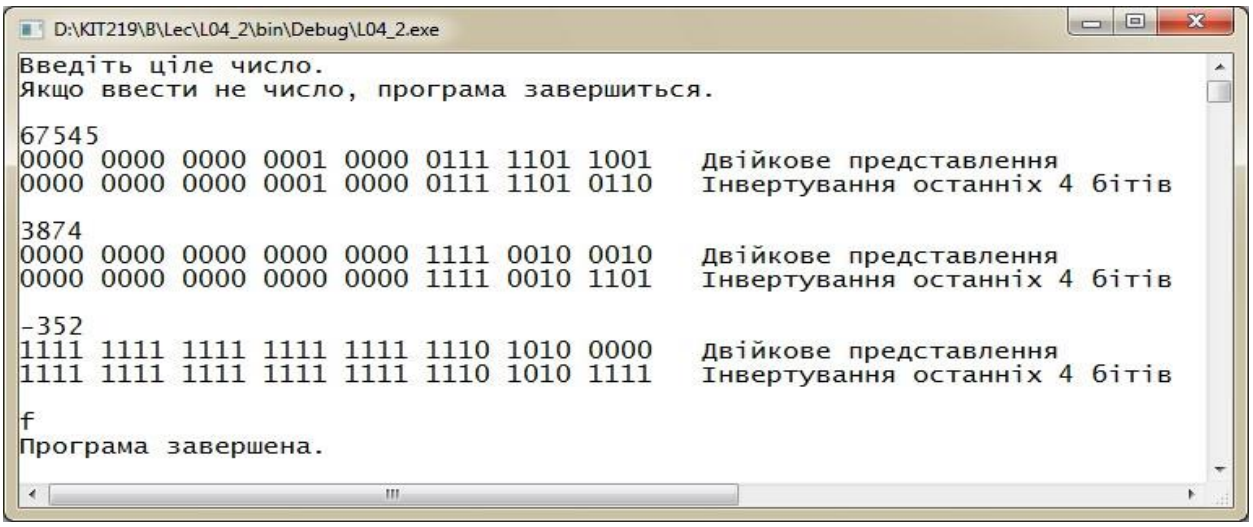


Рисунок 1.20 – Результат роботи програми перетворення чисел з десяткової на двійкову систему числення з інвертуванням останніх чотирьох цифр

1.4.3. Бітові поля та побітові операції

Бітові поля та побітові операції – це два альтернативні підходи до рішення задачі програмування одного й того ж самого типу. Це означає, що часто можна застосовувати будь-який з цих підходів. На попередньому розділі в одному з прикладів для зберігання інформації про графічне вікно використовувалася структура з розміром, як у типі **unsigned int**. Ту ж саму інформацію можна було б зберегти у змінній типу **unsigned int**. Потім замість синтаксису членства в структурі можна було б застосовувати побітові операції. Зазвичай така методика є не дуже зручною. Розглянемо приклад, в якому задіяні обидва підходи.

В якості засобу комбінування підходу на основі структури та підходу на базі побітових операцій можна скористатися об’єднанням. Виходячи з існуючого оголошення типу **struct** box_props, можна оголосити наступне об’єднання:

```
union Views
{
    struct Box_props st_view;
    unsigned short us_view;
};
```

В системі змінна **unsigned short** і структура **Box_props** займають у пам’яті **16** бітів. Об’єднання дозволяє застосовувати член **st_view**, щоб

трактувати відведену пам’ять як структуру, або використовувати член **us_view**, щоб розглядати той самий блок пам’яті як значення **unsigned short**. Які бітові поля структури відповідають окремим бітам змінної типу **unsigned short**? Це залежить від реалізації та обладнання. В наступному прикладі припускається, що структури завантажені в пам’ять, починаючи з молодших бітів і закінчуючи старшими бітами байта. Іншими словами, перше бітове поле в структурі

відповідає біту **0** слова.

В наступній програмі об’єднання **Views** застосовується для порівняння підходів на основі бітових полів і побітових операцій. В ній **box** – це об’єднання **Views**, тому **box.st_view** представляє собою структуру **Box_props**, що використовує бітові поля, а **box.us_view** – ті ж самі дані, але представлені як значення **unsigned short**.

Як ви вже знаєте, об’єднання може мати ініціалізований перший член, тому встановлені значення відповідають представленню структури. Програма відображає властивості вікна за допомогою функції, яка основана на представленні структури, і також за допомогою функції, яка основана на представленні **unsigned short**. Будь-який з підходів забезпечує доступ до даних, але по-різному. Додатково у програмі застосовується визначена раніше функція **itobs()**, яка дозволяє відобразити дані у вигляді рядка двійкових цифр, щоб можна було бачити, які біти встановлені в одиницю, а які в нулі.

```
#include <stdio.h>
#include <windows.h>
#include <stdbool.h>
#include <limits.h>
//=========================================
// КОНСТАНТИ БІТОВИХ ПОЛІВ
//=========================================

// стилі ліній
#define SOLID    0
#define DOTTED   1
#define DASHED   2

 // основні кольори
```

```
#define BLUE    4
#define GREEN   2
#define RED     1

// змішані кольори
#define BLACK   0
#define YELLOW  ( RED   | GREEN )
#define MAGENTA ( RED   | BLUE  )
#define CYAN    ( GREEN | BLUE  )
#define WHITE   ( RED   | GREEN | BLUE )
//=========================================
// ПОБІТОВІ КОНСТАНТИ
//=========================================
#define OPAQUE1        0x1
#define FILL_BLUE      0x8
#define FILL_GREEN     0x4
#define FILL_RED       0x2
#define FILL_MASK      0xE
#define BORDER         0x100
#define BORDER_BLUE    0x800
#define BORDER_GREEN   0x400
#define BORDER_RED     0x200
#define BORDER_MASK    0xE00
#define B_SOLID        0
#define B_DOTTED       0x1000
#define B_DASHED       0x2000
#define STYLE_MASK     0x3000

const char *colors[8] = {
           "чорний", "червоний",  "зелений",   "жовтий",
           "синій",  "пурпурний", "блакитний", "білий"
};

struct Box_props
{
    bool          opaque       : 1;
    unsigned int  fill_color   : 3;
    unsigned int               : 4;
    bool          show_border  : 1;
    unsigned int  border_color : 3;
    unsigned int  border_style : 2;
    unsigned int               : 2;
};
 union Views
{
   // погляд на дані як на struct або як на unsigned short
   struct Box_props st_view;
   unsigned short us_view;
```

```
};

void show_settings(const struct Box_props *pb);
void show_settingsl(unsigned short);
char *itobs(int n, char *ps);
int main(void)
{
    union Views box = {{ true, YELLOW, true, GREEN, DASHED }};
    char bin_str[8 * sizeof(unsigned int) + 1];

SetConsoleOutputCP(1251);
printf("=============================================\n");
printf("Початкові налаштування вікна:\n");
printf("=============================================\n");
show_settings(&box.st_view);

printf("\n=============================================\n");
printf("Налаштування вікна з unsigned short: \n");
printf("=============================================\n");
show_settingsl(box.us_view);
printf("Комбінація бітів  %s\n",
itobs(box.us_view, bin_str));
box.us_view  &=  ~FILL_MASK;      // Очищуємо біти фону
box.us_view  |=  ( FILL_BLUE  |  FILL_GREEN  );  // Колір фону
box.us_view  ^=  OPAQUE1;          // Перемикаємо прозорість
box.us_view  |=  BORDER_RED;     // Встановлюємо колір рамки
box.us_view  &=  ~STYLE_MASK;  // Очищуємо біти стилю рамки
box.us_view  |= B_DOTTED;      // Встановити пунктирний стиль
printf("\n=============================================\n");
printf("Змінені налаштування вікна:\n");
printf("=============================================\n");
show_settings(&box.st_view);
printf("\n=============================================\n");
printf("Налаштування вікна з unsigned short: \n");
printf("=============================================\n");
show_settingsl(box.us_view);
printf("Комбінація бітів  %s\n",
       itobs(box.us_view, bin_str));
return 0;
}

void show_settings(const struct Box_props *pb)
{
    printf("Вікно              %s\n",
           pb->opaque == true ? "непрозоре" : "прозоре");
printf("Колір фону         %s\n", colors[pb->fill_color]);
printf("Рамка              %s\n", pb->show_border == true ?
       "відображається" : "не відображається");
printf("Колір рамки        %s\n", colors[pb->border_color]);
printf("Стиль рамки ");      switch(pb->border_style)
```

```
    {
case  SOLID   : printf("         суцільний\n");          break;
case  DOTTED  : printf("         пунктирний\n");         break;
case  DASHED  : printf("           штрихпунктирний\n");  break;
default       : printf("       невідомого типу\n"); break;
    }
}
 void show_settingsl(unsigned short us)

{
printf("Вікно              %s\n",
        (us & OPAQUE1) == OPAQUE1 ? "непрозоре" : "прозоре");
printf("Колір фону         %s\n", colors[(us >> 1) & 07]);
printf("Рамка              %s\n", (us & BORDER) == BORDER ?
          "відображається" : "не відображається");
printf("Колір рамки        %s\n", colors[(us >> 9) & 07]);
printf("Стиль рамки ");
switch(us & STYLE_MASK)
    {
case  B_SOLID   : printf("      суцільний\n");          break;
case  B_DOTTED  : printf("      пунктирний\n");         break;
case  B_DASHED  : printf("        штрихпунктирний\n");  break;
default         : printf("    невідомого типу\n"); break;
    }
}
 char *itobs(int n, char *ps)
{
int i;
    const static int size = CHAR_BIT * sizeof(unsigned short);
     for(i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
        ps[size] =  '\0';
return ps;
}
```

Відповідність між бітовими полями та позиціями бітів залежить від реалізації. В нашому випадку вони реалізовані по-різному.

Результат виконання програми наведено на рис. 1.21.

1.4.4. Засоби вирівнювання (C11)

Засоби вирівнювання **C11** за своєю природою більш орієнтовані на маніпулювання байтами, ніж бітами, але вони також відображають можливість

мови **C** мати справу з обладнанням. В цьому контексті вирівнювання відноситься до того, як об’єкти розташовуються в пам’яті.

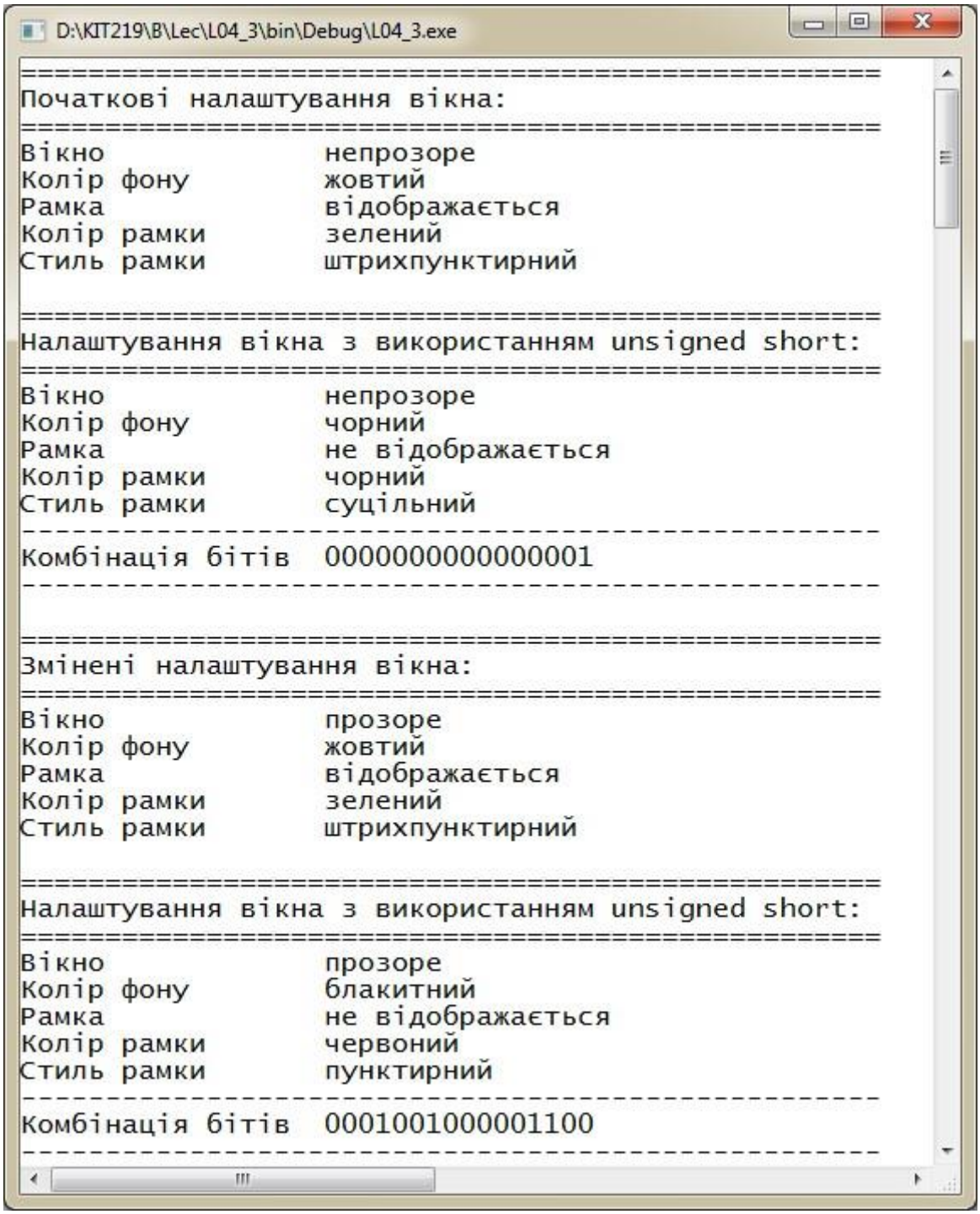


Рисунок 1.21 – Результат роботи програми, в якій порівнюються два підходи на основі бітових полів і побітових операцій

Наприклад, для максимальної ефективності система може вимагати, щоб значення типу `double` зберігалося в пам'яті за адресою, яка кратна `4`, але дозволити значенню типу `char` зберігається за будь-якою адресою. Більшості програмістів рідко коли доведеться турбуватися про вирівнювання. Але в деяких ситуаціях контроль над вирівнюванням дозволяє отримати вигоду, наприклад, при передачі даних з одного фізичного місця в інше або при виклику інструкцій, які оперують на множині елементів даних одночасно.

Операція `_Alignof` висуває вимоги до вирівнювання вказаного типу. Для її використання необхідно після ключового слова `_Alignof` помістити ім'я типу в круглих дужках:

```
size_t d_align = _Alignof(float);
```

Отримане значення, скажімо, `4` для `d_align`, говорить про те, що об'єкти `float` мають вимогу до вирівнювання, що відповідає значенню `4`. Це означає, що `4` є кількістю байтів між адресами для зберігання значень згаданого типу, які йдуть один за одним. В загальному випадку значення вирівнювання повинні бути цілими числами `0`, які представляють собою степінь `2`. Більш високі значення вирівнювання вважаються більш жорсткими або більш строгими, ніж менші значення, в той час як менші значення трактуються як більш слабкі.

За допомогою специфікатора `_Alignas` можна робити запит про конкретне вирівнювання для змінної або типу. Однак ви не повинні робити запит на вирівнювання, яке слабше фундаментального вирівнювання, прийнятого для типу. Наприклад, якщо вимога до вирівнювання для `float` складає `4`, не робіть запит на значення вирівнювання, що дорівнює `1` або `2`. Цей специфікатор застосовується як частина оголошення, і за ним йде пара круглих дужок, яка містить або значення вирівнювання, або тип:

```
_Alignas(double) char c1;
_Alignas(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long
double)];
```

В наступній програмі наведено приклад використання **_Alignas** і **_Alignof**. Текст програми має наступний вигляд:

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

int main(void)
{
double  dx;
char    ca;
char    cx;
double  dz;
char    cb;
char    _Alignas(double) cz;

  SetConsoleOutputCP(1251);

  printf("Вирівнювання char:   %d\n", _Alignof(char));
  printf("Вирівнювання double: %d\n\n",
  _Alignof(double));     printf("&dx: %p\n", &dx);
  printf("&ca: %p\n", &ca);
  printf("&cx: %p\n", &cx);
  printf("&dz: %p\n", &dz);
  printf("&cb: %p\n", &cb);
  printf("&cz: %p\n", &cz);
  return 0;
}
```

Результат виконання програми наведено на рис. 1.22.

Рисунок 1.22 – Результат роботи програми для демонстрації використання ключових слів **_Alignas** і **_Alignof**

В системі значення вирівнювання `8` для типу `double` припускає, що значення цього типу зберігаються за адресами, які кратні `8`. Шістнадцяткові адреси, що закінчуються на `0` або `8`, є кратними `8`, і адреси такого виду застосовувалися для двох змінних `double`, а також змінної `char` на ім'я `cz`, якій було призначено значення вирівнювання для типу `double`. Оскільки значенням вирівнювання для `char` було `1`, компілятор міг використовувати для змінних цього типу будь-які адреси.

Включення файлу `stdalign.h` дозволяє застосовувати псевдоніми `alignas` і `alignof` для `_Alignas` і `_Alignof`. Вони відповідають ключовим словам в `C++`. В `C11` також з'явилася можливість вирівнювання для виділеної пам'яті за рахунок додавання до бібліотеки `stdlib.h` нової функції розподілу пам'яті з наступним прототипом:

```
void *aligned_alloc(size_t alignment, size_t size);
```

В першому параметрі вказується відповідне вирівнювання, а в другому – кількість необхідних байтів, яка повинна бути кратною значенню першого параметра. Як і у випадку інших функцій розподілу пам'яті, по завершенні роботи з виділеною пам'яттю використовуйте функцію `free()`, щоб її звільнити.

Контрольні запитання та завдання

1. Що таке структура?
2. З чого складається формат (шаблон, схема) структури?
3. Як визначити змінну типу «структура»?
4. Як ініціалізувати структуру?
5. Які існують способи доступу до членів структури?
6. Як визначити масив структур?
7. Що означає поняття «вкладені структури»?
8. Як оголосити та ініціалізувати вказівник на структуру?
9. Як здійснити доступ до членів структури за допомогою вказівника?

10. Чи можна присвоювати структури?

11. Що таке об’єднання?

12. З чого складається шаблон об’єднання?

13. Як визначити змінну типу «об’єднання»?

14. Як ініціалізувати об’єднання?

15. Які існують способи доступу до членів об’єднання?

16. Як визначити масив об’єднань?

17. Чи може бути об’єднання членом структури?

18. Що таке перелічення та з яким ключовим словом воно пов’язане?

19. Якого типу повинні бути константи в переліченні?

20. Як визначити бітові поля та для чого вони використовуються?

21. Які два види побітових операцій існує в мові С?

22. Що собою являє операція побітового "І"?

23. Що собою являє операція побітового "АБО"?

24. Що собою являє операція побітового "виключного АБО"?

25. Що таке маска та для чого вона застосовується?

26. Як встановити визначений біт в одиницю або в нуль?

27. Як здійснити перемикання бітів?

28. Як перевірити значення окремого біта?

29. В чому полягає особливість операції побітового зсуву вліво або вправо?

Завдання для самостійного розв’язання

1. Напишіть С-програму, яка вводить відомість успішності групи студентів, які здали зимову сесію з дисциплін «Вища математика», «Фізика» та «Програмування» та оючислити: середній бал кожного студента, середній бал групи з кожної дисципліни. Вивести на екран прізвища відмінників з програмування. Дступ до елементів масиву структур здійснювати з початку за допомогою індексів та використанням вказівників на структуру.

2. Напишіть С-програму, яка обраховує результати екзаменаційної сесії в студентській групі, які містять інформацію про прізвища та ініціали, стать та середній бал успішності студентів із використанням вказівника на масив структур . Вивести повідомлення про те, чий підсумковий середній бал вищий – хлопців або дівчат.

3. Напишіть С-програму, в якій демонструються можливості роботи з об’єднанням, яке повинно мати різні типи даних (short, int, long, float, double, char), а також масив символьного типу розміром 8 байтів. Необхідно створити шаблон об’єднання, присвоїти значення різним його полям та отримати на екрані значення полів різного типу. Крім того необхідно визначити розмір об’єднання за допомогою sizeof та пояснити отримані результати. Під час роботи з кожним типом даних, визначте вміст окремих байтів об’єднання.

4. Напишіть С-програму, яка б продемонструвала можливості роботи з переліченими типами. Для цього сформуйте набір цілочислових констант і виведіть значення кожної з них на екран. Присвойте двом довільним константам будь-які додатні значення та продемонструйте на екрані, як це вплине на інші константи. Присвойте одній константі будь-яке додатне значення, а другій – будь-яке від’ємне значення, та продемонструйте на екрані, як це вплине на інші константи.

5. Напишіть С-програму для роботи з бітовими полями. Присвойте полям різні значення та виведіть їх на екран.

6. Напишіть С-програму, в якій необхідно: - підрахувати кількість одиниць в кожному з операндів a = 0111001011000111 і b = 0000000011000101 типу unsigned short,) та виконати над кожним з операндів одномісну операцію "побітового заперечення"; здійснити над операндами a і b двомісні побітові операції: "І", "АБО" та "виключне АБО"; встановити біти 8-15 операнда a в значення '0'; - встановити біти 7-11 операнда b в значення '1'; в операнді a виділити біти 3-12 та виконати зсув результату вправо таким чином, щоб перший правий виділений біт став у нульову позицію (тобто став нульовим розрядом).

2. КЛАСИ ЗБЕРІГАННЯ ТА КЕРУВАННЯ ПАМ'ЯТТЮ

2.1. Класи зберігання

Однією з переваг мови **C** є можливість керування витонченими аспектами програми. Система керування пам'яттю в **C** є ілюстрацією такого керування, дозволяючи визначати, яким функціям відомі ті або інші змінні та наскільки довго змінна існує в програмі. Використання місця зберігання в пам'яті є ще одним елементом проектного рішення, яке покладається в основу програми.

2.1.1. Класи зберігання даних

Для зберігання даних у пам'яті мова **C** пропонує п'ять різних моделей, або **класів зберігання**. Щоб зрозуміти доступні варіанти, корисно спочатку вивчити декілька концепцій і термінів.

В кожній програмі дані зберігаються в пам'яті. Для цього існує **апаратний аспект** – будь-яке збережене значення знаходиться в фізичній пам'яті. В літературі по **C** для опису такої ділянки пам'яті застосовується термін **об'єкт**. Об'єкт може зберігати одне або декілька значень. В певний момент об'єкт може поки не містити збережене значення, але він буде мати правильний розмір для розміщення потрібного значення.

Існує також і **програмний аспект** – програмі потрібен будь-який спосіб доступу до об'єкту. Цього можна досягти, наприклад, шляхом оголошення змінної:

```
int entity = 3;
```

Показане оголошення призводить до створення ідентифікатора на ім'я **entity**. Ідентифікатор являє собою ім'я, в даному випадку таке, яке може застосовуватися для позначення вмісту окремого об'єкту. Ідентифікатори відповідають домовленостям про іменування змінних, які були розглянуті раніше. В цьому випадку ідентифікатор **entity** відображає спосіб, яким

програмне забезпечення (програма на C) вказує об'єкт, що зберігається в апаратній пам'яті. Таке оголошення також надає значення для зберігання в об'єкті.

Ім'я змінної – не єдиний метод позначення об'єкту. Наприклад, погляньте на наступні оголошення:

```
int *pt = &entity;
int ranks[10];
```

В першому випадку **pt** являє собою ідентифікатор. Він позначає об'єкт, який містить адресу. Вираз ***pt** – не ідентифікатор, оскільки він не є іменем. Проте, він вказує на об'єкт, в даній ситуації – на той самий об'єкт, що й **entity**.

В загальному випадку, вираз, який означає об'єкт, називається **l**-значенням. Таким чином, **entity** – це ідентифікатор, що являє собою **l**-значення, а ***pt** – вираз, що є **l**-значенням. При тих самих оголошеннях вираз **ranks+2*entity** – не ідентифікатор (не ім'я) і не **l**-значення (не вказує на вміст комірки пам'яті). Але вираз ***(ranks+2*entity)** є **l**-значенням, тому що воно вказує на значення певної комірки пам'яті (сьомого елемента масиву **ranks**). До речі, оголошення **ranks** призводить до створення об'єкта, який здатен зберігати **10** значень типу **int**, і кожен елемент масиву також являє собою об'єкт.

Якщо **l**-значення можна використовувати для зміни значення всередині об'єкта, ми маємо справу з **l**-значенням, що модифікується. Тепер розглянемо наступне оголошення:

```
const char *pc = "Це рядковий літерал!";
```

Воно призводить до того, що програма зберігає в пам'яті значення рядкового літерала, і цей масив символьних значень є об'єктом. Кожен символ в масиві також являє собою об'єкт, оскільки до нього можна звертатися індивідуально. Оголошення також створює об'єкт, який має ідентифікатор **pc** і зберігає адресу даного рядка. Ідентифікатор **pc** – це **l**-значення, що модифікується, оскільки його можна заново встановлювати для посилання на інші рядки. Ключове слово **const** запобігає зміні вмісту рядка, на який вказує **pc**,

але не зміні того, на який рядок він вказує. Таким чином, вираз `*pc`, означає об'єкт даних з символом `'ц'`, є `l`-значенням, але не `l`-значенням, що модифікується. Аналогічно, сам рядковий літерал, вказуючи на об'єкт, який містить символьний рядок, являє собою `l`-значення, що не допускає модифікації.

Об'єкт можна описати в термінах його тривалості зберігання, яка вказує, наскільки довго він залишається в пам'яті. Ідентифікатор, що застосовується для доступу до цього об'єкту, може бути описаний за допомогою його області видимості та зв'язування, які разом вказують, в якій частині програми цей ідентифікатор дозволено використовувати. Різні класи зберігання пропонують різні поєднання області видимості, зв'язування та тривалості зберігання. Допускається існування ідентифікаторів, які спільно використовуються в декількох файлах вхідного коду, ідентифікаторів, які можуть застосовуватися в будь-яких функціях всередині одного файлу, ідентифікаторів, які використовуються тільки всередині окремої функції, і навіть ідентифікаторів, що застосовуються лише в певному розділі функції. Одні об'єкти можуть існувати протягом часу життя цілої програми, а інші – тільки протягом часу виконання функції, яка їх містить. В паралельному програмуванні можна мати об'єкти, які існують протягом часу виконання окремого потоку. Можна також зберігати дані в пам'яті, яка явно виділяється та звільняється шляхом викликів функцій. Давайте поглянемо, що означають терміни область видимості, зв'язування та тривалість зберігання. Після цього приступимо до вивчення конкретних класів зберігання.

2.1.2. Область видимості

Область видимості описує ділянку або ділянки програми, де можна звертатися до ідентифікатора. Змінна в `C` має одну з наступних областей видимості:

- в межах блоку;
- в межах функції;

- в межах прототипу функції; - в межах файлу.

У прикладах програм, що були розглянуті до сих пір, для змінних використовувалася в основному область видимості на рівні блоку. Як ви пам'ятаєте, блок – це частина коду, що розміщується між фігурними дужками. Наприклад, блоком є усе тіло функції. Будь-який складений оператор всередині функції також вважається блоком. Змінна, що визначається у блоці, має область видимості в межах блоку, і вона є видимою від місця, де вона визначена, до кінця блоку, що містить це визначення. Крім того, формальні параметри функції, хоча вони з'являються до відкривальної фігурної дужки функції, також мають область видимості в межах блоку та належать блоку, що містить тіло функції. Таким чином, усі локальні змінні, які застосовувалися до сих пір, включаючи формальні параметри функцій, мали область видимості в межах блоку. Відповідно, змінні **cleo** і **patrick** в наведеному нижче коді мають область видимості в межах блоку, що простягається до закривальної фігурної дужки:

```
double blocky(double cleo)
{ double patrick = 0.0;
     ... return
     patrick;
}
```

Змінні, що оголошені у внутрішньому блоці, отримують область видимості, яка обмежена тільки цим блоком:

```
double blocky(double cleo)
{ double patrick = 0.0;
     int i;

     for(i = 0; i < 10; i++)
     { double q = cleo*i; // початок області видимості для q
          ...
          patrick *= q;
     } // кінець області видимості для q
     ... return
     patrick;
}
```

В цьому прикладі область видимості **q** обмежена внутрішнім блоком, і доступ до **q** може мати тільки код всередині цього блоку.

За традицією змінні з областю видимості в межах блоку повинні оголошуватися на початку блоку. В стандарті **C99** ця вимога була послаблена, і змінні дозволено оголошувати в будь-якому місці блоку. Одна з нових можливостей пов'язана з оголошенням всередині керуючого розділу циклу **for**. Тепер можна зробити таким чином:

```
for(int i = 0; i < 10; i++)
     printf("Можливість C99: i = %d", i);
```

Як частина цієї нової можливості, стандарт **C99** розширив концепцію блоку шляхом включення до неї коду, що керується циклами **for**, **while**, **do...while** або оператором **if**, навіть якщо фігурні дужки при цьому не використовуються. Таким чином, у попередньому циклі **for** змінна **i** вважається частиною блоку циклу **for**. Відповідно, її область видимості обмежена циклом **for**. Після того, як виконання покине цикл **for**, ця змінна **i** більше не буде помітною для програми.

Область видимості в межах функції застосовуються тільки до міток, що використовуються з операторами **goto**. Це означає, що коли мітка вперше з'являється у внутрішньому блоці функції, її область видимості простирається на всю функцію. Якщо б можна було використовувати одну й ту ж саму мітку всередині двох окремих блоків, виникла б плутанина, тому область видимості в межах функції для міток запобігає виникненню такої ситуації.

Область видимості в межах прототипу функції застосовується до імен змінних, що використовуються в прототипах функцій, як в наступному вигляді:

```
int mighty(int mouse, double large);
```

Область видимості в межах прототипу функції поширюється від місця визначення змінної до кінця оголошення прототипу. Це означає, що при обробці аргументу прототипу функції компілятор цікавить тільки тип аргументу. Якщо вказані імена, то зазвичай вони не грають жодної ролі та не обов'язково повинні збігатися з іменами, які застосовуються у визначенні функції. Імена грають невелику роль у випадку параметрів, що мають типи масивів змінної довжини:

```
void use_a_VLA(int n, int m, int mas[n][m]);
```

При використанні імен в дужках треба пам’ятати, що це повинні бути імена, які були оголошені раніше в прототипі.

Змінна, визначення якої знаходиться за рамками будь-якої функції, має **область видимості в межах файлу**. Змінна, що має область видимості в межах файлу, є видимою від місця її визначення і до кінця файлу, який містить її визначення.

Погляньте на показаний нижче приклад:

7

```
#include <stdio.h>

int units = 0;            // змінна з областю видимості
                          // в межах файлу
void critic(void);

int main(void)
{ ...
}

void critic(void)
{ ...
}
```

Тут змінна **units** має область видимості в межах файлу та може використовуватися і в функції **main()**, і в функції **critic()**. Оскільки змінні з областю видимості в межах файлу можуть використовуватися в більш ніж одній функції, вони ще називаються **глобальними змінними**.

2.1.3. Одиниці та файли трансляції

То, що ви розглядаєте як декілька файлів, для компілятора може виглядати як єдиний файл. Припустимо для прикладу, і це трапляється досить часто, що ви включаєте один або декілька файлів заголовку (з розширенням **.h**) до файлу вхідного коду (з розширенням **.c**). В свою чергу, файл заголовку може містити інші файли заголовку. У підсумку може бути задіяно багато фізичних файлів. Однак препроцесор **C** по суті замінює директиву **#include** вмістом файлу

заголовку. Таким чином, компілятор бачить єдиний файл, який містить інформацію з вашого файлу вхідного коду та усіх файлів заголовку. Такий файл називається **одиницею трансляції**. Коли ми описуємо змінну як ту, що має область видимості в межах файлу, насправді вона буде видимою цілій одиниці трансляції. Якщо ваша програма складається з декількох файлів вхідного коду, то вона буде нараховувати й декілька одиниць трансляції, кожна з яких відповідає файлу вхідного коду, і файлам, що включаються до нього.

2.1.4. Зв'язування

Змінна в C має одне з наступних зв'язувань:

- зовнішнє зв'язування;
- внутрішнє зв'язування;
- відсутність зв'язування.

Змінні з областю видимості в межах блоку, функції або прототипу функції не мають зв'язування. Це означає, що вони є закритими для блоку, функції або прототипу, в якому визначені. Змінна з областю видимості в межах файлу може мати або внутрішнє, або зовнішнє зв'язування. Змінна з зовнішнім зв'язуванням може застосовуватися у будь-якому місці багатофайлової програми, а змінна з внутрішнім зв'язуванням – де завгодно в одиниці трансляції.

2.1.5. Формальні та неформальні терміни

В стандарті C для опису області видимості, що обмежена однією одиницею трансляції (файл вхідного коду плюс файли заголовку, що підключаються до нього), використовується формулювання «область видимості в межах файлу з внутрішнім зв'язуванням», а для опису області видимості, яка (у всякому випадку, потенційно) поширюється на інші одиниці трансляції – формулювання «область видимості в межах файлу з зовнішнім зв'язуванням». Але у програмістів не завжди є час і терпіння застосовувати такі терміни. Поширення отримали скорочення «область видимості в межах файлу» для «області

видимості в межах файлу з внутрішнім зв'язуванням» і «глобальна область видимості» або «область видимості в межах програми» для «області видимості в межах файлу з зовнішнім зв'язуванням».

Як же тоді з'ясувати, внутрішнє або зовнішнє зв'язування має змінна з областю видимості в межах файлу? Ви повинні подивитися, чи використовується
у зовнішньому визначенні специфікатор класу зберігання **static**:

```
int giants =5;           // область видимості в межах файлу,
                         // зовнішнє зв'язування
static int dodgers = 3; // область видимості в межах файлу,
                         // внутрішнє зв'язування
int main(void)
{
...
}
...
```

Змінна **giants** може застосовуватися в інших файлах, які представляють собою складені частини тієї ж самої програми. Змінна **dodgers** є закритою для даного конкретного файлу, але може використовуватися будь-якою функцією у цьому файлі.

2.1.6. Тривалість зберігання

Область видимості та зв'язування описують видимість ідентифікаторів. Тривалість зберігання характеризує постійність об'єктів, що є доступними за допомогою цих ідентифікаторів. Об'єкт в **C** має одну з наступних чотирьох тривалостей зберігання:

- статичну;
- потокову;
- автоматичну;
- - виділену.

Якщо об'єкт має **статичну тривалість зберігання**, він існує протягом часу виконання програми. Змінні з областю видимості в межах файлу мають

статичну тривалість зберігання. Зверніть увагу, що для змінних з областю видимості в межах файлу ключове слово `static` вказує тип зв'язування, а не тривалість зберігання. Змінна з областю видимості в межах файлу, що оголошена з застосуванням `static`, має внутрішнє зв'язування, але усі змінні з областю видимості в межах файлу, що мають внутрішнє або зовнішнє зв'язування, мають статичну тривалість зберігання.

Потокова тривалість зберігання має місце при паралельному програмуванні, коли виконання програми може бути поділено на декілька потоків. Об'єкт з потоковою тривалістю зберігання існує з моменту його оголошення і до завершення потоку. Такий об'єкт створюється, коли оголошення, яке в іншому випадку призвело б до створення об'єкту з областю видимості в межах файлу, модифіковане за допомогою ключового слова `_Thread_local`. Коли змінна оголошена з таким специфікатором, кожен потік отримує власну закриту копію цієї змінної.

Змінні з областю видимості в межах блоку зазвичай мають автоматичну тривалість зберігання. Пам'ять для цих змінних виділяється, коли потік керування входить до блоку, де вони визначені, та звільняється, коли потік керування покидає цей блок. Ідея полягає в тому, що пам'ять, яка використовується для автоматичних змінних, є робочим простором або тимчасовою пам'яттю, яка може застосовуватися багатократно. Наприклад, після завершення виклику функції, пам'ять, яку функція використовувала для своїх змінних, може бути задіяна під час виклику наступної функції.

Масиви змінної довжини демонструють невеличке виключення в тому, що вони існують від місця свого оголошення і до кінця блоку, а не від початку блоку і до свого кінця.

Локальні змінні, які ми застосовували до сих пір, потрапляють до категорії автоматичної тривалості зберігання. Наприклад, в наступному коді змінні `number` і `index` з'являються кожного разу, коли функція `bore()` викликається, та зникають після її завершення:

```
void bore(int number)
{ int index;
     for(index = 0; index < number; index++)
          puts("Виконання звичної роботи.\n");
     return 0;
}
```

Проте, змінна може мати область видимості в межах блоку, але статичну тривалість зберігання. Щоб створити таку змінну, треба оголосити її всередині блоку та додати до оголошення ключового слова **static**:

```
void more(int number)
{ int index; static
     int ct = 0;
     ...
     return 0;
}
```

Тут змінна **ct** зберігається в статичній пам'яті. Вона існує з моменту завантаження програми в пам'ять і аж до завершення виконання програми. Але область видимості **ct** обмежена блоком функції **more()**. Тільки під час виконання цієї функції програма може використовувати змінну **ct** для доступу до об'єкту, який вона позначує.

Область видимості, зв'язування та тривалість зберігання використовується в **C** з метою визначення декількох схем зберігання для змінних.

Існує п'ять класів зберігання:

- автоматичний;
- регістровий;
- статичний з областю видимості в межах блоку;
- статичний з зовнішнім зв'язуванням;
- статичний з внутрішнім зв'язуванням.

Можливі комбінації представлені в табл. 2.1.

Таблиця 2.1 – П’ять класів зберігання

Клас зберігання	Тривалість зберігання	Область видимості	Зв’язування	Оголошення
Автоматичний	Автоматична	В межах блоку	Ні	В блоці
Регістровий	Автоматична	В межах блоку	Ні	В блоці з зазначенням ключового слова **register**
Статичний з зовнішнім зв’язуванням	Статична	В межах файлу	Зовнішнє	За рамками усіх функцій
Статичний з внутрішнім зв’язуванням	Статична	В межах файлу	Внутрішнє	За рамками усіх функцій з зазначенням ключового слова **static**
Статичний без зв’язування	Статична	В межах файлу	Ні	В блоці з зазначенням ключового слова **static**

2.1.7. Автоматичні змінні

Змінна, що належить до автоматичного класу зберігання, має автоматичну тривалість зберігання, область видимості в межах блоку і не має зв’язування. За замовчуванням будь-яка змінна, яка оголошена в блоці або в заголовку функції, належить до автоматичного класу зберігання. Однак ви можете сформулювати свої наміри, явно вказавши ключове слово **auto**:

```
int main(void)
{ auto int plox;
     ...
}
```

Це можна зробити, скажімо, для документування того факту, що ви навмисно перевизначаєте зовнішню змінну, або для підкреслення важливості того, що клас зберігання змінної не повинен змінюватися. Ключове слово **auto** називається специфікатором класу зберігання. В **C++** ключове слово **auto** призначено для зовсім іншої цілі, тому просто не застосовуйте **auto** в якості специфікатора класу зберігання, щоб домогтися більшої сумісності між **C** і **C++**.

Область видимості в межах блоку та відсутність зв'язування припускає, що доступ до цієї змінної по імені може здійснюватися тільки в блоці, де змінна визначена. Звичайно, за допомогою аргументів значення та адресу змінної можна передати до іншої функції, але це будуть вже непрямі відомості. В іншій функції може використовуватися змінна з тим самим іменем, але це буде незалежна змінна, що зберігається в іншій комірці пам'яті.

Автоматична тривалість зберігання означає, що змінна починає своє існування, коли потік керування входить до блоку, який містить оголошення цієї змінної. Після того як потік керування залишить блок, автоматична змінна зникає. Комірка пам'яті, яку вона займала, може застосовуватися для будь-чого іншого, хоча й необов'язково.

Давайте більш уважно поглянемо на вкладені блоки. Змінна відома тільки в блоці, в якому вона оголошена, і в будь-яких блоках, що розміщені всередині цього блока:

```
int loop(int n)
{ int m;  // m знаходиться в області видимості
     scanf("%d", &m);
     {
     int i; // m та i знаходяться в області видимості
          for(i = m; i < n; i++)
          puts("i є локальною змінною в субблоці \n");
     }
     return m;      // m в області видимості,
                    // i зникла
}
```

В даному коді змінна **i** є видимою тільки у внутрішніх дужках. Якщо ви спробуєте скористатися цією змінною до або після внутрішнього блоку, компілятор повідомить про помилку. Зазвичай такий прийом при проектуванні програм не застосовується. Проте, часом зручно визначати змінну в субблоці, якщо вона не використовується в інших місцях. В цьому випадку можна документувати призначення змінної близько до місця її застосування.

Крім того, змінна не буде залишатися невикористаною, дарма займаючи місце, коли вона більше не потрібна. Оскільки змінні **n** і **m** визначені в заголовку

функції та у зовнішньому блоці, вони находяться в області видимості всієї функції та існують аж до її завершення.

Що станеться, коли ви оголосите у внутрішньому блоці змінну, яка має таке ж саме ім'я, як і змінна у зовнішньому блоці? Тоді ім'я, що визначено всередині блоку, відповідає змінній, яка застосовується в цьому блоці. Ми говоримо, що ім'я приховує зовнішнє визначення. Однак коли потік керування залишає внутрішній блок, зовнішня змінна повертається до області видимості. Ці та інші аспекти проілюстровані в наступній програмі. Текст програми має наступний вид:

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

int main(void)
{
    int x = 30;        // перша змінна x

    SetConsoleOutputCP(1251);

    printf("x у зовнішньому блоці:  %4d за адресою %p\n", x, &x);
    {
        int x = 77;    // друга змінна x, що приховує першу x
printf("x у внутрішньому блоці: %4d за адресою %p\n", x, &x);
    }
    printf("x у зовнішньому блоці:  %4d за адресою %p\n", x, &x);
while(x++ < 33)    // перша змінна x
    {
        int x = 100;  // третя змінна x, що зв'язує першу x
        x++;
        printf("x у циклі while:  %4d за адресою %p\n", x, &x);
    }
    printf("x у зовнішньому блоці:  %4d за адресою %p\n", x, &x);
return 0;
}
```

Результат роботи програми наведено на рис. 2.1.

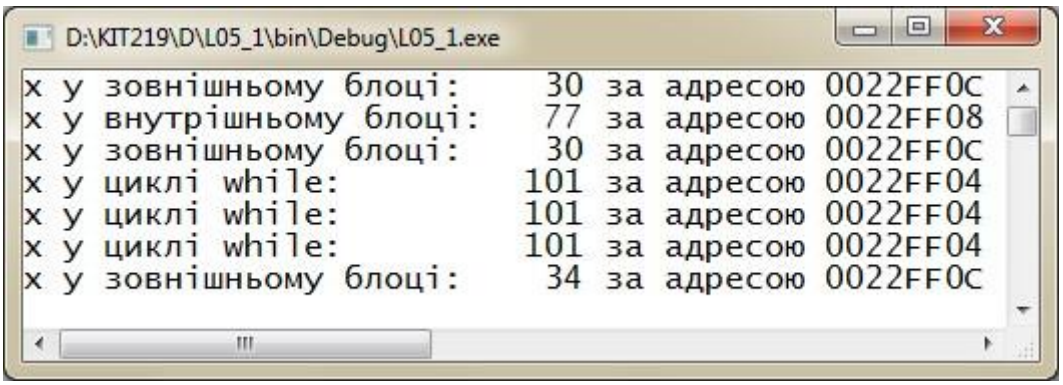


Рисунок 2.1 – Результат роботи програми для відстеження змінних у блоках

Спочатку програма створює змінну **x** зі значенням **30**, як показує перший оператор **printf()**. Потім вона визначає нову змінну **x** зі значенням **77**, про що повідомляє другий оператор **printf()**. Це нова змінна, що приховує першу змінну **x**, значення та адреса якої знову виводяться третім оператором **printf()**. Даний оператор знаходиться після першого внутрішнього блоку та відображає початкове значення **x**, демонструючи тим самим, що ця змінна нікуди не зникла й не змінювалася.

Імовірно, найбільш цікавою частиною програми є цикл **while**. В умові перевірки циклу **while** задіяна початкова змінна **x**:

```
while(x++ < 33)
```

Однак всередині циклу програма бачить третю змінну **x**, тобто ту, яка визначена в рамках блока циклу **while**. Таким чином, коли в тілі циклу використовується вираз **x++**, в ньому беруть участь нова змінна **x**, значення якої інкрементується до **101** і потім відображається. По завершенні кожної ітерації циклу ця нова змінна **x** зникає.

Далі в умові перевірки циклу застосовується та інкрементується початкова змінна **x**, знову відбувається вхід до блоку циклу, і знову створюється нова змінна **x**. В цьому прикладі змінна **x** створюється та знищується три рази.

Зверніть увагу, що для припинення виконання цикл повинен інкрементувати **x** в умові перевірки, тобто інкрементування **x** в тілі циклу

призводить до збільшення значення іншої змінної **x**, а не тієї, яка задіяна в умові перевірки.

Хоча конкретний компілятор не використовує повторно комірку пам'яті змінної **x** внутрішнього блоку для версії **x** з циклу **while**, деякі компілятори це роблять.

Призначення цього прикладу зовсім не в тому, щоб заохочувати написання коду в такому стилі. Він служить лише ілюстрацією того, що відбувається, коли ви визначаєте змінні всередині блоку.

2.1.8. Блоки без фігурних дужок

Згадана раніше можливість стандарту **C99** полягає в тому, що оператори, які є частиною циклу або оператора **if**, кваліфікуються як блок, навіть якщо фігурні дужки **({})** не вказані. Висловлюючись точніше, повний цикл – це субблок блоку, в якому він розміщений, а тіло циклу – субблок блоку повного циклу. Аналогічно, оператор **if** являє собою блок, а зв'язаний з ним оператор – субблок оператора **if**. Описані правила впливають на те, де ви можете оголошувати змінну, і на область видимості цієї змінної.

В наступній програмі показано, як це працює в циклі **for**. Текст програми має наступний вигляд:

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

int
main(void) {
    int n = 8;

    SetConsoleOutputCP(1251);

    printf("Спочатку         n = %d за адресою %p\n", n, &n);
    for(int n = 1; n < 3; n++)
        printf("       цикл 1:    n = %d за адресою %p\n", n, &n);
printf("  Після циклу 1 n = %d за адресою %p\n", n, &n);
for(int n = 1; n < 3; n++)
   {
```

```
    printf(" індекс циклу 2 n = %d за адресою %p\n", n, &n);
    int n = 6;
    printf("       цикл 2:    n = %d за адресою %p\n", n, &n);
    n++;
  }
    printf("  Після циклу 2 n = %d за адресою %p\n", n, &n);
    return 0;
}
```

Результат роботи програми наведено на рис. 2.2.

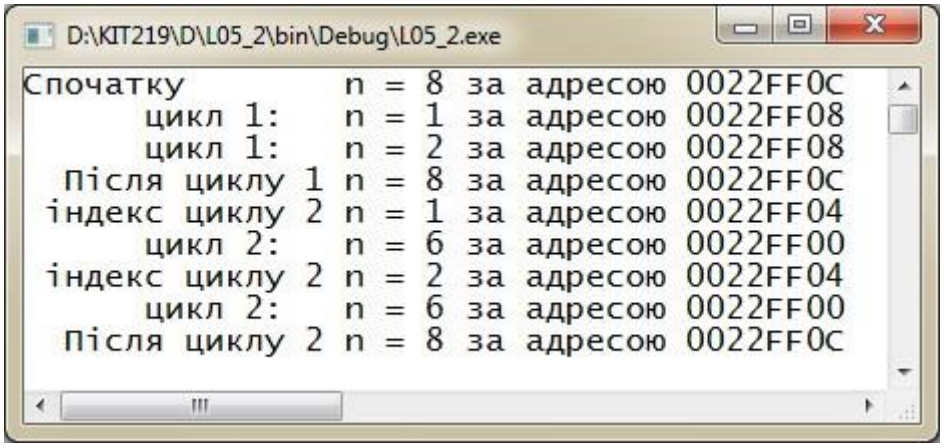


Рисунок 2.2 – Результат роботи програми, яка демонструє використання змінних в блоках

Змінна **n**, яка оголошена в круглих дужка першого циклу **for**, має область видимості до кінця циклу і приховує початкову змінну **n**. Але після того, як керування залишає цикл, початкова змінна **n** повертається до області видимості.

В другому циклі **for** змінна **n**, оголошена як індекс циклу, приховує початкову змінну **n**. Потім змінна **n**, яка оголошена всередині тіла циклу, приховує індекс циклу **n**. Як тільки програма завершить виконання тіла, змінна **n**, що оголошена в тілі, зникає, а в перевірці циклу бере участь індекс **n**. Коли завершиться виконання всього циклу, в області видимості з’являється початкова змінна **n**.

І знову відзначимо, що немає жодної потреби багатократно застосовувати, одне й те ж саме ім’я для змінної, але ви повинні знати, що відбудеться, коли ви все ж таки приймете рішення зробити таким чином.

2.1.9. Ініціалізація автоматичних змінних

Автоматичні змінні не ініціалізуються до тих пір, поки ви не зробите це явно. Погляньте на наступні оголошення:

```
int main(void)
{ int repid; int
     tents = 5;
     ...
}
```

Змінна **tents** ініціалізується значенням **5**, але **repid** отримує значення, яке раніше знаходилось в області пам’яті, що виділяється під цю змінну. Неможна розраховувати, на те, що цим значенням буде **0**. Ви можете ініціалізувати автоматичну змінну неконстантним виразом за умови, що усі задіяні в ньому змінні були визначені раніше:

```
int main(void)
{ int ruth = 1;
    int rance = 8 * ruth; // використовується раніше визначена
                          // змінна
     ...
}
```

2.1.10. Регістрові змінні

Змінні зазвичай зберігаються в пам’яті комп’ютера. За сприятливим збігом обставин регістрові змінні зберігаються в регістрах центрального процесора, або, в загальному випадку, в найшвидшій пам’яті, що забезпечує доступ і маніпулювання ними з меншими витратами часу, ніж для звичайних змінних. Оскільки регістрова змінна може знаходитися в регістрі, або в пам’яті, отримати адресу такої змінної не вдасться. В більшості інших випадків регістрові змінні нічим не відрізняються від автоматичних змінних. Тобто вони мають область видимості в межах блоку, не мають зв’язування та мають автоматичну тривалість зберігання. Змінна оголошується з використанням специфікатора класу зберігання **register**:

```
int main(void)
{
    register int quick;
    ...
}
```

Ми говоримо «за сприятливим збігом обставин», тому що оголошення змінної як регістрової є скоріш запитом, ніж прямою вказівкою. Компілятор повинен зіставити ваші вимоги з кількістю доступних регістрів або об'єму швидкодіючої пам'яті, або ж він може просто проігнорувати запит, і ваше побажання не буде задовільнене. В такому випадку змінна стає звичайною автоматичною змінною. Проте, застосовувати до неї операцію взяття адреси так само неможна. Ви маєте змогу зробити запит, щоб формальні параметри були регістровими змінними. Для цього просто скористайтеся ключовим словом **register** в заголовку функції:

```
void macho(register int n)
```

Типи, які допускається оголошувати як **register**, можуть виявитися обмеженими. Наприклад, регістри в процесорі можуть бути недостатньо великими, щоб вмістити тип **double**.

2.1.11. Статичні змінні з областю видимості в межах блоку

Назва «статична змінна» звучить як взаємовиключне поняття, наче «змінна, яка не може бути змінена». В дійсності характеристика «статична» означає, що змінна залишається розміщеною в пам'яті, а не обов'язково відноситься до значення. Змінні з областю видимості в межах файлу автоматично (і обов'язково) мають статичну тривалість зберігання. Як згадувалося раніше, можна також створювати локальні змінні, що мають область видимості в межах блоку, але статистичну тривалість зберігання. Такі змінні мають таку ж саму область видимості, як автоматичні змінні, однак вони не зникають, коли функція, що їх містить, завершує свою роботу. Іншими словами, такі змінні мають область видимості в межах блоку, не мають зв'язування, але мають статичну тривалість зберігання. Комп'ютер пам'ятає їх значення від одного виклику функції до

наступного. Такі змінні створюються в результаті оголошення в блоці (що забезпечує область видимості в межах блоку і відсутність зв'язування) зі специфікатором класу зберігання **static** (що надає статичну тривалість зберігання).

Наступна програма ілюструє цей прийом. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
 void
trystat(void);

int
main(void) {
    int count;

    SetConsoleOutputCP(1251);

    for(count = 1; count <= 3; count++)
    {
      printf("Ітерація %d:\n", count);
      trystat();
    }
return 0;
}
 void trystat(void)
{
   int fade = 1;
   static int stay = 1;

    printf("=======================\n");
    printf("    fade = %d; stay = %d\n", fade++, stay++);
    printf("=======================\n\n");
}
```

Зверніть увагу, що **trystat()** інкрементує кожну змінну після виводу її значення. Результат роботи програми наведено на рис. 2.3.

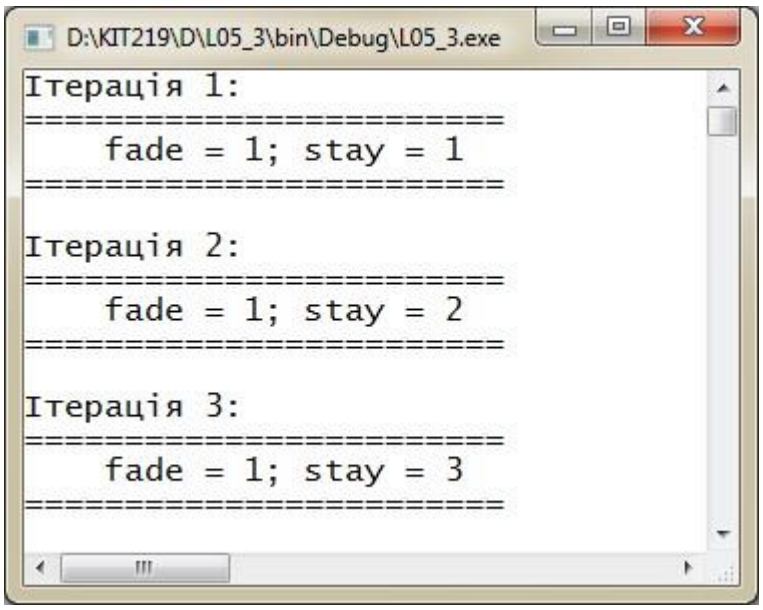


Рисунок 2.3 – Результат роботи програми, яка демонструє статичні змінні з областю видимості в межах блоку

Статична змінна **stay** запам'ятовує, що її значення було збільшено на **1**, але змінна **fade** кожного разу починається заново. Це вказує на відмінність в ініціалізації: **fade** ініціалізується при кожному виклику **trystat()**, а **stay** – тільки одного разу, коли функція **trystat()** компілюється. Статичні змінні ініціалізуються нулем, якщо вони не були явно ініціалізовані іншим значенням.

Два наступних оголошення виглядають схожими:

```
int fade = 1;
static int stay = 1;
```

Проте, перший оператор в дійсності є частиною функції **trystat()** і виконується кожного разу, коли функція викликана. Це дія часу виконання. Другий оператор насправді не стосується функції **trystat()**. Якщо ви застосуєте відлагоджувач для покрокового виконання програми, то побачите, що програма як би пропускає цей крок. Причина в тому, що після того, як програма завантажилася в пам'ять, статичні змінні та зовнішні змінні вже знаходяться в потрібних місцях. Розміщення оператора оголошення до функції **trystat()** повідомляє компілятору, що тільки функції **trystat()** дозволено бачити дану змінну: це не оператор, який виконується під час виконання.

Використовувати модифікатор **static** для параметрів функції неможна:

```
int wontwork(static int flu);       // не дозволено
```

Іншим терміном для статичної змінної з областю видимості в межах блоку є «локальна статична змінна». Крім того, якщо ви читали ранню літературу по **C**, то мабуть побачили, що цей клас зберігання називали внутрішнім статичним класом зберігання. Проте, слово внутрішній застосовувалося для вказування на оголошення всередині функції, а не на внутрішнє зв'язування.

2.1.12. Статичні змінні з зовнішнім зв'язуванням

Статична змінна з зовнішнім зв'язуванням має область видимості в межах файлу, зовнішнє зв'язування і статичну тривалість зберігання. Такий клас іноді називають **зовнішнім класом зберігання**, а змінні цього типу – **зовнішніми змінними**. Зовнішня змінна створюється шляхом розміщення оголошення за рамками усіх функцій. Відповідно до документації, зовнішня змінна може додатково бути оголошена всередині функції, в якій вона використовується, з застосуванням ключового слова **extern**. Якщо будь-яка зовнішня змінна визначена в одному файлі початкового коду та використовується в іншому файлі початкового коду, то оголошення цієї змінної в іншому файлі з ключовим словом **extern** є обов'язковим. Оголошення виглядає наступним чином:

```
int Errupt;          // змінна, що має зовнішнє визначення
double Up[100];      // масив, що має зовнішнє визначення
extern char Coal;    // обов'язкове оголошення, якщо
                     // Coal визначається в іншому файлі
void next(void);

int main(void)
{
  extern int Errupt;     // необов'язкове оголошення
  extern double Up[];    // необов'язкове оголошення
}
 void next(void)
{
...
}
```

Зверніть увагу, що ви не зобов'язані вказувати розмірність масиву в необов'язковому оголошенні `double Up[]`. Це пояснюється тим, що початкове оголошення `double Up[100];` вже надало таку інформацію. Групу оголошень `extern` всередині `main()` можна повністю опустити, оскільки зовнішні оголошення мають область видимості в межах файлу, тому вони відомі від місця оголошення і до кінця файлу. Однак вони призначені для документування намірів застосовувати ці змінні в `main()`.

Якщо ключове слово `extern` відсутнє в оголошенні всередині функції, створюється окрема автоматична змінна. Тобто, заміна `extern int Errupt;` оголошенням `int Errupt;` в `main()` призводить до того, що компілятор створює автоматичну змінну на ім'я `Errupt` – окрему локальну змінну, яка відрізняється від початкової змінної `Errupt`. Ця локальна змінна буде знаходитися в області видимості під час виконання `main()`, але для інших функцій, таких як `next()`, що розташовані в тому ж самому файлі, в області видимості буде зовнішня змінна `Errupt`. Тобто, змінна з областю видимості в межах блоку «приховує» змінну з тим самим іменем, що має область видимості в межах файлу, коли відбувається виконання операторів в цьому блоці. Якщо за будь-якої малоймовірної причини вам дійсно необхідна локальна змінна, що має те ж саме ім'я, що й глобальна змінна, можете скористатися в локальному оголошенні специфікатором класу зберігання `auto`, щоб явно документувати свій вибір.

Зовнішні змінні мають статичну тривалість зберігання. Таким чином, масив `Up` існує і зберігає свої значення незалежно від того, виконується `main()`, `next()` або будь-яка інша функція.

В наступних трьох прикладах показані чотири можливі комбінації зовнішніх і автоматичних змінних. В прикладі 1 присутня одна зовнішня змінна `Hocus`, яка відома і `main()`, і `magic()`.

Приклад 1.

```
int Hocus;
int magic();
int main(void)
{
```

```
    extern int Hocus;   // змінна Hocus оголошена як зовнішня
}
 int magic()
{
    extern int Hocus;   // та ж сама змінна Hocus, що й раніше
    ...
}
```

В прикладі 2 є одна зовнішня змінна **Hocus**, яка відома обом функціям. На цей раз функція **magic()** знає про неї за замовчуванням.

Приклад 2.

```
int Hocus;
int
magic();

int main(void)
{
    extern int Hocus;   // змінна Hocus оголошена як зовнішня

int magic()
{
    // змінна Hocus не оголошена, але відома
    ...
}
```

В прикладі 3 створюються чотири змінні. Змінна **Hocus** в **main()** є автоматичною за замовчуванням і локальною для **main()**. Змінна **Hocus** в **magic()** явно оголошена як автоматична і відома тільки **magic()**. Зовнішня змінна **Hocus** не відома **main()** або **magic()**, але буде відома будь-якій іншій функції в даному файлі, що не має власної локальної змінної **Hocus**. Нарешті, **Pocus** – це зовнішня змінна, яка відома **magic()**, але не **main()**,тому що **Pocus** знаходиться за **main()**.

Приклад 3.

```
int Hocus;
int magic(); 24
int main(void)
{
    int Hocus;          // змінна Hocus оголошена, за
                        // замовчуванням є автоматичною
    ...
}
```

```
int Pocus;
int magic()
{
    auto int Hocus;   // локальна змінна Hocus оголошена
                      // як автоматична
     ...
}
```

Наведені приклади ілюструють область видимості зовнішніх змінних, яка поширюється від місця їх оголошення і до кінця файлу. Вони також відображають час життя змінних. Зовнішні змінні **Hocus** і **Pocus** існують протягом всього часу виконання програми, а оскільки вони не обмежені будь-якою однією функцією, вони не зникають після завершення конкретної функції.

2.1.13. Ініціалізація зовнішніх змінних

Як і автоматичні, зовнішні змінні можуть ініціалізуватися явно. На відміну від автоматичних, зовнішні змінні за замовчуванням ініціалізуються нулем, якщо ви не ініціалізували їх. Це правило можна застосовувати також до елементів зовнішньо визначеного масиву. Однак для ініціалізації змінних з областю видимості в межах файлу можна використовувати тільки константні вирази, що відрізняється від випадку автоматичних змінних.

```
int x = 10;               // допустимо, 10 - це константа
int y = 3 + 20;           // допустимо, константний вираз
size_t z = sizeof(int);   // допустимо, константний вираз
int x2 = 2 * x;           // недопустимо, x — це змінна
```

За умови, що типом не є масив, вираз **sizeof** вважається константним.

2.1.14. Використання зовнішньої змінної

Давайте розглянемо простий приклад, в якому задіяна зовнішня змінна.

Припустимо, що дві функції з іменами **main()** і **critic()** повинні мати доступ до змінної **units**. Це можна зробити, оголосивши **units** за межами двох функцій, що згадувалися раніше, як це показано в наступній програмі. Текст програми має такий вигляд:

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

int units = 0;           // зовнішня змінна
 void critic(void);

int main(void)
{
    SetConsoleOutputCP(1251);

    extern int units;   // необов'язкове повторне оголошення
    printf("Скільки фунтів важить маленький бочонок
    олії?\n");     scanf("%d", &units);
    while(units != 56)
    critic();
    printf("Ви вгадали!\n");
    return 0;
}
 void critic(void)
{
    // необов'язкове повторне оголошення опущене
printf("Вам не пощастило. Спробуйте ще раз.\n");
scanf("%d", &units); }
```

Результат роботи програми наведено на рис. 2.4.

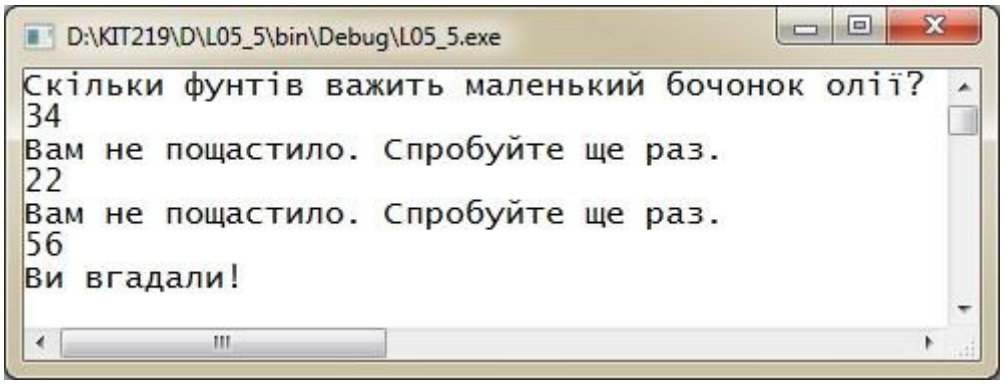


Рисунок 2.4 – Результат роботи програми, яка демонструє використання зовнішньої змінної

Необхідно звернути, що друге значення для **units** читається функцією **critic()**, але **main()** також відоме нове значення після завершення циклу **while**. Таким чином, і **main()**, і **critic()** використовують ідентифікатор **units** для доступу до однієї і тієї ж самої змінної. В рамках термінології **C** ми говоримо,

що змінна **units** має область видимості в межах файлу, зовнішнє зв'язування і статичну тривалість зберігання.

Ми зробили **units** зовнішньою змінною, визначивши її за межами визначень усіх функцій. Це і все, що необхідно зробити для забезпечення доступності **units** усім наступним функціям у файлі.

Давайте поглянемо на деякі деталі. Перш за все, оголошення змінної **units** там, де воно знаходиться, робить її доступною оголошеним далі функціям без додаткових умов. Відповідно, функція **critics()** користується змінною **units**.

Аналогічно, нічого не треба буде робити для надання доступу до **units** функції **main()**. Однак в **main()** є таке оголошення:

```
extern int units;
```

У прикладі, що розглядається, це оголошення є головним чином формою документування. Специфікатор класу зберігання **extern** повідомляє компілятору, що будь-яке згадування **units** в даній функції відноситься до змінної, що оголошена за межами цієї функції, можливо, навіть поза самим файлом. І знову **main()** і **critic()** працюють зі змінною **units**, яка визначена зовні.

2.1.15. Зовнішні імена

Стандарти **C99** і **C11** потребують, щоб компілятори розрізняли перші **63** символи для локальних ідентифікаторів і перші **31** символи для зовнішніх ідентифікаторів. Це корегує попередню вимогу з розпізнавання перших **31** символів для локальних і перших **6** символів для зовнішніх ідентифікаторів. Цілком можливо, що ви маєте справу зі старими правилами. Причина того, що правила для імен зовнішніх змінних є більш обмежуючими, ніж правила для імен локальних змінних, пов'язана з тим, що зовнішні імена повинні дотримуватися правил локального середовища, які можуть бути більш жорсткими.

2.1.16. Визначення та оголошення

Поглянемо більш уважно на різницю між визначенням змінної та її оголошенням. Подивіться на наступний приклад:

```
int tern = 1;           // змінна tern визначена
 int main(void)
{
    extern int tern;    // використання tern, що визначена
                        // десь в іншому місці
    ...
}
```

Тут змінна **tern** оголошується двічі. Перше оголошення призводить до того, що для змінної відводиться місце в пам'яті. Воно створює визначення змінної. Друге оголошення просто вказує компілятору на необхідність застосування змінної **tern**, яка була створена раніше, тому це не є визначенням. Перше оголошення називається **визначальним оголошенням**, а друге – **посилальним оголошенням**. Ключове слово **extern** говорить про те, що оголошення не є визначенням, оскільки воно повідомляє компілятор про необхідність пошуку визначення десь в іншому місці.

Припустимо, що ви записали наступний код:

```
extern int tern;
int main(void)
{
...
}
```

Компілятор припустить, що дане визначення **tern** знаходиться в іншому місці програми, можливо, в іншому файлі. Це оголошення не призводить до виділення пам'яті. Таким чином, не використовуйте ключове слово **extern** для створення зовнішнього визначення. Застосовуйте його тільки для посилання на існуюче зовнішнє визначення.

Зовнішня змінна може бути ініціалізована тільки один раз, і це повинно відбуватися при визначенні змінної. Погляньте на наступний код:

```
// файл one.c
char permis = 'N';
// файл two.c
extern char permis = 'Y';    // помилка
```

Помилка полягає в тому, що визначальне оголошення у файлі **one.c** вже було створено, і воно ініціалізувало змінну **permis**.

2.1.17. Статичні змінні з внутрішнім зв’язуванням

Змінні з цим класом зберігання мають статичну тривалість зберігання, область видимості в межах файлу і внутрішнє зв’язування. Вони створюються шляхом їх визначення поза межами будь-яких функцій (як і у випадку зовнішніх змінних) з вказуванням специфікатора класу зберігання **static**:

```
static int svil = 1;  // статична змінна, внутрішнє зв’язування
int main(void)
{
...
}
```

Змінні подібного роду колись отримали назву зовнішніх статичних змінних, але це дещо заплутує, оскільки вони мають внутрішнє зв’язування. Нажаль, новий компактний термін знайти не вдалося, тому залишається варіант статична змінна з внутрішнім зв’язуванням. Звичайна зовнішня змінна може використовуватися функціями в будь-якому файлі, який є частиною програми, але статична змінна з внутрішнім зв’язуванням може застосовуватися тільки функціями в тому ж самому файлі. Всередині функції можна повторно оголосити будь-яку змінну з областю видимості в межах файлу, використовуючи специфікатор класу зберігання **extern**. Таке оголошення не змінює тип зв’язування.

Розглянемо наступний код:

```
int traveler = 1;              // зовнішнє зв’язування
static int stayhome = 1;       // внутрішнє зв’язування
int main(void)
{
    extern int traveler;       // використання глобальної
                               // змінної traveler
    extern int stayhome;       // використання глобальної
                               // змінної stayhome
    ...
}
```

Змінні **traveler** і **stayhome** є глобальними в цій конкретній одиниці трансляції, але тільки **traveler** можна застосовувати в інших одиницях трансляції. Два оголошення, що використовують **extern**, документують той

факт, що в `main()` застосовуються дві глобальні змінні, але `stayhome` продовжує мати внутрішнє зв'язування.

2.1.18. Використання декількох файлів

Відмінність між внутрішнім і зовнішнім зв'язуваннями важлива тільки в ситуації, коли програма будується з декількох одиниць трансляції, тому давайте коротко розглянемо дане питання.

Складні програми на `C` часто складаються з декількох окремих файлів початкового коду. Іноді в цих файлах виникає необхідність спільного використання будь-якої зовнішньої змінної. Щоб зробити це в `C`, необхідно передбачити визначальне оголошення в одному файлі та посилальне оголошення в інших файлах. Це означає, що в усіх оголошеннях крім одного (визначального оголошення) повинне бути присутнім ключове слово `extern`, а для ініціалізації змінної слід застосовувати тільки визначальне оголошення.

Зверніть увагу, що зовнішня змінна, яка визначена в одному файлі, не буде доступною в іншому файлі до тих пір, поки її також там не оголосити (з використанням `extern`). Саме по собі зовнішнє оголошення лише робить змінну потенційно доступною для іншого файлу.

Однак історично склалося таким чином, що багато компіляторів в цьому відношенні дотримуються інших правил. Наприклад, такі системи як `Unix` дозволяють оголошувати змінну в декількох файлах без зазначення ключового слова `extern` за умови, що тільки одне оголошення містить ініціалізацію.

Оголошення з ініціалізацією вважається визначенням.

2.2. Специфікатори класів зберігання. Виділення пам'яті

2.2.1. Специфікатори класів зберігання

Сенс ключових слів `static` і `extern` залежить від контексту. В мові C існує шість ключових слів, які згруповані разом як специфікатори класів зберігання.

До них належать такі ключові слова, як: **auto**, **register**, **static**, **extern**, **_Thread_local** і **typedef**.

Ключове слово **typedef** нічого не говорить про зберігання в пам'яті, але воно присутнє в цьому списку за синтаксичними причинами. Наприклад, в більшості випадків ви можете використовувати в оголошенні не більше одного специфікатора класу зберігання, а це означає, що ви не можете застосовувати один зі специфікаторів класу зберігання в якості частини **typedef**. Виключенням є специфікатор **_Thread_local**, який можна використовувати разом зі специфікаторами **static** і **extern**.

Специфікатор **auto** вказує змінну з автоматичною тривалістю зберігання. Він може застосовуватися тільки в оголошеннях змінних з областю видимості в межах блоку, які вже мають автоматичну тривалість зберігання, так що **головним його призначенням є документування**.

Специфікатор **register** також може використовуватися тільки зі змінними, що мають область видимості в межах блоку. Він поміщає змінну в регістровий клас зберігання, що рівносильно запиту на мінімізацію часу доступу до неї. Він також запобігає взяттю адреси цієї змінної.

Специфікатор **static** створює об'єкт зі статичною тривалістю зберігання, який з'являється після завантаження програми в пам'ять і зникає при завершенні програми. Якщо **static** застосовується в оголошенні з областю видимості в межах файлу, то область видимості обмежується одним цим файлом. Якщо **static** використовується в оголошенні з областю видимості в межах блоку, то область видимості обмежується цим блоком. Таким чином, об'єкт існує і зберігає своє значення протягом виконання програми, але може бути доступним за допомогою ідентифікатора, тільки коли виконується код всередині його блоку. Статична змінна з областю видимості в межах блоку не має зв'язування. Статична змінна з областю видимості в межах файлу має внутрішнє зв'язування.

Специфікатор **extern** вказує, що ви оголошуєте змінну, яка була визначена в будь-якому іншому місці. Якщо оголошення, яке містить **extern**, має область видимості в межах файлу, то змінна, на яку йде посилання, повинна мати

зовнішнє зв’язування. Якщо оголошення з `extern` має область видимості в межах блоку, то змінна, на яку йде посилання, може мати або зовнішнє, або внутрішнє зв’язування, що залежить від визначального оголошення цієї змінної.

2.2.2. Узагальнені відомості про класи зберігання

Автоматичні змінні мають область видимості в межах блоку, не мають зв’язування і характеризуються автоматичною тривалістю зберігання. Ці змінні є локальними та закриті для блоку (зазвичай функції), в якій вони визначені.

Регістрові змінні мають такі самі властивості, що й автоматичні змінні, але для їх зберігання компілятор може застосовувати більш швидку пам’ять або регістри. Адресу регістрової змінної отримати неможна.

Змінні зі статичною тривалістю зберігання можуть мати зовнішнє зв’язування, внутрішнє зв’язування або взагалі не мати зв’язування. Коли змінна оголошується зовнішньою по відношенню до будь-якої функції у файлі, то вона являє собою зовнішню змінну і має область видимості в межах файлу, зовнішнє зв’язування і статичну тривалість зберігання.

Якщо додати до такого оголошення ключове слово `static`, то можна отримати змінну зі статичною тривалістю зберігання, областю видимості в межах файлу та внутрішнім зв’язуванням. Якщо ви оголошуєте змінну всередині функції та вказуєте ключове слово `static`, то ця змінна отримує статичну тривалість зберігання, область видимості в межах блоку та відсутність зв’язування.

Пам’ять для змінної з автоматичною тривалістю зберігання виділяється, коли потік керування входить до блоку, що містить оголошення змінної, та звільняється після того, як потік керування залишить цей блок. Змінна такого роду, яка не була ініціалізована, має випадкове значення. Пам’ять для змінної зі статичною тривалістю зберігання виділяється на етапі компіляції і зберігається на весь час виконання програми. Змінна такого роду, що не була ініціалізована, отримує значення `0`.

Змінна з областю видимості в межах блоку є локальною по відношенню до блоку, який містить це оголошення.

Змінна з областю видимості в межах файлу відома усім функціям у файлі (або одиниці трансляції), які знаходяться після її оголошення. Якщо змінна з областю видимості в межах файлу має зовнішнє зв'язування, то вона може використовуватися іншими одиницями трансляції в програмі. Якщо змінна з областю видимості в межах файлу має внутрішнє зв'язування, то вона може застосовуватися тільки всередині файлу, в якому вона оголошена.

Нижче показана коротка програма, в якій використовуються усі класи зберігання. Код рознесений на два файли **main.c** і **accumulate.c**, так що ви повинні провести багатофайлову компіляцію. Головна мета цієї програми полягає в демонстрації усіх класів зберігання, а не в тому, щоб запропонувати проектну модель. У якісному проекті немає потреби у змінних з областю видимості в межах файлу.

Текст програми, що знаходиться у файлі **main.c**.

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

void report_count();
void accumulate(int k);
int count = 0;                  // область видимості в межах файлу
                                // зовнішнє зв'язування

int main(void)
{
int value;                  // автоматична змінна
register int i;             // регістрова змінна

SetConsoleOutputCP(1251);
printf("Введіть додатне ціле число (0 для завершення): ");
while(scanf("%d", &value) == 1 && value > 0)
    {
        ++count;                // використання змінної з областю
                                // видимості в межах файлу
for(i = value; i >= 0; i--)
    accumulate(i);
  printf("\nВведіть додатне ціле число (0 для завершення): ");
}
```

```
    report_count();
    return 0;
}
void report_count()
{
printf("\n\n================================================\n");
printf("Кількість разів виконання циклу: %d\n", count);
printf("================================================\n");
}
```

Текст програми, що знаходиться у файлі **accumulate.c**.

```
extern int count;              // посилальне оголошення,
                               // зовнішнє зв'язування
static int total = 0;          // статичне визначення,
                               // внутрішнє зв'язування
void accumulate(int k);        // прототип функції accumulate()
void accumulate(int k)         // k має область видимості в межах
                               // блоку, зв'язування відсутнє
{
    static int subtotal = 0;    // статична змінна,
                                // зв'язування відсутнє
if(k <= 0)
    {

printf("================================================\n");
printf("Ітерація циклу: %d\n", count);
printf("================================================\n");
printf("   subtotal: %10d;      total: %10d\n",
       subtotal, total);
subtotal = 0;
    }
  else
  {
      subtotal += k;
      total    += k;
  }
}
```

В цій програмі статична змінна з областю видимості в межах блоку, що має ім’я **subtotal**, накопичує проміжну суму значень, які передаються функції **accumulate()**, а змінна **total**, з областю видимості в межах файлу і внутрішнім зв’язуванням, накопичує загальну суму. Функція **accumulate()** виводить значення **total** і **subtotal** кожного разу, коли їй передається не додатне значення. В таких ситуаціях вона також скидає **subtotal** в **0**. Прототип

`accumulate()` в програмі `main.c` є обов'язковим, оскільки файл містить виклик функції `accumulate()`. В файлі `accumulate.c` прототип не є обов'язковим, оскільки функція в ньому визначена, але не викликається. В цій функції також застосовується зовнішня змінна `count` для відстеження кількості ітерацій циклу `while`, які виконуються в `main()`. В файлі `main.c` функції `main()` і `report_count()` спільно здійснюють доступ до `count`. Результат роботи програми наведено на рис. 2.5.

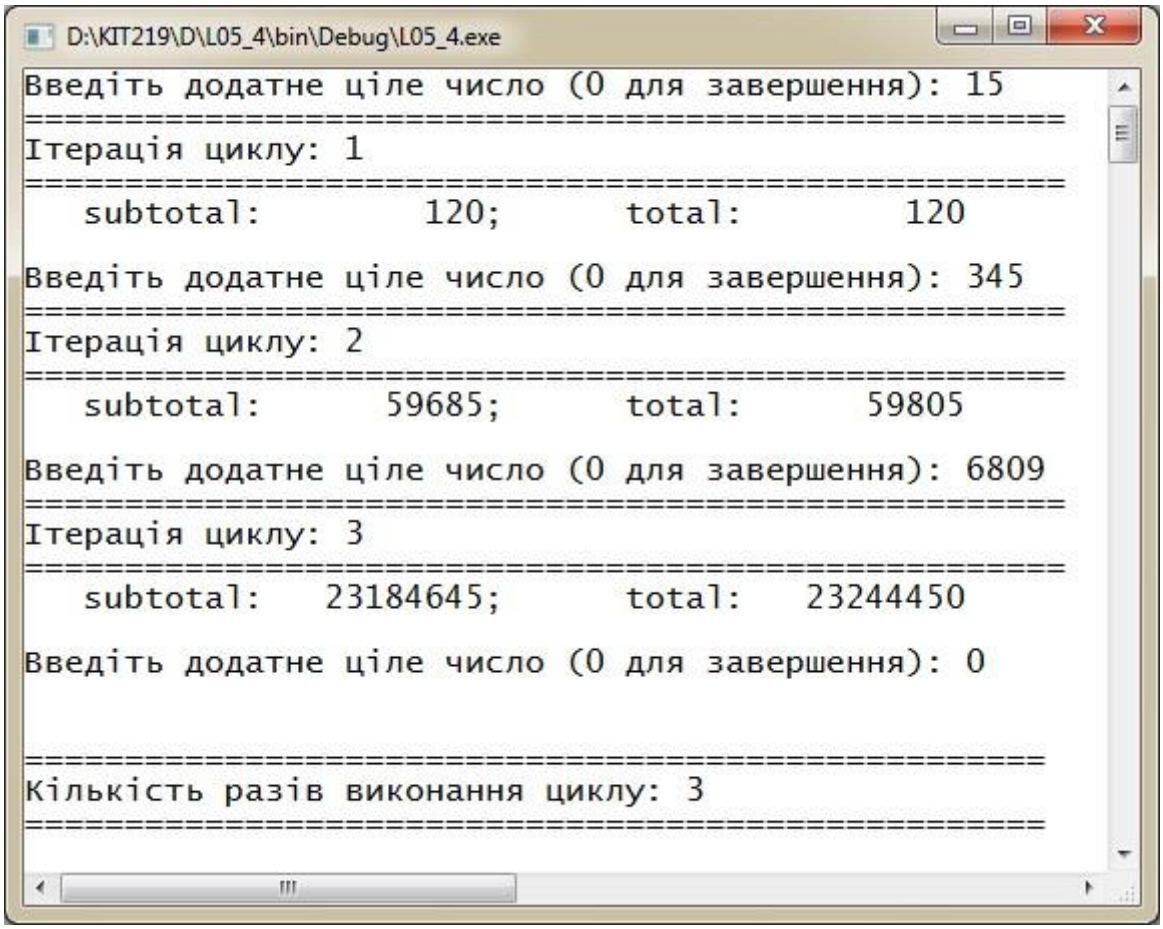


Рисунок 2.5 – Результат роботи програми, в якій використовуються усі класи зберігання

2.2.3. Класи зберігання та функції

Функції також мають класи зберігання. Функція може бути або зовнішньою (за замовчуванням), або статичною. В стандарті `C99` додана третя можливість – вбудована функція. Доступ до зовнішньої функції можуть

отримувати функції в інших файлах, але статична функція може застосовуватися тільки всередині файлу, де вона визначена.

Розглянемо, наприклад, файл з наступними прототипами функцій:

```
double gamma(double);        // за замовчуванням є зовнішньою
static double beta(int, int);
extern double delta(double, int);
```

Функції `gamma()` і `delta()` можуть використовуватися функціями в інших файлах, які є частиною програми, але `beta()` – ні. Через обмеження функції `beta()` одним файлом в інших файлах можна застосовувати інші функції з цим самим іменем. Причина використання класу зберігання `static` пов'язана зі створенням функцій, які закриті по відношенню до конкретного модуля, завдяки чому усувається можливість конфлікту імен.

Звична практика передбачає застосування ключового слова `extern` під час оголошення функції, що визначена в іншому файлі. Головним чином це

стосується ясності, оскільки оголошення функції передбачається як `extern`, якщо тільки не вказано ключове слово `static`.

2.2.4. Вибір класу зберігання

Відповіддю на питання про те, який обрати клас зберігання, частіше за все буде – автоматичний. Взагалі, за якою ще причиною автоматичний клас зберігання був обраний за замовчуванням? Так, ми знаємо, що на перший погляд зовнішній клас зберігання виглядає більш привабливим. Варто лише зробити усі свої змінні зовнішніми, і не прийдеться турбуватися про використання аргументів і вказівників при взаємодії між функціями.

Однак в цьому випадку доведеться переживати про те, що функція `A()` непомітно модифікує значення змінних, що застосовуються в функції `B()`, хоча ваші наміри були зовсім іншими. Безперечне свідоцтво великої кількості років, протягом яких формувався колективний досвід програмістів, говорить про те, що одна ця прихована небезпека далеко перевершує сумнівну привабливість нерозбірливого використання змінних з зовнішнім класом зберігання.

Поширеним виключенням з правила є дані **const**. Оскільки вони не можуть бути змінені, немає потреби турбуватися з приводу їх ненавмисної модифікації:

```
const int DAYS = 1;
const char *MSGS[3] = { "Так", "Ні", "Можливо" };
```

Одним з золотих правил безпечного програмування є принцип «необхідного знання». Тримайте всю внутрішню роботу кожної функції максимально закритою в рамках цієї функції, спільно використовуючи тільки ті змінні, які повинні спільно використовуватися. Інші класи зберігання є зручними та доступними. Однак перш ніж обирати будь-який з них, подумайте,чи є в цьому необхідність.

2.2.5. Функція генерації випадкових чисел і статична змінна

Тепер, коли ви отримали необхідний мінімум знань про класи зберігання, давайте розглянемо програми, в яких вони застосовуються. Спочатку поглянемо на функцію, яка використовує статичну змінну з внутрішнім зв'язуванням: функцію генерації випадкових чисел. Як ви вже знаєте для генерації випадкових чисел бібліотека **ANSI C** пропонує функцію **rand()**. Існують різні алгоритми генерування випадкових чисел, і **ANSI C** дозволяє реалізаціям обирати найкращий алгоритм для конкретної машини. Однак **ANSI C** також пропонує стандартний алгоритм, який видає ті ж самі випадкові числа в різних системах. Насправді функція **rand()** є «генератором псевдовипадкових чисел», тобто фактична послідовність чисел передбачувана, але числа достатньо рівномірно розподілені в межах діапазону можливих значень.

Замість застосування вбудованої функції **rand()** компілятора, ми будемо використовувати версію **ANSI**, щоб ви могли бачити, що відбувається всередині. Схема починається з числа, яке називається «початковим». Функція використовує початкове число для отримання нового числа, яке стає новим початковим. Потім нове початкове число може використовуватися для отримання наступного нового початкового числа і т. д. Щоб ця схема працювала, функція генерації випадкових чисел повинна запам'ятовувати початкове число,

яке застосовувалося при її останньому виклику. Саме тут і виникає потреба в статичній змінній.

Текст функції має наступний вигляд:

```
static unsigned long int next = 1;  // початкове число
int rand0(void)
{
    // магічна формула генерації псевдовипадкових чисел
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}
```

В розглянутій програмі статична змінна **next** починає зі значення **1** і змінюється за допомогою магічної формули при кожному виклику функції. Результатом буде значення, що повертається та знаходиться десь в діапазоні від **0** до **32767**. Зверніть увагу, що **next** є статичною змінною з внутрішнім зв'язуванням, а не просто статичною змінною без зв'язування. Справа в тому, що пізніше приклад буде розширений, щоб змінна **next** спільно використовувалася двома функціями в тому ж самому файлі.

Давайте протестуємо функцію **rand0()** за допомогою **C**-програми. Текст програми має наступний вигляд:

```
#include <stdio.h>
extern int rand0(void);
int main(void)
{
     int count;
     for(count = 0; count < 5; count++)
          printf("%6d\n", rand0());
     return 0;
}
```

Результат виконання програми наведено на рис. 2.6.

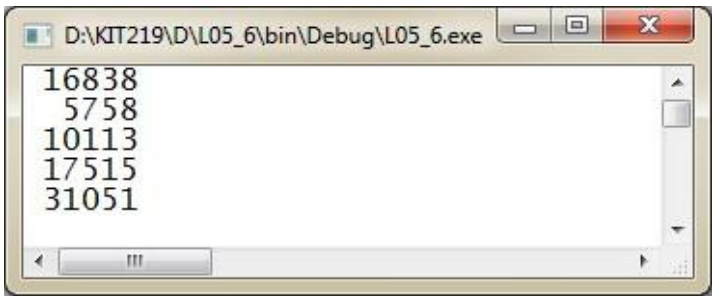


Рисунок 2.6 – Результат виконання програми для генерації псевдовипадкових чисел

Числа виглядають знайомими. В цьому й полягає аспект «псевдо». Кожного разу, коли головна програма запускається, старт відбувається з одного й того ж самого початкового числа `1`. Проблему можна обійти шляхом введення другої функції на ім'я `srand1()`, яка дозволить встановлювати наново початкове число. Трюк полягає в тому, щоб зробити `next` статичною змінною з внутрішнім зв'язуванням, яка відома лише функціям `rand1()` і `srand1()`. Еквівалент функції `srand1()` в бібліотеці `C` називається `srand()`. Додамо функцію `stand1()` до файлу, який містить `rand1()`.

Текст модифікованої програми буде мати наступний вигляд:

Файл `rand1.c`

```
static unsigned long int next = 1; // початкове число
int randl(void)
{
  // магічна формула генерації псевдовипадкових
  чисел      next = next * 1103515245 + 12345;
  return (unsigned int)(next / 65536) % 32768;
}
void srand1(unsigned int seed)
{
    next = seed;
}
```

Зверніть увагу, що `next` – це статична змінна з областю видимості в межах файлу і внутрішнім зв'язуванням. Це означає, що вона може використовуватися як `rand1()`, так і `srand1()`, але не функціями в інших файлах. Для тестування цих функцій застосуємо програму, текст якої має наступний вигляд:

Файл **main.c**

```
#include <stdio.h>
 #include <windows.h>
extern void srand1(unsigned int x);
extern int rand1(void);

int main(void)
{
    int count;
    unsigned seed;
    SetConsoleOutputCP(1251);
    printf("Введіть необхідне початкове число.\n");
    while(scanf("%u", &seed) == 1)
    {
        srand1(seed);   // нове визначення початкового числа
       for(count = 0; count < 5; count++)
           printf("%12d\n", rand1());
       printf("\nВведіть наступне початкове число "
              "(q для завершення): \n");
    }
    printf("Програма завершена.\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 2.7.

2.2.6. Виділення пам’яті: функції malloc( ) і free( )

Розглянуті класи зберігання мають одну спільну особливість. Після вибору класу зберігання автоматично з’являться рішення відносно області видимості та тривалості зберігання. Варіанти дотримуються попередньо укомплектованим правилам керування пам’яттю. Однак існує ще один варіант, який забезпечує більш високу гнучкість. Він передбачає застосування бібліотечних функцій виділення та керування пам’яттю.

Усі програми повинні резервувати простір пам’яті, який буде достатнім для зберігання даних, з якими вони працюють. Деякі операції з виділення пам’яті відбуваються автоматично. Наприклад, в результаті оголошень

```
float x;
char place[] = "Програмування на С";
```

резервується простір пам’яті, достатній для зберігання змінної **float** і рядку.

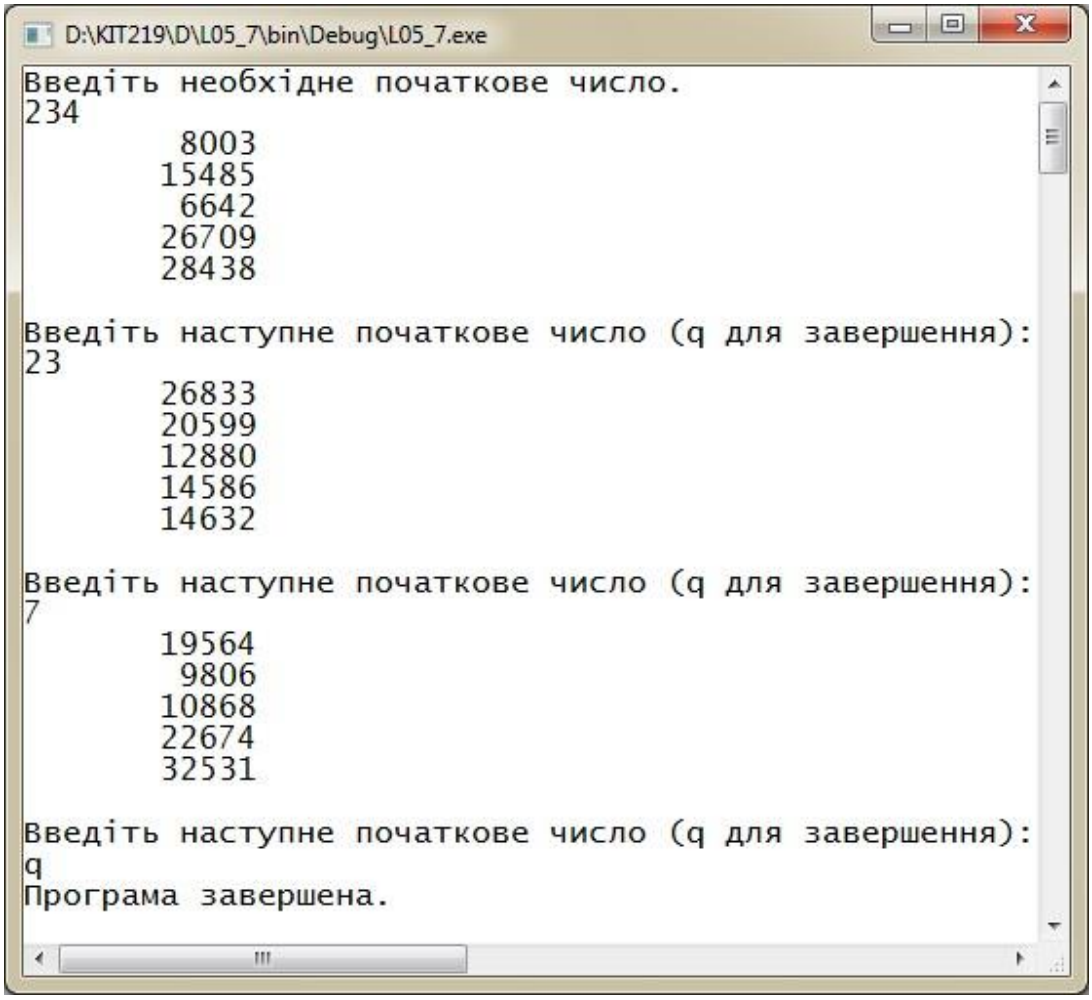


Рисунок 2.7 – Результат виконання модифікованої програми для генерації псевдовипадкових чисел

Можна також явно замовити певний об’єм пам’яті:

```
int plates[100];
```

Це оголошення резервує **100** комірок пам’яті, в кожній з яких можна зберігати значення **int**. В усіх показаних випадках оголошення також надає ідентифікатор виділеній пам’яті, тому для звернення до даних можна використовувати **x** або **place**. Раніше згадувалося, що пам’ять під статичні дані виділяється під час завантаження програми в пам’ять, а пам’ять під автоматичні дані виділяється, коли потік керування програми входить до блоку, і звільняється, коли потік керування залишає блок.

Мова C виходить за ці рамки. Під час виконання програми можна виділяти додаткову пам'ять. Основним інструментом є функція **malloc()**, яка приймає один аргумент: необхідну кількість байтів пам'яті. Потім **malloc()** шукає потрібний блок вільної пам'яті. Пам'ять буде анонімною, тобто функція **malloc()** виділяє блок пам'яті, але не призначає йому ім'я. Проте, вона повертає адресу першого байту в цьому блоці. Відповідно, ви можете присвоїти цю адресу змінній типу вказівника та застосовувати такий вказівник для доступу до пам'яті. Оскільки байт представлений типом **char**, функція **malloc()** традиційно була оголошена с типом вказівника на **char**. Однак в стандарті **ANSI C** використовується новий тип: вказівник на **void**. Цей тип задумувався як узагальнений вказівник.

Функція **malloc()** може застосовуватися для повернення вказівників на масиви, структури та ін., тому зазвичай значення, що повертається, приводиться до потрібного типу. В умовах стандарту **ANSI C** для ясності ви як і раніше повинні здійснювати приведення, але присвоювання значення вказівника на **void** вказівнику іншого типу не вважається конфліктом типів. Якщо функції **malloc()** не вдається знайти необхідний простір, вона повертає нульовий вказівник.

Давайте скористаємося **malloc()** для рішення задачі створення масиву.

За допомогою **malloc()** можна запросити блок пам'яті під час виконання програми. Крім того, знадобиться вказівник, щоб відстежувати, де в пам'яті знаходиться виділений блок. Наприклад, погляньте на наступний код:

```
double *ptd; ptd = (double *) malloc(30 *
sizeof(double));
```

Цей код робить запит на простір під **30** значень типу **double** і встановлює **ptd** для вказування на відповідну комірку пам'яті. Зверніть увагу, що **ptd** оголошено як вказівник на одиночне значення **double**, а не на блок з **30** значень **double**. Згадайте, що ім'я масиву являє собою адресу його першого елементу. Відповідно, якщо ви встановили **ptd** таким чином, щоб він вказував на перший елемент блоку, ви можете використовувати його подібно імені масиву. Тобто ви

можете застосовувати вираз **ptd[0]** для доступу до першого елемента блоку, **ptd[1]** – для доступу до другого елементу і т. д. Як було показано раніше, форму запису з вказівниками можна використовувати для імен масивів, а форму запису з масивами можна застосовувати для вказівників.

Тепер вам відомі три способи створення масиву:

1) Оголосити масив, використовуючи константні вирази для розмірностей, і застосовувати ім'я масиву для доступу до його елементів. Такий масив може бути створений з використанням або статичної, або автоматичної пам'яті.

2) Оголосити масив змінної довжини (**C99**), застосовуючи зміні вирази для розмірностей, і використовувати ім'я масиву для доступу до його елементів. Такий варіант є доступним тільки для автоматичної пам'яті.

3) Оголосити вказівник, викликати функцію **malloc()**, присвоїти вказівнику значення, що повертається, та застосовувати вказівник для доступу до елементів масиву. Цей вказівник може бути або статичним, або автоматичним.

Другий і третій методи можна використовувати для виконання того, що не вдасться зробити зі звичайним оголошеним масивом – створити динамічний масив, пам'ять під який виділяється під час виконання програми, і тоді ж обрати його розмір. Припустимо, наприклад, що **n** – цілочислова змінна. До виходу стандарту **C99** неможна було робити наступним чином:

```
double item[n];
```

Однак можна було записувати (навіть у випадку компілятора, що був випущений до виходу **C99**) наступним чином:

```
ptd = (double *) malloc(n * sizeof(double));    // нормально
```

Цей прийом працює і має дещо більшу гнучкість, ніж масив змінної довжини.

Зазвичай ви повинні компенсувати кожен випадок виклику **malloc()** викликом **free()**. Функція **free()** приймає в якості аргументу адресу, яка була повернена раніше функцією **malloc()**, і звільняє пам'ять, яка була виділена. Таким чином, тривалість існування виділеної пам'яті розраховується з моменту,

коли була викликана функція **malloc()** для виділення пам'яті, і до моменту, коли викликається функція **free()** з метою звільнення пам'яті для її повторного використання. Функції **malloc()** і **free()** можна розглядати як інструменти для керування пулом пам'яті. Кожен виклик **malloc()** виділяє пам'ять для застосування програмою, а кожен виклик **free()** відновлює пам'ять в пулі таким чином, що вона може повторно використовуватися. Аргументом **free()** повинен бути вказівник на блок пам'яті, виділений **malloc()**. Функцію **free()** неможна застосовувати для звільнення пам'яті, що була виділена іншими засобами, такими як оголошення масиву. Функції **malloc()** і **free()** мають прототипи в файлі заголовку **stdlib.h**.

За рахунок використання **malloc()** програма може вирішувати, масив якого розміру нам потрібен, і створювати його під час виконання. Ця можливість демонструється в наступній програмі. В ній вказівнику **ptd** присвоюється адреса блоку пам'яті, після чого **ptd** застосовується ніби це ім'я масиву. Якщо виділити необхідну пам'ять не вдалося, для припинення роботи програми викликається функція **exit()**, прототип якої також міститься в **stdlib.h**. Значення **EXIT_FAILURE** визначено в тому ж самому файлі заголовку. Стандарт надає два значення, що повертаються, які гарантовано розпізнають усі операційні системи: **EXIT_SUCCESS** (еквівалентно значенню **0**) для позначення нормального завершення програми і **EXIT_FAILURE** – для позначення аварійного завершення. Деякі операційні системи, включаючи **Unix**, **Linux** і **Windows**, можуть приймати додаткові цілочислові значення, що позначають конкретні форми відмови.

Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>       // для malloc(), free(), exit()
int main(void)
{
    double *ptd;
    int max = 0;
    int number;
    int i = 0;
```

```
    SetConsoleOutputCP(1251);
    puts("Введіть максимальну кількість елементів типу double.");
if(scanf("%d", &max) != 1)
    {
     puts("Введена кількість є некоректною. Програма завершена.");
     exit(EXIT_FAILURE);
    }
    ptd = (double *) malloc(max * sizeof(double));
    if(ptd == NULL)
    {
    puts("Не вдалося виділити пам'ять. Програма завершена.");
    exit(EXIT_FAILURE);
    }
    // ptd тепер вказує на масив з max елементів
puts("Введіть значення (q для виходу):");
while(i < max && scanf("%lf", &ptd[i]) == 1)
        ++i;
printf("Введено %d елементів:\n", number = i);
for(i = 0; i < number; i++)
   {
     printf("%7.2f", ptd[i]);
     if(i % 7 == 6)
     putchar('\n');
   }
        if(i % 7 != 0)
    putchar('\n');
    puts("Програма завершена.");
    free(ptd);
    return 0;
}
```

Результат виконання програми для виділення пам’яті під масив значень типу **double** наведено на рис. 2.8.

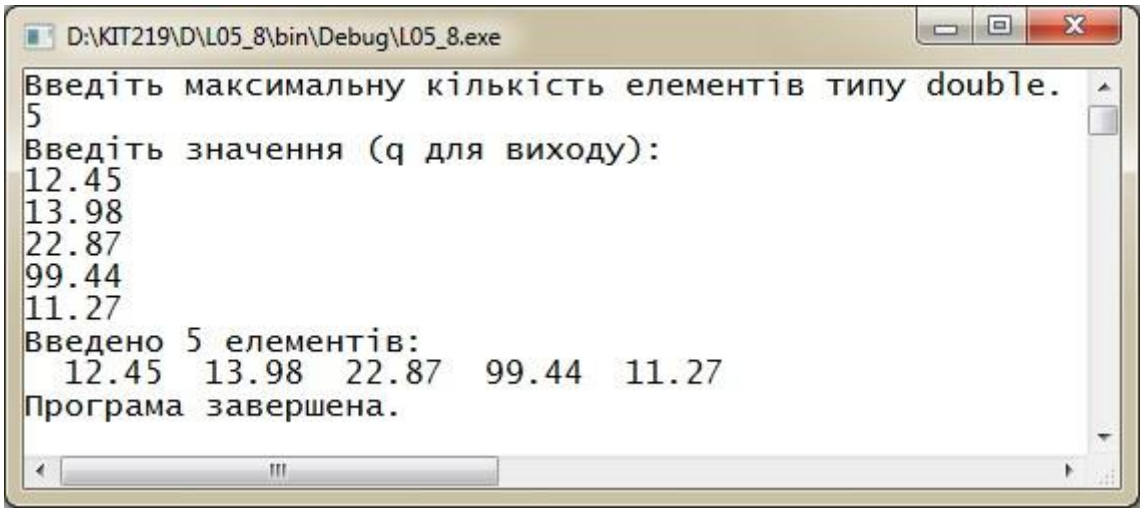


Рисунок 2.8 – Результат виконання програми виділення динамічної пам’яті для масиву за допомогою функції **malloc()**

Давайте розглянемо код програми, яка отримує необхідний розмір масиву за допомогою наступних рядків:

```
if(scanf("%d", &max) != 1)
{
    puts("Кількість введена некоректно — програма завершена.");
    exit(EXIT_FAILURE);
}
```

Показаний нижче рядок коду виділяє в пам'яті простір, достатній для зберігання необхідної кількості елементів, і потім присвоює адресу цього блоку вказівнику **ptd**:

```
ptd = (double *) malloc(max * sizeof(double));
```

Приведення до `(double *)` є необов'язковим в **C**, але потрібне в **C++**, тому використання приведення типу спрощує перенесення програми з **C** в **C++**.

Цілком ймовірно, що функція **malloc()** не зможе надати необхідний об'єм пам'яті. В цьому випадку вона повертає нульовий вказівник, і виконання програми припиняється:

```
if(ptd == NULL)
{
puts("Не вдалося виділити пам'ять. Програма завершена.");
exit(EXIT_FAILURE);
}
```

Якщо програма подолає цю перешкоду, вона може трактувати **ptd** як ім'я масиву з **max** елементів.

Зверніть увагу на виклик функції **free()** ближче до кінця програми. Вона звільняє пам'ять, яка була виділена **malloc()**. Функція **free()** звільняє тільки блок пам'яті, на який вказує її аргумент. Деякі операційні системи будуть звільняти виділену пам'ять автоматично при завершенні програми, але інші можуть цього не робити. Таким чином, застосовуйте **free()** і не сподівайтеся на те, що операційна система виконає очищення пам'яті замість вас.

Яку ж користь ми отримали з того, що скористалися динамічним масивом? В цьому випадку ми збільшили гнучкість програми. Припустимо, вам відомо, що більшість часу програмі буде потрібно не більше **100** елементів, але іноді вона

буде мати потребу в 10000 елементів. Якщо ви оголошуєте масив, то повинні враховувати найгірший випадок і оголосити його з 10000 елементів. Більшу частину часу програма буде витрачати пам'ять дарма. До того ж, якщо настане момент, коли знадобиться мати 10001 елемент, програма отримає відмову. Застосування динамічного масиву дозволяє програмі підлаштовуватися під існуючі обставини.

2.2.7. Важливість функції free()

Об'єм статичної пам'яті фіксується під час компіляції. Він не змінюється протягом виконання програми. Об'єм пам'яті, що використовується для автоматичних змінних, зростає і зменшується автоматично впродовж часу виконання програми. Однак якщо ви забудете викликати функцію free(), то об'єм виділеної пам'яті буде тільки зростати. Наприклад, припустимо, що існує функція, яка створює тимчасову копію масиву. Схематично це продемонстровано в наступному коді:

```
... int
main(void)
{
      double glad[2000];
      int i;
      for(i = 0; i < 1000; i++)
          gobble(glad, 2000);
      ...
}
void gobble(double ar[], int n)
{
   double *temp = (double *)malloc(n * sizeof(double));
    ... // free(temp); // забули скористатися free()
}
```

Коли функція gobble() викликається в перший раз, вона створює вказівник temp і застосовує malloc() для виділення 16000 байтів пам'яті (виходимо з припущення, що тип double займає 8 байтів). Уявимо, що ми не викликали free(), як показано в коді. Коли gobble() завершиться, вказівник

temp, будучі автоматичною змінною, зникає. Але **16000** байтів пам'яті, на які він вказував, як і раніше існують. Доступ до цієї пам'яті є неможливим, оскільки ми більше не маємо її адреси. Вона не може використовуватися повторно, тому що не була викликана функція **free()**.

Коли функція **gobble()** викликається другий раз, вона знову створює **temp** і викликає **malloc()** для виділення **16000** байтів пам'яті. Перший блок з **16000** байтів є недоступним, тому функція **malloc()** повинна знайти другий блок розміром **16000** байтів. Коли функція завершиться, цей блок пам'яті також стає недоступним і не може використовуватися повторно.

Цикл повторюється **1000** разів, і до часу його завершення з пулу пам'яті буде вилучено **16000000** байтів. Насправді програмі може просто не вистачати пам'яті, щоб зайти настільки далеко. Проблема подібного роду називається **витоком пам'яті**. Її можна запобігти за рахунок наявності виклику **free()** наприкінці функції.

2.2.8. Функція calloc()

Ще один спосіб виділення пам'яті передбачає застосування функції **calloc()**. Нижче показаний типовий випадок її використання:

```
long *newmem;
newmem = (long *) calloc(100, sizeof(long));
```

Подібно **malloc()**, функція **calloc()** повертає вказівник на **char** у своїй версії до виходу стандарту **ANSI** і вказівник на **void** в умовах дії стандарту **ANSI**. Якщо треба мати інший тип, ви повинні застосовувати приведення. Ця нова функція приймає два аргументи, які повинні бути цілими числами без знаку (типу **size_t** в **ANSI**). Перший аргумент задає необхідну кількість комірок пам'яті. Другий аргумент задає розмір кожної комірки в байтах. В нашому випадку тип **long** використовує **4** байти, тому вказаний вище оператор встановлює **100** одиниць по **4** байти, задіявши в цілому **400** байтів для зберігання даних.

Застосування `sizeof(long)` замість `4` робить код більш мобільним. Він буде працювати в системах, де тип `long` має розмір, який відрізняється від `4`.

Функція `calloc()` має ще одну властивість: вона встановлює в `0` усі біти в блоці. Функція `free()` може також використовуватися для звільнення пам'яті, що була виділена за допомогою `calloc()`.

Динамічний розподіл пам'яті є ключовим засобом у багатьох розвинутих технологіях програмування. Цілком ймовірно, що ваша бібліотека `C` пропонує ряд інших функцій керування пам'яттю, при цьому частину з них можна переносити до іншої системи, а частину – ні.

2.2.9. Динамічний розподіл пам'яті та масиви змінної довжини

Функціональність масивів змінної довжини і `malloc()` дещо перетинається. Наприклад, обидва засоби можуть застосовуватися для створення масиву, розмір якого визначається під час виконання:

```
int vlamal()
{
   int n;
   int *pi;
   scanf("%d", &n);
   pi = (int *)malloc(n * sizeof(int));

     int ar[n];                  // масив змінної довжини
     pi[2] = ar[2] = -5;
     ...
}
```

Відмінність полягає в тому, що масив змінної довжини є автоматичною пам'яттю. Як наслідок – пам'ять, яку займає масив змінної довжини, звільняється автоматично, коли потік керування залишає блок, в якому масив був визначений (в нашому випадку при завершенні функції `vlamal()`). Таким чином, ви не повинні турбуватися про виклик `free()`. З іншого боку, доступ до масиву, що створений з застосуванням `malloc()`, не обмежується однією функцією.

Наприклад, функція може створювати масив і повертати вказівник, надаючи доступ до нього функції, що викликає. Завершивши роботу з масивом, функція, що викликає, може викликати **free()**. Не буде помилкою, якщо при виклику **free()** задати вказівник, що відрізняється від того, що використовується у виклику **malloc()**: треба тільки, щоб вказівники містили одну й ту ж саму адресу. Однак ви не повинні намагатися звільнити один і той самий блок пам'яті двічі.

Масиви змінної довжини є більш зручними для організації багатовимірних масивів. Ви можете створити двовимірний масив з застосуванням **malloc()**, але синтаксис буде досить незграбним. Якщо компілятор не підтримує масиви змінної довжини, одна з розмірностей повинна бути зафіксована, як у викликах функції:

```
int n = 5;
int m = 6;
int ar2[n][m];  // масив змінної довжини n × m
int (*p2)[6];   // працює до виходу стандарту C99
int (*p3)[m];   // необхідна підтримка масивів змінної довжини
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int));  // масив n * 6
p3 = (int (*)[m]) malloc(n * m * sizeof(int));  // масив n * m

// попередній вираз також потребує підтримки
// масивів змінної довжини
ar2[1][2] = p2[1][2] = 12;
```

Корисно поглянути на оголошення вказівників. Функція **malloc()** повертає вказівник, оскільки **p2** повинен бути вказівником потрібного типу. Оголошення

```
int (*p2)[6];   // працює до виходу стандарту C99
```

говорить про те, що **p2** вказує на масив з шести елементів **int**. Це означає, що **p2[i]** буде розумітися як елемент, що складається з шести значень **int**, а **p2[i][j]** – це одиночне значення **int**.

В другому оголошенні вказівника використовується змінна для повідомлення розміру масиву, на який посилається **p3**. Це означає, що **p3** трактується як вказівник на масив змінної довжини, тому даний код не буде працювати в рамках стандарту **C90**.

2.2.10. Класи зберігання та динамічний розподіл пам’яті

Вас може цікавити, який зв’язок існує між класами зберігання та динамічним розподілом пам’яті. Давайте поглянемо на ідеалізовану модель. Ви можете думати про доступну пам’ять програми як таку, що має три окремих області для:

- статичних змінних з зовнішнім зв’язуванням, внутрішнім зв’язуванням і без зв’язування;
- автоматичних змінних;
- пам’яті, що виділяється динамічно.

Об’єм пам’яті, який є необхідним для класів зі статичною тривалістю зберігання, відомий на етапі трансляції, і дані, які зберігаються в цій області, доступні протягом часу виконання програми. Кожна змінна цих класів з’являється, коли програма запускається, та зникає при її завершенні.

Однак автоматична змінна починає існувати, коли потік керування входить до блоку коду, що містить визначення змінної, та зникає після покидання цього блоку. Отже, під час викликів програмних функцій та їх завершенні, об’єм пам’яті, який був задіяний під автоматичні змінні, зростає і зменшується. Така область пам’яті зазвичай реалізована у вигляді стеку. Це означає, що нові змінні додаються до пам’яті послідовно, в порядку їх створювання, а видаляються в зворотному порядку, коли зникають.

Пам’ять, яка виділяється автоматично, з’являється при виклику `malloc()` або спорідненої до неї функції і звільняється при виклику `free()`. Постійність пам’яті керується програмістом, а не будь-яким набором жорстких правил, тому блок пам’яті може бути створений в одній функції та звільнятися в іншій. За цією причиною область пам’яті, що застосовується для динамічного розподілу пам’яті, може стати фрагментованою, тобто ділянки, що не використовуються, будуть іти вперемiш з активними блоками пам’яті. Крім того, використання динамічної пам’яті має тенденцію бути більш повільним процесом, ніж робота зі стековою пам’яттю.

Зазвичай програма застосовує різні області пам’яті для статичних об’єктів, автоматичних об’єктів і об’єктів, що виділяються динамічно. Сказане демонструється в тексті наступної програми.

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <string.h>

int static_store = 30;
const char *peg = "Рядковий літерал";
int main(void)
{
    int auto_store = 40;
    char auto_string[] = "Автоматичний масив char";
    int *pi;
    char *pel;
    SetConsoleOutputCP(1251);
    pi = (int *)malloc(sizeof(int));
    *pi = 35;
    pel = (char *)malloc(strlen("Динамічний рядок") + 1);
strcpy(pel, "Динамічний рядок");
printf("=============================================\n");
printf("static_store:           %d за адресою %p\n",
       static_store, &static_store);
printf("auto_store:             %d за адресою %p\n",
       auto_store, &auto_store);
printf("*pi:                  %d за адресою %p\n", *pi, pi);
printf("=============================================\n");
printf("%s           за адресою %p\n", peg, peg);
printf("%s    за адресою %p\n", auto_string, auto_string);
printf("%s           за адресою %p\n", pel, pel);
printf("%s             за адресою %p\n",
        "Рядок в лапках", "Рядок в лапках");
printf("=============================================\n\n");
free(pi);
free(pel);
return 0;
}
```

Результат роботи програми наведено на рис. 2.9.

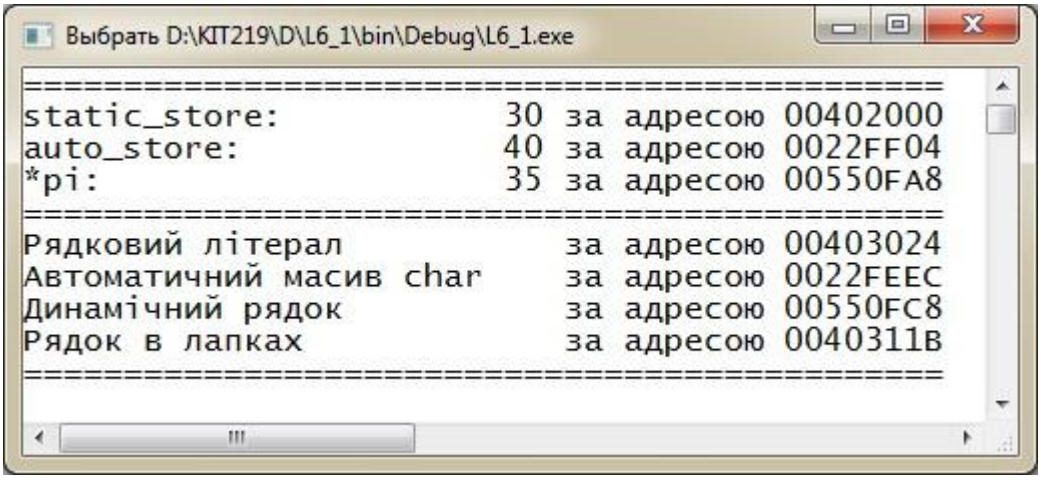


Рисунок 2.9 – Результат роботи програми, яка демонструє виділення областей пам’яті для статичних та автоматичних об’єктів, а також об’єктів, що виділяються динамічно

Як можна побачити, статичні дані, включаючи рядкові літерали, займають одну область пам’яті, автоматичні дані – другу область, а дані, що виділяються динамічно, – третю область, яка часто називається **кучею** або **вільним сховищем**.

2.2.11. Кваліфікатори типів `ANSI C`

Вам вже відомо, що змінна характеризується типом і класом зберігання. В стандарті `C90` були додані ще дві властивості: **постійність** і **непостійність**. Ці властивості оголошуються за допомогою ключових слів `const` і `volatile`, які створюють **кваліфіковані типи**. В стандарті `C99` з’явився третій кваліфікатор, `restrict`, що призначений для підтримки компілятора при оптимізації. Нарешті, в `C11` додано четвертий кваліфікатор – `_Atomic`. Стандарт `C11` надає додаткову бібліотеку, яка керується `stdatomic.h` для підтримки паралельного програмування, і `_Atomic` є частиною цієї необов’язкової підтримки.

Стандарт `C99` наділяє кваліфікатори типів новою властивістю – тепер вони ідемпотентні. Це означає, що один і той самий кваліфікатор можна вказувати в оголошенні декілька разів, і надмірні кваліфікатори ігноруються:

```
const const const int n = 6;
// те ж саме, що й const int n = 6;
```

Це робить прийнятною, наприклад, наступну послідовність:

```
typedef const int zip;
const zip q = 8;
```

2.2.12. Кваліфікатор типу const

Раніше ви вже зустрічалися з випадками використання **const**. В якості нагадування: ключове слово **const** в оголошенні створює змінну, значення якої не може бути змінено шляхом присвоювання або інкрементування/декрементування. У випадку компілятора, сумісного зі стандартом **ANSI**, код

```
const int nochange;          // вказує, що m є константою
          nochange = 12;     // не дозволено
```

призводить до появи повідомлення про помилку. Проте, ініціалізувати змінну **const** можна. Таким чином, наступний код є припустимим:

```
const int nochange = 12;     //  все добре
```

Попереднє оголошення робить **nochange** змінною тільки для читання. Після того, як вона ініціалізована, змінювати її неможна.

Ключове слово **const** можна застосовувати, наприклад, для створення масиву даних, які в програмі не можуть змінюватися:

```
const int daysl[12] = { 31, 28, 31, 30, 31, 30,
                        31, 31, 30, 31, 30, 31 };
```

2.2.13. Використання const з оголошеннями вказівників і параметрів

Застосовувати ключове слово **const** при оголошенні простої змінної та масиву досить просто. Вказівники в цьому сенсі є більш складними, оскільки необхідно проводити різницю між оголошенням самого вказівника як **const** і перетворенням у **const** значення, на яке він вказує. Оголошення

```
const float *pf;    // pf вказує на константне значення float
```

створює вказівник **pf**, що посилається на значення, яке повинне залишатися

постійним. Значення самого **pf** можна змінювати. Наприклад, його можна встановити для вказування на інше значення **const**.

На противагу цьому, оголошення

```
float * const pt;     // pt є вказівником const
```

говорить про те, що значення самого вказівника **pt** не може бути модифіковане.

Він повинен завжди посилатися на одну й ту саму адресу, однак значення, на яке він вказує, може змінюватися. Нарешті, оголошення **const float * const** `ptr;` означає, що **ptr** повинен завжди вказувати на одну й ту ж саму комірку, а значення, що зберігається в цій комірці, не повинне змінюватися.

Існує і третє місце, куди можна помістити

```
const: float const * pfc;    // те ж саме, що й const float * pfc;
```

Як можна побачити в коментарі, розташування **const** після імені типу і попереду символу ***** означає, що вказівник не може використовуватися для зміни значення, на яке він посилається. Висловлюючись коротко, ключове слово **const**, що знаходиться де завгодно ліворуч від символу *****, робить константними дані, а праворуч – робить константним сам вказівник.

Поширеним використанням цього нового ключового слова є оголошення вказівників, які служать формальними параметрами функцій. Наприклад, нехай існує функція на ім'я **display()**, яка відображає вміст масиву. Щоб застосувати дану функцію, необхідно передати їй в якості фактичного аргументу ім'я масиву, але ім'я масиву одночасно являє собою і його адресу. Це дозволить функції змінювати дані з функції, що викликає. Однак наступний прототип запобігає такій зміні:

```
void display(const int array[], int limit);
```

В прототипі та в заголовку функції оголошення параметру **const int** `array[]` аналогічно оголошенню **const int** `* array`, і перше оголошення говорить про те, що дані, на які вказує **array**, не можуть бути змінені.

Цієї практики дотримується бібліотека **ANSI C**. Якщо вказівник використовується тільки для надання функції доступу до значень, він оголошується як вказівник на кваліфікований тип за допомогою **const**. Якщо

вказівник застосовується для зміни даних у функції, що викликає, то ключове слово **const** не використовується. Наприклад, оголошення функції **strcat()**, прийняте в **ANSI C**, має такий вигляд:

```
char *strcat(char * restrict s1, const char * restrict s2);
```

Згадайте, що функція **strcat()** додає копію другого рядка в кінець першого рядка. Це призводить до модифікації першого рядка, але залишає другий рядок незмінним.

2.2.14. Використання const з глобальними даними

Згадайте, що застосування глобальних змінних вважається ризикованим підходом, оскільки дані відкриті для помилкової зміни будь-якої частини програми. Такий ризик зникає, якщо дані є константними, так що цілком розумно позначати глобальні змінні кваліфікатором **const**. Можна мати змінні **const**, масиви **const** і структури **const**.

Однак необхідно дотримуватися обережності при спільному використанні константних даних в різних файлах. Для цього передбачені дві стратегії. Перша полягає в дотриманні правил, які застосовуються по відношенню до зовнішніх змінних – використання визначальних оголошень в одному файлі та посилальних оголошень (з ключовим словом **extern**) в інших файлах:

```
// file1.c – визначення декількох глобальних констант

const double PI = 3.14159;
const char * MONTHS [12] = {
   "Січень",   "Лютий",   "Березень", "Квітень",
   "Травень",  "Червень", "Липень",   "Серпень",
   "Вересень", "Жовтень", "Листопад", "Грудень" };

// file2.c – використання констант, що визначені десь
// в іншому місці

extern const double PI;
extern const * MONTHS[];
```

Другий підхід передбачає розміщення констант у файлі, що підключається. В даному випадку доведеться вжити додаткову дію, пов'язану із застосуванням статичного зовнішнього класу зберігання:

```
// constant.h – визначення декількох глобальних констант

static const double PI = 3.14159;
static const char * MONTHS [12] = {
   "Січень",   "Лютий",   "Березень", "Квітень",
   "Травень",  "Червень", "Липень",   "Серпень",
   "Вересень", "Жовтень", "Листопад", "Грудень" };

// file1.c – використання констант, що визначені
// десь в іншому місці
#include "constant.h"

// file2.c – використання констант, що визначені
// десь в іншому місці
#include "constant.h"
```

Якщо ви не вкажете ключове слово **static**, то включення файлу заголовку **constant.h** в **file1.c** і **file2.c** призведе до того, що кожен файл буде мати визначальне оголошення того ж самого ідентифікатора, що стандартом **ANSI** не підтримується. Проте, деякі компілятори це дозволяють. Зробивши ідентифікатор зовнішнім і статичним, ви фактично надаєте кожному файлу окрему копію даних. Такий прийом не буде працювати, якщо за замислом файли повинні використовувати ці дані для зв'язку один з одним, тому що кожен файл буде бачити тільки свою копію даних. Однак оскільки дані є константними (через наявність ключового слова **const**) та ідентичними (оскільки обидва файли включають той самий файл заголовку), проблеми не виникають.

Перевага підходу з файлом заголовку полягає в тому, що ви не зобов'язані пам'ятати про застосування визначальних оголошень в одному файлі та посилальних оголошень в іншому. Усі файли просто включають той самий файл заголовку. Недолік пов'язаний з тим, що дані дублюються. В попередніх

прикладах це не призводить до значної проблеми, але вона може виникнути, якщо до складу константних даних входять великі масиви.

2.2.15. Кваліфікатор типу volatile

Кваліфікатор **volatile** повідомляє компілятору, що змінна може мати значення, яке змінюється діями, зовнішніми по відношенню до програми. Він зазвичай вказується для апаратних адрес і для даних, які спільно використовуються з іншими програмами або потоками, що виконуються одночасно. Наприклад, адреса може посилатися на поточне показання системного годинника. Значення за цією адресою змінюється разом зі зміною показань часу незалежно від того, що робить програма, або ж адреса може застосовуватися для отримання інформації, яка передається, скажімо, з іншого комп'ютера.

Синтаксис цього кваліфікатора подібний до синтаксису **const**:

```
volatile int loc1;    // loc1 є коміркою, що змінюється
volatile int *ploc;   // ploc вказує на комірку, що змінюється
```

Ці оператори оголошують **loc1** як значення **volatile** і **ploc** як вказівник на значення **volatile**.

Концепція кваліфікатора **volatile** є досить цікавою. Чому саме комітет **ANSI** вирішив зробити **volatile** ключовим словом. Причина в тому, що воно полегшує проведення оптимізації компілятором. Припустимо, наприклад, що є такий код:

```
vai1 = x;
// код, в якому x не використовується
val2 = x;
```

Інтелектуальний компілятор (той, що оптимізує) може помітити, що об'єкт **x** використовується два рази без зміни в проміжку його значення. Він тимчасово може зберегти значення **x** в регістрі. Потім, коли **x** знадобиться для **val2**, з'явиться можливість зекономити час, прочитавши значення з регістру, а не з початкової комірки пам'яті. Така процедура називається **кешуванням**. Зазвичай кешування є корисною оптимізацією, але не у випадку, коли значення **x**

змінюється в проміжку між двома операторами будь-якою іншою дією. Без ключового слова **volatile** у компілятора немає жодних засобів, щоб з'ясувати, чи може це відбутися. Відповідно, щоб уникнути помилки, компілятор не міг реалізувати кешування. Так було до виходу стандарту **ANSI**. Однак тепер, якщо в оголошенні відсутнє ключове слово **volatile**, компілятор може припустити, що значення не змінюється між двома його застосуваннями, та спробувати оптимізувати даний код.

Значення може бути одночасно і **const**, і **volatile**. Наприклад, значення апаратного годинника зазвичай не повинне змінюватися програмою, що робить його **const**, але може бути змінено зовнішню дією, тому воно є **volatile**. Просто додайте обидва кваліфікатора в оголошення, як показано нижче. Порядок їх розташування ролі не грає:

```
volatile const int loc;
const volatile int *ploc;
```

2.2.16. Кваліфікатор типу restrict

Ключове слово **restrict** розширює обчислювальну підтримку, дозволяючи компілятору оптимізацію певних варіантів коду. Воно може бути застосоване тільки до вказівників і повідомляє про те, що той або інший вказівник являє собою єдиний первинний засіб доступу до об'єкту даних. Щоб зрозуміти, чому це корисно, необхідно розглянути декілька прикладів. Погляньте на показані нижче оголошення:

```
int ar[10];
int * restrict restar = (int *)malloc(10 * sizeof(int));
int * par = ar;
```

В них вказівник **restar** є єдиним первинним засобом доступу до пам'яті, що виділена **malloc()**. Відповідно, він може бути кваліфікований за допомогою ключового слова **restrict**. Однак вказівник **par** не є ні первинним, ні єдиним засобом доступу до даних в масиві **ar**, тому він не може бути кваліфікований як **restrict**.

Тепер розглянемо дещо штучний приклад, в якому **n** має тип **int**:

```
for(n = 0; n < 10; n++)
{    par[n] += 5;
     restar[n] += 5;
     ar[n] *= 2;
     par[n] += 3;
     restar[n] += 3;
}
```

Знаючи, що вказівник **restar** – єдиний первинний засіб доступу до блоку даних, на який він посилається, компілятор може замінити два оператора, в яких задіяний **restar**, одним оператором, який дає той самий результат:

```
restar[n] += 8;     // коректна заміна
```

Однак зведення до одного двох операторів, в яких бере участь par, призводить до обчислювальної помилки:

```
par[n]  += 8;       // дає неправильну відповідь
```

Причина отримання неправильної відповіді пов'язана з тим, що всередині циклу **ar** використовується для зміни значення даних між двома випадками доступу до тих самих даних за допомогою **par**.

Без ключового слова **restrict** компілятор повинен розраховувати на найгірший випадок, а саме – на те, що будь-який інший ідентифікатор міг змінити дані між двома застосуваннями вказівника. За наявності **restrict** компілятор отримує свободу в пошуку обчислювальних скорочень.

Ключове слово **restrict** можна використовувати в якості кваліфікатора для параметрів функції, які є вказівниками. Це означає, що компілятор може припустити, що всередині тіла функції дані, які вказуються такими параметрами, не модифікуються за допомогою інших ідентифікаторів, і є можливість спробувати оптимізації, які в іншому випадку не застосовувалися. Наприклад, бібліотека **C** містить дві функції для копіювання байтів з одного місця в інше. В стандарті **C99** вони мають наступні прототипи:

```
void *memcpy(void *restrict s1, const void *restrict s2,
             size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

Кожна функція копіює **n** байтів з місця розташування **s2** в місце розташування **s1**. Функція **memcpy()** потребує, щоб між ними не було

перекриття, але для функції **memmove()** така вимога відсутня. Оголошення **s1** і **s2** як **restrict** означає, що кожен вказівник є єдиним засобом доступу, тому вони не можуть звертатися до одного й того ж самого блоку даних. Це відповідає вимозі відсутності перекриття. Функція **memmove()**, яка дозволяє перекриття, при копіюванні даних повинна дотримуватися більшої обережності, щоб не перезаписати дані до того, як вони будуть використані.

Ключове слово **restrict** повідомляє компілятор про те, що він може робити певні припущення, що стосуються оптимізації. Крім того, **restrict** говорить користувачу про те, що повинні застосовуватися тільки аргументи, які задовольняють вимогам **restrict**. В загальному випадку компілятор не здатний з'ясувати, чи дотримуєтесь ви цього обмеження, але ви берете на себе ризик за його ігнорування.

2.2.17. Кваліфікатор типу _Atomic

Паралельне програмування поділяє виконання програми на потоки, які можуть виконуватися паралельно. При цьому виникають складні задачі програмування, до числа яких входить керування різними потоками, які отримують доступ до одним і тих самих даних.

В **C11** надані (необов'язкові) методи керування, які організовані у вигляді необов'язкових файлів **stdatomic.h** і **threads.h**. Одним з аспектів є концепція атомарного типу, для якого доступ керується різними функціональними макросами. Поки потік виконує атомарну операцію над об'єктом атомарного типу, інші потоки не будуть мати доступу до цього об'єкту. Наприклад, код

```
int hogs;                       // звичайне оголошення
hogs = 12;                      // звичайне присвоювання
```

можна було б замінити наступним чином:

```
_Atomic int hogs;               // hogs - атомарна змінна
atomic_store(&hogs, 12);        // макрос з stdatomic.h
```

Тут зберігання значення **12** в **hogs** являє собою атомарний процес, протягом якого інші потоки не будуть мати доступу до **hogs**.

2.2.18. Нові місця для старих ключових слів

Стандарт **C99** дозволяє поміщати кваліфікатори типу та кваліфікатори класу зберігання **static** в початкові квадратні дужки формального параметра в прототипі та заголовку функції. У випадку кваліфікаторів типу це надає альтернативний синтаксис для існуючої можливості. Наприклад, розглянемо оголошення зі старим синтаксисом:

```
// старий стиль
void ofmouth(int * const a1, int * restrict a2, int n);
```

Оголошення говорить про те, що **a1** – вказівник **const** на **int**, і це означає, що константним є сам вказівник, але не дані, на які він вказує. Крім того, оголошення підкреслює, що **a2** являє собою вказівник **restrict**. Еквівалентний новий синтаксис виглядає наступним чином:

```
// дозволено стандартом C99 void ofmouth(int
a1[const], int a2[restrict], int n);
```

По суті нове правило дозволяє використовувати ці два кваліфікатора або з формою запису з вказівниками, або з формою запису з масивами в оголошенні параметрів функції.

Випадок зі **static** відрізняється, оскільки він вводить нове та незв'язане застосування для цього ключового слова. Замість зазначення області видимості або зв'язування для змінної зі статичним класом зберігання нове використання повідомляє компілятору, як формальний параметр буде застосовуватися. Наприклад, погляньте на наступний прототип:

```
double stick(double ar[static 20]);
```

Таке використання **static** означає те, що фактичний аргумент у виклику функції буде вказівником на перший елемент масиву, який має, принаймні, **20** елементів. Мета полягає в тому, щоб дозволити компілятору застосувати цю інформацію для оптимізації його кодування функції. Навіщо використовувати дане ключове слово в такій манері? Комітет зі стандартів **C** досить неохоче йде на створення нових ключових слів, оскільки це може зробити недійсними старі

програми, в яких такі слова застосовувалися в якості ідентифікаторів, тому якщо вдається реалізувати нове використання будь-якого старого ключового слова, то так вони й вчинять.

Ключове слово `static` повідомляє компілятор про можливість робити певні припущення щодо оптимізації, а користувачу вказує на необхідність надавати тільки такі аргументи, які задовольняють вимогам `static`.

2.2.19. Ключові поняття керування пам'яттю

Мова `C` пропонує декілька моделей керування пам'яттю. Ви повинні ознайомитися з усіма різними варіантами та виробити критерії, коли обирати той або інший тип. Більшу частину часу найкращим вибором буде автоматична змінна. Якщо ви приймете рішення застосовувати інший тип, то для цього потрібна вагома причина. Під час взаємодії між функціями зазвичай краще за все використовувати автоматичні змінні, параметри функцій та значення, що повертаються, а не глобальні змінні. З іншого боку, глобальні змінні є особливо зручними для представлення константних даних.

Ви повинні розуміти властивості статичної пам'яті, автоматичної пам'яті та виділеної пам'яті. Зокрема, майте на увазі, що обсяг застосовуваної статичної пам'яті визначається на етапі компіляції, а статичні дані завантажуються в пам'ять при завантаженні програми. Пам'ять під автоматичні змінні виділяється та звільняється під час виконання, тому об'єм пам'яті, що займають автоматичні змінні, протягом виконання програми змінюється. Автоматичну пам'ять можна представляти як робочий простір, який весь час перезаписується. Виділена пам'ять збільшується та зменшується в об'ємі, але в цьому випадку процес керується викликами функцій, а не відбувається автоматично.

Пам'ять, що задіяна під зберігання даних в програмі, може бути охарактеризована *тривалістю зберігання*, *областю видимості* та *зв'язуванням*.

Тривалість зберігання буває *статичною*, *автоматичною* або *виділеною*. *При статичній тривалості зберігання* пам'ять виділяється на початку виконання програми і залишається зайнятою протягом часу виконання програми.

Якщо тривалість зберігання є автоматичною, то пам'ять під змінну виділяється, коли потік керування програми входить до блоку, в якому змінна визначена, і звільняється, коли керування залишає цей блок. *У випадку виділеної тривалості зберігання* пам'ять виділяється викликом `malloc()` (або спорідненої функції) і звільняється викликом `free()`.

Область видимості визначає, які частини програми можуть мати доступ до даних. Змінна, яка визначена поза будь-яких функцій, має область видимості в межах файлу і видна для будь-якої функції, яка визначена після оголошення цієї змінної. Змінна, яка визначена всередині блоку або в якості параметру функції, має область видимості в межах блоку і видна тільки цьому блоку та усім вкладеним в нього блокам.

Зв'язування описує діапазон, в межах якого змінна, що визначена в одній одиниці програми, може бути прив'язана до будь-якої іншої її одиниці. *Змінні з областю видимості в межах блоку*, будучи локальними, не мають зв'язування. *Змінні з областю видимості в межах файлу*, мають внутрішнє або зовнішнє зв'язування. Внутрішнє зв'язування означає, що змінна може використовуватися тільки в файлі, що містить її визначення. Зовнішнє зв'язування означає, що змінна може застосовуватися також і в інших файлах.

Нижче описані **класи зберігання** в **C** (крім концепцій, що стосуються потоків).

1) *Автоматичний*. Змінна, що оголошена в блоці (або в якості параметру в заголовку функції) без модифікатора класу зберігання або з модифікатором класу зберігання `auto`, належить до автоматичного класу зберігання. Вона характеризується автоматичною тривалістю зберігання, областю видимості в межах блоку та відсутністю зв'язування. Якщо вона не ініціалізована, то її значення не визначене.

2) *Регістровий.* Змінна, що оголошена в блоці (або у вигляді параметру в заголовку функції) з модифікатором класу зберігання `register`, належить до регістрового класу зберігання. Вона характеризується автоматичною тривалістю зберігання, областю видимості в межах блоку та відсутністю зв'язування. Адресу

такої змінної отримати неможна. Оголошення змінної як регістрової – це підказка компілятору про необхідність забезпечити швидкий доступ наскільки це можливо. Якщо вона не ініціалізована, то її значення не визначено.

3) *Статичний, без зв'язування.* Змінна, що оголошена в блоці з модифікатором класу зберігання `static`, належить до класу зберігання «статичний, без зв'язування». Вона характеризується статичною тривалістю зберігання, областю видимості в межах блоку та відсутністю зв'язування. Така змінна ініціалізується тільки один раз на етапі компіляції. Якщо вона не ініціалізована явно, її біти встановлюються в `0`.

4) *Статичний, зовнішнє зв'язування.* Змінна, яка визначена як зовнішня по відношенню до будь-якої функції та без використання модифікатору класу зберігання `static`, належить до класу зберігання «статичний, зовнішнє зв'язування». Вона має статичну тривалість зберігання, область видимості в межах файлу і зовнішнє зв'язування. Така змінна ініціалізується тільки один раз на етапі компіляції. Якщо вона не ініціалізована явно, її біти встановлюються в `0`.

5) *Статичний, внутрішнє зв'язування.* Змінна, яка визначена як зовнішня по відношенню до будь-якої функції та з зазначенням модифікатора класу зберігання `static`, належить до класу зберігання «статичний, внутрішнє зв'язування». Вона має статичну тривалість зберігання, область видимості в межах файлу та внутрішнє зв'язування. Така змінна ініціалізується тільки раз на етапі компіляції. Якщо вона не ініціалізована явно, її біти встановлюються в `0`.

Виділення пам'яті забезпечується функцією `malloc()` (або спорідненою), яка повертає вказівник на блок пам'яті, що має необхідну кількість байтів. Цю пам'ять можна зробити доступною для повторного використання, викликавши функцію `free()` з передачею адреси в якості аргументу.

Кваліфікаторами типу є `const`, `volatile` і `restrict`. Кваліфікатор `const` вказує на константні дані. У випадку застосування з вказівниками `const` може визначати, що константним є сам вказівник або ж дані, на які вказівник посилається, в залежності від місця його розміщення всередині оголошення. Кваліфікатор `volatile` говорить про те, що дані можуть змінюватися процесами,

зовнішніми по відношенню до програми. Він призначений для попередження компілятора про те, що він повинен уникати оптимізацій, які передбачалися б за відсутності `volatile`. Кваліфікатор `restrict` також введений через причини, що пов'язані з оптимізацією. Вказівник, позначений за допомогою `restrict`, ідентифікується як єдиний засіб доступу до блоку даних.

Контрольні запитання та завдання

1. Перерахуте п'ять різних моделей класів зберігання.
2. Яку область видимісті має змінна у мові програмування *C*?
3. До чого застосовуються область видимості в межах функції та межах прототипу функції?
4. Чи мають змінні з областю видимості в межах блоку, функції або прототипу функції зв'язування?
5. Що характеризує тривалість зберігання?
6. Перелічте класи зберігання. Що означає автоматична тривалість зберігання?
7. Як відбувається ініціалізація автоматичних змінних.
8. Розкрийте особливості використання статичних змінних з зовнішнім зв'язуванням.
9. Наведіть узагальнені відомості про класи зберігання.
10. Який клас зберігання вибирається за замовчуванням?
11. Які функції в мові С застосовуються для динамічного виділення пам'яті?
12. Яка функція застосовується для звільнення пам'яті? Як вона виглядає та скільки має параметрів?
13. Яке призначення має функція `malloc()` і в чому її особливість?
14. Чи можна за допомогою функції `malloc()` створювати масиви «неправильної форми»?
15. Яке призначення має функція `calloc()` і в чому її особливість?

16. Для чого потрібна функція `realloc()`? Коли вона застосовується?

17. Які способи виділення динамічної пам'яті для двовимірного масиву ви знаєте? Чим вони відрізняються?

18. Коли відбувається «витік пам'яті»?

19. В чому полягає принцип виділення динамічної пам'яті для багатовимірних масивів?

Завдання для самостійного розв'язання

1. Напишить С-програму для створення двовимірного масиву дійсних чисел розмірністю 5×3 з використанням функції `malloc()`, використайте числа з діапазону від 2.50 до 7.00 та виведіть вміст цього масиву на екран.

2. Напишить С-програму для створення двовимірного трикутного масиву цілих чисел за допомогою функції `malloc()`. Використовуйте масив з 10 рядків, і кожен наступний рядок повинен містити на один елемент більше ніж попередній. Виведіть результат на екран з подальшим звільненням пам'яті.

3. Напишить С-програму для створення в динамічній пам'яті двовимірного масиву цілих чисел, який містить 10 рядків та 7 стовпців. Двовимірний масив розташуйте в оперативній пам'яті в формі стрічки, яка складається з елементів рядків. Виведіть результат на екран з подальшим звільненням пам'яті.

4. Напишить С-програму для виділення пам'яті під символьний масив змінної довжини за допомогою функції `calloc()`, в якій присвойте елементам різні значення кодів символів для цифр від 0 до 9 і виведіть вміст масиву на екран. Наприкінці програми звільніть виділену пам'ять.

5. Напишить С-програму, яка за допомогою функції `malloc()` виділяє пам'ять для 2 слів. Збільшить кількість слів, для збільшення розміру пам'яті використайте функціюї `realloc()`. Введіть слово «Кінець» для припиненя роботи програми та виведіть на екран усі введені раніше слова

3. РОЗШИРЕНЕ ПРЕДСТАВЛЕННЯ ДАНИХ

3.1. Зв’язні списки

На даний момент ви вже маєте певні теоретичні та практичні навички у створенні змінних, структур, функцій і т. д. Однак з часом необхідно переходити на більш високий рівень, де реальним завданням стає проектування та реалізація проекту як єдиного цілого.

Найчастіше найбільш важливим аспектом розробки програми є вибір відповідного представлення даних, якими вона буде маніпулювати. Правильний вибір представлення даних може перетворити написання програми на дуже просту задачу.

Ви вже знайомі з вбудованими типами даних: простими змінними, масивами, вказівниками, структурами та об’єднаннями. Проте, часто вибір правильного представлення даних не обмежується простим вибором типу. Ви повинні також подумати й про те, які операції вам доведеться виконувати. Тобто потрібно обирати спосіб зберігання даних і визначити, які операції є припустимими для обраного типу даних. Наприклад, в реалізаціях `C` тип `int` і тип вказівника зазвичай зберігаються як цілі числа, але для кожного з них визначено свій набір допустимих операцій. Скажімо, одне ціле число можна помножити на інше, але не можна помножити вказівники. Операцію `'*'` можна застосовувати для розіменування вказівника, але вона не має сенсу для цілочислового значення.

У мові `C` визначені допустимі операції для його фундаментальних типів. Проте, при проектуванні схеми представлення даних може знадобитися самостійно визначити допустимі операції. Мовою `C` це можна зробити шляхом розробки функцій, які представляють бажані операції. Тобто, проектування типу даних складається з визначення способу зберігання даних і розробки функцій для керування даними.

Ви також ознайомитеся з деякими алгоритмами, які є готовими рецептами для маніпулювання даними. Як програміст, ви з часом будете мати набір таких

рецептів, які доведеться знову і знову застосовувати для вирішення схожих завдань.

В цьому та наступних пілрозділах будемо розглянуто процес проектування типів даних, тобто процес зіставлення алгоритмів з представленнями даних. Розглянемо такі поширені форми даних, такі як черга, список і двійкове дерево пошуку.

Також буде представлена концепція абстрактного типу даних (**abstract data type – ADT**). Тип **ADT** упаковує методи та представлення даних проблемноорієнтованим, а не мовно-орієнтованим способом. Після того як ви спроектували абстрактний тип даних, його можна легко багаторазово використовувати при різних обставинах. Розуміння типів **ADT** концептуально підготує вас до світу об'єктно-орієнтованого програмування та мов **C++, C#, Java.**

3.1.1. Дослідження представлення даних

Давайте почнемо з обмірковування даних. Припустимо, що потрібно створити програму для адресної книги. Яку форму даних необхідно використовувати для зберігання інформації? Оскільки з кожним записом пов'язана різноманітна інформація, кожен запис має сенс представити у вигляді структури. А як представити кілька записів? За допомогою стандартного масиву структур? За допомогою динамічного масиву? За допомогою якоїсь іншої форми? Чи повинні записи бути впорядковані в алфавітному порядку? Чи потрібна можливість пошуку в записах за поштовим індексом? Чи потрібен пошук за міжміським телефонним кодом? Дії, які потрібно виконувати, можуть впливати на вибір способу зберігання інформації. Тобто, перш ніж приступати до створення коду, доведеться прийняти масу проектних рішень.

А як ви представите растрові графічні зображення, які повинні зберігатися в пам'яті? У растровому зображенні кожен піксель на екрані встановлюється індивідуально. За часів чорно-білих екранів для представлення одного пікселя можна було використовувати один біт (**1** або **0**) – звідси і англійська назва

растрових графічних зображень **bitmapped** (бітове зображення). На кольорових моніторах опис одного пікселя займає більше одного біта. Наприклад, виділення по 8 бітів для кожного пікселя дозволяє отримати 256 кольорів. На даний час відбувся перехід до 65 536 кольорів (16 бітів на піксель), 16 777 216 кольорів (24 біти на піксель); 2 147 483 648 (32 біти на піксель) і навіть більше. При наявності 32-бітових кольорів і роздільної здатності монітора 2560×1440 пікселів для представлення одного екрану растрової графіки вам знадобиться близько 118 мільйонів бітів (14 мегабайтів). Чи слід змиритися з цим або ж розробити якийсь метод стиснення інформації? Чи повинно це стиснення виконуватися без втрат або с втратами (порівняно неважливих даних)? І знову, перш ніж займатися кодуванням, доведеться прийняти безліч проектних рішень.

Розглянемо конкретний випадок представлення даних. Припустимо, що потрібно написати програму, яка дозволяє вводити список усіх фільмів (на відеокасетах, дисках **DVD** і дисках **Blu-ray**), що були переглянуті протягом року. Для кожного фільму бажано реєструвати різноманітну інформацію, таку як назва, рік випуску, імена та прізвища режисера та провідних акторів, тривалість і жанр (комедія, наукова фантастика, романтика, мелодрама та ін.), рейтинг і т. д. Це передбачає застосування структури для кожного фільму та масиву структур для списку фільмів. Для спрощення обмежимо структуру двома членами: назвою фільму та власною оцінкою його рейтингу за 10-бальною шкалою. Текст програми, що використовує цей підхід має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#define TSIZE  45     // розмір масиву для зберігання назви
#define FMAX    5     // максимальна кількість назв фільмів
 struct
Film
{
    char title[TSIZE];
int rating;
};
```

```
char *s_gets(char *st, int n);
int main(void)
{
    struct Film
movies[FMAX];      int i =
0;     int j;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    puts("Введіть назву першого фільму:");

    while(i < FMAX && s_gets(movies[i].title, TSIZE) != NULL
&&           movies[i].title[0] != '\0')
    {
        puts("Введіть своє значення рейтингу <0-
10>:");          scanf("%d", &movies[i++].rating);
while(getchar() != '\n')              continue;
        puts("Введіть назву наступного фільму\n"
             "(або порожній рядок для припинення вводу):");
    }
if(i == 0)
        printf("\nДані не введені.");
else
        printf("\nСписок фільмів:\n");
for(j = 0; j < i; j++)
        printf("Фільм: %s.   Рейтинг: %d\n",
movies[j].title,                  movies[j].rating);
printf("Програма завершена.\n");     return 0;
}
 char *s_gets(char *st,
int n)
{
    char *ret_val;
char *find;

    ret_val = fgets(st, n, stdin);
if(ret_val)
    {
        find = strchr(st, '\n');   // пошук нового рядка
if(find)              // якщо адреса не дорівнює NULL,
*find = '\0';    // помістити туди нульовий символ
else
            while(getchar() != '\n')
                continue;    // відкинути залишок рядка
    }
    return
ret_val; }
```

Результат виконання програми наведено на рис. 3.1.

Програма створює масив структур і заповнює його даними, які вводить користувач. Введення інформації триває аж до заповнення масиву (перевірка `FMAX`), до досягнення кінця файлу (перевірка `NULL`) або до натискання користувачем клавіші `<Enter>` на початку рядка (перевірка `'\0'`).

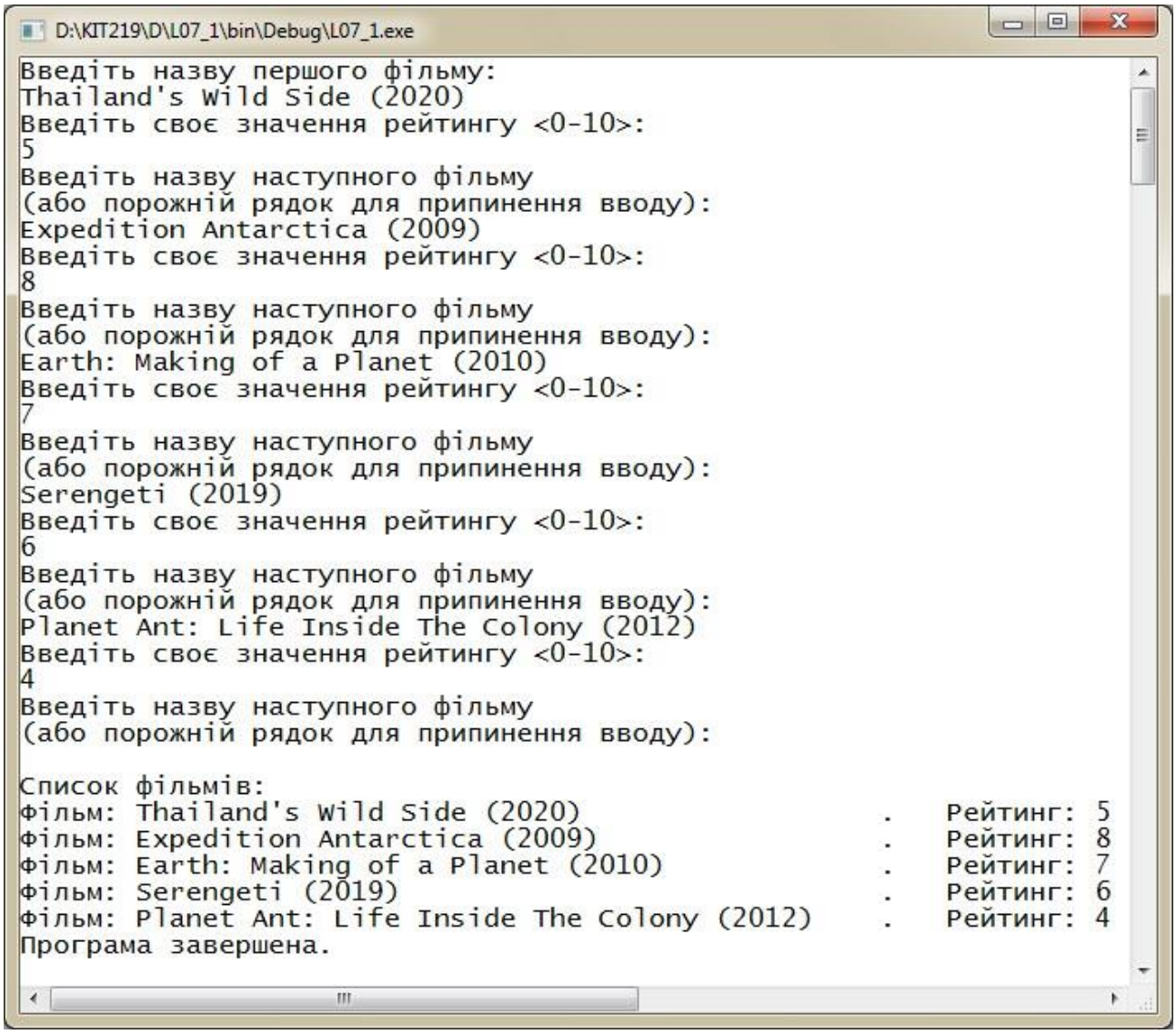


Рисунок 3.1 – Результат виконання програми, яка представляє дані у вигляді структури

Така організація програми пов’язана з низкою проблем. По-перше, швидше за все, програма буде марно витрачати великий об’єм пам’яті, оскільки назви більшості фільмів містять менш ніж `45` символів, але, в той самий час, назви деяких фільмів можуть бути надто довгими. По-друге, обмеження в п’ять

фільмів на рік багатьом буде здаватися надто суворим. Звичайно, цю межу можна збільшити, але якою вона повинна бути? Хтось переглядає до 500 фільмів на рік, тому значення FMAX можна було б збільшити до 500, але для деяких і цього може виявитися замало, в той час як для інших воно приводило б до марної витрати величезного об'єму пам'яті.

Крім того, деякі компілятори за замовчуванням обмежують об'єм пам'яті, доступної для змінних з автоматичним класом зберігання на зразок movies, і такий великий масив міг би перевищити вказане обмеження. Ситуацію можна виправити, зробивши масив статичним або зовнішнім або надавши інструкцію компілятору про необхідність застосування стеку більшого розміру. Однак це не вирішує проблему.

Проблема полягає в тому, що представлення даних визначено абсолютно негнучким чином. На етапі компіляції вам доводиться приймати рішення, які доцільніше приймати під час виконання. Це передбачає перехід до представлення даних, яке використовує динамічний розподіл пам'яті.

Можна спробувати застосувати наступний код:

```
#define TSIZE   45     // розмір масиву для зберігання назви
struct Film
{
char title[TSIZE];
int rating;
};
...
int n, i;
struct Film *movies; // вказівник на структуру
...
printf("Вкажіть максимальну кількість фільмів: \n");
scanf("%d", &n); movies = (struct Film *) malloc(n *
sizeof(struct Film));
```

Вказівник movies можна застосовувати таким чином, ніби він є ім'ям масиву:

```
while(i < FMAX && gets(movies[i].title) != NULL &&
movies[i].title[0] != '\0')
```

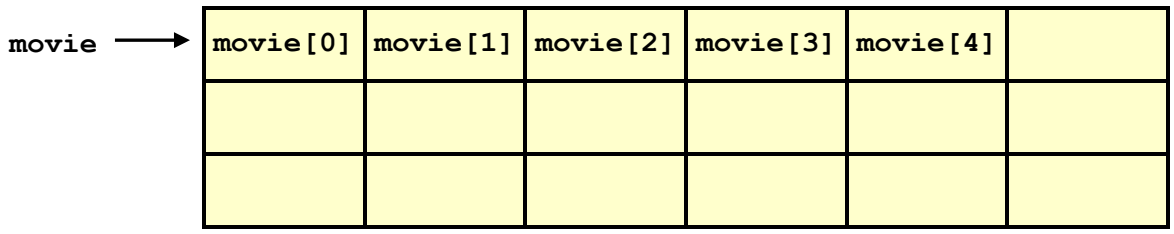
За рахунок використання функції `malloc()` ви можете відкласти визначення кількості елементів до моменту виконання програми, тому не слід буде виділяти пам'ять для `500` елементів, якщо їх необхідно тільки `20`. Але при такому підході обов'язок ввести коректне значення для кількості записів покладається на користувача.

3.1.2. Від масиву до зв'язного списку

В ідеалі бажано мати можливість додавати дані необмежено (або до тих пір, поки програма не вичерпана доступну пам'ять), не вказуючи заздалегідь кількість записів, які будуть створені, і, не змушуючи програму виділяти величезний об'єм пам'яті без реальної на те необхідності. Цієї мети можна досягти, викликаючи `malloc()` після введення кожного запису і виділяючи лише такий об'єм пам'яті, якого достатньо для нового запису. Якщо користувач вводить інформацію про три фільми, програма викликає функцію `malloc()` три рази. Якщо користувач вводить інформацію про `300` фільмів, програма викликає `malloc()` триста разів.

Така прекрасна на перший погляд ідея породжує нову проблему. Щоб побачити, в чому вона полягає, порівняйте одноразовий виклик `malloc()` для виділення пам'яті під `300` структур `Film` з `300`-кратним викликом цієї функції для виділення пам'яті кожного разу тільки для однієї структури `Film`. У першому випадку пам'ять розподіляється у вигляді одного безперервного блоку, і для відстеження вмісту потрібно єдина змінна вказівника на структуру `Film`, яка вказує на першу структуру в блоці. Як було показано в наведеному раніше фрагменті коду, проста форма запису з масивом забезпечує за допомогою вказівника доступ до кожної структурі всередині блоку. Проблема з другим підходом – відсутність будь-якої гарантії того, що послідовні виклики `malloc()` призведуть до виділення суміжних блоків пам'яті. Це означає, що структури не обов'язково будуть збережені безперервно (рис. 3.2).

```
struct Film *movie;
movie = (struct Film *) malloc(5 * sizeof(struct Film));
```




```
int i;
struct Films *movies[5];
for(i = 0; i < 5; i++)
movies[i] = (struct Films *) malloc(sizeof(struct Films));
```

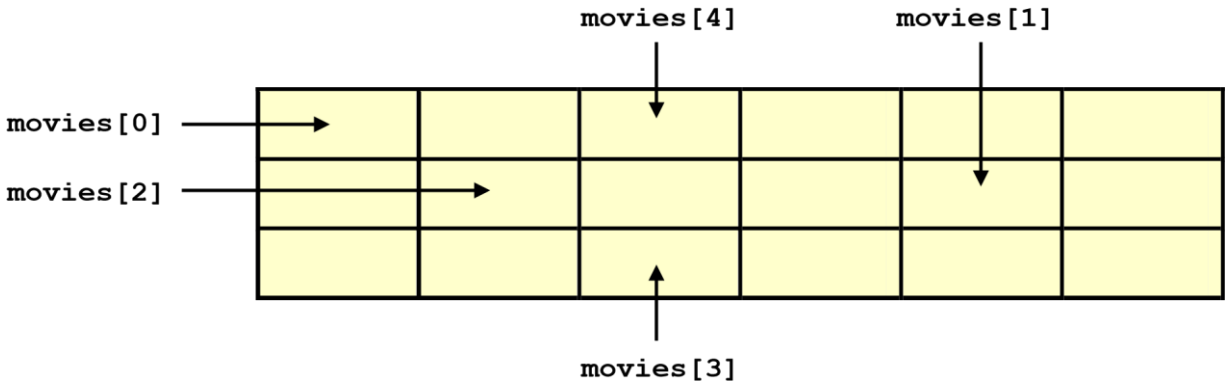


Рисунок 3.2 – Виділення пам’яті під структури одним блоком та індивідуально

Таким чином, замість зберігання одного вказівника на блок з `300` структур доведеться зберігати `300` вказівників – по одному для кожної незалежно виділеної структури.

Одне з можливих рішень, яке, однак, ми застосовувати не будемо, передбачає створення великого масиву вказівників і присвоювання значень вказівниками в міру виділення пам’яті під нові структури:

```
#define TSIZE 45      // розмір масиву для збереження назв
#define FMAX  500     // максимальна кількість назв фільмів
struct Film
{
char title[TSIZE];
int rating;
};
...
```

```
struct Film *movies[FMAX];// масив вказівників на структури
int i;
...
movies[i] = (struct Film *) malloc(sizeof(struct Film));
```

Цей підхід дозволяє зекономити великий об'єм пам'яті, якщо ви не використовуєте повний комплект вказівників, оскільки масив з **500** вказівників займає значно менше пам'яті, ніж масив з **500** структур. Проте, як і раніше простір витрачається марно на зберігання невикористаних вказівників, до того ж продовжує діяти обмеження в **500** структур.

Існує більш ефективний спосіб. При кожному виклику функції **malloc()** для виділення пам'яті під нову структуру одночасно можна виділяти пам'ять і для нового вказівника. Але ви можете заперечити, що тоді треба буде мати ще один вказівник для відстеження вказівника, який щойно був виділений, а для його відстеження необхідний ще один вказівник, і так до нескінченості. Запобігти цій потенційній проблемі можна шляхом перевизначення структури таким чином, щоб вона містила вказівник на наступну структуру. Тоді при кожному створенні нової структури її адресу можна буде зберігати в попередній структурі. Таким чином, структуру **Film** потрібно перевизначити так, як показано нижче:

```
#define TSIZE 45   // розмір масиву для зберігання назв
struct Film
{ char title[TSIZE];
     int rating;
     struct Film
     *next;
};
```

Дійсно, структура не може містити структуру того ж самого типу, але вона може мати вказівник на структуру того ж самого типу. Визначення подібного роду є основою зв'язного списку – списку, в якому кожен елемент містить інформацію про місцезнаходження наступного елемента.

Перш ніж поглянути на код **C**-програми для зв'язного списку, давайте докладніше розглянемо концепції, що лежать в основі такого списку. Припустимо, що в якості назви фільму користувач вводить **Modern Times** і **10**

для значення рейтингу. Програма виділила б пам'ять для структури **Film**, скопіювала б рядок **Modern Times** до члену **title** і встановила б значення члену **rating** у значення, яке дорівнює **10**. Щоб вказати на те, що за цією структурою немає жодних інших структур, програма повинна була б встановити значення членавказівника **next** в **NULL**. Зрозуміло, що при цьому необхідно відстежувати місце зберігання першої структури. Це можна зробити, присвоївши адресу окремому вказівнику, який ми будемо називати вказівником на заголовок списку. Вказівник на заголовок вказує на перший елемент у зв'язному списку елементів. На рис. 3.3 показано, як виглядає ця структура.

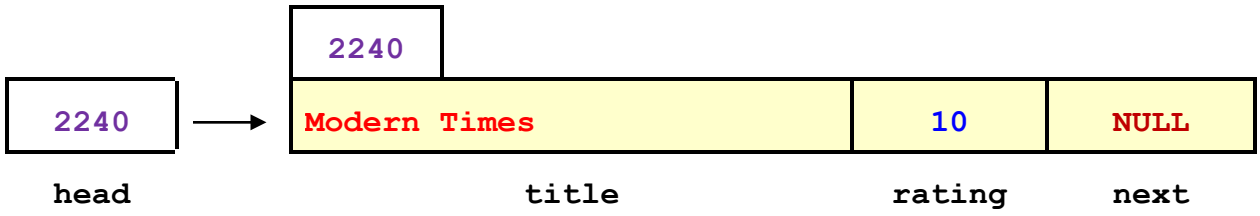


Рисунок 3.3 – Перший елемент у зв'язному списку

Тепер припустимо, що користувач вводить назву і рейтинг другого фільму – скажімо, **Midnight in Paris** і **8**. Програма виділяє пам'ять для другої структури **Film** і зберігає адресу нової структури в члені **next** першої структури (шляхом перезапису значення **NULL**, яке було встановлено раніше), щоб вказівник **next** посилався на наступну структуру у зв'язному списку.

```
#define TSIZE 45
struct Film
{
char title[TSIZE];
int rating;
struct File *next;
}
 struct File
*head;
```

Потім програма копіює значення **Midnight in Paris** і **8** до нової структури і встановлює значення її члену **next** в **NULL**, вказуючи, що тепер ця структура є останньою у списку. Такий список з двох елементів наведено на рис. 3.4.

Обробка інформації про кожен новий фільм буде виконуватися аналогічно.

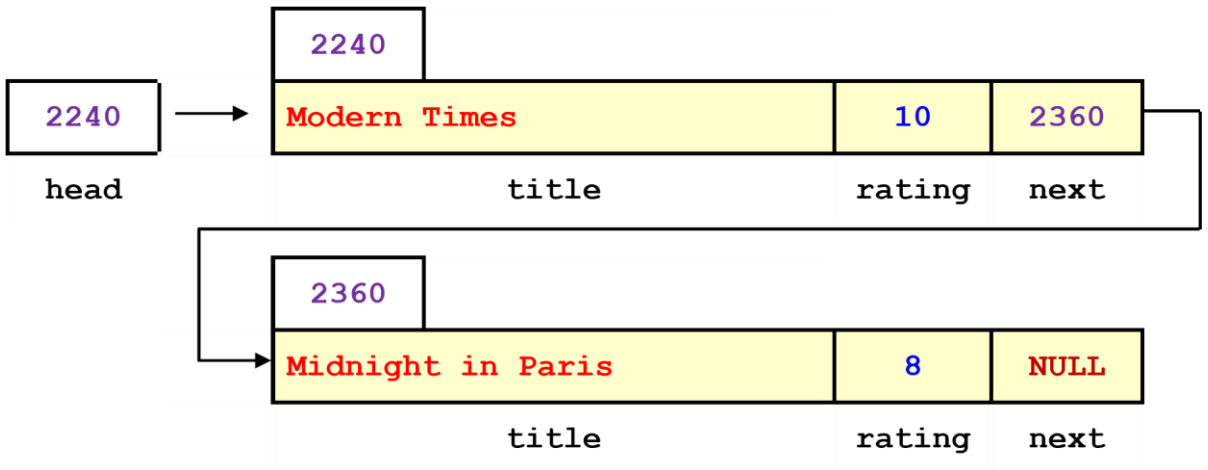


Рисунок 3.4 – Зв’язний список з двома елементами

Адреса нової структури буде зберігатися в попередній структурі, у новій структурі буде міститися введена інформація, а значення члену **next** нової структури буде встановлюватися в **NULL**, що призведе до створення зв’язного списку, подібного до представленого на рис. 3.5.

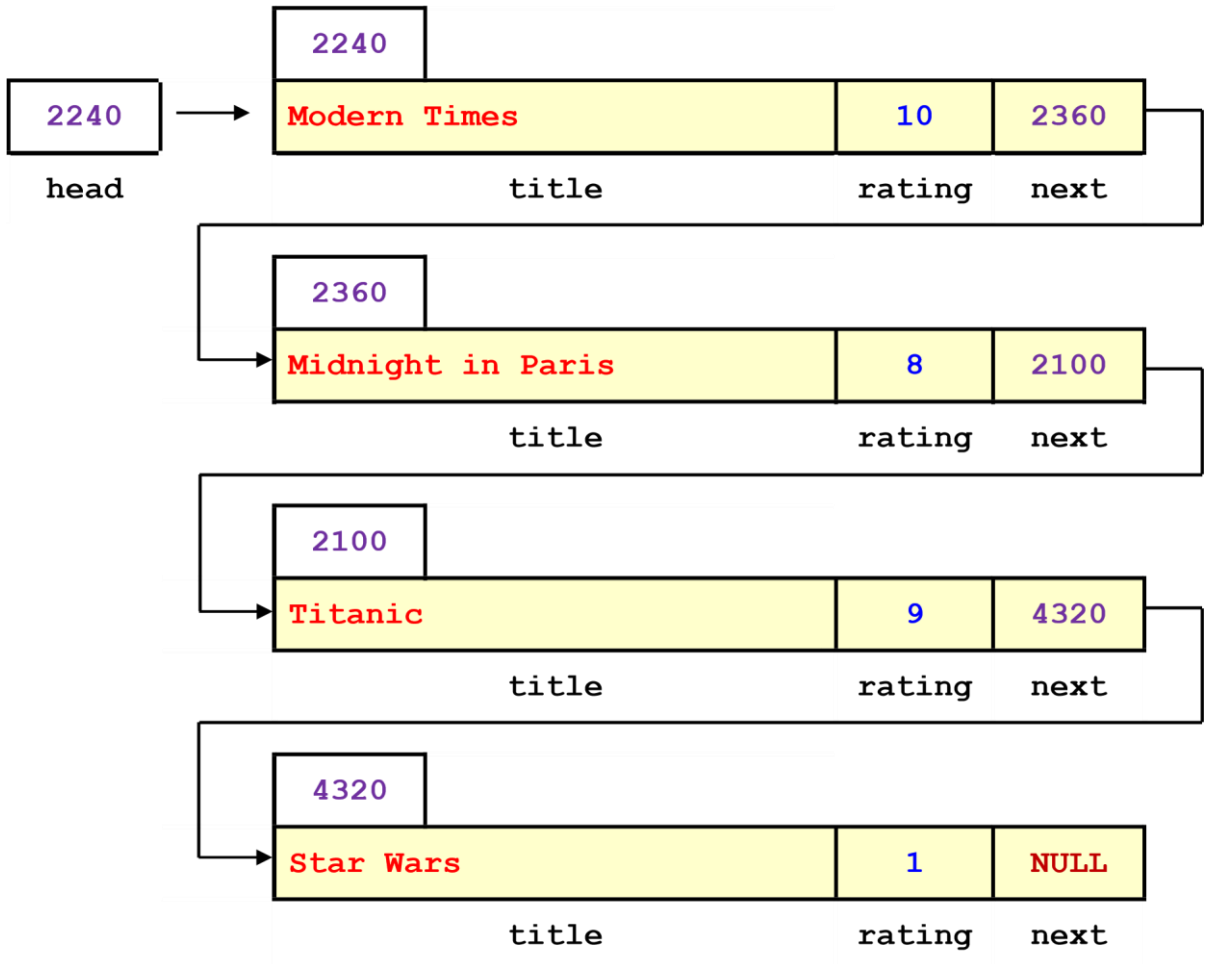


Рисунок 3.5 – Зв’язний список з чотирма елементами

Припустимо, що список необхідно вивести на екран. При кожному виводі елемента для знаходження наступного елементу, можна застосовувати адресу, яка була збережена у відповідній структурі. Однак щоб ця схема працювала, треба мати вказівник, який буде відстежувати самий перший елемент у списку, оскільки жодна структура у списку не зберігає адресу першого елемента. На щастя, це вже зроблено за допомогою вказівника на заголовок списку.

3.1.3. Використання зв'язного списку

Тепер, коли ви отримали уявлення про роботу зв'язного списку, давайте спробуємо його реалізувати. Текст програми, в якому для зберігання інформації про фільми замість масиву застосовується зв'язний список, має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>        // містить прототип функції malloc()
#include <string.h>        // містить прототип функції strcpy()
#define TSIZE 45           // розмір масиву для зберігання назв
 struct
Film
{
    char title[TSIZE];
int rating;
    struct Film *next; // вказує на наступну структуру в списку
}
 char *s_gets(char *st, int n);

int main(void)
{
struct Film *head = NULL;
struct Film *prev, *current;
char input[TSIZE];
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

// Збирання та збереження інформації
puts("Введіть назву першого фільму: ");
    while(s_gets(input, TSIZE) != NULL && input[0] != '\0')
    {
current = (struct Film *) malloc(sizeof(struct Film));
if(head == NULL)     // перша структура
```

```
head = current;
        else                    // наступні структури
prev->next = current;
current->next = NULL;
strcpy(current->title, input);
puts("Введіть своє значення рейтингу <0-10>:");
scanf("%d", &current->rating);
while(getchar() != '\n')                continue;
        puts("Введіть назву наступного фільму\n"
             "(або порожній рядок для припинення вводу):");
prev = current;
    }
     // Відображення списку фільмів
if(head == NULL)
        printf("Дані не введені.");
else
        printf("Список
фільмів:\n");      current = head;
while(current != NULL)
    {
printf("Рейтинг: %d.  Фільм: %s. \n",  current->rating, current-
>title);
current = current->next;      }
// Програма виконана, тому можна звільнити пам'ять
current = head;      while(current != NULL)
    {
        free(current);
        current = current->next;
    }
    printf("Програма завершена.\n");
return 0;
}
 char *s_gets(char *st, int n);
{
char *ret_val;
char *find;
    ret_val = fgets(st, n, stdin);
find = strchr(st, '\n');    // пошук нового рядка
if(find)                    // якщо адреса не дорівнює NULL,
*find = '\0';           // помістити туди нульовий символ
else
            while(getchar() != '\n')
                continue;           // відкинути залишок рядка
    }
    return ret_val;
}
```

Програма вирішує два завдання з використанням зв’язного списку.

По-перше, вона конструює список і заповнює його вхідними даними.

По-друге, вона відображає список.

Відображення списку – це більш просте завдання, тому спочатку розглянемо саме його.

Результат роботи програми наведено на рис. 3.6.

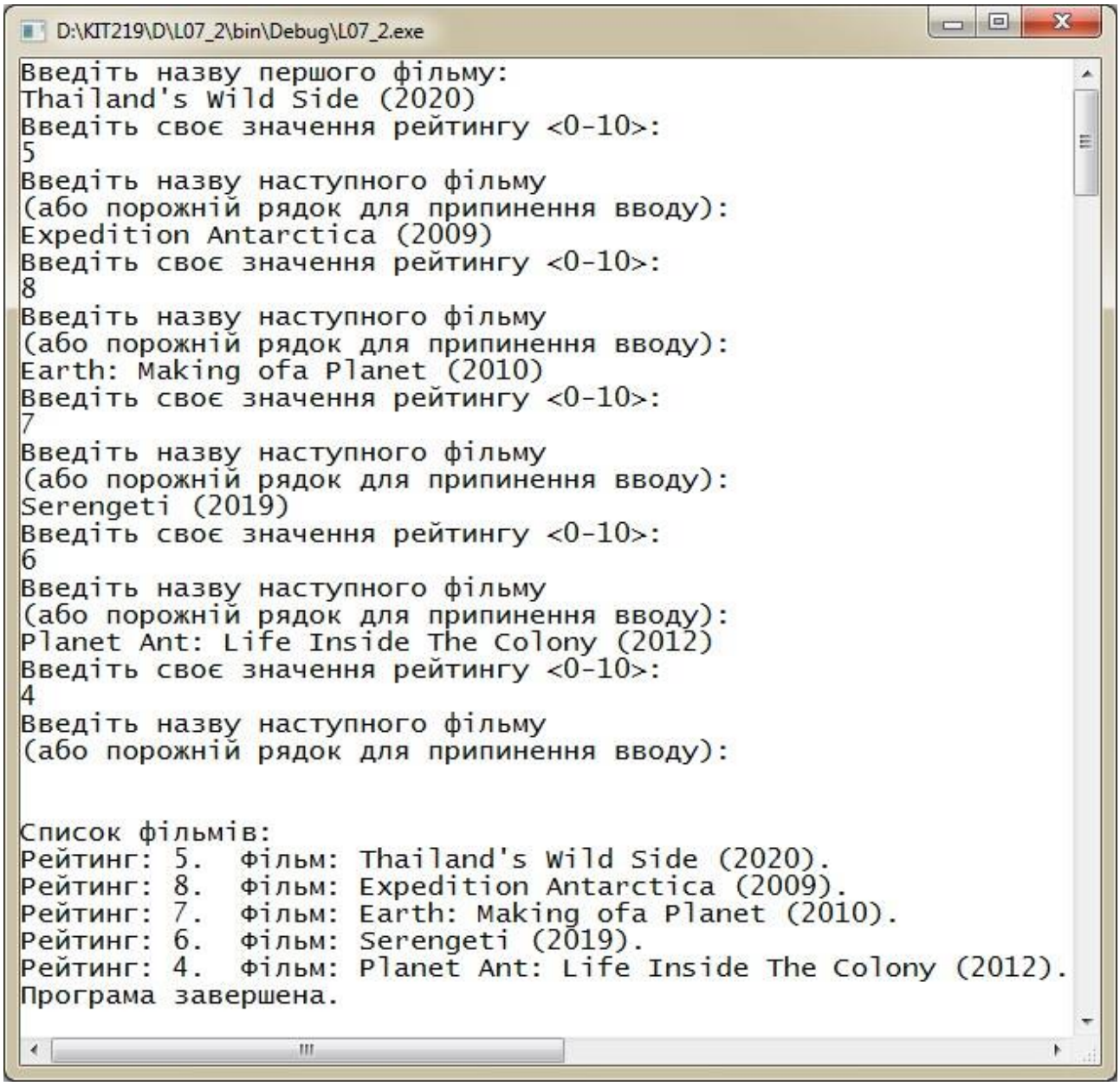


Рисунок 3.6 – Результат виконання програми, яка представляє дані у вигляді зв’язного списку

3.1.4. Відображення списку

Ідея полягає в тому, щоб розпочати зі встановлення вказівника **current** для посилання на першу структуру. Оскільки вказівник на заголовок **head** вже вказує куди треба, наступного коду цілком достатньо:

```
current = head;
```

Потім за допомогою форми запису з вказівником можна звернутися до членів цієї структури:

```
printf("Рейтинг: %d.  Фільм: %s. \n", current->rating,
current->title);
```

Далі вказівник **current** перевстановлюється для посилання на наступну структуру в списку. Ця інформація зберігається в члені **next** структури, тому задача вирішується за допомогою такого коду:

```
current = current->next;
```

Після цього весь процес необхідно повторити. Після відображення останнього елемента в списку вказівник **current** буде встановлений в **NULL**, оскільки це значення члену **next** останньої структури. Даною обставиною можна скористатися для припинення виводу. Фрагмент коду, що застосовується для відображення списку, виглядає наступним чином:

```
while(current != NULL)
{ printf("Рейтинг: %d.  Фільм: %s. \n",
     current->rating, current->title);
     current = current->next;
}
```

Чому б для переміщення по списку не скористатися **head** замість того, щоб створювати новий вказівник (**current**)? Причина полягає в тому, що це призвело б до зміни значення **head**, і програма втратила б можливість знаходити початок списку.

3.1.5. Створення списку

Створення списку передбачає виконання трьох основних дій:

- використання функції **malloc()** для виділення достатнього простору під зберігання структури;

- збереження адреси структури;

- копіювання до структури коректної інформації.

Не має сенсу створювати структуру, якщо вона поки непотрібна, тому для прийому від користувача інформації про назву фільму в програмі застосовується тимчасове сховище (масив **input**). Якщо користувач відтворює за допомогою клавіатури символ **EOF** або вводить порожній рядок, цикл вводу завершується:

```
while(s_gets(input, TSIZE) != NULL && input[0] != '\0')
```

За наявності введених даних програма запитує простір для структури і присвоює її адресу змінній типу вказівника **current**:

```
current = (struct Film *) malloc(sizeof(struct Film));
```

Адреса самої першої структури повинна бути збережена у змінній типу вказівника **head**. Адреса кожної наступної структури повинна зберігатися в члені **next** попередньої структури. Таким чином, програмі потрібен спосіб для з'ясування того, чи є поточна структура першою. Простіше за все вирішити цю задачу, ініціалізуючи вказівник **head** значенням **NULL** на початку програми. Потім у програмі можна використовувати значення вказівника **head** для прийняття рішення про подальші дії:

```
if(head == NULL)                    // перша структура
     head = current;
else                                // наступні структури
prev->next = current;
```

В цьому коді **prev** – вказівник на структуру, яка була виділена минулого разу. Далі знадобиться встановити члени структури у відповідні значення. Зокрема, член **next** повинен бути встановлений в **NULL** для вказівки на те, що поточна структура є останньою в списку. Ви повинні скопіювати назву фільму з масиву **input** до члену **title** і отримати значення для члену **rating**. Ці дії виконує наступний код:

```
current->next = NULL; strcpy(current->title, input);
puts("Введіть своє значення рейтингу <0-10>:");
scanf("%d", &current->rating);
```

Оскільки виклик `s_gets()` обмежує дані, що вводяться, межею в `TSIZE-1` символів, рядок у масиві `input` поміститься до члену `title`, тому цілком

безпечно застосовувати функцію `strcpy()`.

Нарешті, ви повинні підготувати програму до наступного циклу вводу. Зокрема, вказівник `prev` необхідно встановити так, щоб він посилався на поточну структуру, оскільки після вводу назви наступного фільму та розподілу наступної структури поточна структура стане попередньою. Програма встановлює цей вказівник наприкінці циклу: `prev = current;`

3.1.6. Звільнення пам'яті, яка була зайнята списком

В багатьох середовищах програма при своєму завершенні звільнює пам'ять, що була виділена за допомогою функції `malloc()`, але краще, щоб ви звикли кожен виклик функції `malloc()` супроводжувати викликом `free()`. Таким чином, програма очищує використану пам'ять з застосуванням функції `free()` до кожної виділеної структури:

```
current = head;
while(current != NULL)
{ free(current);
     current = current->next;
}
```

Можливості розглянутої програми є дещо обмеженими. Наприклад, в ній відсутня перевірка, чи вдалося функції `malloc()` знайти потрібну пам'ять, і вона позбавлена будь-яких засобів для видалення елементів зі списку. Проте, такі недоліки можуть бути усунені. Скажімо, можна додати код, який перевіряє, чи дорівнює нулю (`NULL` – ознака невдачі в отриманні бажаної пам'яті) значення, що повертається `malloc()`. Якщо програма потребує видалення записів, в ній можна передбачити додатковий код.

Такий спеціалізований підхід до вирішення проблем і додаванню функціональних можливостей в міру необхідності не завжди є найкращим стилем програмування. З іншого боку, як правило, не вдається вгадати абсолютно все, що буде потрібно програмі. У міру зростання масштабу проектів

модель завчасного планування всіх необхідних функціональних засобів стає все менш реалістичною. Було виявлено, що найуспішнішими виявлялися ті великі програми, які поетапно розвивалися від невеликих успішних програм.

З огляду на те, що плани можуть переглядатися, має сенс розробляти початкові ідеї в такий спосіб, що спрощує подальшу модифікацію. Розглянутий приклад не дотримується цього принципу. Так, в ньому проявляється тенденція до змішування деталей кодування та концептуальної моделі. Наприклад, в цьому коді концептуальна модель полягає в тому, що елементи додаються до списку. Програма затінює цей інтерфейс, виносячи на передній план такі деталі, як `malloc()` і вказівник `current->next`. Дуже бажано, якби ви змогли писати програму в стилі, який робить очевидним те, що ви додаєте елемент до списку, і одночасно приховує такі допоміжні дії, як виклик функцій керування пам'яттю і встановлення вказівників. Відокремлення призначеного для користувача інтерфейсу від деталей реалізації спростить розуміння та оновлення програми. Згаданих цілей можна досягти, створивши програму заново. Розпочавши розробку з нуля, ви можете досягти таких цілей.

3.2. Абстракті типи даних

В програмуванні ви намагаєтесь зіставити необхідний тип даних з потребами програмної задачі. Наприклад, для представлення кількості наявних пар взуття можна було б використовувати тип `int`, а для представлення середньої ціни однієї пари – тип `float` або `double`. В наведених вище прикладах програм, що пов'язані з фільмами, дані формували список елементів, кожен з яких складався з назви фільму (рядку символів типу `char`) і значення рейтингу (типу `int`). Жоден з базових типів `C` не відповідає цьому опису, тому для представлення окремих елементів ми визначили структуру, а потім створили пару методів для об'єднання послідовності структур у список. По суті, ми застосували можливості мови `C` з розробки нового типу даних, що задовольняють конкретним вимогам,

але робили це безсистемно. Тепер ми застосуємо систематичний підхід до визначення типів.

Що утворює тип? Тип визначають два види інформації: набір властивостей і набір операцій. Наприклад, властивість типу `int` полягає в тому, що він представляє цілочислове значення і, відповідно, поділяє властивості цілих значень. Дозволеними арифметичними операціями для цього типу є зміна знаку, додавання двох значень `int`, віднімання одного значення `int` від іншого, множення двох значень `int`, ділення одного значення `int` на інше та отримання результату обчислення одного значення `int` за модулем іншого. Оголошення змінної типу `int` означає, що на неї можуть впливати лише ці операції.

Можна вважати, що математика пропонує абстрактну концепцію цілого числа, а мова `C` – реалізацію цієї концепції. Наприклад, в `C` надаються засоби зберігання цілого числа і виконання цілочислових операцій, таких як додавання та множення. Зверніть увагу, що забезпечення підтримки арифметичних операцій є важливою частиною представлення цілих чисел. Тип `int` був би менш корисним, якщо б він дозволяв тільки зберігати значення, але не використовувати їх в арифметичних виразах. Також слід відзначити, що задача представлення цілих чисел в цій реалізації вирішена далеко не ідеально. Наприклад, існує нескінченна кількість цілих чисел, але `4`-байтовий тип `int` може представляти тільки `4 294 967 295` з них (не плутайте абстрактну ідею з конкретною реалізацією).

Припустимо, що ви бажаєте визначити новий тип даних. По-перше, ви повинні надати спосіб для зберігання даних – можливо, за рахунок проектування структури. По-друге, знадобиться забезпечити методи для маніпулювання даними. В якості прикладу розглянемо програму з попередньої підрозділу. Вона містить зв’язаний набір структур для зберігання інформації, а також код для додавання та відображення інформації. Проте, програма не вирішує ці задачі таким чином, щоб зробити очевидним створення нового типу даних. Яким же чином слід було вчинити?

Обчислювальні науки пропонують дуже ефективний спосіб визначення нових типів даних. Він є процесом переходу від абстрактного до конкретного, який складається з трьох етапів:

1) Формування абстрактного опису властивостей типу та операцій, які можна виконувати над цим типом. Такий опис не повинен бути прив'язаний до жодної конкретної реалізації. Він навіть не повинен бути прив'язаний до конкретної мови програмування. Формальний абстрактний опис подібного роду називають **абстрактним типом даних** (`abstract data type – ADT`).

2) Розробка програмного інтерфейсу, що реалізує цей тип `ADT`.

Тобто необхідно зазначити, як слід зберігати дані, та зробити опис набору функцій, що виконують потрібні операції. Наприклад, в `C` ви можете надати визначення структури разом з прототипами функцій для маніпулювання структурами. Ці функції для визначеного користувачем типу відіграють ту ж саму роль, яку вбудовані операції `C` виконують для фундаментальних типів `C`. Будь-хто може скористатися новим типом і використовуватися розроблений інтерфейс у своїх програмах.

3) Написання коду для реалізації інтерфейсу.

Звичайно, цей крок дуже важливий, але програмісту, що використовує новий тип, зовсім необов'язково знати подробиці реалізації.

Щоб поглянути, як працює цей процес, давайте розглянемо конкретний приклад. Оскільки ми вже доклали певних зусиль для створення списку фільмів, переробимо його, застосовуючи новий підхід.

3.2.1. Отримання абстракції

По суті усе, що необхідно для проекту «Інформація про фільми», – це список елементів, кожен з яких містить назву та рейтинг фільму. Нам необхідно мати можливість додавати нові елементи в кінець списку та відображати вміст списку. Давайте назвемо списком абстрактний тип, який буде задовольняти цим потребам. Які властивості він повинен мати? Зрозуміло, що список повинен вміти зберігати послідовність елементів. Іншими словами, список може містити

декілька елементів, причому ці елементи певним чином упорядковані, що дозволяє говорити про перший, другий або останній елемент у списку. Далі, тип списку повинен підтримувати такі операції, як додавання елемента до списку.

Нижче перераховані деякі корисні операції зі списком для даного проекту:

- ініціалізація списку пустим вмістом;

- додавання елементу в кінець списку;

- з’ясування, чи є список пустим;

- з’ясування, чи є список повним;

- визначення кількості елементів у списку;

- перегляд кожного елемента у списку з метою виконання певної дії, такої, наприклад, як відображення елемента списку.

Для цього проекту додаткові операції не потрібні, але більш універсальний перелік операцій зі списками може містити наступні:

- додавання елемента в будь-яке місце списку;

- видалення елемента зі списку;

- отримання елемента зі списку (список залишається незмінним);

- заміна одного елемента у списку іншим; - пошук елемента у списку.

Тоді неформальне, але абстрактне визначення списку виглядає таким чином: список – це об’єкт даних, який може зберігати послідовність елементів, до якого можна застосовувати будь-які з перерахованих раніше операцій. В цьому визначенні не заявлений вид елементів, які можуть зберігатися у списку. В ньому не вказано, чи повинен для зберігання елементів використовуватися масив, зв’язаний набір структур або інша форма даних. Визначення не диктує, який метод застосовувати, наприклад, для з’ясування кількості елементів у списку. Усі ці деталі залишені за реалізацією.

Для простоти давайте приймемо в якості абстрактного типу даних спрощений список, що містить тільки ті функціональні можливості, які потрібні для проекту «Інформація про фільми». Короткий опис типу наведений нижче:

Ім'я типу:	Простий список
Властивості типу:	Може містити послідовність елементів
Операції типу:	Ініціалізація списку пустим вмістом. З'ясування, чи є список пустим. З'ясування, чи є список повним. Визначення кількості елементів у списку. Додавання елемента в кінець списку. Обхід списку з обробкою кожного елемента. Спустошення списку.

Наступним етапом є розробка для ADT простого інтерфейсу на мові С для списку.

3.2.2. Розробка інтерфейсу

Інтерфейс для простого списку складається з двох частин. Перша частина описує спосіб представлення даних, а друга – функції, що реалізують операції **ADT**. Наприклад, інтерфейс буде містити функції для додавання елемента до списку та виводу кількості елементів у списку. Проектне рішення інтерфейсу повинне якомога ближче відображати опис **ADT**. Відповідно, воно повинно бути виражено в термінах деякого загального типу **Item**, а не в термінах будь-якого конкретного типу на зразок **int** або **struct** film. Один зі способів досягнення цього передбачає використання засобу **typedef** мови **C** для визначення **Item** в якості потрібного типу:

```
#define TSIZE 45   // розмір масиву для зберігання назви
struct film
{
char title[TSIZE];
int rating;
};
typedef struct film Item;
```

Потім тип **Item** можна застосовувати в інших визначеннях. Якщо пізніше буде потрібен список елементів будь-якої іншої форми даних, можна буде перевизначити тип **Item** і залишити іншу частину визначення інтерфейсу без змін.

Після того як буде визначено тип **Item**, необхідно прийняти рішення про спосіб зберігання елементів цього типу. В дійсності цей крок належить до етапу реалізації, але прийняття рішення в даний момент спростить розуміння прикладу.

Підхід з використання зв'язаних структур достатньо успішно працював в останній програмі попереднього підрозділу, тому застосуємо його, як це показано нижче:

```
typedef struct node
{
Item item;
struct node *next;
} Node; typedef
Node *List;
```

В реалізації з застосуванням зв'язного списку кожен зв'язок називається вузлом. Кожен вузол містить інформацію, що формує вміст списку, і вказівник на наступний вузол. Щоб підкреслити обрану термінологію, ми назвали структуру вузла іменем **node** і застосували **typedef**, щоб зробити **Node** іменем типу для структури **struct** node. Нарешті, для керування зв'язним списком потрібен вказівник на його початок, тому ми використовували **typedef**, щоб перетворити **List** на ім'я для вказівника цього типу. Таким чином, оголошення **List** movies; розглядає **movies** як вказівник, прийнятний для посилання на зв'язний список.

Чи є цей спосіб визначення типу **List** єдиним? Ні. Наприклад, для відстеження кількості записів можна було б задіяти змінну:

```
typedef struct list
{
Node *head;     // вказівник на заголовок списку
int size;       // кількість записів у списку
}
List;             // альтернативне визначення списку
```

Можна було б додати другий вказівник, призначений для відстеження кінця списку. Пізніше ви побачити відповідний приклад. Поки давайте обмежимося першим визначення типу **List**. Важливо пам'ятати, що оголошення

```
List movies;
```

слід розглядати як визначення списку, а не встановлення вказівника на вузол або структуру.

Точне представлення даних списку **movies** є деталлю реалізації, яка не повинна бути помітною на рівні інтерфейсу.

Наприклад, під час запуску програма повинна ініціалізувати вказівник на заголовок значенням **NULL**, але не слід застосовувати код на зразок `movies = NULL;`

Це пов'язано з тим, що у подальшому може з'ясуватися, що реалізація типу **List** у вигляді структури підходить більше, і тоді буде потрібна наступна ініціалізація:

```
movies.next = NULL;
movies.size = 0;
```

Ніхто з тих, хто використовує тип **List**, не повинен турбуватися про подібні нюанси. Замість цього повинна бути можливість записувати приблизно такий код:

```
InitializeList(movies);
```

Програмістам треба знати тільки про те, що для ініціалізації списку вони повинні застосовувати функцію **InitializeList()**. Вони не зобов'язані знати точну реалізацію даних для змінної **List**. Це є прикладом приховування даних – мистецтва маскування подробиць представлення даних від більш високих рівнів програмування.

Для надання рекомендацій користувачу прототип функції можна супроводжувати наступними рядками:

```
//============================================================
// Операція:     ініціалізація списку
// Передумова:   plist вказує на список list
// Постумова:    список ініціалізований пустим змістом
//============================================================
void InitializeList(List *plist);
```

Є три моменти, на які ви повинні звернути увагу. По-перше, коментарі описують передумови, тобто умови, які повинні бути задовільнені до виклику функції. Наприклад, в даному випадку потрібен список, що призначений для

ініціалізації. По-друге, коментарі описують постумови – умови, які повинні бути задовільнені після виконання функції. Нарешті, по-третє, в якості свого аргументу функція використовує вказівник на список, а не сам список, тому виклик функції буде мати такий вигляд:

```
InitializeList(&movies);
```

Причина полягає в тому, що в **C** аргументи передаються по значенню. Таким чином, єдиний спосіб дозволити функції **C** змінювати значення з програми, що викликає, передбачає використання вказівника на цю змінну. Як можна побачити, в даному випадку обмеження мови призводять до певної відмінності інтерфейсу від його абстрактного опису.

Прийнятий в мові **C** метод об'єднання інформації про тип і функції до єдиного пакету передбачає розміщення визначень для типу та прототипів функцій (в тому числі коментарів з перед- і постумовами) у файлі заголовку. Цей файл повинен надавати всю інформацію, яка необхідна програмісту для використання типу. Файл заголовку **list.h** для простого типу **List** має наступний вигляд:

```
//=============================================================
// list.h - файл заголовку для простого типу списку
//=============================================================
#ifndef LIST_H_
#define LIST_H_
#include <stdbool.h>     // Функціональна можливість C99

//=============================================================
// Оголошення, які є специфічними для програми
//=============================================================
#define TSIZE 45         // розмір масиву для зберігання назви
struct film
{
    char title[TSIZE];
int rating;
};

//=============================================================
// Оголошення спільних типів
```

```
//============================================================
typedef struct film Item; typedef struct node
{
    Item item;
struct node *next;
} Node; typedef
Node *List;
//============================================================
// Прототипи функцій
//============================================================
// Операція:   ініціалізація списку
// Передумова: plist вказує на список list
// Постумова:  список ініціалізований пустим змістом
//============================================================
void InitializeList(List *plist);
//============================================================
// Операція:   визначення, чи є список пустим
// Передумова: plist вказує на ініціалізований список
// Постумови:  функція повертає значення True, якщо список
//             пустий, і False в іншому випадку
//============================================================
bool ListIsEmpty(const List *plist);
//============================================================
// Операція:   визначення, чи є список повним
// Передумова: plist вказує на ініціалізований список
// Постумови:  функція повертає значення True, якщо список
//             повний, і False в іншому випадку
//============================================================
bool ListIsFull(const List *plist);
//============================================================
// Операція:   визначення кількості елементів у списку
// Передумова: plist вказує на ініціалізований список
// Постумови:  функція повертає кількість елементів у списку
//============================================================
unsigned int ListItemCount(const List *plist);
//============================================================
// Операція:   додавання елемента в кінець списку
// Передумови: item – елемент, що додається до списку
//             plist вказує на ініціалізований список
// Постумови:  якщо це можливо, функція додає елемент в
//             кінець списку і повертає значення True;
//             в іншому випадку повертає значення False
//============================================================
bool AddItem(Item item, List *plist);
//============================================================
// Операція:   застосування функції до кожного елемента списку
// Передумови: plist вказує на ініціалізований список
//             pfun вказує на функцію, яка приймає
//             аргумент Item і не має значення, що повертається
// Постумови:  функція, на яку вказує pfun, виконується один
```

```
//             раз для кожного елемента в списку
//=============================================================
void Traverse(const List *plist, void (* pfun)(Item item));
//=============================================================
// Операція:    звільнення виділеної пам'яті, якщо вона є
// Передумова:  plist вказує на ініціалізований список
// Постумови:   будь-яка пам'ять, що виділяється для списку,
//              звільняється, і список встановлюється
//              в пустий стан
//=============================================================
void EmptyTheList(List *plist);
#endif
```

В ньому конкретна структура визначена як та, що належить до типу **Item**, після чого тип **Node** визначений в термінах **Item** і тип **List** – в термінах **Node**. Потім у функціях, які надають операції над списком, типи **Item** і **List** застосовуються для аргументів. Якщо функції необхідно модифікувати аргумент, вона використовує вказівник на відповідний тип, а не сам тип напряму. В файлі імена функцій починаються з прописних літер для їх позначення як частини інтерфейсного пакету. Крім того, для захисту від множинного включення файлу застосовується прийом з **#ifndef**. Якщо ваш компілятор не підтримує тип **bool** зі стандарту **C99**, можете замінити у файлі заголовку рядок

```
#include <stdbool.h>           // функціональна можливість C99 на
enum bool { false, true };     // визначення bool як типу,
                               // і false, true - як значень
```

Список модифікує тільки функції **InitializeList()**, **AddItem()** і **EmptyTheList()**, тому формально тільки вони потребують на аргумент типу вказівника. Однак якщо б користувачу довелося пам'ятати про необхідність передавання аргументу **List** одним функціям і його адреси іншим, то це могло б призводити до плутанини. Таким чином, для спрощення задачі користувача в усіх функціях використовується аргументи типу вказівників.

Один з прототипів у файлі заголовку дещо складніший за інші:

```
//=============================================================
// Операція:   застосування функції до кожного елемента списку
// Передумови: plist вказує на ініціалізований список
```

```
//              pfun вказує на функцію, яка приймає
//              аргумент Item і не має значення, що повертається
// Постумови:  функція, на яку вказує pfun, виконується один
//              раз для кожного елемента в списку
//============================================================
void Traverse(const List *plist, void (* pfun)(Item item));
```

Аргумент **pfun** представляє собою вказівник на функцію. В цьому випадку він є вказівником на функцію, яка приймає значення **item** в якості аргументу і не має значення, що повертається. Вказівник на функцію можна передавати у вигляді аргументу іншій функції, яка зможе викликати цю вказану функцію. Так, наприклад, **pfun** може вказувати на функцію, що відображає елемент. Функція **Traverse()** буде застосовувати цю функцію до кожного елемента списку, в результаті відображаючи увесь список.

3.2.3. Використання інтерфейсу

Цей інтерфейс можна використовувати для написання програми, не маючи жодних додаткових деталей – наприклад, нічого не знаючи про те, як реалізовані функції інтерфейсу. Напишемо нову версію програми виводу інформації про фільми ще до створення допоміжних функцій. Оскільки інтерфейс визначений в термінах типів **List** і **Item**, програма повинна бути створена із застосуванням саме цих типів.

Нижче за допомогою псевдокоду наведено один з можливих планів:

1) Створити змінну **List**.

2) Створити змінну **Item**.

3) Ініціалізувати список пустим змістом.

4) Поки список не заповнений та є вхідні дані:

- прочитати вхідні дані та помістити їх до змінної **Item**; - додати елемент в кінець списку.

5) Переглянути кожен елемент списку та відобразити його.

Програма, яка відповідає цьому базовому плану, має наступний вигляд:

```
//=============================================================
// main.c - файл для використання зв'язного списку в стилі ADT
//=============================================================
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>       // прототип для exit()
#include "list.h"         // визначення List, Item

void showmovies(Item item);
char *s_gets(char *st, int
n);

int main(void)
{
    List movies;
    Item temp;

    SetConsoleOutputCP(1251);

    InitializeList(&movies);
if(ListIsFull(&movies))
    {
        fprintf(stderr, "Доступна пам'ять відсутня! "
                "Програма завершена.\n");
        exit(1);
    }
    // Збір та зберігання інформації
puts("Введіть назву першого фільму:");
while(s_gets(temp.title, TSIZE) != NULL &&
temp.title[0] != '\0')
    {
        puts("Введіть своє значення рейтингу <0-
10>:");          scanf("%d", &temp.rating);
while(getchar() != '\n')              continue;
        if(AddItem(temp, &movies) == false)
        {
            fprintf(stderr, "Проблема з виділенням пам'яті\n");
break;
        }
        if(ListIsFull(&movies))
        {
            puts("Список повний.");
break;
        }
        puts("Введіть назву наступного фільму");
puts("(або порожній рядок для припинення вводу):");
    }
```

```
    // відображення списку
if(ListIsEmpty(&movies))
printf("Дані не введені.");      else
    {
        printf("=========================================\n");
printf("Список фільмів:\n");
        printf("=========================================\n");
        Traverse(&movies, showmovies);
    }
    printf("=========================================\n");
printf("Ви ввели %d фільмів.\n", ListItemCount(&movies));
printf("=========================================\n");
    // очищення списку
EmptyTheList(&movies);
printf("Програма завершена.\n");
return 0;
}
 void showmovies(Item
item)
{
    printf("Рейтинг: %3d.  Фільм: %s\n", item.rating,
item.title);
}
 char *s_gets(char *st,
int n)
{
    char *ret_val;
char *find;

    ret_val = fgets(st, n, stdin);
    if(ret_val)
    {
        find = strchr(st, '\n');
if(find)
            *find = '\0';
else
            while(getchar() != '\n')
                continue;       // відкинути залишок рядка
    }
    return ret_val;
}
```

Даний файл необхідно компілювати разом з **list.c**.

Крім того, до неї додано код для перевірки помилок. Погляньте, яким чином в ній використовується інтерфейс, що описаний у файлі **list.h**. Зверніть також увагу, що текст програми містить код функції **showmovies()**, яка відповідає прототипу, що потребує функція **Traversed()**. Тому програма може

передавати вказівник **showmovies** до функції **Traverse()**, щоб та могла застосовувати функцію **showmovies()** до кожного елемента списку (ім'я функції є вказівником на цю функцію).

3.2.4. Реалізація інтерфейсу

Реалізуємо інтерфейс **List**. Підхід, що прийнятий в **C**, передбачає збір визначень функцій у файлі на ім'я **list.c**. Тоді повна програма буде складатися з трьох файлів: **list.h**, в якому визначені структури даних і наведені прототипи для інтерфейсу користувача, **list.c**, що містить код функцій для реалізації інтерфейсу, і **main.c**, що являє собою файл початкового коду, де інтерфейс списку застосовується для вирішення конкретної задачі. Одна з можливих реалізацій файлу **list.c** має наступний вигляд:

```
//=========================================================
// list.c - файл для функцій підтримки операцій зі списком
//=========================================================
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

// прототип локальної функції static void
CopyToNode(Item item, Node *pnode); 14

//=========================================================
// Функції інтерфейсу
//=========================================================
//=========================================================
// Встановлює список в пустий стан (ініціалізація)
//=========================================================
void InitializeList(List *plist)
{
    *plist = NULL;
}

//=========================================================
// Повертає true, якщо список пустий
//=========================================================
bool ListIsEmpty(const List *plist)
{
```

```
    if(*plist == NULL)
return true;     else
        return false;
}

//================================================================
// Повертає true, якщо список повний
//================================================================
bool ListIsFull(const List *plist)
{
    Node *pt;
bool full;

    pt = (Node *)
malloc(sizeof(Node));     if(pt ==
NULL)          full = true;     else
        full =
false;
free(pt);
return full;
}
//================================================================
// Повертає кількість вузлів
//================================================================
unsigned int ListItemCount(const List *plist)
{
    unsigned int count = 0;
    Node *pnode = *plist;     // встановлюємо на початок списку
while(pnode != NULL)
    {
        ++count;
        pnode = pnode->next;  // встановлюємо на наступний вузол
    }
return count;
}
//================================================================
// Створює вузол для зберігання елемента і додає його в кінець
// списку, на який вказує змінна plist (повільна реалізація)
//================================================================
bool AddItem(Item item, List *plist)
{
    Node *pnew;
    Node *scan = *plist;
    pnew = (Node *) malloc(sizeof(Node));
if(pnew == NULL)
        return false;     // вихід з функції у разі помилки
    CopyToNode(item, pnew);
pnew->next = NULL;
    if(scan == NULL)      // список пустий, тому треба помістити
```

```
        *plist = pnew;     // pnew на початок списку
else
    {
        while(scan->next != NULL)
            scan = scan->next;         // пошук кінця списку
scan->next = pnew;             // додавання pnew в кінець
    }
return true;
}
//=================================================================
// Відвідує кожен вузол і виконує функцію, на яку вказує pfun
//=================================================================
void Traverse(const List *plist, void (*pfun)(Item item))
{
    Node *pnode = *plist;       // встановлюємо на початок списку
while(pnode != NULL)
    {
        (*pfun)(pnode->item);  // застосовуємо функцію до
елементу          pnode = pnode->next;    // перехід до наступного
елемента
    }
}
//=================================================================
// Звільняє пам'ять, яка була виділена функцією malloc()
// Встановлює вказівник списку в NULL
//=================================================================
void EmptyTheList(List *plist)
{
    Node *psave;
while(*plist != NULL)
    {
        psave = (*plist)->next; // зберігання адреси наступного
                                // вузла
        free(*plist);           // звільнення поточного вузла
*plist = psave;          // перехід до наступного вузла
    }
}
//=================================================================
// Визначення локальної функції
// Копіює елемент у вузол
//=================================================================
static void CopyToNode(Item item, Node *pnode)
{
    pnode->item = item;        // копіювання структури
}
```

Щоб запустити програму, необхідно скомпілювати обидва файли **main.c** і **list.c** та скомпонувати їх. Разом файли **list.h**, **list.c** і **main.c** утворюють завершену програму. Результат роботи програми наведено на рис. 3.7.

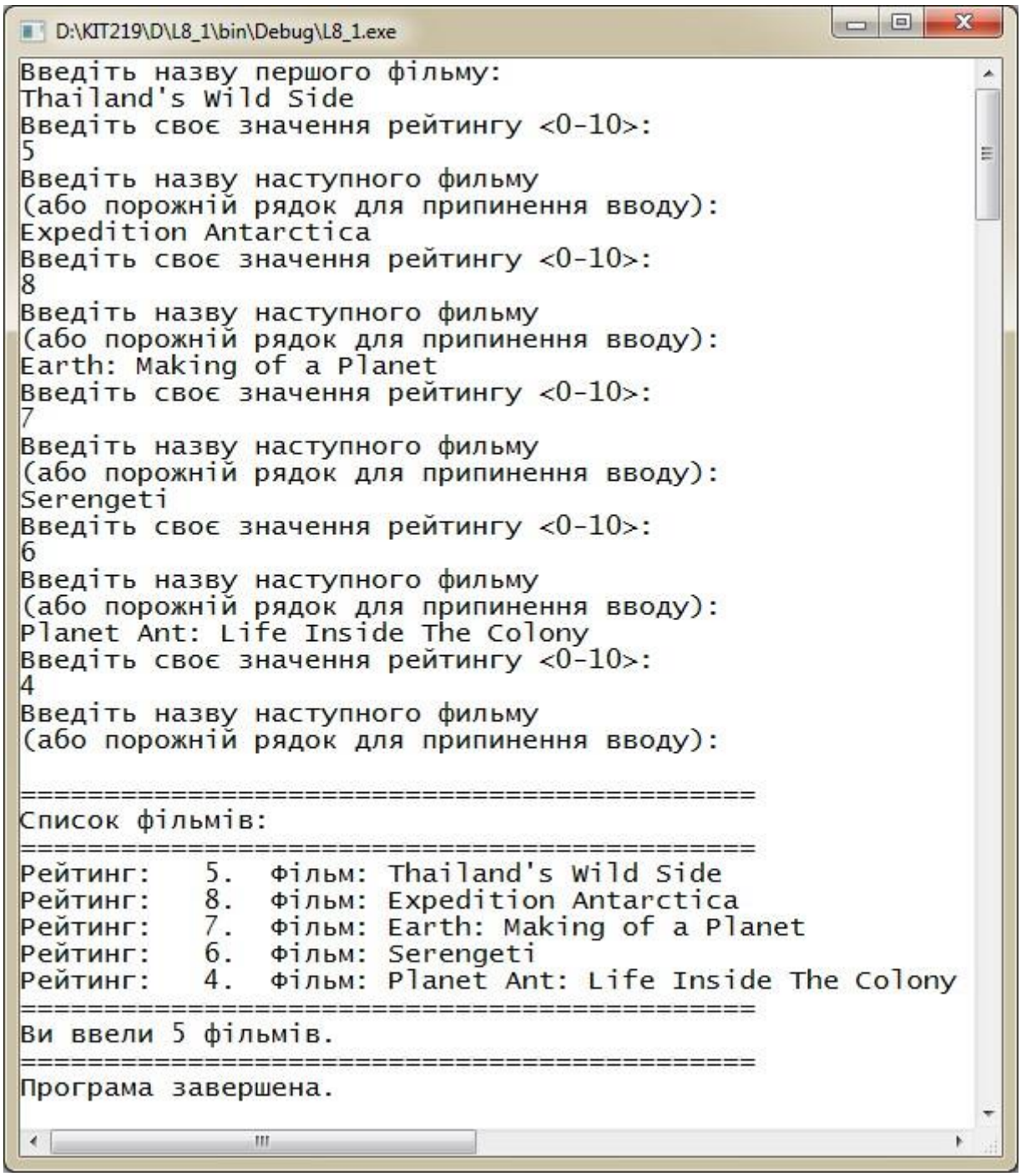


Рисунок 3.7 – Результат виконання програми для демонстрації роботи зі списками

З файлом `list.c` пов'язано багато цікавих особливостей. Наприклад, він ілюструє ситуацію, коли можна використовувати функції з внутрішнім зв'язуванням, які відомі тільки у файлі, де вони визначені. При реалізації інтерфейсу іноді зручно застосовувати допоміжні функції, які не є частиною офіційного інтерфейсу. Наприклад, в наведеній програмі функція `CopyToNode()`

використовується для копіювання значення типу **Item** до змінної типу **Item**. Оскільки ця функція є частиною реалізації, але не інтерфейсу, за допомогою кваліфікатора класу зберігання **static** ми приховали її у файлі **list.c**. А тепер давайте проаналізуємо інші функції.

Функція **InitializeList()** ініціалізує список пустим вмістом. В нашій реалізації це означає встановлення змінної типу **List** в **NULL**. Як згадувалося раніше, це потребує надати до функції вказівник на змінну типу **List**.

Функція **ListEmpty()** є досить простою, але вона спирається на те, що змінна списку встановлена в **NULL**, коли список є пустим. Таким чином, важливо ініціалізувати список до першого виклику функції **ListEmpty()**.

Крім того, якщо ви розширите інтерфейс, включивши до нього засіб для видалення елементів, то треба переконатися, що функція видалення переводить список в пустий стан після видалення останнього елемента. У випадку застосування зв'язного списку його розмір обмежений об'ємом доступної пам'яті. Функція **ListIsFull()** намагається виділити об'єм пам'яті, який буде достатнім для нового елемента. Якщо це їй не вдасться, значить, список заповнений. Якщо спроба була успішною, функція повинна звільнити щойно виділену пам'ять, щоб вона була доступною для реального елемента.

Функція **ListItemCount()** використовує звичний алгоритм обходу зв'язного списку, підраховуючи при цьому кількість елементів:

```
unsigned int ListItemCount(const List *plist)
{
  unsigned int count = 0;
  Node *pnode = *plist;     // встановлення на початок списку
while(pnode != NULL)
    {
     ++count;
     pnode = pnode->next; // встановлення на наступний вузол
    }
return count;
}
```

Функція **AddItem()** є найбільш складною з усіх:

```
bool AddItem(Item item, List *plist)
{
```

```
    Node *pnew;
    Node *scan = *plist;
    pnew = (Node *) malloc(sizeof(Node));
if(pnew == NULL)
        return false;          // вихід з функції у разі
помилки
    CopyToNode(item, pnew);
pnew->next = NULL;
    if(scan == NULL)         // список пустий, тому помістити
        *plist = pnew;       // pnew на початок списку
else
    {
        while(scan->next != NULL)
            scan = scan->next; // пошук кінця списку
scan->next = pnew;      // додавання pnew в кінець
    }
return true;
}
```

Спочатку функція **AddItem()** виділяє пам'ять для нового вузла. Якщо це їй вдається, вона застосовує функцію **CopyToNode()** для копіювання елемента у вузол. Потім вона встановлює член **next** вузла в **NULL**. Це є сигналом того, що даний вузол є останнім у зв'язному списку. І, нарешті, після створення вузла і присвоювання відповідних значень його членам, функція приєднує вузол до кінця списку. Якщо це перший доданий елемент списку, програма встановлює вказівник на заголовок на перший елемент. В протилежному випадку код робить прохід по зв'язному списку до тих пір, поки не виявить елемент, член **next** якого встановлений в **NULL**. В поточний момент цей вузол є останнім у списку, тому функція перевстановлює його член **next**, щоб він вказував на новий вузол.

Прийнята практика програмування потребує виклику функції **ListIsFull()** перед намаганням додавання елемента до списку. Однак користувач може упустити цей момент, тому функція **AddItem()** самостійно перевіряє успішність виклику функції **malloc()**. Крім того, цілком імовірно, що між викликами функцій **ListIsFull()** і **AddItem()** користувач міг виконати ще будь-які дії з виділення пам'яті, тому краще на всяк випадок перевірити, чи спрацювала функція **malloc()**.

Функція **Traverse()** аналогічна до **ListItemCount()**, але в ній функція застосовується до кожного елемента списку:

```
void Traverse(const List *plist, void (* pfun)(Item item))
{
    Node *pnode = *plist;       // встановлення на початок списку
while(pnode != NULL)
    {
    (*pfun)(pnode->item);  // застосування функції до елемента
pnode = pnode->next;    // перехід до наступного елемента

} }
```

Згадайте, що **pnode->item** представляє дані, що зберігаються у вузлі, а **pnode->next** ідентифікує наступний вузол у зв'язному списку. Наприклад, **Traverse(movies, showmovies);** застосовує функцію **showmovies()** до кожного елемента у списку.

Нарешті, функція **EmptyTheList()** звільнює пам'ять, яка раніше була виділена за допомогою **malloc()**:

```
void EmptyTheList(List *plist)
{
    Node *psave;
while(*plist != NULL)
   {
psave = (*plist)->next; // збереження адреси наступного вузла
free(*plist);           // звільнення поточного вузла
*plist = psave;         // перехід до наступного вузла
   }
}
```

В цій реалізації пустий список позначається шляхом встановлення змінної **List** в **NULL**. Відповідно, щоб можна було модифікувати змінну **List**, до функції треба передати адресу цієї змінної. Оскільки **List** вже є вказівником, то **plist** – це вказівник на вказівник. Таким чином, всередині коду вираз ***plist** має тип вказівника на **Node**. Коли список закінчується, значення ***plist** дорівнює **NULL**, тобто початковий фактичний аргумент тепер встановлено в **NULL**.

Код зберігає адресу наступного вузла, оскільки в принципі виклик функції **free()** може зробити недоступним вміст поточного вузла, на який посилається вказівник ***plist**.

Треба звернути увагу на те, що деякі функції обробки списку мають у якості параметру вираз **List *plist**. Це означає той факт, що такі функції не модифікують список. В даному випадку **const** забезпечує певний захист, запобігаючи зміні вказівника ***plist** (величини, на яку вказує **plist**). В нашій програмі **plist** вказує на **movies**, тому специфікатор **const** запобігає модифікації цими функціями змінної **movies**, яка, у свою чергу, вказує на перше посилання у списку. Таким чином, код на зразок того, що вказаний нижче є неприпустимим, скажімо, у функції **ListItemCount()**:

```
*plist = (*plist)->next; // не дозволено, якщо *plist - константа
```

Це добре, оскільки зміна ***plist** і, відповідно, **movies** призвела б до втрати програмою можливості відстеження даних. Однак те, що змінні ***plist** і **movies** трактуються як **const**, зовсім не означає, що дані, на які вказує ***plist** або **movies**, є константами. Наприклад, наступний код є цілком припустимим:

```
(*plist)->item.rating = 3;       // дозволено, навіть якщо
                                 // *plist - константа
```

Причина полягає в тому, що цей код не модифікує змінну ***plist**. Він змінює дані, на які вказує ***plist**. У зв'язку з цим, на **const** неможна повністю покладатися при виявленні програмних помилок, які призводять до випадкової модифікації даних.

3.2.5. Висновки щодо використання ADT

Зробимо оцінку того, що нам дав підхід з використанням **ADT**. Спочатку порівняємо програми з цього підрозділу та яка наведена у п3.2. В обох програмах для вирішення задачі зі створення списку фільмів застосовується один і той самий фундаментальний метод (динамічне виділення пам'яті для зв'язаних структур). Але програма минулого підрозділу показує усі програмні нюанси, поміщаючи **malloc()** і **prev->next** у відкрите представлення. З іншого боку, код наведений в підроздлі 3.3. приховує ці деталі та подає програму на мові, яка безпосередньо пов'язана з вирішуваним завданням. Це означає, що в ньому мова йде про створення списку та додавання до нього елементів, а не про виклик

функцій керування пам'яттю або про переустановлення вказівників. Іншими словами він представляє програму в термінах вирішуваної задачі, а не в термінах низькорівневих інструментів, що необхідні для її вирішення. Версія з **ADT** орієнтована на проблеми кінцевого користувача, тому читати її значно легше.

Разом файли **list.h** і **list.c** утворюють ресурс, що може використовуватися багатократно. Якщо вам потрібен простий список елементів іншого типу, достатньо звернутися до цих файлів.

Припустимо, що вам необхідно зберігати відомості про своїх родичів: імена, родинні відносини, адреси та номери телефонів. Перш за все, слід звернутися до файлу **list.h** і перевизначити тип **Item**:

```
typedef struct itemtag
{
char fname[14];
char lname[24];
char relationship[36];
char address[60];
char phonenum[20]; }
Item;
```

В даному випадку це все, що ви повинні були зробити, оскільки усі функції простого списку визначені в термінах типу **Item**. В деяких випадках прийшлось би також перевизначити функцію **CopyToNode()**. Наприклад, якщо б елемент був масивом, то його не вдалось би копіювати за допомогою операції присвоювання.

Ще один важливий момент пов'язаний з тим, що інтерфейс користувача визначений в термінах операцій абстрактного списку, а не конкретного набору представлень даних та алгоритмів. Це дозволяє вільно маніпулювати реалізацією, не переробляючи кінцеву програму. Наприклад, створена нами функція **AddItem()** є дещо неефективною, оскільки вона завжди починає роботу з початку списку і потім здійснює пошук його кінця. Вказаний недолік можна усунути, відстежуючи кінець списку. Наприклад, тип **List** можна перевизначити наступним чином:

```
typedef struct list
{
    Node *head;          // вказує на початок списку
```

```
    Node *end;          // вказує на кінецьсписку
}
List;
```

Звичайно, після цього довелося б переписувати функції обробки списку, застосувавши це нове визначення, але не треба було б змінювати щось у самій програмі. Такий вид ізолювання реалізації від фінального інтерфейсу особливо корисний у великомасштабних програмних проектах. Цей підхід називається **приховуванням даних**, оскільки докладне представлення даних приховано від кінцевого користувача.

Зверніть увагу, що цей конкретний тип **ADT** навіть не потребує реалізації простого списку у вигляді зв'язного списку. Нижче показана ще одна можливість:

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSTZE];    // масив елементів
    int items;                // кількість елементів у списку
 }
List;
```

Це знову потребує переписування файлу **list.c**, але програма, що використовує такий список, не потребує змін.

І, нарешті, подумайте про переваги, які даний підхід надає процесу розробки програм. Якщо щось працює не так як треба, цілком імовірно, що проблему вдасться локалізувати з точністю до функції. Якщо вдасться придумати більш ефективний спосіб вирішення однієї з задач, такої як додавання елемента, то доведеться переписати тільки цю одну функцію. Якщо треба нова функціональна можливість, задачу можна вирішити шляхом додавання нової функції до пакету. Якщо здається, що масив або двобічно зв'язаний список є більш зручними, можна модифікувати реалізацію, не змінюючи програми, які користуються цією реалізацією.

3.3. Черги

3.3.1. Створення черги за допомогою ADT

Підхід до програмування на **C** з застосуванням абстрактних типів даних передбачає виконання:

- опису типу в абстрактній узагальненій манері разом з його операціями; - визначення інтерфейсу у вигляді функцій для представлення нового типу; - написання коду для реалізації інтерфейсу.

Цей підхід був задіяний при створенні простого списку. Тепер скористаємося ним для побудови дещо складнішого об’єкта – черги.

3.3.2. Визначення абстрактного типу даних для представлення черги

Черга – це список, який має дві особливі властивості: - нові елементи можуть додаватися тільки наприкінці списку; - елементи можуть видалятися тільки на початку списку.

Чергу можна порівняти з ланцюжком людей, які стоять один за одним до квиткової каси. Кожна нова людина стає в кінець ланцюжка та залишає її на самому початку (після придбання квитків).

Черга є формою даних типу «першим прибув – першим був обслуговани́й» (**First In First Out – FIFO**), подібною до черги в касу (якщо тільки ніхто не уклиниться в чергу).

Нижче наведене неформальне абстрактне визначення черги:

Ім’я типу:	Черга
Властивість типу:	Може містити упорядковану послідовність елементів
Операції типу:	1) Ініціалізація черги пустим вмістом. 2) З’ясування, чи є черга пустою. 3) З’ясування, чи є черга повною. 4) Визначення кількості елементів у черзі. 5) Додавання елемента наприкінці черги. 6) Видалення та відновлення елемента на початку черги. 7) Спустошення черги.

3.3.3. Визначення інтерфейсу

Визначення інтерфейсу буде поміщено до файлу `queue.h`. За допомогою засобу `typedef` мови `C` ми створимо імена для двох типів: `Item` і `Queue`. Точна реалізація відповідних структур повинна знаходитися у файлі `queue.h`, але концептуально проектування структур є частиною етапу детальної реалізації. Будемо вважати, що типи визначені, і зосередимо увагу на прототипах функцій.

Перш за все, слід подумати про ініціалізацію. Вона передбачає зміну типу `Queue`, тому функція повинна приймати в якості аргументу адресу змінної `Queue`:

```
void InitializeQueue(Queue *pq);
```

З'ясування, чи є черга пустою або повною, передбачає застосування функцій, які повинні повертати істинне або хибне значення. В нашому випадку будемо вважати, що файл заголовку `stdbool.h` стандарту `C99` є доступним. Якщо це не так, можна використовувати тип `int` або визначити тип `bool` самостійно. Оскільки функція не змінює чергу, вона може приймати аргумент `Queue`. З іншого боку, в залежності від реального розміру об'єкта типу `Queue`, передавання тільки адреси змінної `Queue` може проходити швидше та з меншими витратами пам'яті. Ще одна перевага такого підходу полягає в тому, що усі функції будуть приймати в якості аргументу адресу. Для позначення того, що функції не змінюють чергу, краще застосовувати кваліфікатор `const`:

```
bool QueueIsFull(const Queue *pq);
bool QueueIsEmpty(const Queue *pq);
```

Інакше кажучи, вказівник `pq` посилається на об'єкт даних `Queue`, який не може змінюватися за допомогою `pq`. Аналогічний прототип можна визначити для функції, яка повертає кількість елементів у черзі:

```
int QueueItemCount(const Queue *pq);
```

Додавання елемента наприкінці черги передбачає ідентифікацію елемента та черги. Цього разу черга змінюється, таким чином використання вказівника є обов'язковим. Функція може мати тип `void` або ж значення, що повертається,

можна застосовувати для позначення успішності або неуспішності виконання операції з додавання елементу. Давайте застосуємо другий підхід:

```
bool EnQueue(Item item, Queue *pq);
```

Нарешті, видалення елемента може бути реалізовано декількома способами. Якщо елемент визначається як структура або один з фундаментальних типів, функція може його повертати. Аргументом функції могла б бути змінна `Queue` або вказівник на неї. Таким чином, один з можливих прототипів має наступний вигляд:

```
Item DeQueue(Queue q);
```

Однак наступний прототип є більш загальним:

```
bool DeQueue(Item *pitem, Queue *pq);
```

Елемент, що видаляється з черги, поміщається в місце, на яке посилається вказівник `pitem`, а значення, що повертається, відповідає на питання, чи успішно виконана операція. Єдиним аргументом, який повинен бути наданий функції спустошення черги, є адреса черги, що і демонструє наведений нижче прототип:

```
void EmptyTheQueue(Queue *pq);
```

3.3.4. Реалізація представлення даних інтерфейсу

Перший крок передбачає рішення про те, яка форма даних `C` буде використовуватися для черги. Одним з варіантів є масив. Переваги масивів пов'язані з простотою їх застосування та легкістю додавання елемента наприкінці заповненої частини масиву. Проблема виникає, коли справа доходить до видалення елемента на початку черги.

Якщо знову скористатися аналогією черги за квитками, видалення елемента на початку черги полягає в копіюванні значення першого елемента масиву і подальшим переміщенням кожного елемента, що залишилися в масиві, на одну позицію в напрямку його початку. Хоча ці дії легко програмувати, вони займають багато процесорного часу. На рис. 3.8 наведено перший спосіб використання масиву в якості черги.

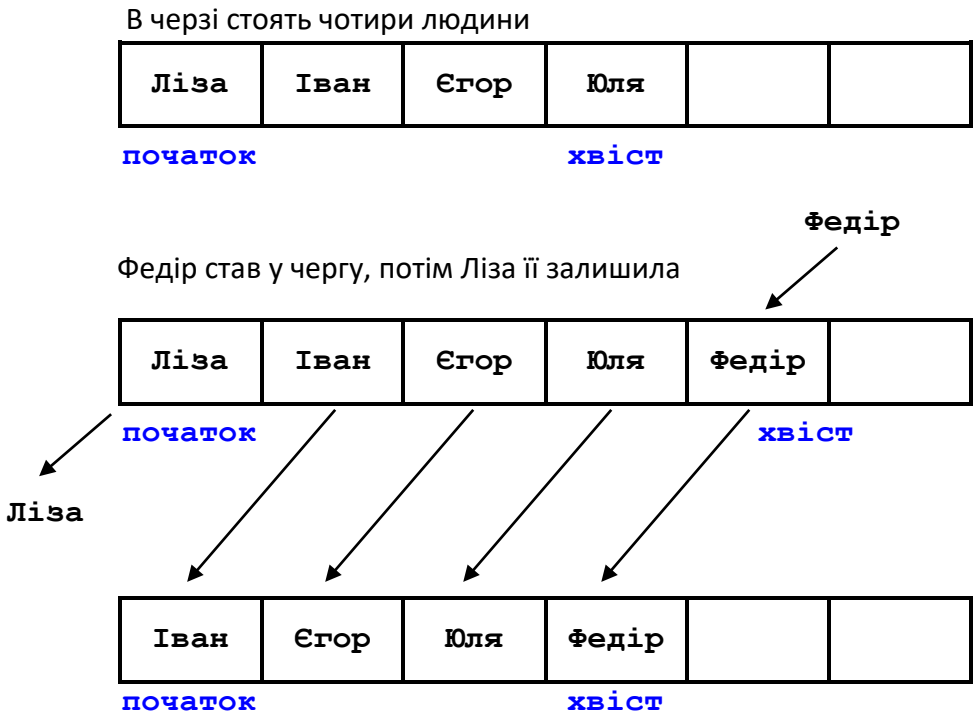


Рисунок 3.8 – Використання масиву в якості черги

Другий спосіб вирішення задачі видалення в реалізації з застосуванням масиву – залишити елементи в позиціях, де вони знаходяться, і потім змінити елемент, який вважається початковим (рис. 3.9). Проблема цього метода полягає в тому, що місце, яке раніше було зайняте видаленими елементами, витрачається даремно, що веде до зменшення доступного простору в черзі.

Більш вдалим рішенням проблеми простору, що витрачається даремно, є перетворення черги на **кільцеву**. Це означає, що кінець масиву повинен бути поєднаний з його початком. Тобто уявіть, що перший елемент масиву йде безпосередньо за останнім елементом, тому при досягненні кінця масиву ви починаєте додавати елементи в початкові позиції, немов вони були звільнені (рис. 3.10). Такий процес можна порівняти з рисуванням на паперовій стрічці, яка склеєна в кільце. Звичайно, тепер доведеться виконувати додаткові дії щодо забезпечення того, щоб кінець черги не перекривав її початок.

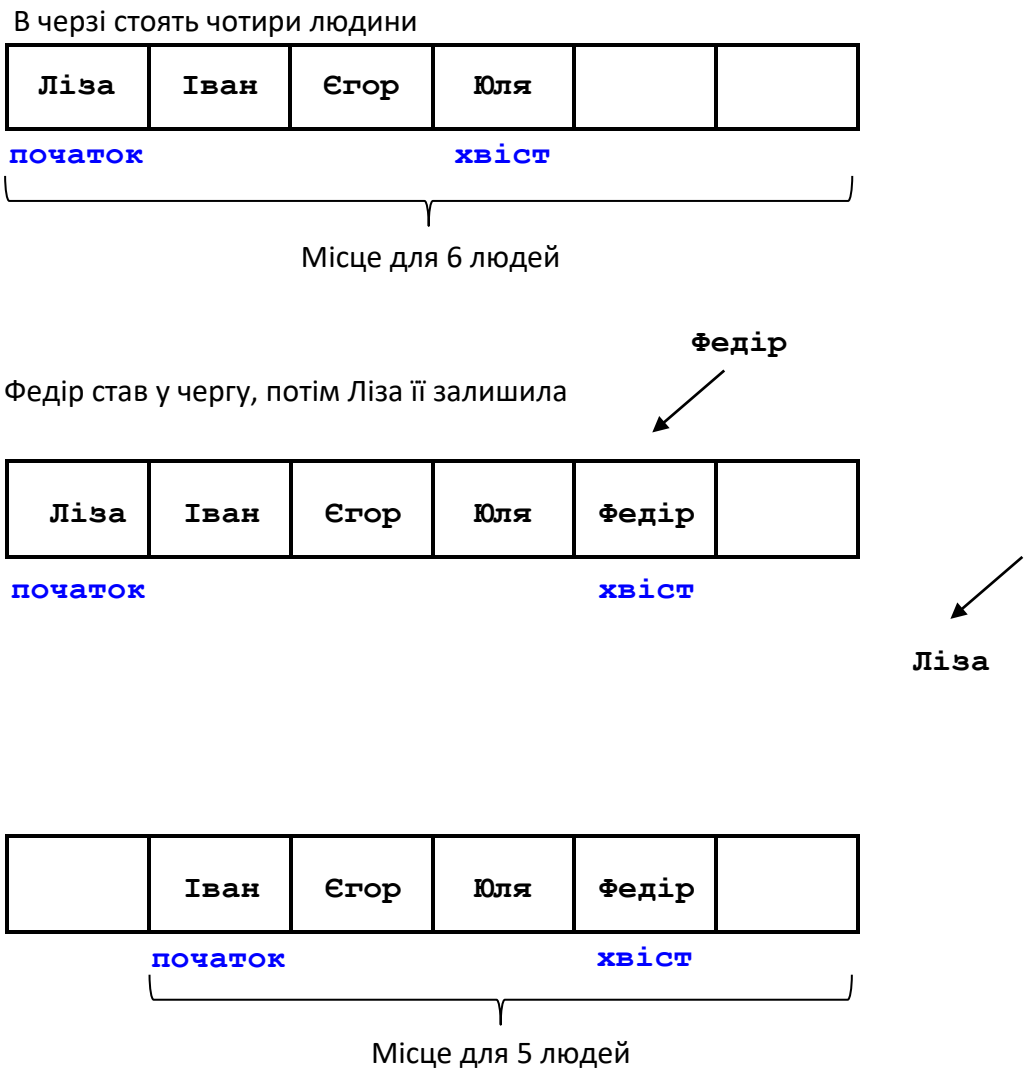


Рисунок 3.9 – Перевизначення початкового елемента в черзі

Ще одне можливе рішення передбачає використання зв’язного списку. Перевага цього підходу полягає в тому, що видалення початкового елемента не потребує переміщення усіх інших елементів. Натомість потрібно просто перевстановити вказівник на початок, щоб він вказував на новий перший елемент.

Оскільки ми вже працювали зі зв’язними списками, то підемо саме цим шляхом.

Щоб перевірити свої ідеї, почнемо зі створення черги цілих чисел:

```
typedef int Item;
```

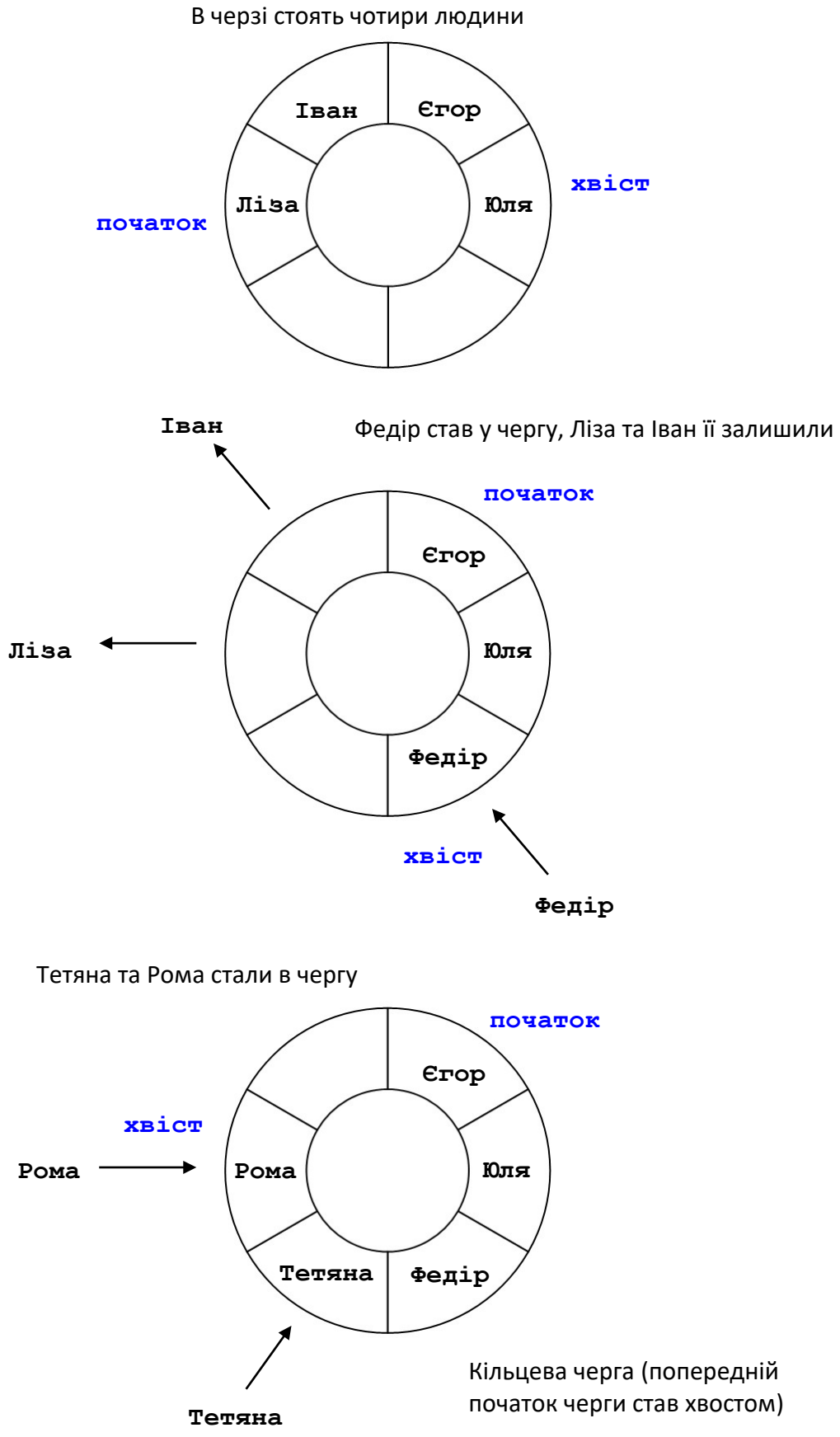


Рисунок 3.10 – Кільцева черга

Зв’язний список буде побудовано з вузлів, тому давайте визначимо вузол:

```
typedef struct node
{
    Item item;
struct node *next;
} Node;
```

Для черги необхідно відстежувати початковий і кінцевий елементи. Це можна робити з застосуванням вказівників. Крім того, можна використовувати лічильник для відстеження кількості елементів у черзі. Таким чином, структура буде містити два члени типу «вказівник» і один член типу **int**:

```
typedef struct queue
{
    Node *front;     // вказівник на початок черги
Node *rear;      // вказівник на кінець черги
int items;       // кількість елементів у черзі }
Queue;
```

Зверніть увагу, що **Queue** – це структура з трьома членами, тому раніше прийняте рішення про використання в якості аргументів вказівників на черги, а не самих черг, економить час і об’єм пам’яті, що витрачається.

Тепер пора подумати про розмір черги. У випадку зв’язного списку розмір черги обмежений об’ємом доступної пам’яті, але часто має сенс застосовувати чергу значно меншого розміру. Наприклад, чергу можна використовувати для емуляції літаків, що очікують приземлення в аеропорту. Якщо кількість літаків, що очікують, стає значно більшою, нові літаки, які прибувають, можуть спрямовуватися в інші аеропорти. Ми встановимо максимальний розмір черги, який дорівнює **10**. Визначення та прототипи інтерфейсу черги наведені у файлі заголовку **queue.h**, який має наступний вигляд:

```
//=====================================================================
// queue.h - файл заголовку для інтерфейсу черги
//=====================================================================
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>
#define MAXQUEUE 10     // максимальна кількість елементів в черзі
typedef int Item; typedef struct node
{
```

```
Item item;
struct node *next;
} Node;
 typedef struct
queue
{
    Node *front;          // вказівник на початок черги
    Node *rear;           // вказівник на хвіст черги
    int items;            // кількість елементів у черзі
}
Queue;

//=============================================================
// Операція:   ініціалізація черги
// Передумова: pq вказує на чергу
// Постумова:  черга ініціалізована пустим вмістом
//=============================================================
void InitializeQueue(Queue *pq);

//=============================================================
// Операція:   перевірка, чи заповнена черга
// Передумова: pq вказує на чергу, що була ініціалізована раніше
// Постумова:  повертає True, якщо черга заповнена,
//             і False у протилежному випадку
//=============================================================
bool QueueIsFull(const Queue *pq);

//=============================================================
// Операція:   перевірка, чи є черга пустою
// Передумова: pq вказує на чергу, що була ініціалізована раніше
// Постумова:  повертає True, якщо черга пуста,
//             і False у протилежному випадку
//=============================================================
bool QueueIsEmpty(const Queue *pq);

//=============================================================
// Операція:   визначення кількості елементів у черзі
// Передумова: pq вказує на чергу, що була ініціалізована раніше
// Постумова:  повертає кількість елементів у черзі
//=============================================================
int QueueItemCount(const Queue *pq);

//=============================================================
// Операція:   додавання елемента наприкінці черги
// Передумова: pq вказує на чергу, що була ініціалізована раніше
//             елемент повинен бути поміщений в кінець черги
// Постумова:  якщо черга не є пустою, елемент поміщається
//             наприкінці черги і функція повертає True;
//             у протилежному випадку черга залишається незмінною,
```

```
//             а функція повертає False
//=============================================================
bool EnQueue(Item item, Queue *pq);

//=============================================================
// Операція:   видалення елемента на початку черги
// Передумова: pq вказує на чергу, що була ініціалізована раніше
// Постумова:  якщо черга не є пустою, елемент на початку черги
//             копіюється в *pitem і видаляється з черги,
//             сама функція повертає True;
//             якщо операція спустошує чергу, то черга
//             перевстановлюється в пустий стан.
//             Якщо черга є пустою з самого початку, вона
//             залишається незмінною, сама функція повертає False
//=============================================================
bool DeQueue(Item *pitem, Queue *pq);

//=============================================================
// Операція:   спустошення черги
// Передумова: pq вказує на чергу, що була ініціалізована раніше
// Постумова:  черга стає пустою
//=============================================================
void EmptyTheQueue(Queue *pq);

//=============================================================
// Операція:   вивід вмісту черги на екран
// Передумова: pq вказує на чергу, що була ініціалізована раніше
// Постумова:  черга виведена на екран
//=============================================================
void PrintQueue(Queue *pq);
#endif
```

В даному тексті відсутнє конкретне визначення типу **Item**. Під час застосування інтерфейсу в нього буде поміщено визначення, яке відповідає потребам конкретної програми.

3.3.5. Реалізація функцій інтерфейсу

Тепер можна приступити до написання коду інтерфейсу. Ініціалізація черги «пустим вмістом» означає встановлення вказівників на початок і кінець черги в **NULL**, а лічильника (члена **item**) – в **0**:

```
void InitializeQueue(Queue *pq)
{
pq->front = pq->rear = NULL;
pq->items = 0; }
```

За допомогою члена `item` дуже легко перевірити, чи є черга повною або пустою, і повернути кількість елементів у черзі:

```
bool QueueIsFull(const Queue *pq)
{
    return pq->items = MAXQUEUE;
}
 bool QueueIsEmpty(const Queue
*pq)
{
    return pq->items = 0;
}
 int QueueItemCount(const Queue
*pq)
{
    return pq->items;
}
```

Додавання елемента в чергу передбачає виконання наступних дій:

1) Створення нового вузла.

2) Копіювання елемента в цей вузол.

3) Встановлення вказівника `next` цього вузла в `NULL`, що ідентифікує його як останній у черзі.

4) Встановлення вказівника `next` поточного кінцевого вузла таким чином, щоб він посилався на новий вузол, зв'язуючи його з чергою.

5) Встановлення вказівника `rear` для посилання на новий вузол з метою спрощення пошуку останнього вузла.

6) Збільшення на `1` лічильника елементів черги.

Крім того, функція повинна обробляти два особливих випадки.

По-перше, якщо черга є пустою, вказівник `front` повинен бути встановлений для посилання на новий вузол. Причина в тому, що при наявності тільки одного вузла цей вузол є одночасно і початковим, і кінцевим вузлом черги.

По-друге, якщо функції не вдається виділити пам'ять для вузла, вона повинна щось зробити. Оскільки ми припускаємо використання невеликих черг, така відмова буде виникати рідко, тому у випадку нестачі пам'яті функція буде просто припиняти виконання програми. Нижче наведено код функції `EnQueue()`:

```
bool EnQueue(Item item, Queue *pq)
{
    Node *pnew;
    if(QueueIsFull(pq))
return false;
    pnew = (Node *) malloc(sizeof(Node));
if(pnew == NULL)
    {
        fprintf(stderr, "Не вдається виділити пам'ять!\n");
exit(1);
    }
    CopyToNode(item, pnew);
pnew->next = NULL;
if(QueueIsEmpty(pq))
        pq->front = pnew;     // елемент поміщається на початок
черги     else
        pq->rear->next = pnew; // зв'язування з кінцем черги
pq->rear = pnew;          // запис місця розташування кінця черги
pq->items++;              // збільшення кількості елементів на 1
return true;
}
```

Функція `CopyToNode()` – це статична функція, яка виконує копіювання елемента у вузол:

```
static void CopyToNode(Item item, Node *pn)
{
    pn->item =
item; }
```

Видалення елемента на початку черги потребує виконання наступних дій:

1) Копіювання елемента до визначеної змінної.
2) Звільнення пам'яті, яка використовувалася вузлом, що видаляється.
3) Переустановлення вказівника на початок черги, щоб він посилався на наступний елемент у черзі.
4) Встановлення вказівників на початок і на кінець черги в `NULL`, якщо видалено останній елемент.
5) Зменшення на `1` лічильника елементів черги.

Усі ці дії реалізовані в наступному коді:

```
bool DeQueue(Item *pitem, Queue *pq)
{
    Node *pt;
```

```
if(QueueIsEmpty(pq))
return false;
CopyToItem(pq->front, pitem);
pt = pq->front;
pq->front = pq->front->next;
free(pt);
pq->items--;
if(pq->items == 0)
pq->rear = NULL;
return true;
}
```

Тут необхідно відзначити декілька важливих моментів.

По-перше, в коді не проводиться явне встановлення вказівника **front** в **NULL**, коли видаляється останній елемент. Причина полягає в тому, що вказівник **front** вже встановлено в значення вказівника **next** вузла, що видаляється. Якщо ж цей вузол є останнім у черзі, то значення його вказівника **next** дорівнює **NULL**, тому вказівник **front** отримує значення **NULL**.

По-друге, код використовує тимчасовий вказівник **pt** для відстеження місця знаходження видаленого вузла. Це пов'язано з тим, що офіційний вказівник першого вузла **pq->front** перевстановлюється таким чином, щоб він вказував на наступний вузол. Тому без застосування тимчасового вказівника програма втратила б можливість відстеження того, який блок пам'яті звільнювати.

Для спустошення черги можна використовувати функцію **DeQueue()**. Для цього достатньо викликати її в циклі до тих пір, поки черга не стане пустою:

```
void EmptyTheQueue(Queue *pq)
{
Item dummy;
while(!QueueIsEmpty(pq))
DeQueue(&dummy, pq);
}
```

Після визначення інтерфейсу **ADT** ви повинні застосувати одну з його функцій для підтримки типу даних. Наприклад, зверніть увагу, що функція **DeQueue()** покладається на функцію **EnQueue()** у виконанні роботи з коректного встановлення вказівників і з встановлення вказівника **next** вузла **rear** в **NULL**.

Коли у програмі, що використовує **ADT**, ви вирішите маніпулювати частинами черги напряму, це може призвести до порушення координації між функціями в пакеті інтерфейсу.

В наступному тексті програми представлені усі функції інтерфейсу, включаючи функцію **CopyToItem()**, що застосовується в **EnQueue()**.

```
//=====================================================================
// queue.c - файл реалізації черги (Queue)
//=====================================================================
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

// локальні функції
static void CopyToNode(Item item, Node *pn);
static void CopyToItem(Node *pn, Item *pi);

void InitializeQueue(Queue *pq)
{
pq->front = pq->rear = NULL;
pq->items = 0;
}

bool QueueIsFull(const Queue *pq)
{
    return pq->items == MAXQUEUE;
}
 bool QueueIsEmpty(const Queue *pq)
{
    return pq->items == 0;
}
int QueueItemCount(const Queue *pq)
{
    return pq->items;
}
 bool EnQueue(Item item, Queue *pq)
{
    Node *pnew;
    if(QueueIsFull(pq))
return false;
    pnew = (Node *) malloc(sizeof(Node));
if(pnew == NULL)
    {
fprintf(stderr, "Не вдається виділити пам'ять!\n");
exit(1);
```

```
    }
    CopyToNode(item, pnew);
pnew->next = NULL;
if(QueueIsEmpty(pq))
        pq->front = pnew; // елемент поміщається на початок черги
else
pq->rear->next = pnew; // зв'язування з кінцем черги
pq->rear = pnew;  // запис місця розташування кінця черги
pq->items++;      // збільшення на 1 кількості елементів в черзі
return true;
}
 bool DeQueue(Item *pitem, Queue *pq)
{
    Node *pt;
     if(QueueIsEmpty(pq))
return false;
    CopyToItem(pq->front, pitem);
pt = pq->front;
pq->front = pq->front->next;
free(pt);
pq->items--;
if(pq->items == 0)
pq->rear = NULL;
return true;
}
void PrintQueue(Queue *pq)
{
    Node *pt;
     pt = pq->front;
printf("Черга:");
    for(int i = 0; i < pq->items; i++)
    {
    printf("%5d", pt->item);
    pt = pt->next;
    }
printf("\n");
}
// спустошення черги
void EmptyTheQueue(Queue *pq)
{
    Item dummy;
    while(!QueueIsEmpty(pq))
    DeQueue(&dummy, pq);
}
// локальні функції
static void CopyToNode(Item item, Node *pn)
{
    pn->item = item;
}
```

```
static void CopyToItem(Node *pn, Item *pi)
{
    *pi = pn->item;
}
```

3.3.6. Тестування черги

Перш ніж включати нову структуру, таку як пакет черги, до важливої програми, цю структуру необхідно протестувати. Один з підходів до тестування передбачає створення короткої програми, єдиним призначенням якої є тестування пакету. Наприклад, в коді, що наведений в наступному тексті програми, черга використовується для додавання та видалення цілих чисел. Перш ніж компілювати програму, переконайтеся в наявності наступного рядка у файлі **queue.h**:

```
typedef int item;
```

Крім того, не забудьте про необхідність виконання компонування цього файлу з **queue.c** і **main.c**.

```
//=====================================================================
// main.c – тестування інтерфейсу черги
//=====================================================================
#include <stdio.h>
#include <windows.h>
#include "queue.h"

// визначення Queue, Item
int main(void)
{
    Queue line;
    Item temp;
char ch;
     SetConsoleOutputCP(1251);
    InitializeQueue(&line);
puts("============================================");
puts("Тестування інтерфейсу черги (Queue)          ");
puts("============================================");
puts("Введіть символ:");
puts("    1 - додати значення до черги");
puts("    2 - видалити значення з черги");
puts("    3 - вихід з програми");
```

```
puts("=============================================");
    while((ch = getchar()) != '3')
    {
        if(ch != '1' && ch != '2')  // ігнорувати інші дані
continue;
if( ch == '1')
        {
           printf("Введіть ціле число для додавання: ");
           scanf("%d", &temp);
  if(!QueueIsFull(&line))
            {
                printf("Поміщаємо %d до черги\n", temp);
                EnQueue(temp, &line);
                PrintQueue(&line);
            }
else
                puts("Черга заповнена!");
        }
else
        {
            if(QueueIsEmpty(&line))
                puts("Елементи для видалення відсутні!");
else
            {
                DeQueue(&temp, &line);
                printf("Видалення %d з черги\n", temp);
                PrintQueue(&line);
            }
        }
        printf("Кількість елементів в черзі: %d\n",
QueueItemCount(&line));
puts("=============================================");
puts("Введіть символ:");
puts("    1 - додати значення до черги");
puts("    2 - видалити значення з черги");
puts("    3 - вихід з програми");
puts("=============================================");
    }
EmptyTheQueue(&line);
puts("Програма завершена.");
return 0;
}
```

Вікно проектів **Code::Blocks** після процесу компіляції повинно мати приблизно такий вигляд (рис. 3.11). Зверніть увагу на склад проекту.

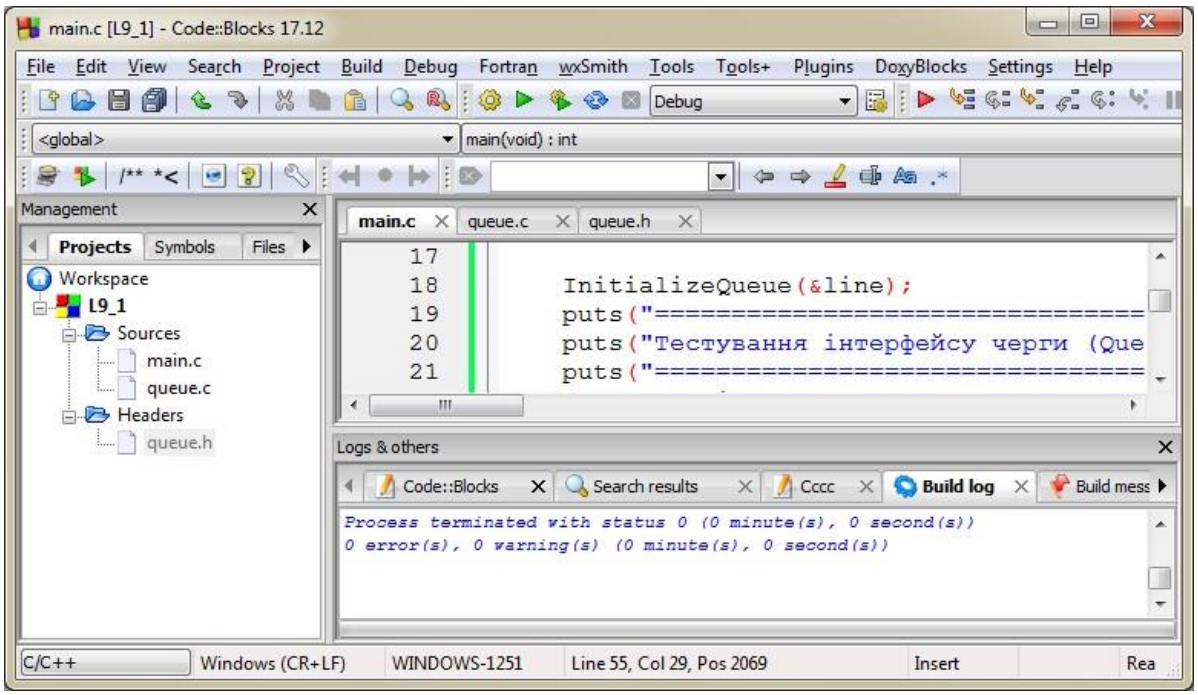


Рисунок 3.11 – Зовнішній вигляд вікна **`Code::Blocks`** після компіляції

Нижче показані результати пробного запуску програми (рис. 3.12). Ви повинні також протестувати коректність роботи програми у випадку, коли черга є повною.

3.3.7. Моделювання реальної черги

Черги зустрічаються в багатьох реальних ситуаціях. Це можуть бути, наприклад, черги клієнтів у банках та універсамах, черги літаків в аеропортах і черги задач у багатозадачних комп'ютерних системах. Пакет черги можна застосовувати для моделювання ситуацій подібного роду.

Припустимо, що дехто встановив консультаційний кіоск у торговому центрі для проведення консультацій. Клієнти можуть заплатити за одну, дві або три хвилини консультації. Правила для забезпечення вільного проходу, які діють у торговому центрі, обмежують кількість клієнтів у черзі до 10 (що легко визначає максимальний розмір черги в програмі). Уявимо собі, що люди підходять до кіоску випадковим чином, а час, який вони витрачають на отримання консультації, довільно здійснює розподіл між трьома можливими варіантами (одна, дві або три хвилини). Скільки в середньому клієнтів доведеться обслужити власнику кіоску впродовж години? Скільки в середньому

кожному клієнту доведеться очікувати свою чергу? Якою буде середня довжина черги? Моделювання може дати відповіді на питання такого роду.

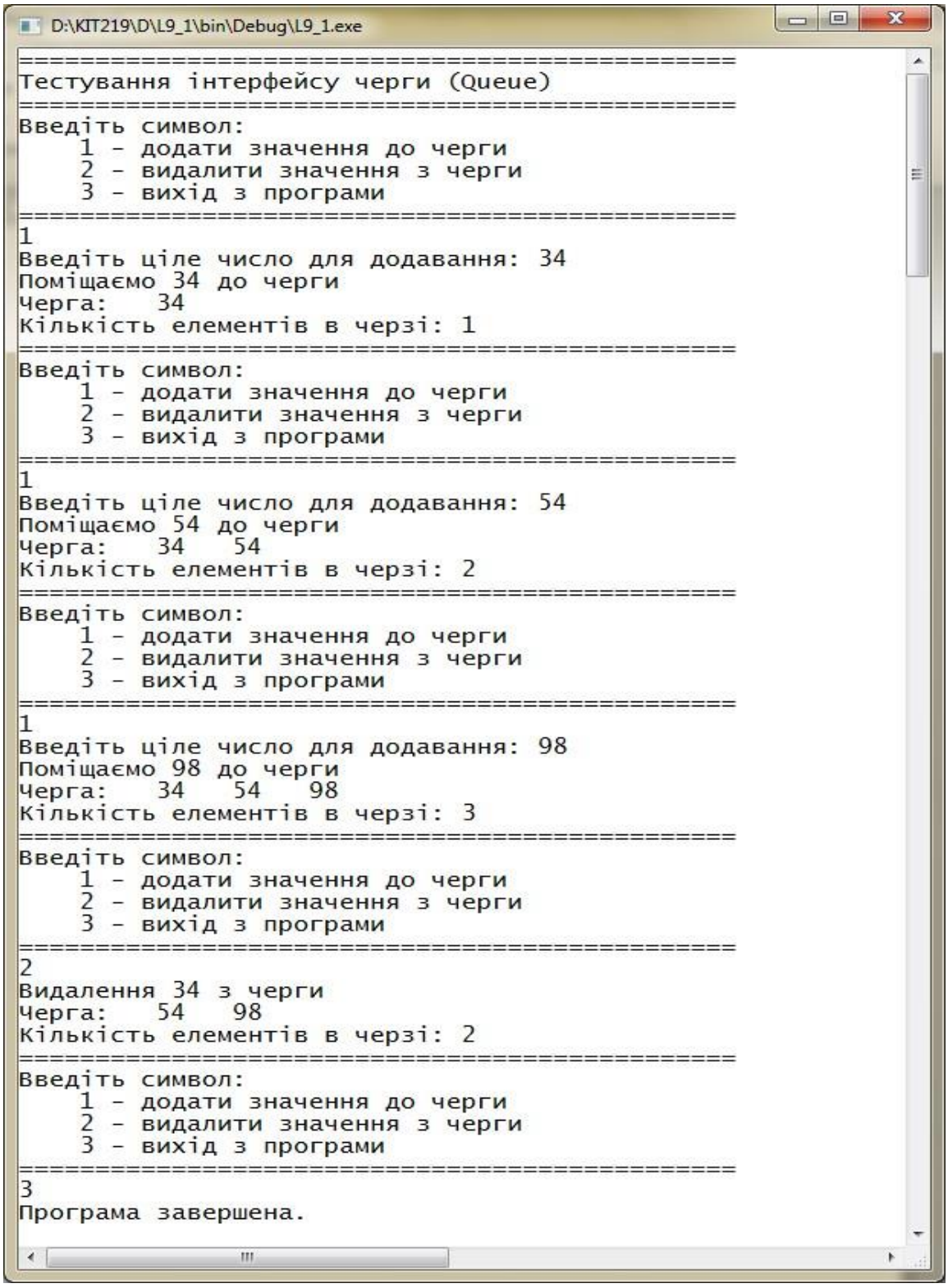


Рисунок 3.12 – Результат виконання програми для демонстрації черги

Перш за все, давайте вирішимо, що саме поміщати в чергу. Кожного клієнта можна описувати в термінах часу, коли він стає в чергу, і кількість хвилин, які він намагається витратити на консультацію. Це передбачає наступне визначення елемента **Item**:

```
typedef struct item
{
long arrive;     // час приєднання клієнта до черги
int  processtime;   // кількість хвилин консультації
}
Item;
```

Для перетворення пакету черги таким чином, щоб він обробляв цю структуру, а не тип int, який використовувався в останньому прикладі, достатньо замінити попереднє визначення **typedef** типу **Item**, що наведені вище. Після цього вам не доведеться турбуватися про деталі функціонування черги. Замість цього ви зможете зосередити всю увагу на реальній задачі – моделюванні черги до консультаційного кіоску.

Розглянемо один з можливих підходів. Нехай відлік часу здійснюється інтервалами тривалістю одна хвилина. Тоді кожну хвилину необхідно перевіряти, чи не з'явився новий клієнт. Якщо клієнт підійшов, і черга не переповнена, клієнта необхідно додати до черги. Це передбачає запис до структури **Item** часу прибуття клієнта та тривалості консультації, яку клієнт бажає оплатити, з подальшим додаванням елемента до черги. Однак якщо черга є повною, клієнту треба «відмовити». З метою обліку ми будемо відстежувати загальну кількість клієнтів і загальну кількість «відмов» (людей, які не можуть стати в чергу, оскільки вона переповнена).

Далі потрібно обробити початок черги. Тобто, якщо черга не є пустою і консультант не зайнятий обслуговуванням попереднього клієнта, необхідно видалити елемент з початку черги. Згадайте, що елемент містить значення часу приєднання клієнта до черги. Порівнюючи це значення з поточним часом, ми отримуємо час надходження клієнта в черзі (у хвилинах). Елемент містить також кількість хвилин, впродовж яких клієнт бажає отримати консультацію. Це значення визначає інтервал, впродовж якого консультант буде зайнятий

обслуговуванням нового клієнта. Для відстеження часу очікування ми застосовуємо змінну. Якщо консультант зайнятий, з черги ніхто не видаляється, але значення змінної для відстеження часу очікування повинно декрементуватися (зменшуватися на 1).

Основний код може виглядати схожим на показаний далі, при цьому кожен цикл відповідає одній хвилині активності:

```
for(cycle = 0; cycle < cyclelimit; cycle++)
{
    if(newcustomer(min_per_cust))
    {
if(QueueIsFull(&line)) turnaways++;
else
        {
            customers++;
            temp = customertime(cycle);
            EnQueue(temp, &line);
        }
    }
    if(wait_time <= 0 && !QueuelsEmpty(&line))
    {
        DeQueue(&temp, &line);
        wait_time = temp.processtime;
line_wait += cycle - temp.arrive;
served++;
    }
    if(wait_time > 0) wait_time—;
    sum_line +=
QueueltemCount(&line); }
```

Зверніть увагу, що інтервал часу є відносно грубим (одна хвилина), оскільки максимальна кількість клієнтів на годину складає усього 60.

Нижче представлені короткий опис деяких змінних і функцій:

- `min_per_cust` – середня кількість хвилин між прибуттям клієнтів;
- `newcustomer()` – використовує функцію `rand()` мови C для визначення, чи з'являється клієнт протягом цієї конкретної хвилини;
- `turnaways` – кількість клієнтів, які прибули, але їм було відмовлено в обслуговуванні;
- `customers` – кількість клієнтів, які прибувають і стають у чергу;
- `temp` – змінна типу `Item`, що описує нового клієнта;

- **customertime()** – встановлює члени **arrive** і **processtime** структури **temp**;

- **wait_time** – кількість хвилин, що залишилися до того моменту, коли Зигмунд завершить консультування поточного клієнта;

- **line_wait** – значення часу, що витрачається в черзі всіма клієнтами на поточний момент (значення накопичується);

- **served** – кількість клієнтів, які дійсно були обслуговані;

- **sumline** – значення довжини черги на поточний момент (значення накопичується).

Тільки подумайте, наскільки більш заплутаним і незрозумілим виглядав би код, якщо б він містив багато викликів функцій **malloc()**, **free()** і вказівників на вузли. Наявність пакету черги дозволяє зосередитися на задачі моделювання, не відволікаючись на деталі програмування.

Повний код для моделювання консультаційного кіоску в торговому центрі наведений в тексті програми, розміщеному в файлі **main.c**. Для генерації випадкових значень застосовуються стандартні функції **rand()**, **srand()** і **time()**. Щоб можна було використовувати програму, необхідно обновити визначення типу **Item** у файлі **queue.h** наступним чином:

```
typedef struct item
{
long arrive;      // час приєднання клієнта до черги
int  processtime;   // кількість хвилин консультації
}
Item;
```

Не забудьте також виконати компонування файлів **main.c**, **queue.c** і **queue.h**, а у файлі **queue.c** видаліть функцію **PrintQueue()**, оскільки вона не використовується в даному проекті.

```
//=============================================================
// main.c – тестування інтерфейсу консультаційного кіоску
//=============================================================
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>            // для rand() і srand()
#include <time.h>              // для time()
```

```
#include "queue.h"              // змініть визначення типу Item
#define MIN_PER_HR 60.0
bool newcustomer(double x);     // чи є новий клієнт?
Item customertime(long when);   // встановлення параметрів клієнта
int main(void)
{
    Queue line;
    Item temp;                  // дані про нового клієнта
    int hours;                  // тривалість моделювання в годинах
    int perhour;                // середня кількість клієнтів,
                                // які надходять за годину
long cycle, cyclelimit;   // лічильник і граничне значення циклу
long turnaways = 0;       // кількість відмов через переповнення
                          // черги
   long customers = 0;       // кількість клієнтів, які приєдналися
                             // до черги
    long served = 0;          // кількість клієнтів, які були
                              // обслуговані за час моделювання
long sum_line = 0;        // довжина черги, що накопичується
int wait_time = 0;        // час до звільнення консультанта
double min_per_cust;      // середній час між прибуттям клієнтів
long line_wait =0;        // час у черзі, що накопичується


    SetConsoleOutputCP(1251);     InitializeQueue(&line);
srand((unsigned int)time(0));

printf("====================================================\n");
printf("Консультаційний кіоск                                \n");
printf("====================================================\n");
printf("Введіть тривалість моделювання в годинах:        ");
    scanf("%d", &hours);     cyclelimit = MIN_PER_HR * hours;
    printf("Введіть середню кількість клієнтів, "            "які
надходять за годину: ");     scanf("%d", &perhour);
    min_per_cust = MIN_PER_HR / perhour;

printf("====================================================\n");
for(cycle = 0; cycle < cyclelimit; cycle++)
    {
        if(newcustomer(min_per_cust))
        {
             if(QueueIsFull(&line))                    turnaways++;
else              {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
             }          }
        if(wait_time <= 0 && !QueueIsEmpty(&line))
        {
            DeQueue(&temp, &line);
```

```
wait_time = temp.processtime;
line_wait += cycle - temp.arrive;
served++;
        }
        if(wait_time > 0)       wait_time--;
        sum_line += QueueItemCount(&line);
    }
    if(customers > 0)
    {
printf("Кількість прийнятих клієнтів:  %5ld\n", customers);
printf("Кількість клієнтів, які обслуговані: %5ld\n",served);
printf("Кількість відмов:   %5ld\n",  turnaways);
printf("Середня довжина черги:
%.2f\n",(double)sum_line/cyclelimit);
printf("Середній час очікування: %.2f хв\n",
(double)line_wait / served);
    }     else
puts("Клієнти відсутні!");
EmptyTheQueue(&line);

printf("================================================\n");
return 0;
}
// x – середній час між прибуттям клієнтів у хвилинах
// повертає true, якщо клієнт з'являється впродовж даної хвилини
bool newcustomer(double x)
{
    if(rand() * x / RAND_MAX < 1)          return true;      else
return false;
}


// when - час прибуття клієнта
// функція повертає структуру Item з часом прибуття,
// який встановлений в when, і часом обслуговування,
// що встановлений у випадкове значення з діапазону від 1 до 3

Item customertime(long when)
{
    Item cust;
    cust.processtime = rand() % 3 + 1;      cust.arrive = when;
return cust;
}
```

Результати виконання програми з різними параметрами **hours** і **perhour** представлені на рис. 3.13 – 3.16.

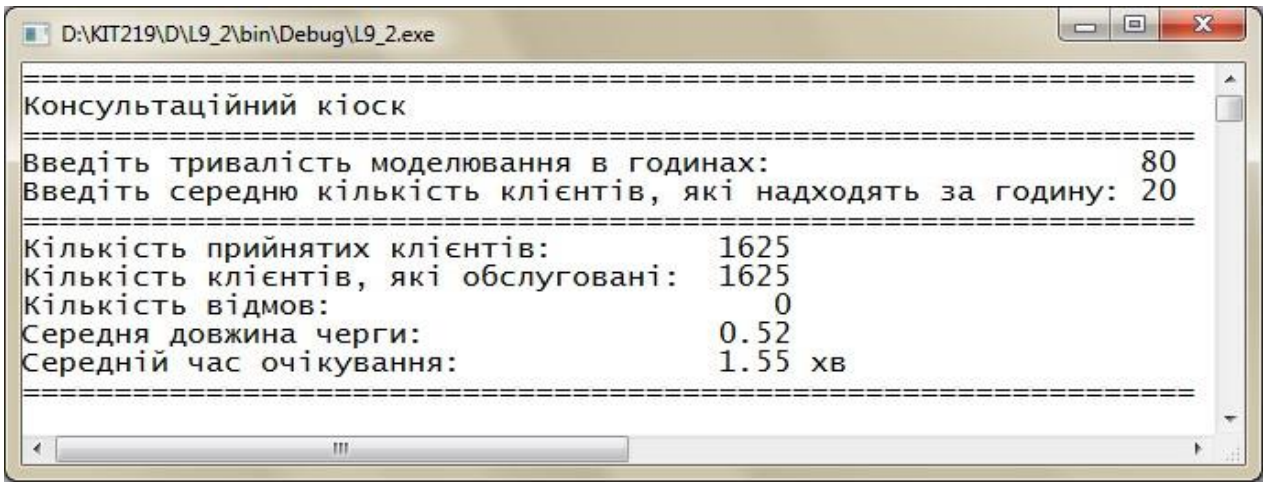


Рисунок 3.13 – Результат виконання програми для моделювання консультаційного кіоску в торговому центрі (`hours = 80`, `perhour = 20`)

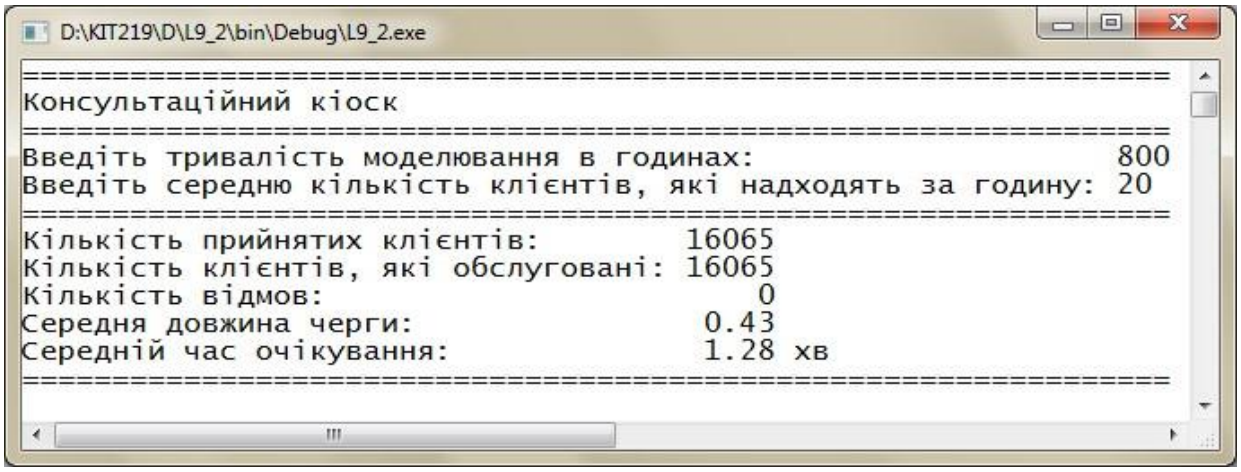


Рисунок 3.14 – Результат виконання програми для моделювання консультаційного кіоску в торговому центрі (`hours = 800`, `perhour = 20`)

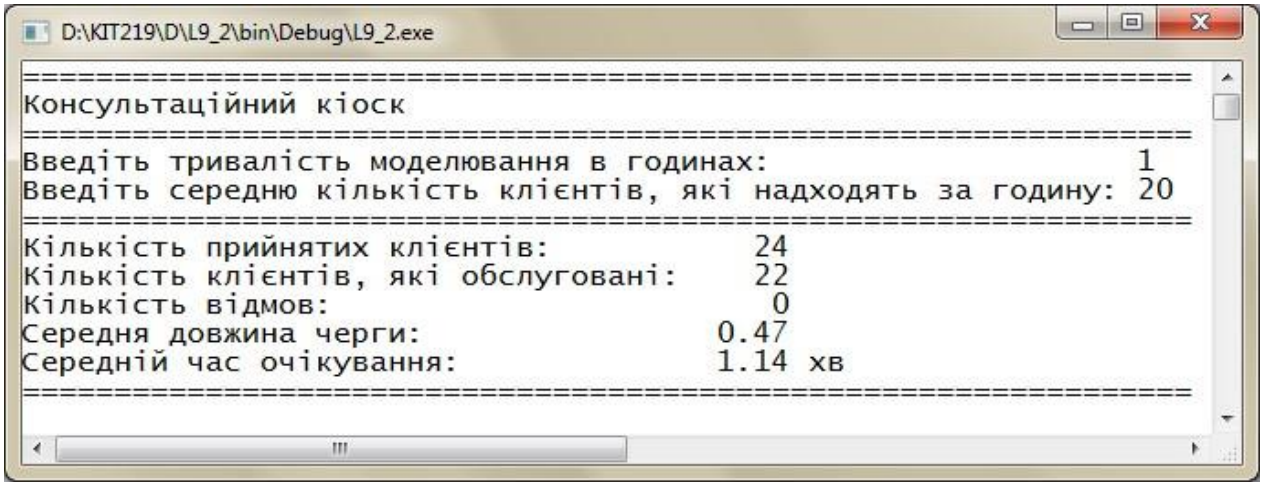


Рисунок 3.15 – Результат виконання програми для моделювання консультаційного кіоску в торговому центрі (`hours = 1`, `perhour = 20`)

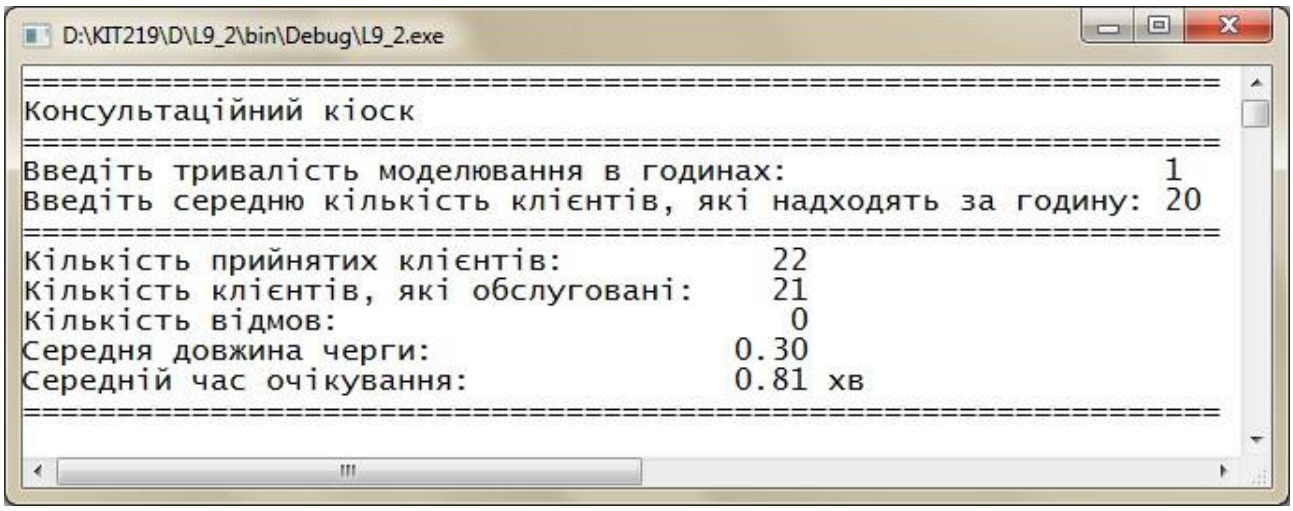


Рисунок 3.16 – Результат виконання програми для моделювання консультаційного кіоску в торговому центрі (`hours = 1, perhour = 20`)

Програма дозволяє вказувати кількість годин моделювання і середнє число клієнтів, що звертаються за консультацією протягом години. Вибір більшої кількості годин моделювання забезпечить отримання доволі точних середніх значень, тоді як мала кількість годин дає свого роду випадкову варіацію, яка може мати місце від години до години. Ці моменти демонструють показані нижче результати пробних запусків. Зверніть увагу, що середні значення довжини черги та час очікування для `80` годин і для `800` годин майже збігаються, але результати двох одногодинних вибірок суттєво відрізняються як один від одного, так і від середніх значень для більш тривалих періодів. Це обумовлено тим, що менші статистичні вибірки характеризуються більшими відносними варіаціями.

Ще один спосіб застосування цієї програми передбачає збереження тривалості моделювання незмінною, але зазначення різних середніх значень числа клієнтів, що прибули протягом години. Нижче приведені результати двох пробних запусків програми для дослідження такої варіації (рис. 3.17 і рис. 3.18).

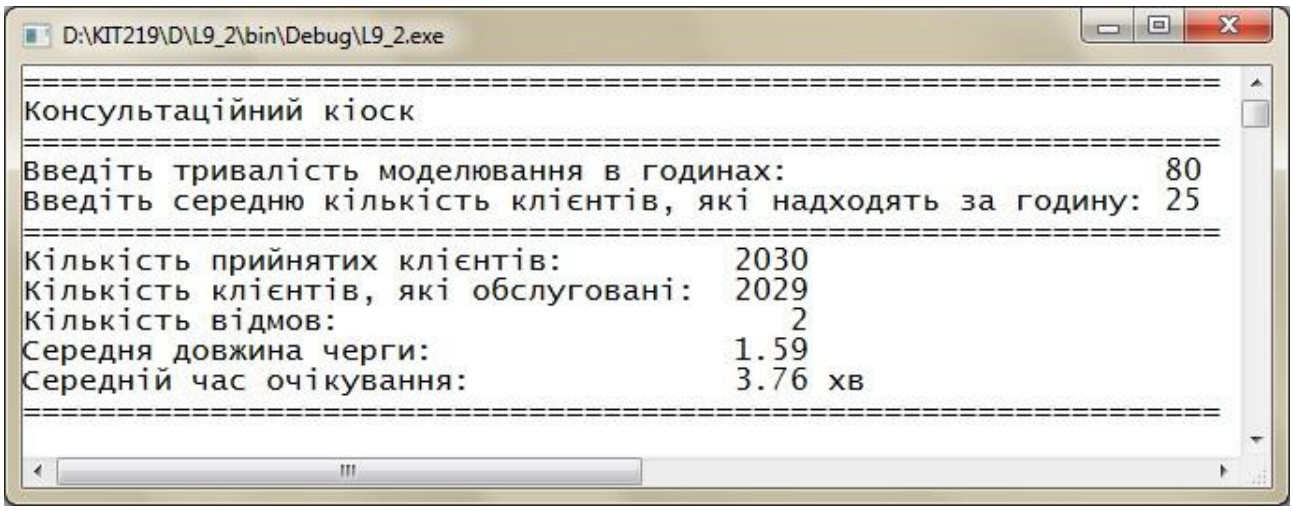


Рисунок 3.17 – Результат виконання програми для моделювання консультаційного кіоску в торговому центрі (`hours = 80`, `perhour = 25`)

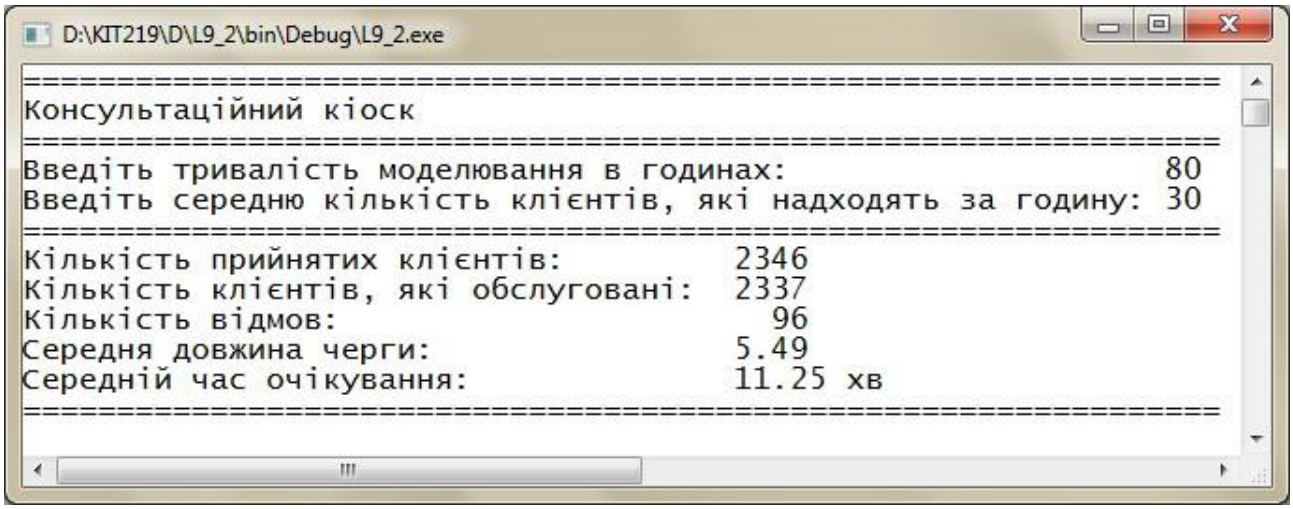


Рисунок 3.18 – Результат виконання програми для моделювання консультаційного кіоску в торговому центрі (`hours = 80`, `perhour = 30`)

Зверніть увагу на різке зростання середнього часу очікування зі збільшенням частоти прибуття клієнтів. Середній час очікування при `20` клієнтах за годину (`80`-годинне моделювання) склало `1,55` хвилини. Це значення збільшується до `3,76` хвилини при `25` клієнтах за годину і до `11,25` хвилини при `30` клієнтах на годину. Крім того, кількість відмов зростає від `0` до `2` і до `96` відповідно. Власник кіоску міг би скористатися подібним аналізом для прийняття рішення про необхідність відкриття другого кіоску.

3.4. Стек

Стек – це мабуть, найпростіша структура даних, яку ми будемо вивчати, і яку будемо час від часу застосовувати.

Стек – структура даних, в якій елементи підтримують принцип `LIFO` (`Last In - First Out`) – останнім зайшов – першим вийшов.

Стек дозволяє зберігати елементи різного типу. Зазвичай він підтримує дві базові операції:

- `push` – поміщає елемент до вершини стеку;
- `pop` – витягає елемент зі стеку, переміщуючи вершину до наступного елемента.

Також доволі часто зустрічається операція `peek`, яка отримує елемент, що знаходиться на вершині стеку, але не витягає його звідти.

Стек є однією з базових структур даних, що використовується у програмуванні, схемотехніці, на виробництві (для реалізації технологічних процесів) і т. д. Стек також використовується в якості допоміжної структури даних в багатьох алгоритмах і в інших більш складних структурах.

Нехай, наприклад, у нас є стек для зберігання цілих чисел. Спочатку стек пустий. Вершиною стеку є вказівник на перший елемент, який нікуди не вказує. У випадку мови `C` він може дорівнювати `NULL`.

Застосуємо до стеку декілька операцій (рис. 3.19).

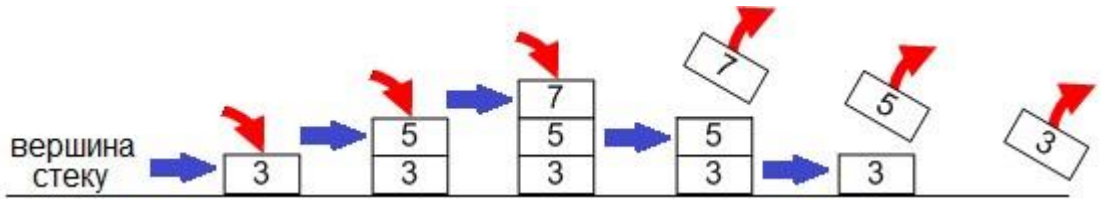


Рисунок 3.19 – Послідовне виконання операцій `push(3)`, `push(5)`, `push(7)`, а також `pop()`, `pop()`, `pop()`

Алгоритм виконання цих операцій має наступний вигляд:

1) **push(3)**

Стек містить один елемент – число **3**. Вершина стеку вказує на число **3**.

2) **push(5)**

Стек містить два елементи – **5** і **3**, при цьому вершина стеку вказує на число **5**.

3) **push(7)**

Стек містить три елементи, вершина стеку вказує на число **7**.

4) **pop()**

Поверне значення **7**. У стеку залишиться **5** і **3**. Вершина буде вказувати на наступний елемент – число **5**.

5) **pop()**

Поверне число **5**. У стеку залишиться лише один елемент – число **3**, на яке буде вказувати вершина стеку.

6) **pop()**

Поверне число **3**. Стек стане пустим.

Часто стек порівнюють з купкою тарілок (рис. 3.20). Щоб дістати з купки тарілку, необхідно зняти всі попередні. Вершина стеку – це вершина купки тарілок.

Рисунок 3.20 – Купка тарілок (спрощена модель стеку)

Коли ми будемо працювати зі стеком, можливі дві основні помилки, які дуже часто зустрічаються:

1) **STACK_UNDERFLOW** – намагання витягти елемент з пустого стеку;

2) **STACK_OVERFLOW** – намагання помістити новий елемент до стеку, який не може більше зростати (наприклад, не вистачає оперативної пам'яті).

Розглянемо три основні питання з реалізації стеку на мові **C**:

- формування стеку фіксованого розміру на основі масиву;

- формування динамічно зростаючого стеку на основі масиву;
- формування динамічно зростаючого стеку на основі зв’язного списку.

3.4.1. Формування стеку фіксованого розміру на основі масиву

Особливістю стеку фіксованого розміру з використанням масиву є простота реалізації та максимальна швидкодія. Такий стек може застосовуватися тоді, коли його максимальний розмір відомий заздалегідь або відомо, що він є невеликим.

Спочатку визначимо максимальний розмір масиву і тип даних, які будуть в ньому зберігатися:

```
#define STACK_MAX_SIZE 20
typedef int Type;
```

Сама структура буде мати наступний вигляд:

```
typedef struct stack
{
    Type data[STACK_MAX_SIZE];
    size_t size;
} Stack;
```

В цьому визначенні змінна `size` – це кількість елементів у стеку, і разом з цим вказівник на його вершину. Вершина буде вказувати на наступний елемент масиву, в який буде заноситися чергове значення.

Поміщати до стеку новий елемент будемо за допомогою функції `push()`:

```
void push(Stack *stack, const Type value)
{
stack->data[stack->size] = value;
stack->size++;

}
```

Єдина проблема полягає в тому, що можна вийти за межі масиву. Тому завжди треба перевіряти, щоб не було помилки `STACK_OVERFLOW`:

```
#define STACK_OVERFLOW   100
#define STACK_UNDERFLOW  101

void push(Stack *stack, const Type value)
{
    if(stack->size >= STACK_MAX_SIZE)
```

```
            exit(STACK_OVERFLOW);
        stack->data[stack->size] = value;
        stack->size++;
}
```

Аналогічним чином, визначимо функцію `pop()`, яка буде витягати елемент з вершини стеку і переходити до наступного:

```
Type pop(Stack *stack)
{
  if(stack->size == 0)
      exit(STACK_UNDERFLOW);
stack->size--;
    return stack->data[stack->size];
}
```

Функція `peek()` буде повертати поточний елемент, який знаходиться у вершині стеку, але не буде витягати його зі стеку:

```
Type peek(const Stack *stack)
{
    if(stack->size <= 0)
         exit(STACK_UNDERFLOW);
    return stack->data[stack->size-1];
}
```

Ще одне важливе зауваження полягає в тому, що у нас немає функції створення стеку, тому необхідно вручну обнулити значення `size`.

Допоміжні функції `printStackValue()` і `printStack()`, які будуть застосовуватися для виводу на екран елементів стеку, мають наступний вигляд:

```
void printStackValue(const Type
value)
{
    printf("%d", value);
}

void printStack(const Stack *stack,
                void (*printStackValue)(const Type))
{
int i;
int len = stack->size - 1;
printf("stack %d > ", stack->size);
for(i = 0; i < len; i++)
    {
```

```
    printStackValue(stack->data[i]);
    printf(" | ");
    }
    if (stack->size != 0)
        printStackValue(stack->data[i]);
    printf("\n");
    }
```

Треба зауважити, що у функції виводу стеку на екран ми використовуємо `int`, а не `size_t`, тому що значення `len` може стати від'ємним. Функція виводить спочатку розмір стеку, а потім його вміст, поділяючи елементи символом `|`.

Для перевірки роботи функцій і реалізації операцій, які були розглянуті на рис. 3.19, можна застосувати наступний код:

```
Stack stack;
stack.size = 0;

push(&stack, 3);
printStack(&stack, printStackValue);
push(&stack, 5);
printStack(&stack, printStackValue);
push(&stack, 7);
printStack(&stack, printStackValue);
printf("%d\n", pop(&stack));
printStack(&stack, printStackValue);
printf("%d\n", pop(&stack));
printStack(&stack, printStackValue);
printf("%d\n", pop(&stack));
printStack(&stack, printStackValue);
```

Розглянемо також ситуації, коли виникають помилки використання:

```
UNDERFLOW                // намагання читати з пустого стеку

int main(void)

{
Stack stack;
stack.size = 0;

push(&stack, 3);
pop(&stack);
pop(&stack);
return 0;
}
```

```
OVERFLOW                // переповнення стеку
int main(void)
{
   Stack stack;
   size_t i;
   stack.size = 0;

    for(i = 0; i < 100; i++)
    {
        push(&stack, i);
    }
return 0;
}
```

Розглянемо приклад використання стеку фіксованого розміру з використанням масиву. Текст програми має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>

#define STACK_OVERFLOW   100
#define STACK_UNDERFLOW  101
#define STACK_MAX_SIZE    20

typedef int Type;
typedef struct stack
{
     Type data[STACK_MAX_SIZE];
     size_t size;
} Stack;

void push(Stack *stack, const Type value);
void printStackValue(const Type value);
void printStack(const Stack *stack,
                void (*printStackValue)(const Type));
Type pop(Stack *stack);

int main(void)
{
int k;
Stack stack;
stack.size = 0;
```

```
    SetConsoleOutputCP(1251);

    srand(time(NULL));
for(int i = 0; i < rand() % 10 + 5; i++)
    {
        k = rand() % 50 + 10;
        printf("Додати до стеку число %d\n", k);
        push(&stack, k);
        printStack(&stack, printStackValue);
    }
    for(int i = 0; i < rand() % stack.size + 1; i++)
    {
        printf("Витягти число з вершини стеку\n");
        pop(&stack);
        printStack(&stack, printStackValue);
    }
return 0;
}
 void push(Stack *stack, const Type value)
{
    if(stack->size >= STACK_MAX_SIZE)
exit(STACK_OVERFLOW);
    stack->data[stack->size] = value;
    stack->size++;
}
 void printStackValue(const Type value)
{
    printf("%4d", value);
}

void printStack(const Stack *stack,
                void (*printStackValue)(const Type))
  {
  int i;
   int len = stack->size - 1;
printf("=========================================\n");
for(i = 0; i < len; i++)
printStackValue(stack->data[i]);
if(stack->size != 0)
        printStackValue(stack->data[i]);
printf("\n");
printf("=========================================\n");
printf("Розмірність стеку:                      %d\n",
       stack->size);
printf("=========================================\n");
}

Type pop(Stack *stack)
{
```

```
    if(stack->size == 0)
exit(STACK_UNDERFLOW);
stack->size--;
    return stack->data[stack->size]; }
```

Результат виконання програми наведено на рис. 3.21.

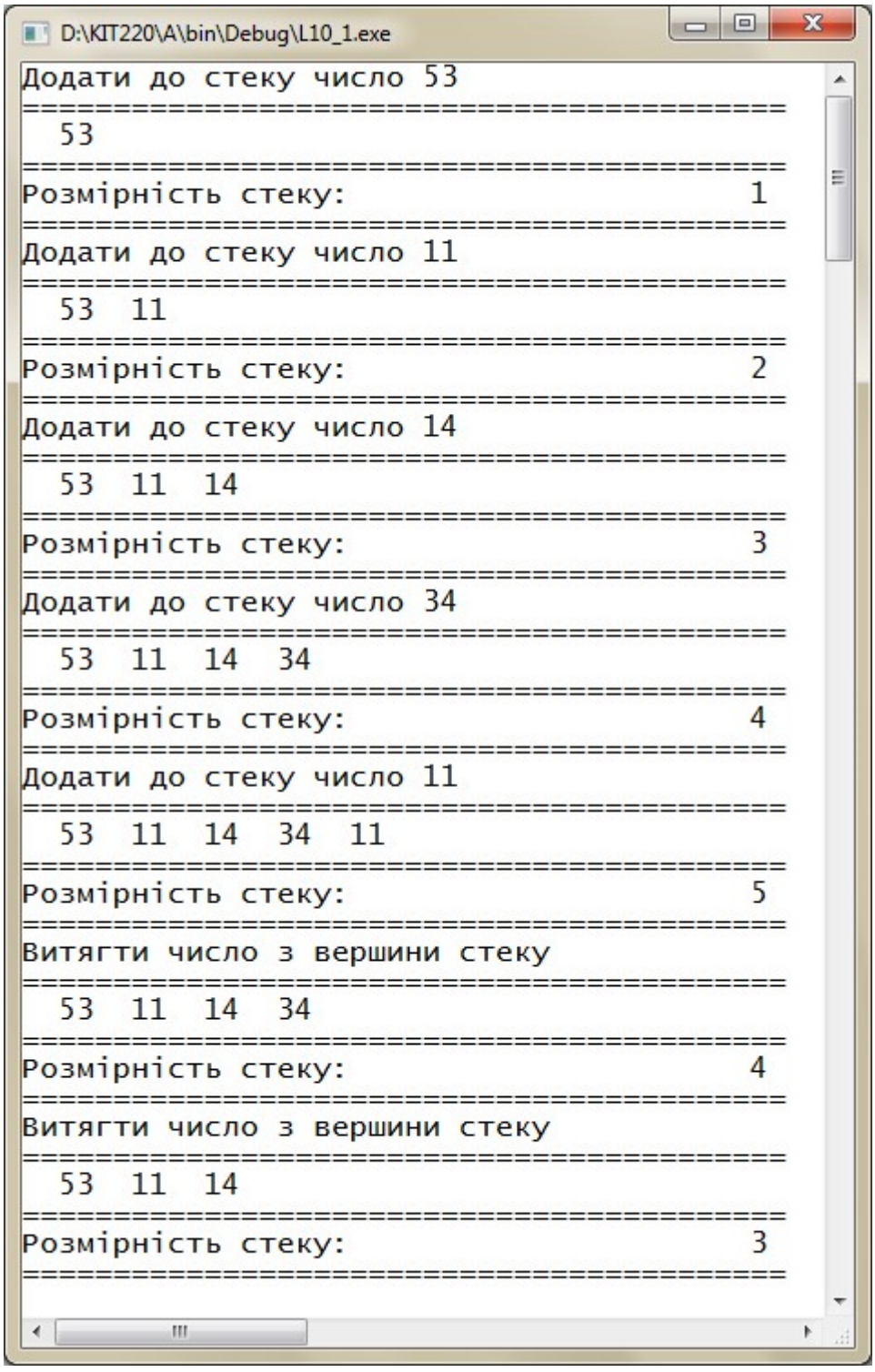


Рисунок 3.21 – Результат виконання програми для роботи зі стеком

3.4.2. Формування динамічно зростаючого стеку на основі масиву

Динамічно зростаючий стек використовується в тому випадку, коли число елементів може бути значним і воно не відоме на момент рішення задачі. Максимальній розмір стеку може бути обмежений будь-яким числом, або розміром оперативної пам'яті.

Стек буде містити вказівник на дані, розмір масиву (максимальний), і число елементів у масиві. Це саме число також буде вказувати на вершину стеку:

```
typedef struct stack
{
  Type *data;
  size_t size;
  size_t top;
 } Stack;
```

Спочатку знадобиться деякий початковий розмір масиву. Нехай він буде дорівнювати **10**.

```
#define INIT_SIZE 10
```

Алгоритм роботи буде таким: ми перевіряємо, чи не перевищило значення **top** величину **size**. Якщо це значення більше, то збільшуємо розмір масиву.

В даному випадку можливі декілька варіантів збільшення масиву. Можна додавати до **size** якесь число або помножувати його на певне значення. Який з варіантів кращий – залежить від специфіки задачі. В нашому випадку будемо помножувати розмір на іменовану константу **MULTIPLIER**:

```
#define MULTIPLIER 2
```

Максимального розміру задавати не будемо. Програма буде завершуватися при переповненні стеку (**STACK_OVERFLOW**) або намаганні витягти дані з пустого стеку (**STACK_UNDERFLOW**). Будемо реалізовувати ті ж самі функції, що й раніше: **push()**, **pop()**, **peek()**. Крім того, оскільки масив є динамічним, напишемо деякі допоміжні функції, щоб створювати або видаляти стек, змінювати його розмір або очищувати вміст стеку.

По-перше, розглянемо функції для створення стеку – `createStack()` і його видалення – `deleteStack()`. Текст цих функцій наведено нижче.

Спочатку створюємо стек з початковою довжиною і обнуляємо значення:

```
#define STACK_OVERFLOW  100
#define STACK_UNDERFLOW 101
#define OUT_OF_MEMORY   102

Stack *createStack()
{
    Stack *p = NULL;
    p = (Stack *) malloc(sizeof(Stack));
    if(p == NULL)
         exit(OUT_OF_MEMORY);
    p->size = INIT_SIZE;
    p->data = (Type *) malloc(p->size * sizeof(Type));
if(p->data == NULL)
    {
      free(p);
      exit(OUT_OF_MEMORY);
    }
    p->top = 0;
    return p;
}

void deleteStack(Stack **stack)
{
 free((*stack)->data);
 free(*stack);
 *stack = NULL;
 }
```

Тепер напишемо допоміжну функцію для зміни розміру:

```
void resize(Stack *stack)
{
stack->size *= MULTIPLIER;
stack->data = (Type *) realloc(stack->data,
                        stack->size * sizeof(Type));
if   (stack->data == NULL)
      exit(OUT_OF_MEMORY);
}
```

Якщо не вдалося виділити достатньо пам’яті, буде здійснено вихід з програми у зв’язку з нестачею пам’яті (`OUT_OF_MEMORY`).

Функція `push()` перевіряє, чи вийшли ми за межі масиву. Якщо так, то збільшуємо його розмір:

```
void push(Stack *stack, Type value)
{
    if(stack->top >= stack->size)
            resize(stack);
   stack->data[stack->top] = value;
   stack->top++;
}
```

Функції `pop()` і `peek()` виглядають аналогічно до тих, що використовувалися для масиву фіксованого розміру:

```
Type pop(Stack *stack)
{
    if(stack->top == 0)
       exit(STACK_UNDERFLOW);
    stack->top--;
    return stack->data[stack->top];
}

Type peek(const Stack *stack)
{
     if(stack->top <= 0)
            exit(STACK_UNDERFLOW);
return stack->data[stack->top-1];
}
```

Перевіримо роботу розглянутих функцій:

```
int main(void)
{
    int i;
    Stack *s = createStack();
   for(i = 0; i < 30; i++)
         push(s, i);
   for(i = 0; i < 30; i++)
    {
        printf("%d ", peek(s));
        printf("%d ", pop(s));
    }
deleteStack(&s);
return 0;
}
```

Напишемо ще одну функцію **impact()**, яка зменшує масив до розміру, що дорівнює числу елементів у масиві. Вона може бути використана тоді, коли вже відомо, що більше елементи вставлятися не будуть, і пам'ять може бути частково звільнена:

```
void impact(Stack *stack)
{
 stack->size = stack->top;
 stack->data = (Type *) realloc(stack->data,
                                stack->size * sizeof(Type));
}
```

Для перевірки роботи цієї функції ми можемо застосувати наступний код:

```
for(i = 0; i < 30; i++)
    push(s, i);
 impact(s);
for(i = 0; i < 30; i++)
{
    printf("%d ", peek(s));
    printf("%d ", pop(s));
}
```

Ця однопотокова реалізація стеку використовує мало звернень до пам'яті, є досить простою та універсальною, працює швидко і може бути реалізована, за необхідністю, за декілька хвилин. В подальшому вона буде використовуватися завжди, якщо не вказано інше.

У неї є певний недолік, який пов'язаний з методом збільшення пам'яті, що витрачається. Під час множення (в нашому випадку у **2** рази) треба мало звернень до пам'яті, але при цьому кожне наступне збільшення може призвести до помилки, особливо при невеликій кількості пам'яті в системі. Якщо ж використовувати більш «щадний» спосіб виділення пам'яті (наприклад, кожного разу додаючи по **10**), то число звернень збільшиться і швидкість спаде. На сьогодні, проблем з розміром пам'яті зазвичай немає, а менеджери пам'яті та збирачі сміття (яких немає в **C**) працюють швидко.

Розглянемо приклад використання динамічно зростаючого стеку на основі масиву. Текст програми в цьому випадку буде мати наступний вигляд:

```c
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#define INIT_SIZE         10
#define MULTIPLIER         2
#define STACK_OVERFLOW   100
#define STACK_UNDERFLOW  101
#define OUT_OF_MEMORY    102


typedef int Type;
typedef struct stack
{
    Type *data;
    size_t size;
    size_t top;
} Stack;


Stack *createStack();
void deleteStack(Stack **stack);
void resize(Stack *stack);
void push(Stack *stack, Type value);
Type pop(Stack *stack);
Type peek(const Stack *stack);
void impact(Stack *stack);


int main(void)
{
int i;
SetConsoleOutputCP(1251);
Stack *s = createStack();
printf("Заповнюємо стек цілими числами\n");
printf("=====================================\n");
for(i = 0; i < 25; i++)
    {
        push(s, i);
        printf("%3d", peek(s));
    }
printf("\n=====================================\n");
printf("УСІ ЧИСЛА ЗАНЕСЕНІ ДО СТЕКУ          \n");
printf("=====================================\n");
impact(s);
printf("ВМІСТ СТЕКУ\n");
printf("=====================================\n");
for(i = 1; i <= 25; i++)
    {
        printf("%3d", pop(s));
        if(i % 10 == 0) printf("\n");
    }
```

```
    if(i % 10 != 0)

printf("\n=====================================\n\n");
deleteStack(&s);
return 0;
}

Stack *createStack()
{
    Stack *p = NULL;
    p = (Stack *) malloc(sizeof(Stack));
if(p == NULL)
    exit(OUT_OF_MEMORY);
p->size = INIT_SIZE;
p->data = (Type *) malloc(p->size * sizeof(Type));
printf("=====================================\n");
printf("Спрацювала функція createStack()     \n");
printf("=====================================\n");
printf("Розмірність створеного стеку: %d\n", p->size);
printf("=====================================\n");
if(p->data == NULL)
    {          free(p);
exit(OUT_OF_MEMORY);
    }     p-
>top = 0;
return p;
}

void deleteStack(Stack **stack)
{
    free((*stack)->data);
    free(*stack);
    *stack = NULL;
}

void resize(Stack *stack)
{
printf("Спрацювала функція resize()\n");
stack->size *= MULTIPLIER;
stack->data = (Type *) realloc(stack->data,
                               stack->size * sizeof(Type));
printf("=====================================\n");
printf("Розмірність стеку збільшено: %d\n", stack->size);
printf("=====================================\n");
printf("Заповнюємо стек цілими числами\n");
printf("=====================================\n");
if(stack->data == NULL)
   exit(OUT_OF_MEMORY);
}
```

```
void push(Stack *stack, Type value)
{
static int kil = 0;
kil++;
    if(stack->top >= stack->size)
    {
printf("\n=====================================\n");
printf("Спрацювала функція push()\n");
printf("=====================================\n");
printf("Не вистачає пам'яті для елемента №%d\n", kil);
printf("Треба збільшити пам'ять\n");
printf("=====================================\n");
resize(stack);
    }
    stack->data[stack->top] = value;
    stack->top++;
}
Type pop(Stack *stack)
{
    if(stack->top == 0)
       exit(STACK_UNDERFLOW);
stack->top--;
return stack->data[stack->top];
}

Type peek(const Stack *stack)
{
    if(stack->top <= 0)
    exit(STACK_UNDERFLOW);
    return stack->data[stack->top-1];
}
 void impact(Stack *stack)
{
  stack->size = stack->top;
  stack->data = (Type *) realloc(stack->data,
                                 stack->size * sizeof(Type));
printf("Спрацювала функція impact()\n");
printf("=====================================\n");
printf("Зменшуємо розмір стеку\n");
printf("=====================================\n");
printf("Кількість елементів у стеку: %d\n", stack->size);
printf("=====================================\n"); }
```

Результат виконання програми наведено на рис. 3.22.

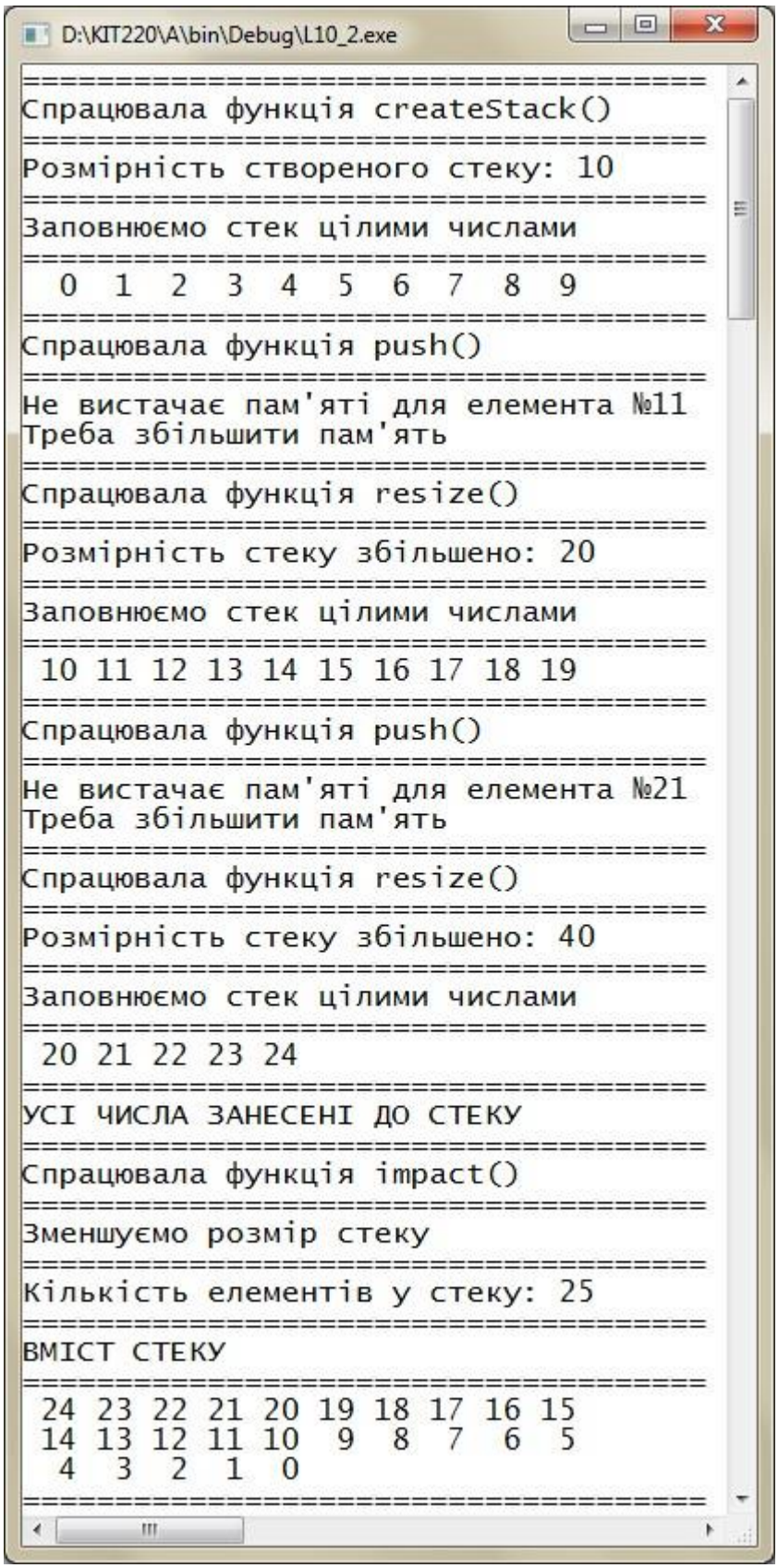


Рисунок 3.22 – Результат виконання програми для роботи з динамічно зростаючим стеком та використанням масиву

3.4.3. Формування динамічно зростаючого стеку на основі зв'язного списку

Зв'язний список складається з вузлів, кожен з яких містить корисну інформацію та посилання на наступний вузол. Останній вузол посилається на `NULL`. Жодного максимального і мінімального розмірів у нас не буде (хоча в загальному випадку може бути). Кожен новий елемент створюється заново.

Спочатку визначимо структуру для вузла:

```
#define STACK_OVERFLOW  100
#define STACK_UNDERFLOW 101
#define OUT_OF_MEMORY   102

typedef int Type;

typedef struct node
{
   Type value;
   struct node *next;
} Node;
```

Реалізація функції `push()` – додавання елемента до стеку – буде складатися з декількох кроків. Спочатку створимо новий вузол, потім вказівник `next` спрямуємо на старий вузол. Далі вказівник на вершину стека спрямуємо на щойно створений вузол. Таким чином вершина стеку буде вказувати на новий вузол.

```
void push(Node **head, Type value)
{
    Node *tmp = (Node *) malloc(sizeof(Node));
    if(tmp == NULL)
        exit(OUT_OF_MEMORY);
    tmp->next = *head;
    tmp->value = value;
    *head = tmp;
}
```

Сутність функції `pop()` – видалення елемента зі стеку – полягає в тому, що вона бере перший елемент (той, на який вказує вершина), спрямовує вказівник на наступний елемент і повертає перший. В нашому випадку існує два варіанти:

1) Повернути вказівник на вузол:

```
Node *popPointer(Node **head)
{
    Node *p;
    if(*head == NULL)
     exit(STACK_UNDERFLOW);
     p = *head;
     *head = (*head)->next;
    return p;            // повертає вказівник на вузол
}
```

2) Повернути значення:

Якщо повернути значення, то доведеться видаляти вузол всередині функції.

```
Type popValue(Node **head)
{
    Node *p;
    Type value;
     if(*head == NULL)
          exit(STACK_UNDERFLOW);
     p = *head;
        *head = (*head)->next;
        value = p->value;
    free(p);                // видалення вузла
    return value;           // повертає значення
}
```

Тепер замість перевірки довжини масиву, всюди буде використовуватися перевірка на рівність нулю вершини стеку (`if(*head == NULL)`).

Функція `peek()` буде мати наступний вигляд:

```
Type peek(const Node *head)
{
    if(head == NULL)
       exit(STACK_UNDERFLOW);
    return head->value;
 }
```

Вивід на екран вмісту стека для функції `printStack()` буде відбуватися в циклі. Просто переходимо від одного вузла до іншого, поки не дійдемо до кінця:

```
void printStack(const Node *head)
{
```

```
    printf("stack >");
while(head)
    {
        printf("%d ", head->value);
head = head->next;
    }
}
```

Оскільки стек реалізується на основі зв'язного списку, з'являється ще одна проблема. Вже неможна просто взяти і подивитися розмір стеку. Тепер треба пройти від початку до кінця стеку і підрахувати усі елементи. Наприклад, так:

```
size_t getSize(const Node *head)
{
    size_t size = 0;
    while(head)
    {
        size++;
        head = head->next;
    }
return size;
}
```

Звичайно, можна зберігати розмір окремо, можна обгорнути стек з усіма даними ще в одну структуру і т. д.

Проведемо тестування роботи розроблених функцій. Це можна зробити за допомогою наступного коду:

```
int main(void)
{
int i;
    Node *head = NULL;
    for(i = 0; i < 30; i++)
    push(&head, i);
    printf("size = %d\n", getSize(head));
while(head)
    {
printf("%d ", peek(head));
printf("%d ", popValue(&head));
    }
return 0;
}
```

Можна також застосувати другий варіант:

```
int main(void)
{
int i;
```

```
    Node *head = NULL;
    Node *tmp;
        for(i = 0; i < 30; i++)
      push(&head, i);
      printf("size = %d\n", getSize(head));
    while(head)
        {
    printf("%d ", peek(head));
    tmp = popPointer(&head);
    printf("%d ", tmp->value);
    free(tmp);
        }
    return 0;
    }
```

Розглянемо приклад використання динамічно зростаючого стеку на основі зв’язного списку.

Текст програми буде мати наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define STACK_OVERFLOW  100
#define STACK_UNDERFLOW 101
#define OUT_OF_MEMORY   102
 typedef int Type;
 typedef struct node
{
Type value;
struct node *next;
} Node;
void push(Node **head, Type value);
Node *popPointer(Node **head);
Type peek(const Node *head);
void printStack(const Node *head);
size_t getSize(const Node *head);
 int main(void)
{
    int i;
    Node *head = NULL;
    Node *tmp;
    SetConsoleOutputCP(1251);
     srand(time(NULL));
    printf("Заносимо елементи до стеку\n");

printf("================================================\n");
for(i = 0; i < 10; i++)
```

```
    {
        push(&head, rand() % 35 + 10);
        printf("Заносимо  %d   СТЕК: ", peek(head));
        printStack(head);
    }
printf("\n=================================================\n");
printf("Розмірність стеку: %d\n", getSize(head));

printf("=================================================\n");
printf("\nВитягаємо елементи зі стеку \n");

printf("=================================================\n");
    while(head)
    {
printf("Витягаємо %d   СТЕК: ",peek(head));
tmp = popPointer(&head);
printStack(head);
free(tmp);
    }
printf("=================================================\n");
printf("СТЕК ПУСТИЙ                                      \n");
printf("=================================================\n");
return 0;
}
void push(Node **head, Type value)
{
    Node *tmp = (Node *) malloc(sizeof(Node));
    if(tmp == NULL)
         exit(OUT_OF_MEMORY);
     tmp->next = *head;
     tmp->value = value;
     *head = tmp;
}
Node *popPointer(Node **head)
{
    Node *p;
if((*head) == NULL)
     exit(STACK_UNDERFLOW);
p = *head;
    *head = (*head)->next;
return p;
}
Type peek(const Node *head)
{
    if(head == NULL)
exit(STACK_UNDERFLOW);
return head->value;
}
 void printStack(const Node *head)
{
```

```
    while(head)
    {
        printf("%3d", head->value);
        head = head->next;
    }
printf("\n");
}
 size_t getSize(const Node *head)
{
     size_t size = 0;
     while(head)
    {
    size++;
    head = head->next;
    }
return size;
}
```

Результат виконання програми для роботи з динамічно зростаючим стеком на основі зв’язного списку наведено на рис. 3.23.

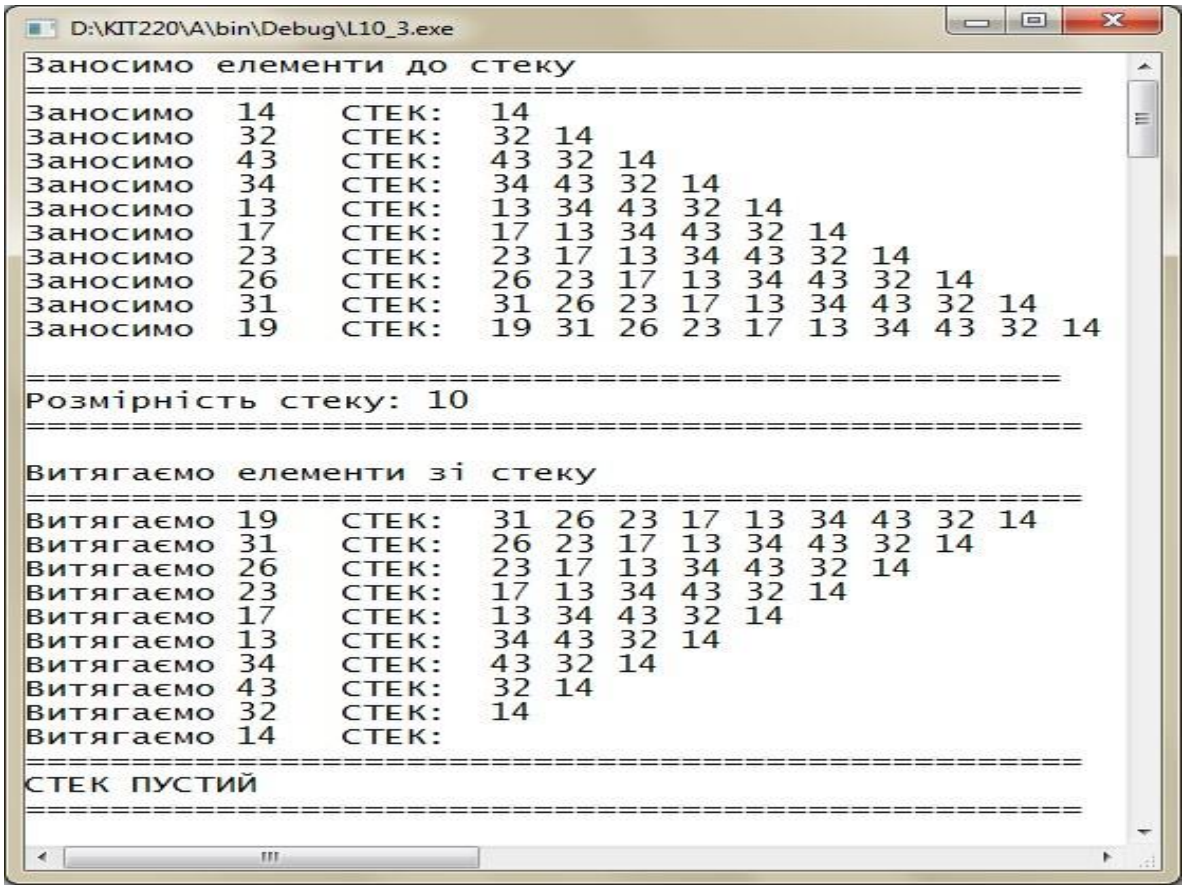


Рисунок 3.23 – Результат виконання програми для роботи з динамічно зростаючим стеком на основі зв’язного списку

3.5. Дек. Реалізація деку за допомогою списку

3.5.1. Дек (двостороння черга)

Дек (**`deque – double ended queue`**, «двостороння черга») – структура даних типу «список», яка функціонує одночасно за двома принципами організації даних: **`FIFO`** і **`LIFO`**. Визначити дек можна як чергу з двома сторонами, так і стек, що має два кінця. Тобто даний підвид списку має двосторонній доступ: виконання поелементної операції над деком передбачає можливість вибору однієї з його сторін в якості активної.

Раніше було з'ясовано, що число основних операцій, які виконуються над стеком і чергою, дорівнює трьом: додавання елемента, видалення елемента та читання елемента. При цьому не вказувалося місце структури даних, що було активним на момент їх виконання, оскільки раніше воно однозначно визначалося властивостями (визначенням) самої структури. Тепер, внаслідок використання деку як узагальненого випадку, для наведених операцій слід вказати цю область. Розділивши кожну з операцій на дві (одна буде стосуватися «голови» деку, друга буде сточуватися його «хвоста»), отримаємо набір з шести операцій:

- додавання елемента на початку деку;
- додавання елемента наприкінці деку;
- видалення першого елемента;
- видалення останнього елемента;
- читання першого елемента;
- читання останнього елемента.

На практиці цей список може бути доповнений перевіркою деку на порожність, отриманням його розміру та деякими іншими операціями.

В плані реалізації двостороння черга є дуже близькою до стеку та звичайної черги: в якості її базису можна використовувати як масив, так і список.

3.5.2. Реалізація деку за допомогою списку

Двозв'язний список – це структура даних, що складається з вузлів, які зберігають корисні дані, вказівники на попередній вузол і наступний вузол (рис. 3.24).

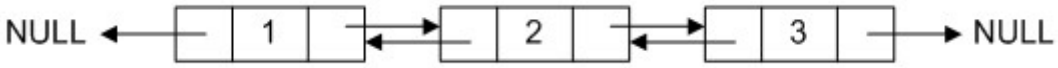


Рисунок 3.24 – Двозв'язний список

Для реалізації списку нам знадобиться структура `DNode`:

```
typedef struct DNode
{
void *value;
struct DNode *next;
struct DNode *prev;
} Node;
```

Вказівник **`prev`** зберігає адресу попереднього вузла. Якщо його немає (тобто, це перший вузол), то змінна дорівнює **`NULL`**. Вказівник **`next`** зберігає адресу наступного вузла. Змінна **`value`** має тип **`void *`**. Він використовується, коли тип змінної заздалегідь невідомий і може бути будь-яким: **`int`**, **`float`** і т. д.

Структура `DLList` буде зберігати свій розмір (щоб кожного разу не перераховувати кількість елементів), а також вказівник **`head`**, що посилається на перший елемент, і вказівник **`tail`**, що посилається на останній елемент.

```
typedef struct DList
{
   size_t size;
   Node *head;
   Node *tail;
} DLList;
```

У випадку, коли список не містить елементів, обидва вказівники дорівнюють нулю (рис. 3.25).

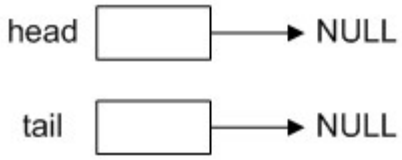


Рисунок 3.25 – Порожня структура

Якщо у списку один елемент, то обидва вказівника посилаються на один і той самий елемент (тобто, вони збігаються). Про це постійно слід пам'ятати: кожного разу, коли видаляється або додається елемент, доведеться перевіряти, щоб вказівники head і tail правильно зберігали адреси (рис. 3.26).

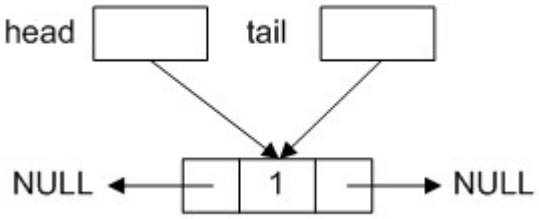


Рисунок 3.26 – Структура типу **DLList** з одним елементом

Перша функція **створення деку** `createDLList()`, створює екземпляр структури **DLList**:

```
DLList *createDLList()
{
    DLList *tmp = (DLList *) malloc(sizeof(DLList));
    tmp->size = 0;
    tmp->head = tmp->tail = NULL;
    return tmp;
}
```

Функція **видалення деку** `deleteDLList()` видаляє дек:

```
void deleteDLList(DLList **list)
{
    Node *tmp = (*list)->head;
    Node *next = NULL;
    while(tmp)
    {
```

```
            next = tmp->next;
            free(tmp);
            tmp = next;
        }
        free(*list);
        (*list) = NULL;
    }
```

До **стандартних функцій роботи з деком** належать: Тепер, визначимо набір стандартних функцій для роботи з деком:

- `pushFront()` і `popFront()` для роботи з головою списку;
- `pushBack()` і `popBack()` для роботи з останнім елементом списку;
- `insert()` і `getNth()` для додавання вузла в довільне місце списку та `deleteNth()` для видалення довільного елемента списку.

Додавання вузла на початку деку (рис. 3.27) дуже схоже на додавання до однозв'язного списку.

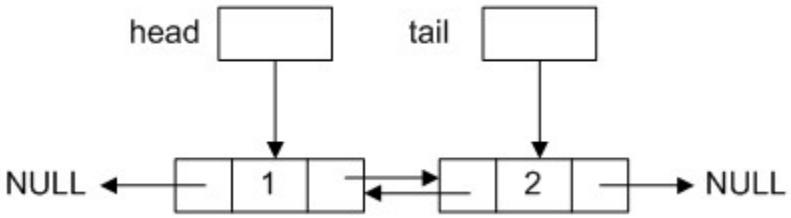


Рисунок 3.27 – Додавання нового елемента на початку деку

Спочатку створюється новий елемент (рис. 3.28),

```
Node *tmp = (Node *) malloc(sizeof(Node));
if(tmp == NULL)
exit(1);
```

а потім йому присвоюються відповідні значення (рис. 3.29).

```
tmp->value = data;
tmp->next = list->head;
tmp->prev = NULL;
```

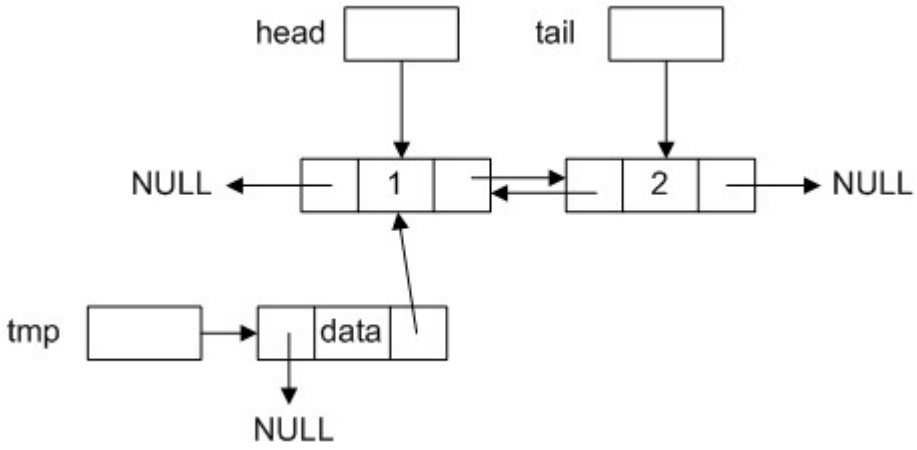


Рисунок 3.28– Створили новий елемент і задали йому значення

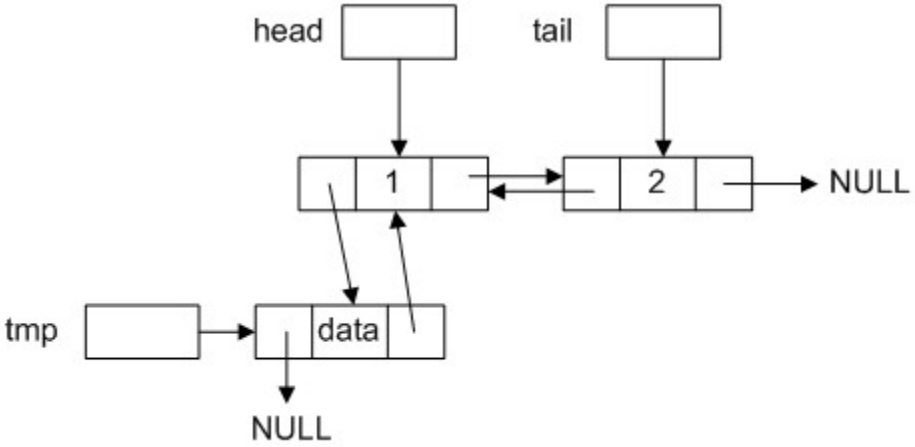


Рисунок 3.29 – Створили новий елемент і задали йому значення

Оскільки він став першим, то вказівник **`next`** посилається на стару голову списку, а попереднього елемента немає. Якщо у списку вже був головний елемент, то його вказівник **`prev`** повинен посилатися на знов створений елемент

```
if(list->head)
list->head->prev = tmp;
```

Тепер перевіримо вказівник **`tail`**. Якщо він порожній, то після додавання нового елемента він повинен посилатися на нього.

```
if(list->tail == NULL)
    list->tail = tmp;
```

Далі спрямовуємо вказівник **head** на щойно створений елемент (рис. 3.30) і збільшуємо значення лічильника **size**.

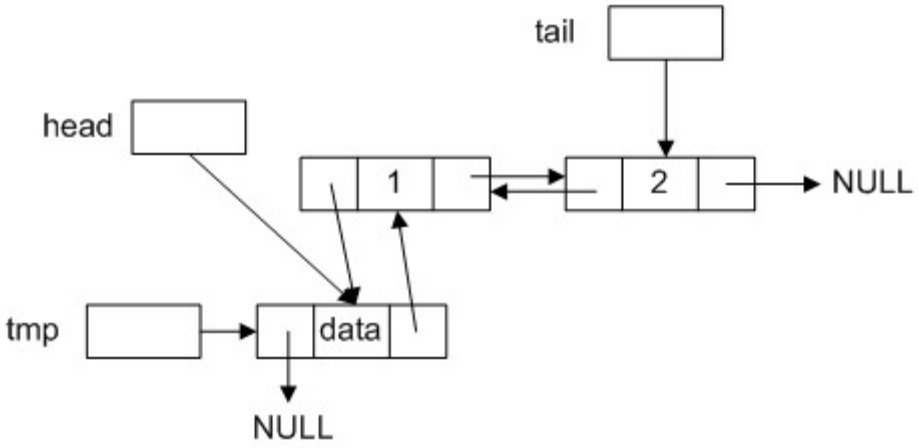


Рисунок 3.30 – Спрямовуємо вказівник **head** на щойно створений елемент

```
list->head = tmp;
list->size++;
```

Таким чином, повний текст функції **pushFront()** для додавання вузла на початку деку має наступний вигляд:

```
void pushFront(DLList *list, void *data)
{
    Node *tmp = (Node *) malloc(sizeof(Node));
    if(tmp == NULL)
        exit(1);
tmp->value = data;
tmp->next = list->head;
tmp->prev = NULL;
if(list->head)
    list->head->prev = tmp;
list->head = tmp;
if(list->tail == NULL)
         list->tail = tmp;
list->size++;
}
```

Видалення елемента на початку деку також схоже на видалення вузла для однозв’язного списку. Додаються лише посилання додаткових вказівників і перевірка умови, щоб вказівник на останній елемент (у випадку, якщо елементів більше не залишилось) став дорівнювати нулю.

Спочатку створимо вказівник на перший елемент деку (рис. 3.31). Він знадобиться для того, щоб після модифікації усіх вказівників **prev** і **next** ми змогли видалити вузол.

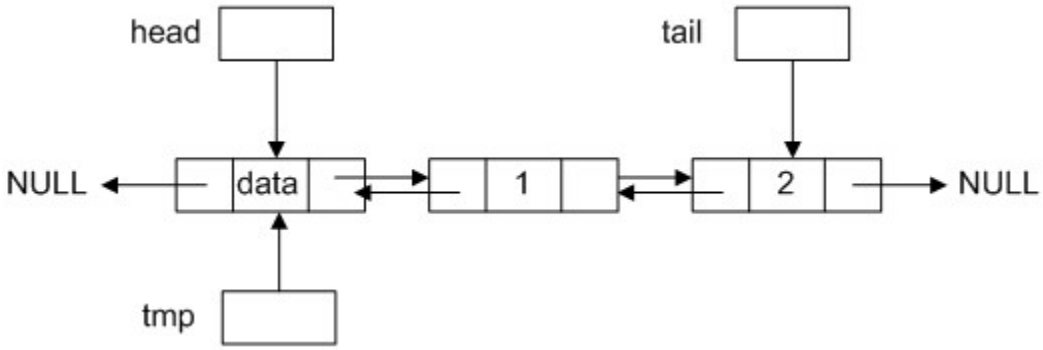


Рисунок 3.31 – Створили вказівник на перший елемент деку

```
Node *prev;
void *tmp; if(list->head == NULL)
     exit(2);
prev = list->head;
```

Після цього спрямовуємо вказівник head на елемент, що йде за ним (рис. 3.32).

```
list->head = list->head->next;
```

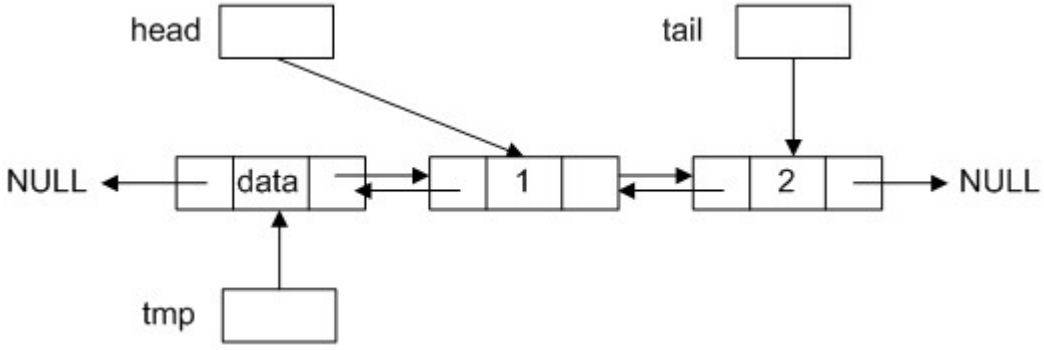


Рисунок 3.32 – Спрямовуємо вказівник **head** на наступний елемент

Далі перевіряємо, що елемент, який видаляється, не є останнім (тобто, коли в списку є лише один елемент), після чого звільняємо пам’ять (рис. 3.33).

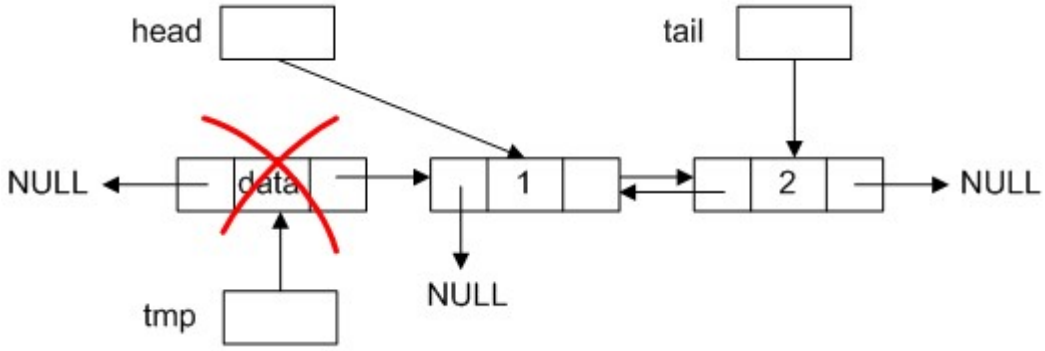


Рисунок 3.33 – Звільняємо пам'ять, яка була виділена під даний елемент

Повний код для функції **`popFront()`** для видалення вузла на початку деку має наступний вигляд:

```
void *popFront(DLList *list)
{
    Node *prev;
    void *tmp;
    if(list->head == NULL)
       exit(2);
    prev = list->head;
    list->head = list->head-
    >next;     if(list->head)
            list->head->prev = NULL;
    if(prev == list->tail)
    list->tail = NULL;
    tmp = prev->value;
    free(prev);
    list->size--;
    return tmp;
}
```

Додавання елементу наприкінці деку та видалення останнього елемента деку є дуже схожими – просто ми перевертаємо двосторонній список. Відповідно, усі **`prev`** змінюються на **`next`**, а **`head`** на **`tail`**.

Функція **`pushBack()`** додавання вузла наприкінці деку має наступний вигляд:

```
void pushBack(DLList *list, void *value)
```

```
{
    Node *tmp = (Node *) malloc(sizeof(Node));
    if(tmp == NULL)
         exit(3);
    tmp->value = value;
    tmp->next = NULL;
    tmp->prev = list->tail;
    if(list->tail)
             list->tail->next = tmp;
        list->tail = tmp;
    if(list->head == NULL)
             list->head = tmp;
    list->size++;
}
```

Функція **видалення вузла наприкінці деку** `popBack()` має наступний вигляд:

```
void *popBack(DLList *list)
{
    Node *next;
    void *tmp;
    if(list->tail == NULL)
           exit(4);
    next = list->tail;
    list->tail = list->tail->prev;
    if(list->tail)
            list->tail->next = NULL;
    if(next == list->head)
             list->head = NULL;
    tmp = next->value;
    free(next);
    list->size--;
    return tmp;
}
```

Функція **отримання n-го елемента деку** `getNth()` є дуже простою і не відрізняється від подібної функції для однозв'язного списку. Вона має наступний вигляд:

```
Node *getNth(DLList *list, size_t index)
{
    Node *tmp = list->head;
    size_t i = 0;
```

```
    while(tmp && i < index)
    {
       tmp = tmp->next;
       i++;
    }
return tmp;
}
```

Цю функцію можна покращити. Якщо список є досить довгим, то в залежності від індексу можна проходити або з початку до кінця деку, або з кінця до початку. Це дозволяє завжди використовувати не більш ніж **N/2** кроків.

```
Node *getNthq(DLList *list, size_t index)
{
    Node *tmp = NULL;
    size_t i;
    if(index < list->size/2)
    {
      i = 0;
      tmp = list->head;
        while(tmp && i < index)
        {
            tmp = tmp->next;
            i++;
        }
    }
else
    {
        i = list->size - 1;
        tmp = list->tail;
while(tmp && i > index)
        {
            tmp = tmp->prev;
            i--;
        }
    }
    return tmp;
}
```

Функції **видалення та додавання вузла в довільному місці деку.** Спочатку знаходимо потрібний елемент і потім створюємо або новий вузол (якщо це додавання), або вказівник на елемент, що видаляється. Потім змінюємо усі вказівники. Функція **insert()** додавання вузла в довільному місці деку має наступний вигляд:

```
void insert(DLList *list, size_t index, void *value)
{
    Node *elm = NULL;
    Node *ins = NULL;
    elm = getNth(list, index);
    if(elm == NULL)
          exit(5);
    ins = (Node *) malloc(sizeof(Node));
    ins->value = value;
    ins->prev = elm;
    ins->next = elm->next;
    if(elm->next)
       elm->next->prev = ins;
       elm->next = ins;
    if(!elm->prev)
        list->head = elm;
    if(!elm->next)
        list->tail = elm;
    list->size++;
  }
```

Функція **deleteNth()** видалення вузла в довільному місці деку має наступний вигляд:

```
void *deleteNth(DLList *list, size_t index)
{
    Node *elm = NULL;
    void *tmp = NULL;
    elm = getNth(list, index);
    if(elm == NULL)
     exit(5);
    if(elm->prev)
        elm->prev->next = elm->next;
    if(elm->next)
        elm->next->prev = elm->prev;
    tmp = elm->value;
    if(!elm->prev)
        list->head = elm->next;
    if(!elm->next)
        list->tail = elm->prev;
    free(elm);
    list->size--;
    return tmp;
}
```

Не забуваємо контролювати значення **head** і **tail**, щоб вони вказували на елементи, що існують на даний момент.

3.5.3. Допоміжні функції для роботи з деком

Додамо дві допоміжні функції `printDLList()` і `fromArray()`, які допоможуть в подальшій роботі.

Функція `printDLList()` **виводить елементи списку на екран**. Оскільки тип значення, що повертається, – `void *`, то необхідно передавати функцію `printDLList()` одного елемента.

```
void printDLList(DLList *list, void (*fun)(void *))
{
   Node *tmp = list->head;
   while(tmp)
    {
        fun(tmp->value);
        tmp = tmp->next;
    }
        printf("\n");
}
```

Далі в прикладах будемо використовувати змінні типу `int`. Для цього будемо застосовувати функцію `printInt()`:

```
void printInt(void *value)
{
    printf("%d ", *((int *)value));
}
```

Функція `fromArray()` призначена для **створення списку з масиву**:

```
DLList *fromArray(void *arr, size_t n, size_t size)
{
    DLList *tmp = NULL;
    size_t i = 0;
    if(arr == NULL)
       exit(7);
    tmp = createDLList();
    while(i < n)
    {
        pushBack(tmp, ((char *)arr + i * size));
        i++;
    }
    return tmp;
}
```

3.5.4. Робота з деком

Тепер можна користуватися двозв'язним списком, застосовуючи ті функції, які були розглянуті раніше. Для цього напишемо головну функцію, текст якої має наступний вигляд:

```
int main(void)
{
    DLList *list = createDLList();
int a, b, c, d, e, f, g, h;


    a = 10;
    b = 20;
    c = 30;
    d = 40;
    e = 50;
    f = 60;
    g = 70;
    h = 80;
pushFront(list, &a);
pushFront(list, &b);
pushFront(list, &c);
pushBack(list, &d);
pushBack(list, &e);
pushBack(list, &f);
printDLList(list, printInt);
printf("length %d\n", list->size);
printf("nth 2 %d\n", *((int *)(getNthq(list, 2))->value));
printf("nth 5 %d\n", *((int *)(getNthq(list, 5))->value));
printf("popFront %d\n", *((int *)popFront(list)));
printf("popFront %d\n", *((int *)popFront(list)));
printf("head %d\n", *((int *)(list->head->value)));
printf("tail %d\n", *((int *)(list->tail->value)));
printf("popBack %d\n", *((int *)popBack(list)));
printf("popBack %d\n", *((int *)popBack(list)));
printf("length %d\n", list->size);
printDLList(list, printInt);
insert(list, 1, &g);
printDLList(list, printInt);
deleteNth(list, 0);
printDLList(list, printInt);
deleteDLList(&list);
return 0;
}
```

Повний текст програми для роботи з деком має наступний вигляд:

```c
#include <stdio.h>
#include <windows.h>
typedef struct DNode
{
    void *value;
    struct DNode *next;
    struct DNode *prev;
} Node;

typedef struct DList
{
    size_t size;
    Node *head;
    Node *tail;
} DLList;

DLList *createDLList();
void    deleteDLList(DLList **list);
void    pushFront(DLList *list, void *data);
void   *popFront(DLList *list);
void    pushBack(DLList *list, void *value);
void   *popBack(DLList *list);
Node   *getNthq(DLList *list, size_t index);
void    insert(DLList *list, size_t index, void *value);
void   *deleteNth(DLList *list, size_t index);
void   printDLList(DLList *list, void (*fun)(void *));
void    printInt(void *value);
DLList *fromArray(void *arr, size_t n, size_t size);

int main(void)
{
    DLList *list = createDLList();
    int a, b, c, d, e, f, g;
    SetConsoleOutputCP(1251);
a = 10;
b = 20;
c = 30;
d = 40;
e = 50;
f = 60;
g = 70;
printf("================================================\n");
printf("Додаємо елемент %d на початку списку\n", a);
printf("================================================\n");
pushFront(list, &a);
printDLList(list, printInt);
```

```
printf("================================================\n");
printf("Додаємо елемент %d на початку списку\n", b);
printf("================================================\n");
pushFront(list, &b);
printDLList(list, printInt);
printf("================================================\n");
printf("Додаємо елемент %d на початку списку\n", c);
printf("================================================\n");
pushFront(list, &c);
printDLList(list, printInt);
printf("================================================\n");
printf("Додаємо елемент %d в кінець списку\n", d);
printf("================================================\n");
pushBack(list, &d);
printDLList(list, printInt);
printf("================================================\n");
printf("Додаємо елемент %d в кінець списку\n", e);
printf("================================================\n");
pushBack(list, &e);
printDLList(list, printInt);
printf("================================================\n");
printf("Додаємо елемент %d в кінець списку\n", f);
printf("================================================\n");
pushBack(list, &f);
printDLList(list, printInt);
printf("================================================\n");
printf("Довжина списку: %d\n", list->size);
printf("================================================\n");
printf("%d-й елемент списку: %d\n",
       2, *((int *)(getNthq(list, 2))->value));
printf("================================================\n");
printf("%d-й елемент списку: %d\n",
       5, *((int *)(getNthq(list, 5))->value));
printf("================================================\n");
printf("Витягаємо елемент %d на початку списку\n",
       *((int *)popFront(list)));
printf("================================================\n");
printDLList(list, printInt);
printf("================================================\n");
printf("Витягаємо елемент %d на початку списку\n",
       *((int *)popFront(list)));
printf("================================================\n");
printDLList(list, printInt);
printf("================================================\n");
printf("Голова списку: %d\n", *((int *)(list->head->value)));
printf("================================================\n");
printf("Хвіст списку:  %d\n", *((int *)(list->tail->value)));
printf("================================================\n");
printDLList(list, printInt);
printf("================================================\n");
printf("Витягаємо елемент %d з кінця списку\n",
       *((int *)popBack(list)));
```

```
printf("=============================================\n");
printDLList(list, printInt);
printf("=============================================\n");
printf("Витягаємо елемент %d з кінця списку\n",
       *((int *)popBack(list)));
printf("=============================================\n");
printDLList(list, printInt);
printf("=============================================\n");
printf("Довжина списку: %d\n", list->size);
printf("=============================================\n");
printf("=============================================\n");
printf("Додаємо елемент %d після %d-го елемента списку\n",
g, 0);
printf("=============================================\n");
insert(list, 0, &g);     printDLList(list, printInt);
printf("=============================================\n");
printf("Витягаємо %d-й елемент на початку списку\n", 0);
printf("=============================================\n");
deleteNth(list, 0);     printDLList(list, printInt);
printf("=============================================\n");
deleteDLList(&list);
return 0;
}
DLList *createDLList()
{
    DLList *tmp = (DLList *) malloc(sizeof(DLList));
    tmp->size = 0;
    tmp->head = tmp->tail = NULL;
    return tmp;
}
deleteDLList(DLList **list)
{
    Node *tmp = (*list)->head;
    Node *next = NULL;
    while(tmp)
    {
        next = tmp->next;
        free(tmp);
        tmp = next;
    }
    free(*list);
    *list) = NULL;
}
 void pushFront(DLList *list, void *data)
{
    Node *tmp = (Node *) malloc(sizeof(Node));
    if(tmp == NULL)
        exit(1);
     tmp->value = data;
     tmp->next = list->head;
```

```
     tmp->prev = NULL;
     if(list->head)
         list->head->prev = tmp;
     list->head = tmp;
     if(list->tail == NULL)
         list->tail = tmp;
     list->size++;
}

void *popFront(DLList *list)
{
    Node *prev;
    void *tmp;
    if(list->head == NULL)
        exit(2);
    prev = list->head;
    list->head = list->head->next;
     if(list->head)
        list->head->prev = NULL;
     if(prev == list->tail)
         list->tail = NULL;
      tmp = prev->value;
      free(prev);
      list->size--;
      return tmp;
}

void pushBack(DLList *list, void *value)
{

Node *tmp = (Node *) malloc(sizeof(Node));
if(tmp == NULL)
   exit(3);
tmp->value = value;
tmp->next = NULL;
tmp->prev = list->tail;
if(list->tail)
    list->tail->next = tmp;
list->tail = tmp;
if(list->head == NULL)
    list->head = tmp;
list->size++;
}

void *popBack(DLList *list)
{
    Node *next;
    void *tmp;
    if(list->tail == NULL)
```

```
        exit(4);
    next = list->tail;
    list->tail = list->tail->prev;
    if(list->tail)
        list->tail->next = NULL;
    if(next == list->head)
        list->head = NULL;
    tmp = next->value;
    free(next);
   list->size--;
    return tmp;
}

Node *getNthq(DLList *list, size_t index)
{
    Node *tmp = NULL;
    size_t i;
    if(index < list->size/2)
    {
       i = 0;
       tmp = list->head;
       while(tmp && i < index)
        {
            tmp = tmp->next;
            i++;
        }
     }
     else
     {
        i = list->size - 1;
        tmp = list->tail;
        while(tmp && i > index)
        {
            tmp = tmp->prev i--;
        }
     }
     return tmp;
}

void insert(DLList *list, size_t index, void *value)
{
    Node *elm = NULL;
    Node *ins = NULL;
    elm = getNthq(list, index);
    if(elm == NULL) exit(5);
    ins = (Node *) malloc(sizeof(Node));
    ins->value = value;
    ins->prev = elm;
```

```
    ins->next = elm->next;
    if(elm->next)
        elm->next->prev = ins;
    elm->next = ins;
    if(!elm->prev)
        list->head = elm;
    if(!elm->next)
        list->tail = elm;
    list->size++;
}

 void *deleteNth(DLList *list, size_t index)
{
    Node *elm = NULL;
    void *tmp = NULL;
    elm = getNthq(list, index);
    if(elm == NULL)
        exit(5);
    if(elm->prev)
         elm->prev->next = elm->next;
    if(elm->next)
         elm->next->prev = elm->prev;
    tmp = elm->value;
    if(!elm->prev)
         list->head = elm->next;
    if(!elm->next)
         list->tail = elm->prev;
    free(elm);
    list->size--;
    return tmp;
}

void printDLList(DLList *list, void (*fun)(void *))
{
    Node *tmp = list->head;
    while(tmp)
    {
        fun(tmp->value);
        tmp = tmp->next;
    }
    printf("\n");
}

void printInt(void *value)
{
    printf("%d ", *((int *)value));
}
DLList *fromArray(void *arr, size_t n, size_t size)
{
```

```
    DLList *tmp = NULL;
    size_t i = 0;
    if(arr == NULL)
         exit(7);
    tmp = createDLList();
    while(i < n)
    {
        pushBack(tmp, ((char *)arr + i * size));
        i++;
    }
return tmp;
}
```

Результат роботи програми наведені на рис. 3.34 і рис. 3.35.

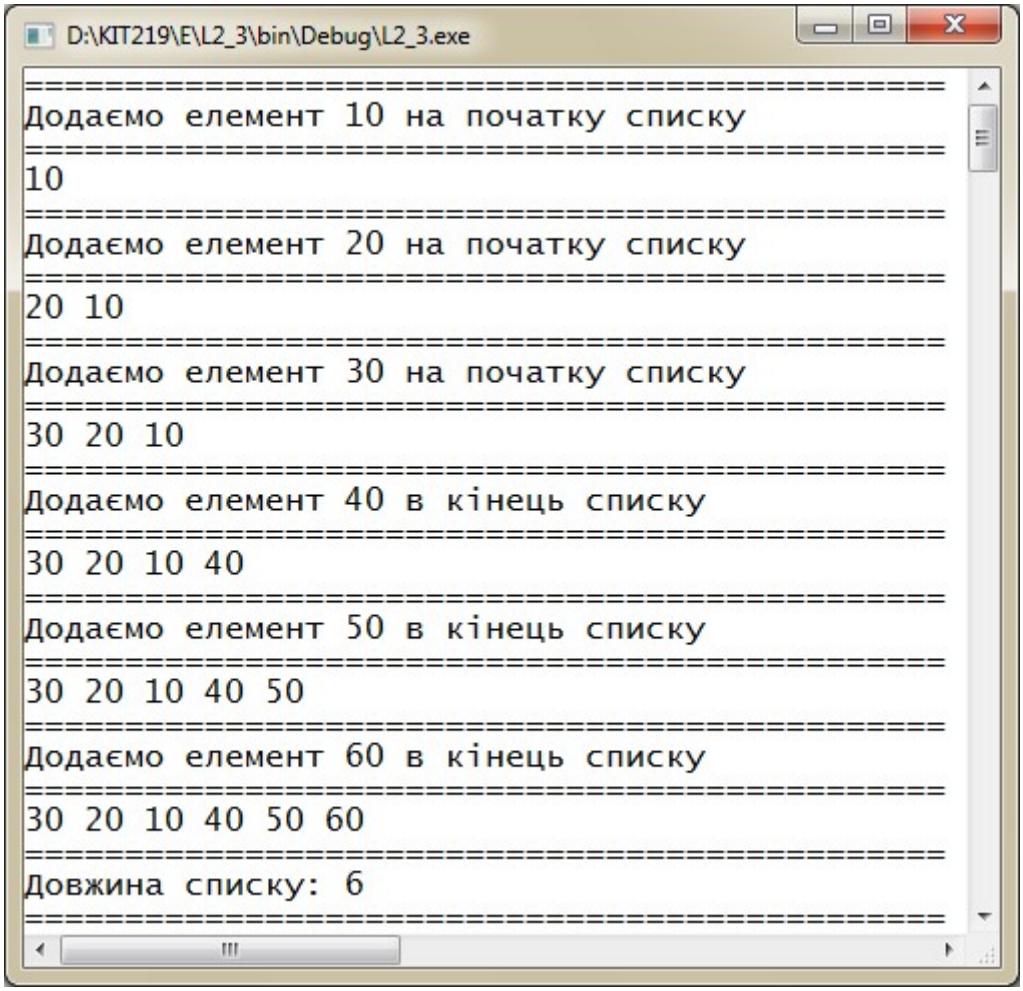


Рисунок 3.34 – Результат роботи програми з деком, що реалізований за допомогою списку (перша частина)

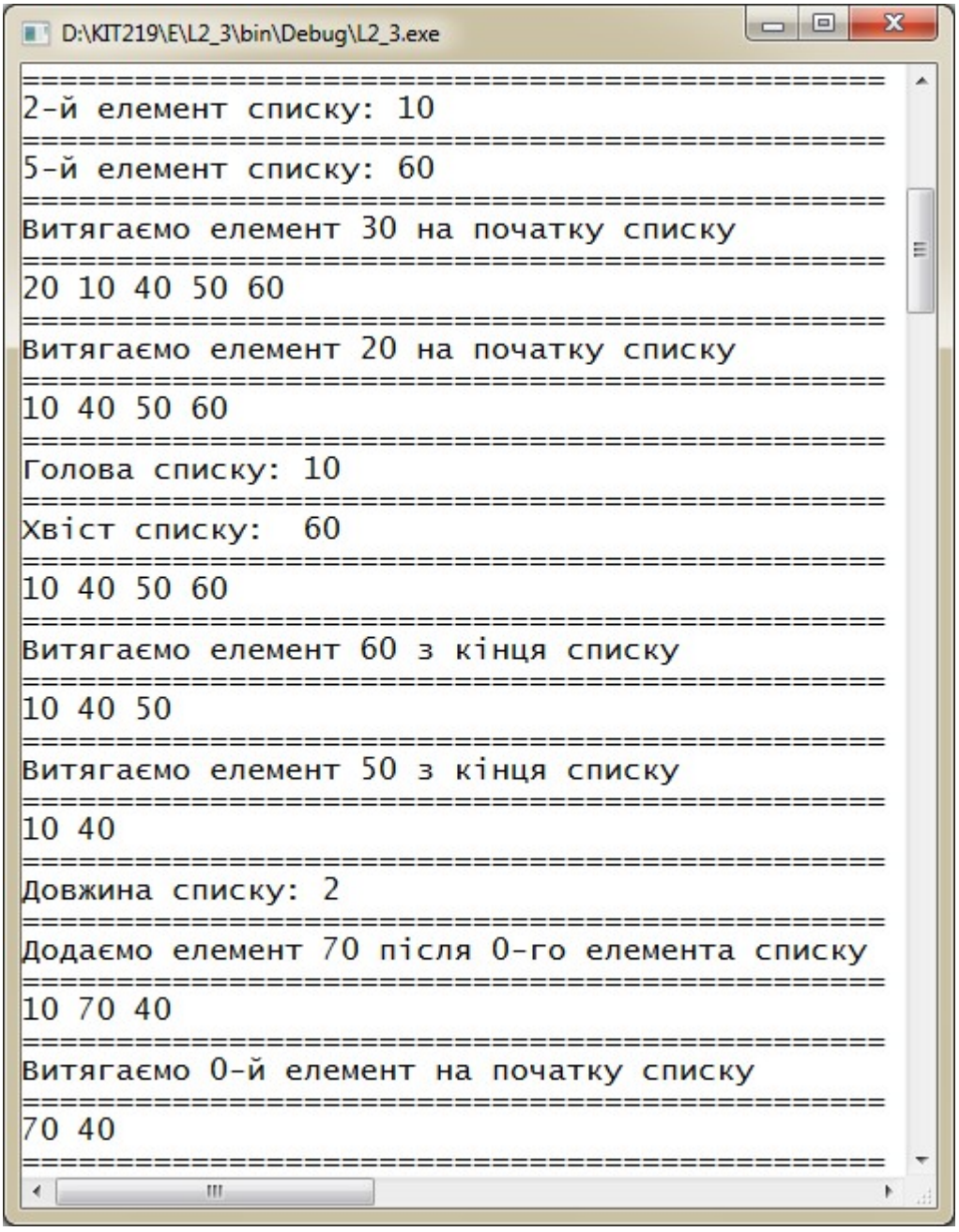


Рисунок 3.35 – Результат роботи програми з деком, що реалізований за допомогою списку (друга частина)

3.6. Дек. Реалізація деку на основі масиву

Розглянемо реалізацію деку на мові `C` з використанням стандарту `C99` на основі масиву. Оскільки, разом з деком, ми будемо обговорювати і масиви, доведеться мати справу з необхідністю розрізняти два види елементів: елементів деку та елементів масиву. Щоб не було плутанини, будемо завжди уточнювати,

про які елементи йде мова, за виключенням випадків, коли інформація про вигляд однозначно витікає з контексту. Наприклад, говорячи про голову та хвіст, завжди будемо мати на увазі елементи деку. Іноді будемо говорити про індекси елементів деку, розуміючи під ними індекси елементів масиву, в яких вони містяться.

Оскільки елементи деку будуть зберігатися в масиві, з'ясуємо як вони будуть в ньому розташовані.

Очевидним є рішення розташовувати елементи деку таким чином, щоб їх індекси завжди зростали в напрямку від голови до хвоста. Але в цьому випадку виникає проблема. Як бути, якщо треба додати елемент до голови деку, але елемент масиву з індексом `0` вже зайнятий, хоча при цьому є вільними елементи масиву, що розташовуються за хвостом? Один з можливих варіантів – зсунути усі елементи деку в бік хвоста, щоб звільнити один або декілька елементів масиву на його початку. Безумовно, місця може не виявитися не на початку масиву, а в кінці, тоді зсув елементів деку доведеться робити в бік голови. Однак цей варіант не кращим чином впливатиме на продуктивність, особливо, якщо до обох кінців деку постійно застосовуються операції додавання та видалення.

Набагато більш розумним представляється рішення «закільцювати» масив. Якщо в такому масиві індекс одного елемента на одиницю перевищує індекс другого, то перший елемент по відношенню до другого називається «попереднім», а другий по відношенню до першого – «наступним». Крім цього, елемент з нульовим індексом називається «наступним» по відношенню до елементу з максимальним індексом, а останній називається «попереднім» по відношенню до елементу з нульовим індексом.

Таким чином, у кожного без виключення елемента закільцьованого масиву є як попередній елемент, так і наступний.

3.6.1. Принцип додавання елементів деку до масиву

Під час додавання до хвоста ми поміщаємо елемент деку до елемента масиву, що йде після елемента, в якому міститься «старий» хвостовий елемент.

Під час додавання до голови – поміщаємо елемент деку до елемента масиву, що передує елементу, в якому зберігається «старий» головний елемент.

Таким чином, якщо елемент масиву з нульовим (максимальним) індексом є зайнятим, то при додаванні до голови (до хвоста) елемент деку поміщається до елементу масиву з максимальним (нульовим) індексом. Такий підхід гарантує, що завжди, коли у масиві є вільні елементи, можна застосовувати операції додавання як до голови, так і до хвоста.

Слід зауважити, що індекс елемента масиву, що містить головний елемент деку, може бути більшим за індекс елемента масиву, який містить хвостовий елемент деку (і навпаки). Індекси також можуть і збігатися. Цей випадок відповідає наявності в деку лише одного елемента.

3.6.2. Структура програми

Програма буде складатися з трьох файлів. Сам дек буде реалізований в файлах **deq.h** і **deq.c**. У файлі **main.c** будуть викликатися основні функції для тестування деку.

У файлі **deq.h** будуть описані типи, які необхідні для роботи деку, та прототипи функцій, що реалізують операції з деком. Реалізації функцій будуть міститися у файлі **deq.c**. Таким чином, ці два файли будуть складати бібліотеку для роботи з деком.

Файл **deq.h** починається з директив **#include**, які підключають необхідні стандартні бібліотеки:

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

До файлу **deq.c** буде підключений єдиний файл заголовку:

```
#include "deq.h"
```

3.6.3. Типи Item і Deq

Елементи, що містяться в деку, будуть мати тип **Item**, який описується наступним чином:

```
typedef int Item;          // тип елементів деку
```

Як ми бачимо, **Item** – це просто псевдонім для типу **int**. Якщо ми маємо намір зберігати в деку елементи будь-якого іншого типу, нам треба буде лише відредагувати цей рядок у файлі **deq.h**, замінивши **int** потрібним нам типом.

Сам дек буде екземпляром структури **deq**, що міститься у файлі **deq.h** і має наступне визначення:

```
typedef struct deq
{
    int head;              // індекс голови
    int tail;              // індекс хвоста
    int size;              // розмір масиву elements
    int count;             // кількість елементів у деку
    Item elements[];       // масив для зберігання елементів
} Deq;
```

Розглянемо поля структури **deq**:

- **elements[]** – це масив, в якому будуть зберігатися елементи деку. Розмір масиву на етапі компіляції є невідомим (про це свідчать порожні квадратні дужки). Розмір буде задаватися безпосередньо в ході створення деку. Використання такого поля змушує нас до динамічного створення об’єктів типу **Deq** і керування ними за допомогою вказівників (при автоматичному створенні екземплярів цієї структури пам’ять для масиву **elements[]** виділятися не буде);

- **size** – це розмір масиву **elements[]**, який можна розглядати як ємність або розмір нашого деку;

- **count** – кількість елементів, що зберігаються в деку;

- **head** і **tail** – індекси елементів масиву **elements[]**, що містять головний і хвостовий елементи деку відповідно. Якщо дек пустий, то значення цих полів дорівнюють **-1**.

3.6.4. Створення деку

За створення деку відповідає функція **createDeq()**, яка визначена наступним чином:

```
// створити дек
Deq *createDeq(int size)
{
    if(size <= 0)
        return 0;
    Deq *deq = malloc(sizeof(Deq) + size * sizeof(Item));
    if(deq)
    {
        deq->size = size;
        deq->head = deq->tail = -1;
        deq->count = 0;
     }
     return deq;
}
```

Функція **createDeq()** приймає в якості параметра розмір деку і, у випадку, якщо розмір більше нуля, намагається динамічно створити дек заданого розміру. Якщо розмір дорівнює нулю або пам'ять динамічно виділити не вдалося, функція повертає нульову адресу. В протилежному випадку поля створеного деку ініціалізуються початковими значеннями, після чого адреса деку повертається до функції, яка її викликала.

Саме функція, що викликає, і несе відповідальність за знищення деку після його використання. Пам'ять, яка зайнята деком, необхідно звільняти за допомогою функції **free()**.

3.6.5. Операції додавання елементів до деку

Додавання елементу до хвоста деку здійснюється функцією **pushBack()**, яка має наступний вигляд:

```
// додати елемент наприкінці деку
Deq *pushBack(Deq *deq, Item item)
{
    if(deq->count)
    {
      if(++(deq->tail) == deq->size)
          deq->tail = 0;
```

```
    }
else
        deq->tail = deq->head = 0;
     deq->elements[deq->tail] = item;
     deq->count++;
     return deq;
}
```

Функція приймає в якості параметрів адресу деку та значення елемента, що додається. Вона додає елемент наприкінці деку і повертає адресу деку, тобто значення `1`-го параметру. Зверніть увагу на те, що коли в момент виклику функції дек є пустим, то новий елемент поміщається в нульовий елемент масиву `elements[]`. В протилежному випадку додавання елемента відбувається у відповідності до вже розглянутого раніше принципу.

Код функції `pushBack()` є достатньо простим, як і код наступної функції `pushFront()`, що додає елемент до голови деку:

```
// додати елемент на початку деку
Deq *pushFront(Deq *deq, Item item)
{
    if(deq->count)
    {
         if(--(deq->head) < 0)
              deq->head = deq->size-1;
    }
else
        deq->tail = deq->head = 0;
  deq->elements[deq->head] = item;
  deq->count++;
  return deq;
}
```

3.6.6. Операції видалення елементів з деку

Нижче наведено код функції `popBack()`, яка видаляє елемент наприкінці деку:

```
Item popBack(Deq *deq)    // видалити елемент наприкінці деку
{
    int index = deq->tail;
if(deq->count == 1)
    deq->tail = deq->head = -1;
else
        if(--(deq->tail) < 0)
```

```
            deq->tail = deq->size-1;
    deq->count--;
    return deq->elements[index];
}
```

Функція приймає в якості параметра адресу деку, видаляє його хвостовий елемент і повертає цей елемент. Безумовно, необхідності в очищенні елемента масиву **elements[]**, що містить хвостовий елемент, при цьому немає. Таким чином, головна дія функції полягає в коректній зміні поля **tail** деку, яке адресується формальним параметром **deq**. Вміст поля замінюється індексом елемента масиву, що передує елементу, чий індекс знаходився в полі раніше, за умови, що елемент, який видаляється, – не є останнім.

Якщо ж з деку видаляється останній елемент, то полям **tail** і **head** об'єкту, який адресується параметром **deq**, присвоюється **-1**.

Код функції **popFront()**, що видаляє елемент з початку деку, має наступний вигляд:

```
// видалити елемент на початку деку
Item popFront(Deq *deq)
{
    int index = deq->head;
    if(deq->count == 1)
         deq->tail = deq->head = -1;
    else
        if(++(deq->head) == deq->size)
           deq->head = 0;
    deq->count--;
    return deq->elements[index];
}
```

Якщо елемент, що видаляється, не є останнім, значення поля **head** об'єкту, який адресується параметром **deq**, замінюється індексом елемента масиву, що йде за елементом, який містить елемент, що видаляється з деку.

Для комфортної роботи з деком можуть знадобиться і додаткові операції.

3.6.7. Отримання інформації про стан деку

Може виявитися корисною функція **isEmpty()**, що приймає в якості параметра адресу деку і повертає **true**, якщо дек не містить жодного елемента, або **false** – у протилежному випадку. Код цієї функції має наступний вигляд:

```
bool isEmpty(Deq *deq)    // повертає true, якщо дек пустий
{
    return !deq->count;
}
```

Код функції **isFull()**, що приймає адресу деку і повертає **true**, якщо дек заповнений, або **false** – в протилежному випадку, має наступний вигляд:

```
// повертає true, якщо дек повний
bool isFull(Deq *deq)
{
    return deq->count == deq->size; }
```

3.6.8. Отримання крайніх елементів без видалення їх з деку

Наступні дві функції приймають в якості параметра адресу деку і повертають хвостовий і головний елементи деку відповідно. При цьому самі елементи з деку не видаляються.

```
// отримати елемент з кінця деку без видалення
Item peekBack(Deq *deq)
{
    return deq->elements[deq->tail];
}

// отримати елемент на початку деку без видалення
Item peekFront(Deq *deq)
{
    return deq->elements[deq->head];
}
```

3.6.9. Перебір елементів деку за допомогою ітератора

Припустимо, що ми бажаємо вивести на екран вміст деку, не видаляючи з нього елементи. Які дії слід вжити в цьому випадку?

Один з можливих способів – написати спеціальну функцію, яка перебирає елементи деку і виводить кожен з них на екран. Однак, оскільки наш дек є універсальним (щоб перейти від одного типу елементів, що зберігаються в деку, до другого, треба лише змінити цей тип у файлі заголовку), ми не можемо поміщати до бібліотеки для роботи з деком функції, які виводять на екран елементи конкретних типів. Відповідно, функція виводу на екран елемента повинна знаходитися поза бібліотекою, а вказівник на неї повинен бути переданий функції, що виводить на екран вміст усього деку. Для рішення цієї задачі будемо використовувати ітератори (не плутати з ітераторами з бібліотеки шаблонів **C++**).

Ітератор – це об'єкт, звертаючись до якого за допомогою відповідних функцій, можна перебирати елементи певної колекції, за якою цей ітератор закріплений (використовують навіть словосполучення «ітератор колекції»). Саме дек і є окремим випадком колекції елементів.

Ітератор для нашого деку буде являти собою екземпляр структури **iterator**, яку ми визначимо у файлі **deq.h** наступним чином:

```
typedef struct iterator
{
 Deq *deq;    // містить адресу деку, за яким він закріплений
 int index;   // індекс поточного елемента масиву
} Iterator;
```

Поле **deq** ітератора буде містити адресу деку, за якою даний ітератор закріплений. У полі **index** повинен зберігатися індекс елемента масиву, де знаходиться певний елемент деку, який ми назвемо «поточним».

Створенням ітератора деку буде займатися функція **getIterator()**, що має наступний вигляд:

```
// отримати ітератор
Iterator getIterator(Deq *deq, bool head)
{
   Iterator it;
   it.deq = deq;
   it.index = head ? deq->head : deq->tail;
   return it;
}
```

Функція приймає в якості параметрів адресу деку і значення типу **bool**.

Вона створює об'єкт типу **Iterator**, присвоюючи його полю **index** індекс або головного, або хвостового елемента деку, в залежності від значення **2**-го формального параметру. Якщо воно дорівнює **true**, то присвоюється індекс головного елемента, а якщо **false**, – то хвостового. Полю **deq** ітератора присвоюється адреса деку, що передається функції **getIterator()**. Таким чином, поточним елементом деку стає або його перший елемент, або останній.

Функція **getIterator()** повертає значення створеного ітератора.

Перебір елементів деку за допомогою ітератора відбувається з використанням функцій **next()** і **prev()**. Код першої з них має такий вигляд:

```
// отримати поточний елемент і зробити поточним наступний
Item next(Iterator *it)
{
int index = it->index;
if(index == it->deq->tail)
it->index = -1;
else
    if(++(it->index) == it->deq->size)
      it->index = 0;
return it->deq->elements[index];
}
```

Функція **next()** приймає адресу ітератора і повертає елемент деку, зв'язаного з цим ітератором, який на момент виклику функції був для ітератора поточним. Але перед поверненням значення функція робить поточним наступний елемент деку. Іншими словами, функція заміняє значення поля **index** ітератора індексом елемента масиву, який йде за тим елементом, чий індекс містився в полі до цього. Безумовно, ми говоримо про масив, в якому містяться елементи деку.

Якщо ж поточним елементом для ітератора на момент виклику функції був хвостовий елемент, то полю **index** ітератора присвоюється **-1**.

Код функції **prev()** має наступний вигляд:

```
// отримати поточний елемент і зробити поточним попередній
Item prev(Iterator *it)
{
    int index = it->index;
```

```
if(index == it->deq->head)
    it->index = -1;
else
        if(--(it->index) < 0)
     it->index = it->deq->size-1;
return it->deq->elements[index];
}
```

Функція `prev()` відрізняється від `next()` тим, що в якості нового поточного елемента деку для ітератора вона встановлює попередній (по відношенню до старого поточного) елемент деку, а якщо старий поточний елемент був головним, то полю `index` ітератора присвоюється `-1`.

За допомогою створених нами інструментів перебирати усі елементи деку від початку до кінця можна наступним чином. Спочатку отримуємо ітератор деку, викликавши функцію `getIterator()` та передавши їй в якості `2`-го параметру `true`. Далі багатократно викликаємо функцію `next()`, передаючи їй кожного разу адресу ітератора до тих пір, поки поле `index` ітератора не стане дорівнювати `-1`. Значення, що повертаються функцією `next()`, і будуть елементами деку, що перебираються від голови до хвоста.

Елементи деку у зворотному порядку перебираються таким самим чином, з тією лише різницею, що функції `getIterator()` передається в якості `2`-го параметру `false`, а замість функції `next()` викликається функція `prev()`.

Метод перебору колекції за допомогою ітератора є більш гнучким, ніж той, що розглядався раніше. Завдяки щойно розглянутому методу ми отримуємо доступ до усіх елементів колекції, тому можемо обробляти будь-які з них довільним чином. Усе, що ми могли робити раніше – це лише вивести одразу усі елементи (у прямому або зворотному порядку) на екран.

Функції додавання та видалення елементів, отримання крайніх елементів, а також функції перебору елементів за допомогою ітераторів, передбачають, що аргументи, які передані їм, є коректними. Наприклад, при додаванні елементів дек не повинен бути повним, а при видаленні – пустим. Під час виклику функцій перебору поле `index` ітератора не повинне містити `-1`, а значення поля повинне бути індексом елемента масиву, який дійсно містить елемент деку.

Жодних перевірок значень фактичних аргументів у згаданих функціях не передбачено, тому коректність даних значень повинна контролюватися функціями, що їх викликають. Такий підхід дозволяє дещо зекономити час роботи функцій, які призначені для роботи з деком.

3.6.10. Тестування розроблених функцій

Для тестування розроблених функцій будемо використовувати файл **main.c**.

Підключимо до нього файли заголовків, один з яких – це створений файл **deq.h**:

```
#include <stdio.h>
#include <windows.h>
#include "deq.h"
 int main(void)
{
    SetConsoleOutputCP(1251);
   Deq *deq = createDeq(10);
    while(!isFull(deq))
    {
        int r = rand() % 77 + 10;
        printf("Додаємо ");
        if(r % 2)
        {
         printf("до хвоста: %2d\n", r);
         pushBack(deq, r);
        }
else
        {
         printf("до голови: %2d\n", r);
         pushFront(deq, r);
        }
    }
Iterator it = getIterator(deq, true);
puts("Дек від голови до хвоста:");
while(it.index != -1)
    printf("%3d", next(&it));
it = getIterator(deq, false);
puts("\nДек від хвоста до голови:");
while(it.index != -1)
     printf("%3d", prev(&it));
printf("\n");
printf("Голова: %2d\n", peekFront(deq));
printf("Хвіст:  %2d\n", peekBack(deq));
```

```
    while(!isEmpty(d))
        {
        printf("Видаляємо з ");
        if(rand() % 2)
                printf("хвоста: %2d\n", popBack(deq));
    else
                printf("голови: %2d\n", popFront(deq));
        }
    free(deq);
    return 0;
    }
```

В тексті функції `main()` задіяні усі функції, що були створені раніше.

Спочатку динамічно створюємо дек, який розрахований на `10` елементів. Далі додаємо до деку елементи до тих пір, поки він не виявиться заповненим. Елементи, що додаються, створюються генератором псевдовипадкових чисел, причому непарні числа додаються до хвоста деку, а парні – до голови.

Далі за допомогою ітератора перебираємо і виводимо на екран усі елементи деку в прямому напрямку (від голови до хвоста), а потім – у зворотному напрямку. Виводимо на екран спочатку головний елемент, а потім і хвостовий. Далі видаляємо усі елементи з деку і виводимо їх на екран. Вибір частини деку, з якої видаляється черговий елемент (хвіст або голова) відбувається випадковим чином. По закінченню роботи функції видаляємо дек.

Повний текст програми з файлу `main.c` без вмісту файлу `deq.h`, який також разом з файлом `deq.c` повинен входити до складу проекту, має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include "deq.h"
 int
main(void)
{
    SetConsoleOutputCP(1251);

    Deq *deq = createDeq(10);
    while(!isFull(deq))
    {
        int r = rand() % 77 + 10;
printf("Додаємо ");
```

```
if(r % 2)
{
printf("до хвоста: %2d\n", r);
pushBack(deq, r);
        }
else
        {
printf("до голови: %2d\n", r);
pushFront(deq, r);
        }
    }
Iterator it = getIterator(deq, true);
printf("==============================\n");
puts("Дек від голови до хвоста:");
printf("==============================\n");
while(it.index != -1)
      printf("%3d", next(&it));
printf("\n==============================\n");
printf("==============================\n");
printf("Голова: %d\n", peekFront(deq));
printf("Хвіст:  %d\n", peekBack(deq));
printf("==============================\n");
it = getIterator(deq, false);
printf("==============================\n");
puts("Дек від хвоста до голови:");
printf("==============================\n");
while(it.index != -1)
     printf("%3d", prev(&it));
printf("\n==============================\n");
printf("\n");
while(!isEmpty(deq))
    {
        printf("Видаляємо з ");
        if(rand() % 2)
            printf("хвоста: %2d\n", popBack(deq));
        else
            printf("голови: %2d\n", popFront(deq));
    }
        free(deq);
        return 0;
}
```

Файл **deq.c** буде містити реалізації всіх функцій для роботи з деком:

```
#include "deq.h"
// отримати поточний елемент і зробити поточним попередній
Item prev(Iterator *it)
{
    int index = it->index;
    if(index == it->deq->head)
```

```
       it->index = -1;
      else
         if(--(it->index) < 0)
            it->index = it->deq->size-1;
      return it->deq->elements[index];
}

// отримати поточний елемент і зробити поточним наступний
Item next(Iterator *it)
{
     int index = it->index;
     if(index == it->deq->tail)
        it->index = -1;
      else
         if(++(it->index) == it->deq->size)
            it->index = 0;
      return it->deq->elements[index];
}

// отримати ітератор
Iterator getIterator(Deq *deq, bool head)
{
     Iterator it;
     it.deq = deq;
     it.index = head ? deq->head : deq->tail;
     return it;
}

// отримати елемент з кінця деку без видалення
Item peekBack(Deq *deq)
{
     return deq->elements[deq->tail];
}

// отримати елемент на початку деку без видалення
Item peekFront(Deq *deq)
{
     return deq->elements[deq->head]; }

// повертає true, якщо дек повний
bool isFull(Deq *deq)
{
     return deq->count == deq->size;
}

// повертає true, якщо дек порожній
bool isEmpty(Deq *deq)
{
```

```
    return !deq->count;
}

// видалити елемент на початку деку
Item popFront(Deq *deq)
{
    int index = deq->head;
if(deq->count == 1)
    deq->tail = deq->head = -1;
else
        if(++(deq->head) == deq->size)
         deq->head = 0;
    deq->count--;
    return deq->elements[index];
}

// видалити елемент наприкінці деку
Item popBack(Deq *deq)
{
    int index = deq->tail;
    if(deq->count == 1)
      deq->tail = deq->head = -1;
    else
        if(--(deq->tail) < 0)
          deq->tail = deq->size-1;
    deq->count--;
    return deq->elements[index];
}

// додати елемент на початку деку
Deq *pushFront(Deq *deq, Item item)
{
    if(deq->count)
    {
        if(--(deq->head) < 0)
          deq->head = deq->size-1;
    }
else
        deq->tail = deq->head = 0;
        deq->elements[deq->head] = item;
        deq->count++;
  return deq;
}

// додати елемент наприкінці деку
Deq *pushBack(Deq *deq, Item item)
{
    if(deq->count)
    {
```

```
        if(++(deq->tail) == deq->size)
            deq->tail = 0;
    }
else
        deq->tail = deq->head = 0;
        deq->elements[deq->tail] = item;
        deq->count++;
return deq;
}

// створити дек
Deq *createDeq(int size)
{
    if(size <= 0)
return 0;
    Deq *deq = malloc(sizeof(Deq) + size * sizeof(Item));
if(deq)
    {
        deq->size = size;
        deq->head = deq->tail = -1;
        deq->count = 0;
    }
return deq;
}
```

У файлі **deq.h** описані типи, які необхідні для роботи деку, та прототипи функцій, що реалізують операції з деком:

```
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
typedef int Item;          // тип елементів деку
typedef struct deq
{
    int head;          // індекс голови
    int tail;          // індекс хвоста
    int size;          // розмір масиву elements
    int count;         // кількість елементів у деку
    Item elements[];   // масив для зберігання елементів
} Deq;
typedef struct iterator
{
    Deq *deq;  // містить адресу деку, за яким він закріплений
    int index; // індекс поточного елемента масиву
} Iterator;
Item prev(Iterator *it);
Item next(Iterator *it);
```

```
Iterator getIterator(Deq *deq, bool head);
Item peekBack(Deq *deq);
Item peekFront(Deq *deq);
bool isFull(Deq *deq);
bool isEmpty(Deq *deq);
Item popFront(Deq *deq);
Item popBack(Deq *deq);
Deq *pushFront(Deq *deq, Item item);
Deq *pushBack(Deq *deq, Item item);
Deq *createDeq(int size);
```

Результат роботи програми наведено на рис. 3.36.

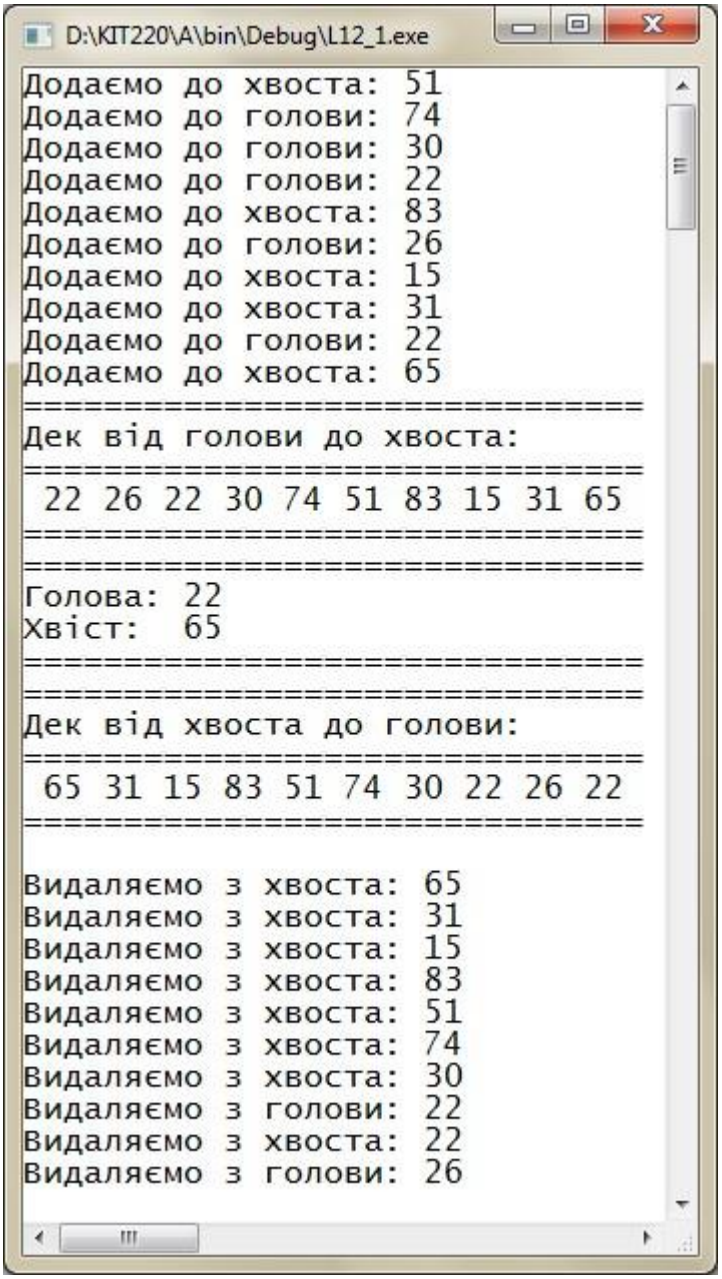


Рисунок 3.36 – Результат виконання програми для роботи з деком

3.6.11. Зміна розміру деку

В ході роботи з деком може виникнути ситуація, при якій дек є вже повним, але треба додати до нього елементи. А може трапитися так, що в деку дуже багато вільних місць, і необхідно зменшити його розмір, щоб звільнити пам'ять.

В обох випадках є необхідність змінити розмір деку. Напишемо функцію, що вирішує цю задачу. Однак треба мати на увазі, що при створенні деку пам'ять виділяється динамічно для всієї змінної типу **Deq** цілком. Тому ми не можемо один масив, що використовується для зберігання елементів деку, замінити іншим (тобто таким, що має розмір, який відрізняється від розміру першого). Міняти доведеться дек цілком, тобто створювати новий дек, копіювати до нього елементи старого деку, після чого видаляти старий дек. Саме цим і буде займатися функція **resize()**. Її код має наступний вигляд:

```
Deq *resize(Deq *deq, int new_size)    // Змінити розмір деку
{
    if(new_size <= 0)
     return 0;
    if(new_size == deq->size)
        return deq;
    Deq *ndeq = malloc(sizeof(Deq) + new_size * sizeof(Item));
if(!ndeq)
     return 0;
ndeq->size = new_size;
ndeq->count =
        new_size > deq->count ? deq->count : new_size;
ndeq->head = 0;
ndeq->tail = ndeq->count-1;
int r = deq->size - deq->head;
if(ndeq->count <= r)
        memcpy(ndeq->elements, deq->elements + deq->head,
                 ndeq->count * sizeof(Item));
else
    {
        memcpy(ndeq->elements, deq->elements + deq->head,
                r * sizeof(Item));
        memcpy(ndeq->elements + deq->size - deq->head,
               deq->elements, (ndeq->count-r) * sizeof(Item));
    }
free(deq);
return ndeq; }
```

Функція `resize()` приймає в якості першого параметра адресу деку, а в якості другого – бажаний розмір деку. Якщо останній співпадає з розміром деку, то функція повертає його адресу. Якщо другий параметр не є додатним, то функція повертає нульову адресу. В решті випадків функція намагається створити новий дек необхідного розміру.

У випадку успіху функція копіює до нового деку найбільше можливе число елементів старого деку (якщо кількість елементів старого деку перевищує розмір нового, то до нового деку потрапляють тільки перші елементи старого. Іншими словами, від старого «відрізається» хвіст). Потім старий дек знищується, а адреса нового повертається до функції, що викликає. У випадку невдачі функція повертає нульову адресу.

Голова деку тепер буде зберігатися в елементі масиву з індексом `0`. Кількість елементів, що зберігаються в новому деку, обчислюється як найменше з двох чисел. Перше з них – це кількість елементів, що знаходяться в старому деку, а друге – це розмір нового деку.

Обчислюємо значення змінної `r` – кількість елементів деку, чиї індекси не менш ніж індекс голови деку. Якщо кількість елементів, які будуть міститися в новому деку, не перевищує значення `r`, то копіюємо необхідні елементи деку з масиву старого деку до масиву нового за один прийом. Обмежитися одним прийомом ми можемо тому, що в цьому випадку перебору елементів від голови до хвоста відповідають повільне збільшення їх індексів.

В іншому випадку перебір елементів деку, що підлягають копіюванню, в напрямку від голови до хвоста вже не буде відповідати повільному збільшенню їх індексів (він порушиться при переході від елемента деку з найбільшим індексом до елементу з нульовим індексом). На цей раз копіюємо елементи у два прийоми: частина з них з кінця масиву старого деку, а частина – з початку.

В обох випадках скопійовані елементи деку будуть розташовуватися в новому масиві так, що голова буде мати нульовий індекс, а кожен наступний елемент буде мати індекс, що на одиницю перевищує індекс попереднього.

Наприкінці функції `resize()` видаляємо пам'ять під старий дек і повертаємо вказівник на ділянку пам'яті під новий дек.

3.6.12. Тестування функції resize()

Для тестування розробленої функції `resize()` будемо використовувати функцію `main()`:

```
int main(void)
{
SetConsoleOutputCP(1251);
Deq *deq = createDeq(7);
pushBack(pushBack(pushBack(pushBack(deq, 2), 4), 6), 8);
pushFront(pushFront(pushFront(deq, 1), 3), 5);
Iterator it = getIterator(deq, true);
puts("Дек розміру 7 від голови до хвоста:");
printf("=========================================\n");
while(it.index != -1)
      printf("%3d", next(&it));
printf("\n=========================================\n");
puts("Збільшуємо розмір деку\n");
deq = resize(deq, 14);
pushBack(pushBack(pushBack(pushBack(deq, 10), 12), 14), 16);
pushFront(pushFront(pushFront(deq, 7), 9), 11);
it = getIterator(deq, true);
puts("Дек розміру 14 від голови до хвоста:");
printf("=========================================\n");
while(it.index != -1)         printf("%3d", next(&it));
    printf("\n=========================================\n");
puts("\nЗменшуємо розмір деку");
deq = resize(deq, 10);
it = getIterator(deq, true);
puts("Дек розміру 10 від голови до хвоста:");
while (it.index != -1) printf("%3d", next(&it));
puts("\nЗнову зменшуємо розмір деку");
deq = resize(deq, 5);
it = getIterator(deq, true);
puts("Дек розміру 5 від голови до хвоста:");
while(it.index != -1) printf("%3d", next(&it));
free(deq);
return 0;
}
```

В цій програмі ми створюємо дек з `7` елементів. Далі заповнюємо його значеннями типу `int` і виводимо вміст деку на екран. Потім збільшуємо розмір деку до `14` елементів, додаємо до нього ще `7` елементів і знову виводимо на екран

його вміст. Далі зменшуємо розмір деку до 10 елементів і виводимо його вміст на екран. Нарешті, знову зменшуємо розмір деку, на цей раз до 5, і виводимо усі його елементи на екран. Не забуваємо наприкінці програми видалити дек.

Дана програма використовує як операцію збільшення розміру деку, так і операцію його зменшення. Зазначимо також, що збільшуючи розмір деку і зменшуючи його в перший раз, ми копіюємо елементи деку зі старого масиву до нового у два прийоми. Зменшуючи дек другий раз, ми обмежуємося одним прийомом. Таким чином, обидва види копіювання піддаються перевірці.

Повний текст програми з файлу main.c без вмісту файлу deq.h, який також разом з файлом deq.c повинен входити до складу проекту, має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include "deq.h"
 int
main(void)
{
    SetConsoleOutputCP(1251);
    Deq *deq = createDeq(7);
    pushBack(pushBack(pushBack(pushBack(deq, 2), 4), 6), 8);
    pushFront(pushFront(pushFront(deq, 1), 3), 5);
Iterator it = getIterator(deq, true);
puts("Дек розміру 7 від голови до хвоста:");
while(it.index != -1)
     printf("%3d", next(&it));
puts("\nЗбільшуємо розмір деку");
deq = resize(deq, 14);
pushBack(pushBack(pushBack(pushBack(deq, 10), 12), 14), 16);
pushFront(pushFront(pushFront(deq, 7), 9), 11);
it = getIterator(deq, true);
puts("Дек розміру 14 від голови до хвоста:");
while(it.index != -1)
       printf("%3d", next(&it));
puts("\nЗменшуємо розмір деку");
deq = resize(deq, 10);
it = getIterator(deq, true);
puts("Дек розміру 10 від голови до хвоста:");
while(it.index != -1)
   printf("%3d", next(&it));
puts("\nЗнову зменшуємо розмір деку");
deq = resize(deq, 5);
```

```
it = getIterator(deq, true);
    puts("Дек розміру 5 від голови до хвоста:");
while(it.index != -1)  printf("%3d", next(&it));
free(deq);
return 0;
}
```

Файл **deq.c** буде містити реалізації всіх функцій для роботи з деком:

```
#include "deq.h"

// отримати поточний елемент і зробити поточним попередній
Item prev(Iterator *it)
{
    int index = it->index;
if(index == it->deq->head)
it->index = -1;
else
        if(--(it->index) < 0)
it->index = it->deq->size-1;
return it->deq->elements[index];
}

// отримати поточний елемент і зробити поточним наступний
Item next(Iterator *it)
{
    int index = it->index;
if(index == it->deq->tail)
it->index = -1;
else
        if(++(it->index) == it->deq->size)
it->index = 0;
return it->deq->elements[index];
}

// отримати ітератор
Iterator getIterator(Deq *deq, bool head)
{
    Iterator it;
    it.deq = deq;
    it.index = head ? deq->head : deq->tail;
    return it;
}

// отримати елемент з кінця деку без видалення
Item peekBack(Deq *deq)
{
    return deq->elements[deq->tail];
}
```

```
// отримати елемент на початку деку без видалення
Item peekFront(Deq *deq)
{
    return deq->elements[deq->head];
}

// повертає true, якщо дек повний
bool isFull(Deq *deq)
{
    return deq->count == deq->size;
}

// повертає true, якщо дек порожній
bool isEmpty(Deq *deq)
{
    return !deq->count;
}

// видалити елемент на початку деку
Item popFront(Deq *deq)
{
    int index = deq->head;
    if(deq->count == 1)
        deq->tail = deq->head = -1;
    else
        if(++(deq->head) == deq->size)
          deq->head = 0;
    deq->count--;
    return deq->elements[index];
}

// видалити елемент наприкінці деку
Item popBack(Deq *deq)
{
    int index = deq->tail;
if(deq->count == 1)
   deq->tail = deq->head = -1;
else
        if(--(deq->tail) < 0)
           deq->tail = deq->size-1;
        deq->count--;
    return deq->elements[index];
}

// додати елемент на початку деку
Deq *pushFront(Deq *deq, Item item)
{
    if(deq->count)
```

```
    {
       if(--(deq->head) < 0
         deq->head = deq->size-1;
    }
else
        deq->tail = deq->head = 0;
deq->elements[deq->head] = item;
deq->count++;
return deq;
}

// додати елемент наприкінці деку
Deq *pushBack(Deq *deq, Item item)
{
    if(deq->count)
    {
        if(++(deq->tail) == deq->size)
                deq->tail = 0;
    }
    else
        deq->tail = deq->head = 0;
deq->elements[deq->tail] = item;
deq->count++;
return deq;
}

// створити дек
Deq *createDeq(int size)
{
    if(size <= 0)
return 0;
    Deq *deq = malloc(sizeof(Deq) + size * sizeof(Item));
if(deq)
    {
deq->size = size;
deq->head = deq->tail = -1;
deq->count = 0;
    }
return deq;
}

// Змінити розмір деку
Deq *resize(Deq *deq, int new_size)
{
    if(new_size <= 0)
return 0;
 if(new_size == deq->size)
        return deq;
```

```
    Deq *ndeq = malloc(sizeof(Deq) + new_size * sizeof(Item));
if(!ndeq)
return 0;
ndeq->size = new_size;
ndeq->count = new_size > deq->count ? deq->count : new_size;
ndeq->head = 0;
ndeq->tail = ndeq->count-1;
int r = deq->size - deq->head;
if(ndeq->count <= r)
memcpy(ndeq->elements, deq->elements + deq->head,
        ndeq->count * sizeof(Item));
else
    {
      memcpy(ndeq->elements, deq->elements + deq->head,
                r * sizeof(Item));
        memcpy(ndeq->elements + deq->size - deq->head,
deq->elements, (ndeq->count-r) * sizeof(Item));
    }
free(deq);
return ndeq;
}
```

У файлі **deq.h** описані типи, які необхідні для роботи деку, та прототипи функцій, що реалізують операції з деком:

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

typedef int Item;          // тип елементів деку

typedef struct deq
{
int head;               // індекс голови
int tail;               // індекс хвоста
int size;               // розмір масиву elements
int count;              // кількість елементів у деку
Item elements[];        // масив для зберігання елементів }
} Deq;

typedef struct iterator
{
   Deq *deq;    // містить адресу деку, за яким він закріплений
   int index;   // індекс поточного елемента масиву
} Iterator;

Item prev(Iterator *it);
```

```
    Item next(Iterator *it);
    Iterator getIterator(Deq *deq, bool head);
    Item peekBack(Deq *deq);
    Item peekFront(Deq *deq);
    bool isFull(Deq *deq);
    bool isEmpty(Deq *deq);
    Item popFront(Deq *deq);
    Item popBack(Deq *deq);
    Deq *pushFront(Deq *deq, Item item);
    Deq *pushBack(Deq *deq, Item item);
    Deq *createDeq(int size);
    Deq *resize(Deq *deq, int
new_size);
```

Результат виконання програми наведено на рис. 3.37.

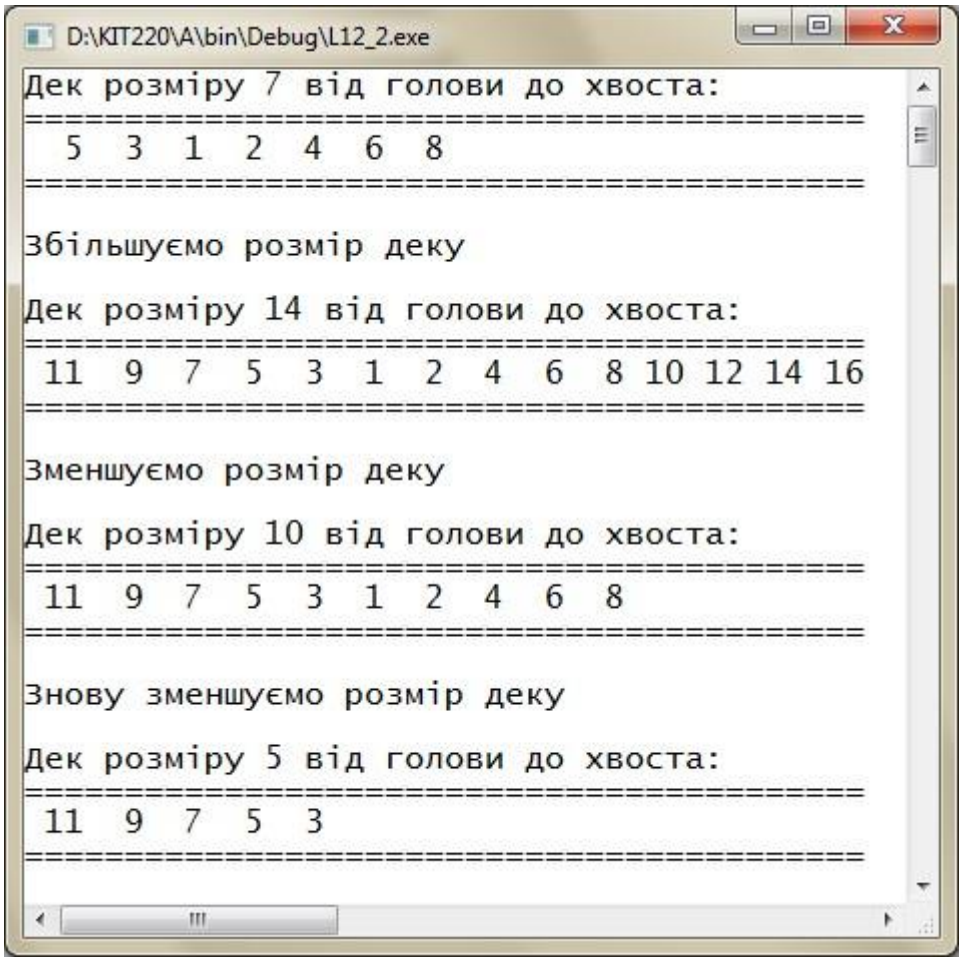


Рисунок 3.37 – Результат роботи програми, яка змінює розміри деку

3.7. Двійкове дерево пошуку

Двійкове дерево пошуку – це зв'язна структура, яка підтримує стратегію двійкового пошуку. Кожен вузол дерева містить елемент і два вказівники на інші

вузли – **дочірні вузли**. Зв’язок між вузлами в двійковому дереві пошуку наведено на рис. 3.38. Основна ідея цієї структури полягає в тому, що кожен вузол має два дочірніх вузла – лівий і правий. Порядок елементів визначається тим, що елемент у лівому вузлі передує елементу в батьківському вузлі, а елемент у правому вузлі йде за елементом батьківського вузла. Це відношення зберігається для всіх вузлів серед дочірніх вузлів. Більш того, всі елементи, чий родовід може бути простежено до лівого вузла батьківського вузла, містять елементи, які передують батьківського елементу, а всі елементи, які є потомками правого вузла, містять елементи, які йдуть за батьківським елементом. Слова в дереві на рис. 1.1 зберігаються саме таким чином.

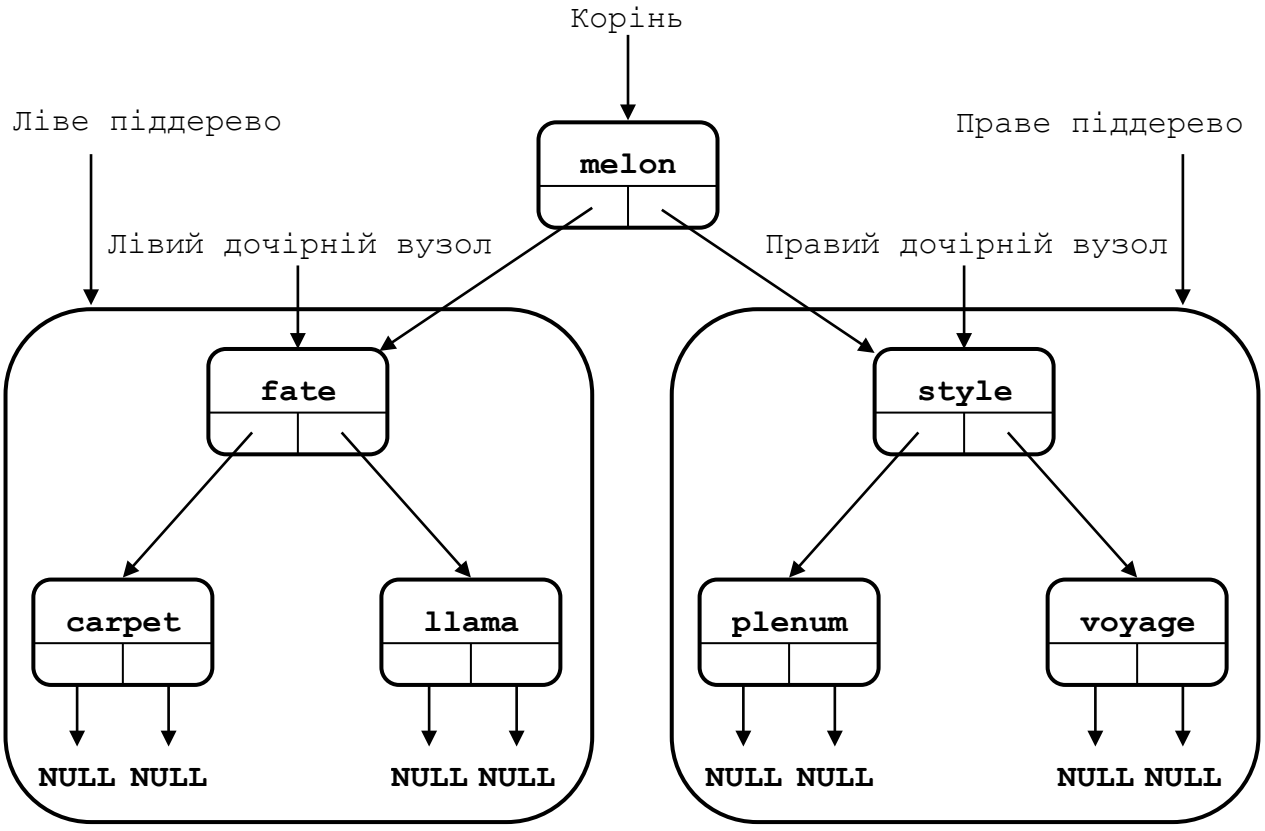


Рисунок 3.38 – Двійкове дерево пошуку

Верхня частина дерева, на відміну від ботаніки, називається **коренем**. Дерево являє собою **ієрархічну** організацію даних, тобто дані організовані за рангами, або рівнями, причому в загальному випадку кожному рангу відповідають ранги, що розташовані над та під ним. Якщо двійкове дерево

пошуку повністю заповнене, кожен рівень містить вдвічі більше вузлів, ніж рівень, який розташований над ним.

Кожен вузол у двійковому дереві пошуку сам є коренем вузлів, які виходять з нього. Це перетворює вузол і його потомків на **піддерево**.

Наприклад, на рис. 3.38 вузли, що містять слова `fate`, `carpet` і `llama`, утворюють ліве піддерево усього дерева, а слово `voyage` є правим піддеревом піддерева `style-plenum-voyage`.

Припустимо, що в такому дереві необхідно знайти елемент – назвемо його цільовим. Якщо елемент передує кореневому елементу, пошук треба буде робити тільки в лівій половині дерева. Якщо ж цільовий елемент йде за кореневим елементом, пошук повинен виконуватися тільки в правому піддереві кореневого вузла. Таким чином, одне порівняння виключає з пошуку половину дерева.

Припустимо, що пошук здійснюється в лівій половині. Це означає порівняння цільового елемента з елементом у лівому дочірньому вузлі. Якщо цільовий елемент передує елементу лівого дочірнього вузла, то пошук необхідно робити тільки в лівій половині дочірніх вузлів і т. д. Як і при двійковому пошуку, кожне порівняння зменшує кількість потенційних зіставлень у два рази.

Давайте застосуємо цей метод, щоб з'ясувати присутність слова `puppy` в дереві, яке наведено на рис. 3.38.

Порівнюючи слово `puppy` з `melon` (елементом кореневого вузла) ми бачимо, що слово `puppy`, якщо воно присутнє, повинне розташовуватися в правій половині дерева. Тому ми переходимо до правого дочірнього вузла кореневого вузла і порівнюємо `puppy` зі словом `style`. В даному випадку цільовий елемент передує елементу вузла, тому треба рухатися за зв'язком до лівого вузла. Тут знаходиться слово `plenum`, яке передує слову `puppy`. Тепер необхідно йти правою гілкою для цього вузла, але вона пуста. Таким чином, три операції порівняння дозволили встановити, що слово `puppy` в дереві відсутнє.

Двійкове дерево пошуку поєднує переваги зв'язної структури з ефективністю двійкового пошуку. З точки зору програмування реалізація дерева

є більш трудомістким процесом, ніж створення зв'язного списку. Далі ми побудуємо двійкове дерево для фінального проекту **ADT**.

3.7.1. Створення абстрактного типу даних для двійкового дерева

Як і раніше, почнемо з загального визначення двійкового дерева. Це конкретне визначення передбачає, що дерево не містить дубльованих елементів. Багато його операцій збігаються з операціями зі списками. Різниця полягає в ієрархічній організації даних.

Неформальне визначення цього типу **ADT** виглядає наступним чином:

Ім'я типу:	Двійкове дерево пошуку
Властивості типу:	Двійкове дерево є або пустим набором вузлів (пусте дерево), або набором вузлів, один з яких означає корінь. Кожен вузол має в точності два дерева, що виходять з нього. Вони мають назву лівого піддерева та правого піддерева. Кожне піддерево саме є двійковим деревом, з можливістю бути пустим деревом. Двійкове дерево пошуку — це упорядковане двійкове дерево, де кожен вузол містить елемент, всередині якого усі елементи в лівому піддереві передують кореневому елементу, а кореневий елемент передує усім елементам у правому піддереві.
Операції типу:	Ініціалізація дерева пустим значенням З'ясування, чи не є дерево пустим З'ясування, чи не є дерево заповненим Визначення кількості елементів у дереві Додавання елемента до дерева Видалення елемента з дерева Пошук елемента в дереві Перегляд кожного елемента в дереві Спустошення дерева

5

3.7.2. Інтерфейс двійкового дерева пошуку

В принципі, двійкове дерево пошуку можна реалізувати різними способами. Його можна реалізувати навіть у вигляді масиву, відповідним чином маніпулюючи індексами масиву. Найбільш прямолінійний спосіб

реалізації двійкового дерева пошуку передбачає використання вузлів, що виділяються динамічно та зв'язані між собою за допомогою вказівників. Розглянемо декілька визначень, що стосуються двійкового дерева:

```
typedef SOMETHING Item;   // SOMETHING – це будь-який базовий
                          // тип, або структура
typedef struct tnode
{
    Item item;
struct tnode *left;      // вказівник на ліву гілку
struct tnode *right;     // вказівник на праву гілку
} Tnode;

typedef struct tree
{
    Tnode *root;
    int size;
} Tree;
```

Кожен вузол містить елемент, вказівник на лівий дочірній вузол і вказівник на правий дочірній вузол. Структуру **Tree** можна було б визначити як тип вказівника на **Tnode**, оскільки для доступу до усього дерева достатньо знати тільки місце розташування кореневого вузла. Однак застосування структури з членом **size** спрощує відстеження розміру дерева.

Розробимо програму для ведення реєстру домашніх тварин у клубі Nerfville Pet Club причому кожен елемент буде містити кличку тварини та його вид. Враховуючи це, визначимо інтерфейс, що буде мати наступний вигляд (файл **tree.h**):

```
//=====================================================================
// tree.h – двійкове дерево пошуку
// Дубльовані елементи в цьому дереві використовувати не
дозволено
//=====================================================================
#ifndef _TREE_H_
#define _TREE_H_
#include <stdbool.h>
#define SLEN 20
// перевизначення типу Item
typedef struct item
{
    char petname[SLEN];
    char petkind[SLEN];
```

```
} Item;
 #define MAXITEMS 10
 typedef struct tnode
{
    Item item;
    struct tnode *left;    // вказівник на ліву гілку
    struct tnode *right;       // вказівник на праву гілку
} Tnode;
 typedef struct tree
{
    Tnode *root;           // вказівник на корінь дерева
    int size;              // кількість елементів у дереві
} Tree;

//=============================================================
// Прототипи функцій
//=============================================================
// Операція:     ініціалізація дерева пустим вмістом
// Передумова:   ptree вказує на дерево
// Постумова:    дерево встановлено в пустий стан
//=============================================================
void InitializeTree(Tree *ptree);

//=============================================================
// Операція:     з'ясування, чи є дерево пустим
// Передумова:   ptree вказує на дерево
// Постумова:    функція повертає true, якщо дерево є пустим,
//               і false — у протилежному випадку
//=============================================================
bool TreeIsEmpty(const Tree *ptree);

//=============================================================
// Операція:     з'ясування, чи є дерево повним
// Передумова:   ptree вказує на дерево
// Постумова: функція повертає true, якщо дерево є заповненим
//               і false — у протилежному випадку
//=============================================================
bool TreeIsFull(const Tree *ptree);

//=============================================================
// Операція:     визначення кількості елементів у дереві
// Передумова:   ptree вказує на дерево
// Постумова:    функція повертає кількість елементів у дереві
//=============================================================
int TreeItemCount(const Tree *ptree);
//=============================================================
// Операція:     додавання елемента до дерева
// Передумови:   pi - адреса елемента, що додається
//               ptree вказує на ініціалізоване дерево
```

```
// Постумови:    якщо можливо, функція додає елемент
//               до дерева і повертає true;
//               у протилежному випадку вона повертає false
//=============================================================
bool AddItem(const Item *pi, Tree *ptree);

//=============================================================
// Операція:     пошук елемента в дереві
// Передумови:   pi вказує на елемент
//               ptree вказує на ініціалізоване дерево
// Постумови:    функція повертає true, якщо елемент присутній
//               в дереві, і false - у протилежному випадку
//=============================================================
bool InTree(const Item *pi, const Tree *ptree);

//=============================================================
// Операція:     видалення елемента з дерева
// Передумови:   pi - адреса елемента, що видаляється
//               ptree вказує на ініціалізоване дерево
// Постумови:    якщо можливо, функція видаляє елемент з дерева
//              і повертає true; у протилежному випадку функція
//               повертає false
//=============================================================
bool DeleteItem(const Item *pi, Tree *ptree);

//=============================================================
// Операція:  застосування вказаної функції до кожного елемента
//               в дереві
// Передумови:   ptree вказує на дерево
//            pfun вказує на функцію, яка приймає аргумент Item
//               і не має значення, що повертається
// Постумови:    функція, що вказується за допомогою pfun,
//           виконується один раз для кожного елемента в дереві
//=============================================================
void Traverse(const Tree *ptree, void (*pfun)(Item item));

//=============================================================
// Операція:     видалення усіх елементів з дерева
// Передумова:   ptree вказує на ініціалізоване дерево
// Постумова:    дерево є пустим
//=============================================================
void DeleteAll(Tree *ptree);
#endif
```

Ми обмежили розмір дерева до **10**. Невеликий розмір полегшує тестування програми при заповненні дерева. За необхідності значення **MAXITEMS** завжди можна збільшити.

3.7.3. Реалізація двійкового дерева

Тепер приступимо до реалізації декількох функцій, що описані у файлі `tree.h`. Функції `InitializeTree()`, `EmptyTree()`, `FullTree()` і `TreeItems()` достатньо прості і працюють подібно до своїх аналогів у абстрактних типах даних списку та черги, тому ми приділимо основну увагу іншим функціям.

Додавання елемента до дерева. Спочатку треба перевірити, чи є місце для нового вузла. Оскільки двійкове дерево пошуку визначено таким чином, що не може містити дубльованих елементів, далі необхідно з'ясувати, чи не існує цей елемент у дереві. Якщо новий елемент задовольняє цим двом початковим умовам, треба створити новий вузол, скопіювати в нього елемент і встановити лівий і правий вказівники вузла в `NULL`. Це говорить про відсутність дочірніх вузлів у дочірнього вузла. Потім слід обновити елемент `size` структури `Tree` з метою відображення факту додавання нового елемента. Далі необхідно з'ясувати, в яку позицію дерева повинен бути поміщений новий вузол. Якщо дерево є пустим, кореневий вказівник необхідно встановити таким чином, щоб він посилався на новий вузол. У протилежному випадку слід переглянути дерево, щоб знайти в ньому місце для додавання вузла. Функція `AddItem()` виконує ці дії, надаючи частину роботи функціям, які поки ще не визначені: `SeekItem()`, `MakeNode()` і `AddNode()`.

```
bool AddItem(const Item *pi, Tree *ptree)
{
    Tnode *newnode;
    if(TreeIsFull(ptree))
    {
        fprintf(stderr, "Дерево переповнено\n");
        return false;            // передчасне повернення з функції
    }
    if(SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Намагання додати дубльований елемент\n");
return false;    // передчасне повернення з функції
    }
    newnode = MakeNode(pi);       // вказує на новий вузол
if(newnode == NULL)
    {
```

```
        fprintf(stderr, "Не вдалося створити вузол\n");
        return false;            // передчасне повернення з функції
    }
    // успішне створення нового вузла
ptree->size++;
if(ptree->root == NULL)       // випадок 1: дерево є пустим
ptree->root = newnode;    // новий вузол - корінь дерева
 else                          // випадок 2: дерево не є пустим
AddNode(newnode, ptree->root); // додавання вузла до дерева
return true;                  // передчасне повернення з функції }
```

Функції **SeekItem()**, **MakeNode()** і **AddNode()** не є частиною відкритого інтерфейсу для типу **Tree**. Замість цього вони являють собою статичні функції, які приховані у файлі **tree.c**. Вони мають справу з такими деталями реалізації, як вузли, вказівники та структури, що не відносяться до відкритого інтерфейсу.

Функція **MakeNode()** забезпечує динамічне виділення пам'яті та ініціалізацію вузла. Аргументом функції є вказівник на новий елемент, а її значеним, що повертається, – вказівник на новий вузол. Згадайте, що функція **malloc()** повертає нульовий вказівник, якщо вона не може виділити необхідну пам'ять. Функція **MakeNode()** ініціалізує новий вузол тільки у випадку успішного виділення пам'яті. Її код має наступний вигляд:

```
static Tnode *MakeNode(const Item *pi)
{
    Tnode *newnode;
    newnode = (Tnode *) malloc(sizeof(Tnode));
    if(newnode != NULL)
    {
        newnode->item = *pi;
        newnode->left = NULL;
        newnode->right = NULL;
    }
    return
newnode;

}
```

Функція **AddNode()** є другою за складністю в пакеті двійкового дерева пошуку. Вона повинна визначити, куди повинен бути поміщений новий вузол, і потім додати його. Зокрема, їй необхідно порівняти новий елемент з кореневим

елементом, щоб з'ясувати, в яке піддерево повинен бути поміщений новий елемент – ліве або праве.

Якщо елемент є числом, то для виконання порівнянь можна використовувати операції '<' і '>', а якщо він є рядком, – то функцію strcmp().

В нашому випадку елемент є структурою, яка містить два рядки, тому для виконання порівнянь доведеться передбачити власні функції. Функція ToLeft(), яка буде визначена пізніше, повертає значення True, якщо новий елемент повинен бути поміщений до лівого піддерева, а функція ToRight() повертає значення True, якщо новий елемент повинен увійти до правого піддерева. Ці дві функції являють собою аналоги операцій '<' і '>'. Припустимо, що новий елемент повинен бути поміщений до лівого піддерева. Воно цілком може виявитися пустим. В такому випадку функція просто встановлює вказівник на лівий дочірній вузол таким чином, щоб він посилався на новий вузол. Якщо ліве піддерево не є пустим, тоді функція повинна порівняти новий елемент з елементом у лівому дочірньому вузлі, щоб з'ясувати, до якого піддерева дочірнього вузла повинен бути поміщений новий вузол – лівого чи правого. Цей процес повинен тривати до тих пір, поки функція не досягне пустого піддерева, в яке може бути доданий новий вузол. Один з можливих способів реалізації такого пошуку пов'язаний з рекурсією, а саме застосуванням функції AddNode() до дочірнього, а не кореневого вузла. Рекурсивна послідовність викликів функції завершується, коли ліве або праве піддерево виявляється пустим, тобто коли root->left або root->right дорівнює NULL. Майте на увазі, що root – це вказівник на верхівку поточного піддерева, тому в кожному рекурсивному виклику він вказує на нове піддерево, що розташоване на більш низькому рівні.

```
static void AddNode(Tnode *newnode, Tnode *root)
{
    if(ToLeft(&newnode->item, &root->item))
    {
      if(root->left == NULL)      // якщо піддерево є пустим,
root->left = newnode;  // додаємо до нього вузол
else                     // інакше обробляємо піддерево
            AddNode(newnode, root->left);
    }
else
```

```
    {
        if(ToRight(&newnode->item, &root->item))
        {
            if(root->right == NULL)
root->right = newnode;              else
                AddNode(newnode, root->right);
        }
        else     // дублікати не допускаються
        {
            fprintf(stderr, "Помилка місця розташування "
                    "в AddNode()\n");
exit(1);
        }
    }
}
```

Функції `ToLeft()` і `ToRight()` залежать від сутності типу `Item`. Члени клубу `Nerfville Pet Club` будуть впорядковані в алфавітному порядку за кличками. Якщо дві тварини мають однакові клички, вони повинні бути впорядковані за видом. Якщо їх вид також збігається, то два елементи є дублікатами, що в базовому дереві пошуку не допускається. Згадайте, що функція `strcmp()` зі стандартної бібліотеки `C` повертає від'ємне число, якщо рядок, представлений її першим аргументом, передує рядку в другому аргументі, нуль, якщо обидва рядки збігаються, і додатне число, якщо перший рядок йде за другим. Функція `ToRight()` містить аналогічний код. Використання цих двох функцій замість виконання порівнянь безпосередньо в `AddNode()` спрощує адаптацію коду до нових вимог. Замість того щоб переписувати функцію `AddNode()`, коли необхідна друга форма порівняння, достатньо модифікувати функції `ToLeft()` і `ToRight()`.

```
static bool ToLeft(const Item *i1, const Item *i2)
{
    int comp1;
    if((comp1 = strcmp(i1->petname, i2->petname)) < 0)
return true;
else
   if(comp1 == 0 && strcmp(i1->petkind, i2->petkind) < 0)
return true;
else
            return
false;
}
```

Пошук елемента. В трьох функціях інтерфейсу – `AddItem()`, `InTree()` і `DeleteItem()` – передбачено пошук в дереві конкретного елемента. В реалізації, що розглядається, для цього використовується функція `SeekItem()`. З функцією `DeleteItem()` пов'язана додаткова вимога: вона повинна знати батьківський вузол елемента, що видаляється, щоб дочірній вказівник батьківського вузла можна було обновити, коли видаляється дочірній елемент. Таким чином, функція `SeekItem()` спроектована таким чином, щоб повертати структуру, яка містить два вказівники: один вказує на вузол, який містить необхідний елемент (`NULL`, якщо елемент не знайдено), а другий вказує на батьківський вузол (`NULL`, якщо даний вузол є кореневим і не має батьківського вузла). Тип структури визначений наступним чином:

```
typedef struct pair
{
    Tnode *parent;
    Tnode *child;
} Pair;
```

Функцію `SeekItem()` можна реалізувати рекурсивно. Однак щоб ознайомитися з різними прийомами програмування, для спадного обходу дерева застосовуємо цикл `while`. Подібно до функції `AddNode()`, для навігації по дереву функція `SeekItem()` використовує функції `ToLeft()` і `ToRight()`. Спочатку `SeekItem()` встановлює вказівник `look.child` так, щоб він посилався на корінь дерева, а потім, по мірі проходу по шляху до можливого місцезнаходження елемента, переустановлює цей вказівник на подальші піддерева. Одночасно вказівник `look.parent` встановлюється для посилання на подальші батьківські вузли. Якщо потрібного елемента не знайдено, значенням вказівника `look.child` буде `NULL`. Якщо необхідний елемент знаходиться в кореневому вузлі, `look.parent` дорівнює `NULL`, оскільки кореневий вузол не має батьківського вузла. Нижче наведено код функції `SeekItem()`:

```
static Pair SeekItem(const Item *pi, const Tree *ptree)
{
Pair look;
```

```
look.parent = NULL;
look.child = ptree->root;
if(look.child == NULL)
        return look;       // передчасне повернення з функції
while(look.child != NULL)
    {
        if(ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.chlld->left;
        }
        else if(ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
 else          // якщо елемент не розташований ні ліворуч,
      break; // ні праворуч, він повинен бути таким самим
             // look.child - це адреса вузла,
             // що містить елемент
    }
return look;
}
```

Зверніть увагу, що оскільки функція **SeekItem()** повертає структуру, її можна застосовувати з операцією членства в структурі. Наприклад, у функції **AddItem()** використовується наступний код:

```
if(SeekItem(pi, ptree).child != NULL)
```

За наявності функції **SeekItem()**, написання коду функції **InTree()** відкритого інтерфейсу є нескладним завданням. Код буде мати наступний вигляд:

```
bool InTree(const Item *pi, const Tree *ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false :
true; }
```

Видалення елемента двійкового дерева. Найбільш трудомістка задача, оскільки необхідно заново поєднати піддерева, що залишилися після видалення, для формування допустимого дерева. Перш ніж приступити до програмування цієї задачі, має сенс візуально представити дії, які повинні бути при цьому

виконані. На рис. 3.39 проілюстровано найпростіший випадок видалення вузла, де *вузол, що видаляється, не має дочірніх вузлів*.

Такий вузол називається **листом**. В цьому випадку знадобиться тільки переустановити вказівник у батьківському вузлі в `NULL` і за допомогою функції `free()` звільнити пам'ять, яку займає цей вузол.

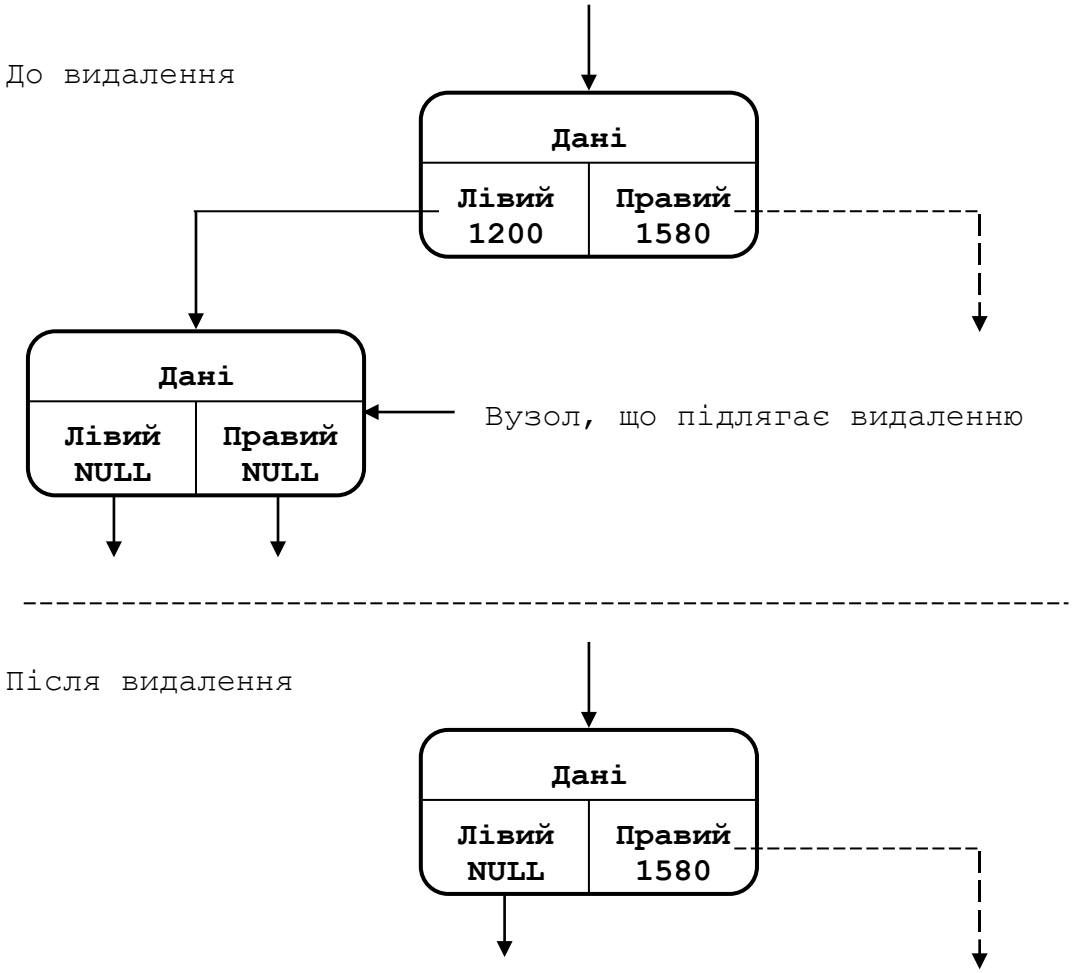


Рисунок 3.39 – Видалення листа, що не має дочірніх вузлів

Наступною за складністю задачею є **видалення вузла з одним дочірнім вузлом**. Видалення вузла призводить до відокремлення дочірнього піддерева від іншої частини дерева. Для виправлення такої ситуації адреса дочірнього піддерева повинна бути збережена в батьківському вузлі в позиції, яка раніше була зайнята адресою видаленого вузла (рис. 3.40).

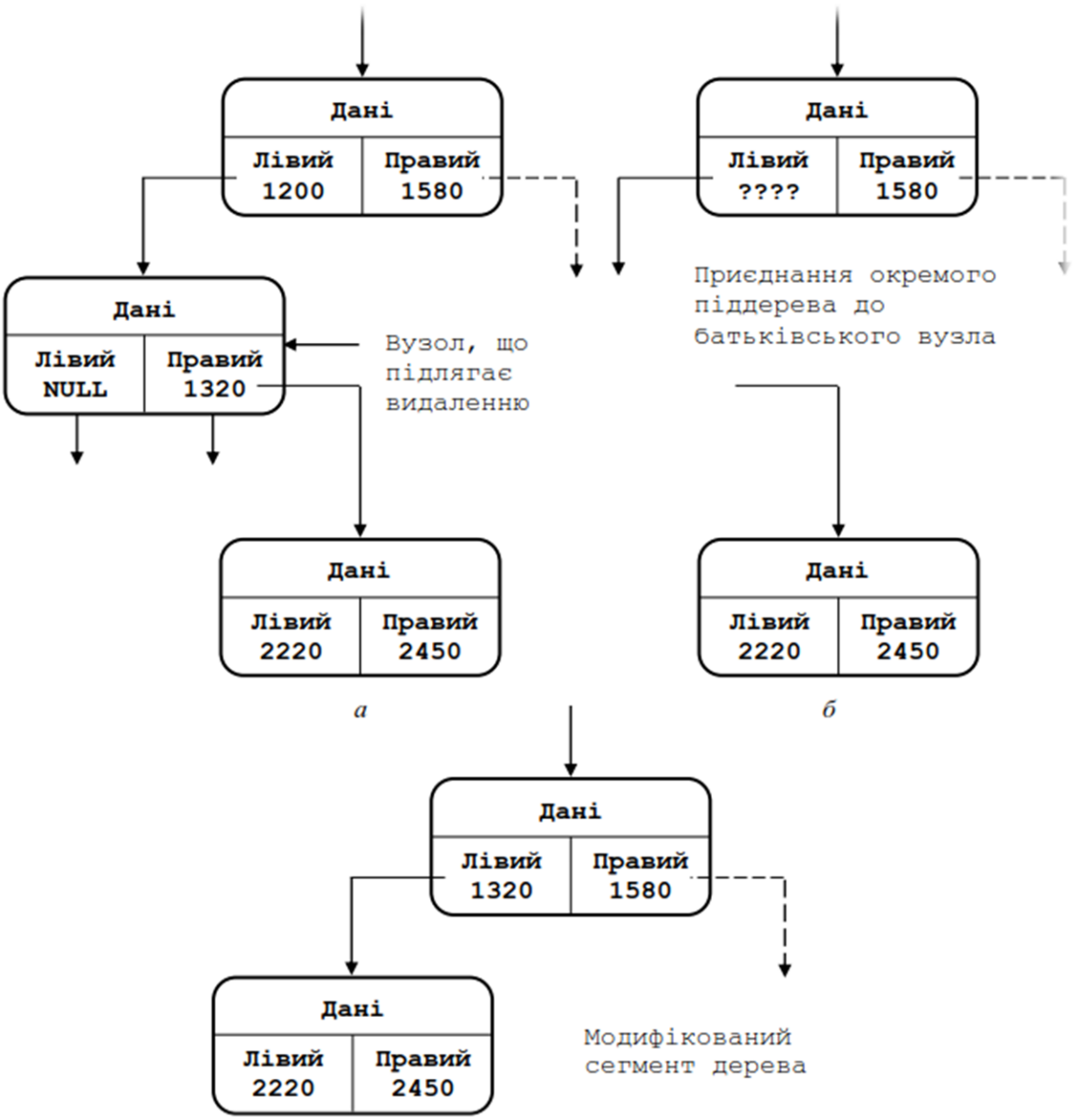


Рисунок 3.40 – Видалення вузла з одним дочірнім вузлом

Останній випадок пов’язаний з **видаленням вузла, що має два піддерева**. Одне піддерево, скажімо, ліве, може бути приєднано до того вузла, до якого спочатку був приєднаний видалений вузол. Але куди помістити піддерево, що залишилося? Згадаємо базовий принцип формування деревоподібної структури. Кожен елемент в лівому піддереві передує елементу в батьківському вузлі, а кожен елемент у правому піддереві йде за елементом у батьківському вузлі. Це

означає, що кожен елемент у правому піддереві розташований у структурі далі будь-якого елемента з лівого піддерева.

Крім того, оскільки праве піддерево раніше було частиною піддерева, що починається з видаленого вузла, кожен елемент у правому піддереві передує батьківському вузлу видаленого вузла. Уявіть собі спуск по дереву в пошуках позиції для розміщення початку правого піддерева. Він передує батьківському вузлу, тому далі необхідно йти вниз по лівому піддереву. Однак початок піддерева повинен бути розташований після всіх елементів у лівому піддереві, тому необхідно йти правою гілкою лівого піддерева та з'ясувати, чи є в ній місце для нового вузла. Якщо ні, треба буде продовжувати спуск правою гілкою лівого піддерева до тих пір, поки вільне місце не буде знайдено. Цей підхід продемонстрований на рис. 3.41.

Тепер можна перейти до планування необхідних функцій, розділивши роботу на дві задачі. Перша задача передбачає зв'язування конкретного елемента з вузлом, що підлягає видаленню, а друга полягає в безпосередньому видаленні вузла. Слід відзначити, що в усіх випадках буде потрібна модифікація вказівника в батьківському вузлі, а це призводить до двох важливих наслідків:

- програма повинна ідентифікувати батьківський вузол вузла, що видаляється;
- для зміни вказівника код повинен передавати функції видалення адреси цього вказівника.

До першого моменту ми повернемося дещо пізніше, а поки проаналізуємо другий момент. Вказівник, який треба змінювати, має тип `Tnode *`, тобто є вказівником на `Tnode`. Оскільки аргумент функції – це адреса цього вказівника, типом аргументу буде `Tnode **`, або вказівник на вказівник на `Tnode`. Припускаючи, що потрібна адреса є доступною, функцію видалення вузла `DeleteNode()` можна реалізувати наступним чином:

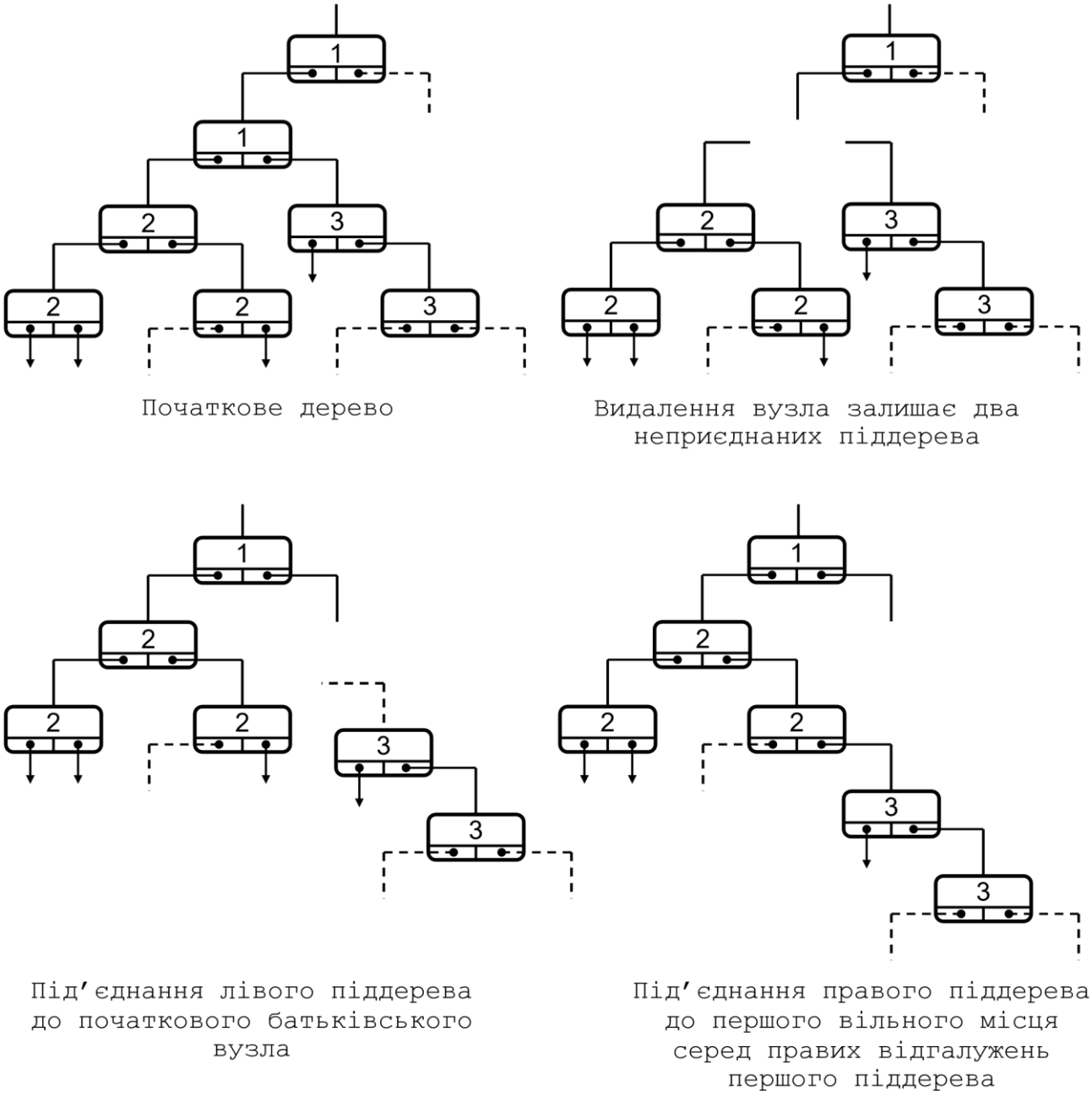


Рисунок 3.41 – Видалення вузла з двома дочірніми вузлами

```
static void DeleteNode(Tnode **ptr)
{// адреса батьківського елемента, що вказує на цільовий вузол
Tnode *temp;
if((*ptr)->left == NULL)
     {
          temp = *ptr;
          *ptr = (*ptr)->right;
free(temp);
     }
     else if((*ptr)->right == NULL)
     {
temp = *ptr;
```

```
*ptr = (*ptr)->left;
free(temp);
    }
    else  // вузол, що видаляється, має два дочірніх вузла
    {
        // з'ясування місця приєднання правого піддерева
for(temp = (*ptr)->left; temp->right != NULL;
                                    temp = temp->right)
continue;
temp->right = (*ptr)->right;
temp = *ptr;
*ptr =(*ptr)->left;
free(temp);
    }
}
```

В цій функції явно обробляються три випадки: вузол без лівого дочірнього вузла, вузол без правого дочірнього вузла та вузол з двома дочірніми вузлами.

Вузол без дочірніх вузлів можна вважати особливим випадком вузла без лівого дочірнього вузла. Якщо вузол не має лівого дочірнього вузла, код присвоює адресу правого дочірнього вузла вказівнику на батьківський вузол. Але, якщо вузол не має також і правого вузла, то значенням цього вказівника буде **NULL**, яке є потрібним значенням для випадку вузла без дочірніх вузлів.

Зверніть увагу, що для відстеження адреси вузла, що видаляється, в коді застосовується тимчасовий вказівник. В результаті переустановлення батьківського вказівника (***ptr**) програма втратила б інформацію про місце розташування видаленого вузла, а ця інформація потрібна для функції **free()**. Таким чином, початкове значення ***ptr** зберігається у змінній **temp**, а потім використовується для звільнення пам'яті, яку займав видалений вузол.

В коді для випадку вузла з двома дочірніми вузлами спочатку застосовується вказівник **temp** в циклі **for** для пошуку вільного місця в правій частині лівого піддерева. Після його знаходження до нього приєднується праве піддерево. Потім знову використовується вказівник **temp** для відстеження місця розташування видаленого вузла. І, нарешті, ліве піддерево приєднується до батьківського вузла, після чого вузол, на який вказує **temp**, звільняється.

Зверніть увагу, що оскільки `ptr` має тип `Tnode **`, то `*ptr` відноситься до типу `Tnode *` і робить його таким, що збігається за типом з вказівником `temp`.

3.7.4. Видалення вузла

Частина задачі, що залишилася невирішеною, стосується зв'язку вузла з певним елементом. Щоб зробити це, можна скористатися функцією `SeekItem()`. Згадайте, що вона повертає структуру, що містить вказівник на батьківський вузол і вказівник на вузол, в якому знаходиться елемент. Відповідно, вказівник батьківського вузла можна застосовувати для отримання потрібної адреси та його передачі до функції `DeleteNode()`. Такий план реалізований у функції `DeleteItem()`, код якої має наступний вигляд:

```
bool DeleteItem(const Item *pi, Tree *ptree)
{
    Pair look;
    look = SeekItem(pi, ptree);
if(look.child == NULL)
return false;
    if(look.parent == NULL)   // видалення кореневого елемента
        DeleteNode(&ptree->root);
else
    {
      if(look.parent->left == look.child)
DeleteNode(&look.parent->left);
else
            DeleteNode(&look.parent->right);
    }
    ptree->size--;
return true;

}
```

Значення, що повертається функцією `SeekItem()` присвоюється змінній `look` типу структури. Якщо значення `look.child` дорівнює `NULL`, пошук елемента не буде мати успіху, і функція `DeleteItem()` завершить роботу, повернувши `false`. Якщо елемент `Item` знайдено, функція обробляє три випадки. Перш за все, значення `NULL` змінної `look.parent` говорить про те, що елемент був знайдений в кореневому вузлі. В такому випадку батьківський вузол, який треба було б оновити, відсутній. Замість цього треба оновити

вказівник `root` в структурі `Tree`. Відповідно, функція передає адресу цього вказівника до функції `DeleteNode()`. В протилежному випадку код з'ясовує, в лівому чи правому дочірньому вузлі батьківського вузла розташований вузол, що видаляється, і потім передає адресу відповідного вказівника.

Зверніть увагу, що функція `DeleteItem()` відкритого інтерфейсу оперує поняттями, близькими до кінцевого користувача (елементами і деревами), а прихована функція `DeleteNode()` виконує звичайні дії з вказівниками.

3.7.5. Обхід дерева

Обхід дерева є більш складною задачею, ніж обхід зв'язного списку, оскільки кожен вузол має дві гілки, якими слід просуватися. Така природа розгалуження робить рекурсію природним методом рішення цієї задачі. В кожному вузлі необхідно виконати наступні дії:

- обробити елемент у вузлі;
- обробити ліве піддерево (рекурсивний виклик);
- обробити праве піддерево (рекурсивний виклик);

Даний процес можна поділити на дві функції: `Traverse()` і `InOrder()`. Зверніть увагу, що функція `InOrder()` обробляє ліве піддерево, потім елемент і після цього праве піддерево. Така організація обробки призводить до обходу дерева в алфавітному порядку. При бажанні можете самостійно побачити, що відбувається у випадку використання інших порядків обробки, скажімо, «елемент, ліве піддерево, праве піддерево» і «ліве піддерево, праве піддерево, елемент».

```
void Traverse(const Tree *ptree, void (*pfun)(Item item))
{
    if(ptree != NULL)
        InOrder(ptree->root, pfun);
}
static void InOrder(const Tnode *root, void (*pfun)(Item item))
{
    if(root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
```

```
        InOrder(root->right, pfun);
    }
}
```

3.7.6. Спустошення дерева

По суті спустошення дерева являє собою такий самий процес, що і його обхід. Іншими словами, коду необхідно відвідати кожен вузол і застосувати до нього функцію **free()**. Код повинен також переустановити члени структури **Tree**, щоб відобразити пусте дерево. Функція **DeleteAll()** потурбується про структуру **Tree** і надасть задачу звільнення пам'яті функції **DeleteAllNodes()**. Остання функція аналогічна до функції **InOrder()**. Вона зберігає значення вказівника `root->right`, щоб він залишався доступним після звільнення кореня. Код згаданих двох функцій має наступний вигляд:

```
void DeleteAll(Tree *ptree)
{
if(ptree != NULL)
    DeleteAllNodes(ptree->root);
ptree->root = NULL;
ptree->size = 0;
}
static void DeleteAllNodes(Tnode *root)
{
    Tnode *pright;
if(root != NULL)
    {
     pright = root->right;
     DeleteAllNodes(root->left);
     free(root);
     DeleteAllNodes(pright);
    }
}
```

Завершений пакет. Повний код файлу **tree.c** наведений нижче. Разом з файлами **tree.h** і **tree.c** формується програмний пакет для деревовидного представлення.

```
//=============================================================
// tree.c – функції підтримки дерева
//=============================================================
#include <string.h> #include <stdio.h>
#include <stdlib.h>
```

```
#include "tree.h"

// локальний тип даних typedef struct pair
{
    Tnode *parent;
    Tnode *child;
} Pair;
// прототипи локальних функцій
static Tnode *MakeNode(const Item *pi);
static bool ToLeft(const Item *i1, const Item *i2);
static bool ToRight(const Item *i1, const Item *i2);
static void AddNode(Tnode *newnode, Tnode *root);
static void InOrder(const Tnode *root, void (*pfun)(Item item));
static Pair SeekItem(const Item *pi, const Tree *ptree);
static void DeleteNode(Tnode **ptr);
static void DeleteAllNodes(Tnode *ptr);

// визначення функцій void InitializeTree(Tree *ptree)
{
    ptree->root = NULL;
    ptree->size = 0;
}
bool TreeIsEmpty(const Tree *ptree)
{
    if(ptree->root == NULL)
      return true;
    else
     return false;
}
 bool TreeIsFull(const Tree *ptree)
{
    if(ptree->size == MAXITEMS)
      return true;
else
        return false;
}
 int TreeItemCount(const Tree *ptree)
{
    return ptree->size;
}
bool AddItem(const Item *pi, Tree *ptree)
{
    Tnode *newnode;
    if(TreeIsFull(ptree))
    {
        fprintf(stderr, "Дерево переповнено\n");
 return false;          // передчасне повернення
    }
```

```
    if(SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Спроба додавання дублюючого елемента\n");
        return false;          // передчасне повернення
    }
    newnode = MakeNode(pi);    // вказує на новий вузол
if(newnode == NULL)
    {
        fprintf(stderr, "Не вдалося створити вузол\n");
return false;          // передчасне повернення
    }
    // успішне створення нового вузла
ptree->size++;
    if(ptree->root == NULL)      // випадок 1: дерево є пустим
ptree->root = newnode;  // новий вузол - корінь дерева
else                           // випадок 2: дерево не є пустим
AddNode(newnode, ptree->root); // додавання вузла до дерева
return true;                   // повернення у разі успіху }

bool InTree(const Item *pi, const Tree *ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true; }

bool DeleteItem(const Item *pi, Tree *ptree)
{
    Pair look;
    look = SeekItem(pi, ptree);      if(look.child == NULL)
return false;
    if(look.parent == NULL)    // видалення кореневого елемента
        DeleteNode(&ptree->root);      else
    {
        if(look.parent->left == look.child)
DeleteNode(&look.parent->left);          else
            DeleteNode(&look.parent->right);
    }
    ptree->size--;      return true;
}

void Traverse(const Tree *ptree, void (*pfun)(Item item))
{
    if(ptree != NULL)
        InOrder(ptree->root, pfun);
}

void DeleteAll(Tree *ptree)
{
    if(ptree != NULL)
```

```
        DeleteAllNodes(ptree->root);      ptree->root = NULL;
ptree->size = 0;
}
// локальні функції
static void InOrder(const Tnode *root, void (*pfun)(Item item))
{
    if(root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}

static void DeleteAllNodes(Tnode *root)
{
    Tnode *pright;      if(root != NULL)
    {
        pright = root->right;
 DeleteAllNodes(root->left);
free(root);
        DeleteAllNodes(pright);
    } }

static void AddNode(Tnode *newnode, Tnode *root)
{
    if(ToLeft(&newnode->item, &root->item))
    {
        if(root->left == NULL)      // піддерево є пустим,
root->left = newnode;  // додаємо вузол
        else                        // інакше обробляємо піддерево
            AddNode(newnode, root->left);
    }      else
    {
        if(ToRight(&newnode->item, &root->item))
        {
            if(root->right == NULL)
 root->right = newnode;              else
                AddNode(newnode, root->right);
        }
        else        // дублікати не дозволяються
        {
         fprintf(stderr, "Помилка місця розташування "
                    "в AddNode()\n");
         exit(1);
        }
    }
}
```

```
static bool ToLeft(const Item *i1, const Item *i2)
{
    int comp1;
    if((comp1 = strcmp(i1->petname, i2->petname)) < 0)
return true;     else
    {
     if(comp1 == 0 && strcmp(i1->petkind, i2->petkind) < 0)
return true;
    else
      return false;
    }
}

static bool ToRight(const Item *i1, const Item *i2)
{
    int comp1;
    if((comp1 = strcmp(i1->petname, i2->petname)) > 0)
return true;
    else
    {
        if(comp1 == 0 && strcmp(i1->petkind, i2->petkind) > 0)
return true;
        else
          return false;
    }
}

static Tnode *MakeNode(const Item *pi)
{
    Tnode *newnode;
    newnode = (Tnode *) malloc(sizeof(Tnode));
 if(newnode != NULL)
    {
        newnode->item = *pi;
        newnode->left = NULL;
        newnode->right = NULL;
    }
    return newnode;
}

static Pair SeekItem(const Item *pi, const Tree *ptree)
{
    Pair look;
look.parent = NULL;
look.child = ptree->root;
    if(look.child == NULL)
```

```
        return look;          // передчасне повернення
while(look.child != NULL)
    {
        if(ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }         else
        {
           if(ToRight(pi, &(look.child->item)))
           {
               look.parent = look.child;
               look.child = look.child->right;
           }
           else         // якщо елемент не розміщений ні ліворуч,
break;  // ні праворуч, він повинен бути таким самим
                        // look.child - це адреса вузла, що
                        // містить елемент
        }     }
    return look;        // повернення у разі успіху }
 static void DeleteNode(Tnode **ptr)
// адреса батьківського елемента, що вказує на цільовий вузол
{
    Tnode *temp;     if((*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
   else
    {
        if((*ptr)->right == NULL)
        {
            temp = *ptr;
            *ptr = (*ptr)->left;
         free(temp);
        }
        else        // вузол, що видаляється, має два дочірніх вузла
        {
            // з'ясування місця приєднання правого піддерева
for(temp = (*ptr)->left; temp->right != NULL;
temp = temp->right)                    continue;
            temp->right = (*ptr)->right;
temp = *ptr;
*ptr = (*ptr)->left;
free(temp);
        }
    }
}
```

3.7.7. Тестування пакету для деревовидного представлення

Тепер, коли реалізації інтерфейсу та функцій створені, давайте застосуємо їх. В наступній програмі використовується меню з пунктами, що призначені для додавання домашніх тварин до реєстру членів клубу, виводу списку членів клубу, виводу кількості членів, перевірки членства і виходу з програми. Коротка функція

main() зосереджена на основній схемі програми. Більшу частину роботи виконують додаткові функції, які підтримують функцію **main()**.

```
//=============================================================
// main.c – використання двійкового дерева пошуку
//=============================================================
#include <stdio.h>
#include <windows.h>
#include <string.h>
#include <ctype.h>
#include "tree.h"

char  menu(void);
void  addPet(Tree *pt);
void  dropPet(Tree *pt);
void  showPets(const Tree *pt);
void  findPet(const Tree *pt);
void  printItem(Item item);
void upperCase(char *str);
char *s_gets(char *st, int n);

int main(void)
{
Tree pets;
char choice;
    SetConsoleCP(1251);
     SetConsoleOutputCP(1251);
     InitializeTree(&pets);
     puts("Програма членства у клубі Nerfville Pet Club");
     printf("=========================================\n");
while((choice = menu()) != '6')
     {
          switch(choice)
          {
               case '1': addPet(&pets);
break;
```

```
            case '2':
printf("================================\n");
printf("Тварина             Вид           \n");
printf("================================\n");
showPets(&pets);
printf("================================\n");
break;
case '3': findPet(&pets);
break;
case '4': printf("================================\n");
printf("Кількість тварин у клубі: %d\n",
TreeItemCount(&pets));
printf("================================\n");
break;
case '5': dropPet(&pets);
break;
default: puts("Помилка у switch");
break;
        }
    }
    DeleteAll(&pets);
puts("Програма завершена.");
return 0;
}
char menu(void)
{
int ch;
printf("============================================\n");
puts("Введіть цифру, що відповідає вашому вибору:");
printf("============================================\n");
printf(" 1 - додати тварину                         \n");
printf(" 2 - вивести список тварин                  \n");
printf(" 3 - пошук тварини                          \n");
printf(" 4 - кількість тварин у клубі               \n");
printf(" 5 - видалення тварини                      \n");
printf(" 6 - вихід                                  \n");
printf("============================================\n");
while((ch = getchar()) != EOF)
    {
        while(getchar() != '\n') // відкинути решту рядка
continue;          ch = tolower(ch);
        if(strchr("123456", ch) == NULL)
            puts("Введіть цифру 1, 2, 3, 4, 5 або 6:");
else              break;
    }
    if(ch == EOF)      // символ EOF призводить до виходу з програми
ch = '6';      return ch;
}
void addPet(Tree *pt)
{
```

```
    Item temp;
if(TreeIsFull(pt))
        puts("У клубі більше немає місць!");
else
    {
printf("Введіть кличку тварини: ");
s_gets(temp.petname, SLEN);
printf("Введіть вид тварини:    ");
s_gets(temp.petkind, SLEN);
upperCase(temp.petname);
upperCase(temp.petkind);
AddItem(&temp, pt);
    }
}

void showPets(const Tree *pt)
{
    if(TreeIsEmpty(pt))
puts("Записи відсутні!");
else
        Traverse(pt, printItem);
}

void printItem(Item item)
{
    printf("%-19s%-19s\n", item.petname, item.petkind); }
void findPet(const Tree *pt)
{
    Item temp;
if(TreeIsEmpty(pt))
    {
        puts("Записи відсутні!");
        return;   // якщо дерево є пустим, вийти з функції
    }
printf("Введіть кличку тварини,\n");
printf("яку треба знайти:             ");
s_gets(temp.petname, SLEN);
printf("Введіть вид тварини:          ");
s_gets(temp.petkind, SLEN);
upperCase(temp.petname);
upperCase(temp.petkind);
printf("=========================================\n");
printf("%s на ім'я %s ", temp.petkind, temp.petname);
if(InTree(&temp, pt))  printf("є членом клубу.\n");
else
    printf("не є членом клубу.\n");
    printf("=========================================\n");
}
```

```
 void dropPet(Tree
*pt)
{
    Item temp;
    if(TreeIsEmpty(pt))
    {
        puts("Записи відсутні!");
        return;   // якщо дерево є пустим, вийти з функції
    }

printf("=============================================\n");
printf("Введіть кличку тварини,\n");     printf("яку треба
виключити з клубу: ");     s_gets(temp.petname, SLEN);
    printf("Введіть вид тварини:          ");
s_gets(temp.petkind, SLEN);
upperCase(temp.petname);
upperCase(temp.petkind);
    printf("=============================================\n");
printf("%s на ім'я %s ", temp.petkind, temp.petname);
if(DeleteItem(&temp, pt))
        printf("виключений(на) з клубу.\n");
else
        printf("не є членом клубу.\n");

printf("=============================================\n"); }
void upperCase(char *str)
{
    while(*str)
    {
        *str = toupper(*str);
str++;
} }


char *s_gets(char *st, int n)
{
    char *ret_val;
char *find;
    ret_val = fgets(st, n, stdin);
if(ret_val)
    {
        find = strchr(st, '\n');
if(find)
            *find = '\0';
else
            while(getchar() != '\n')
continue;
    }
    return ret_val;
}
```

Програма перетворює усі літери на великі, тому `JEREMY`, `Jeremy` і `jeremy` не вважаються різними кличками.

Результат роботи програми наведено на рис. 3.42 – рис. 3.44.

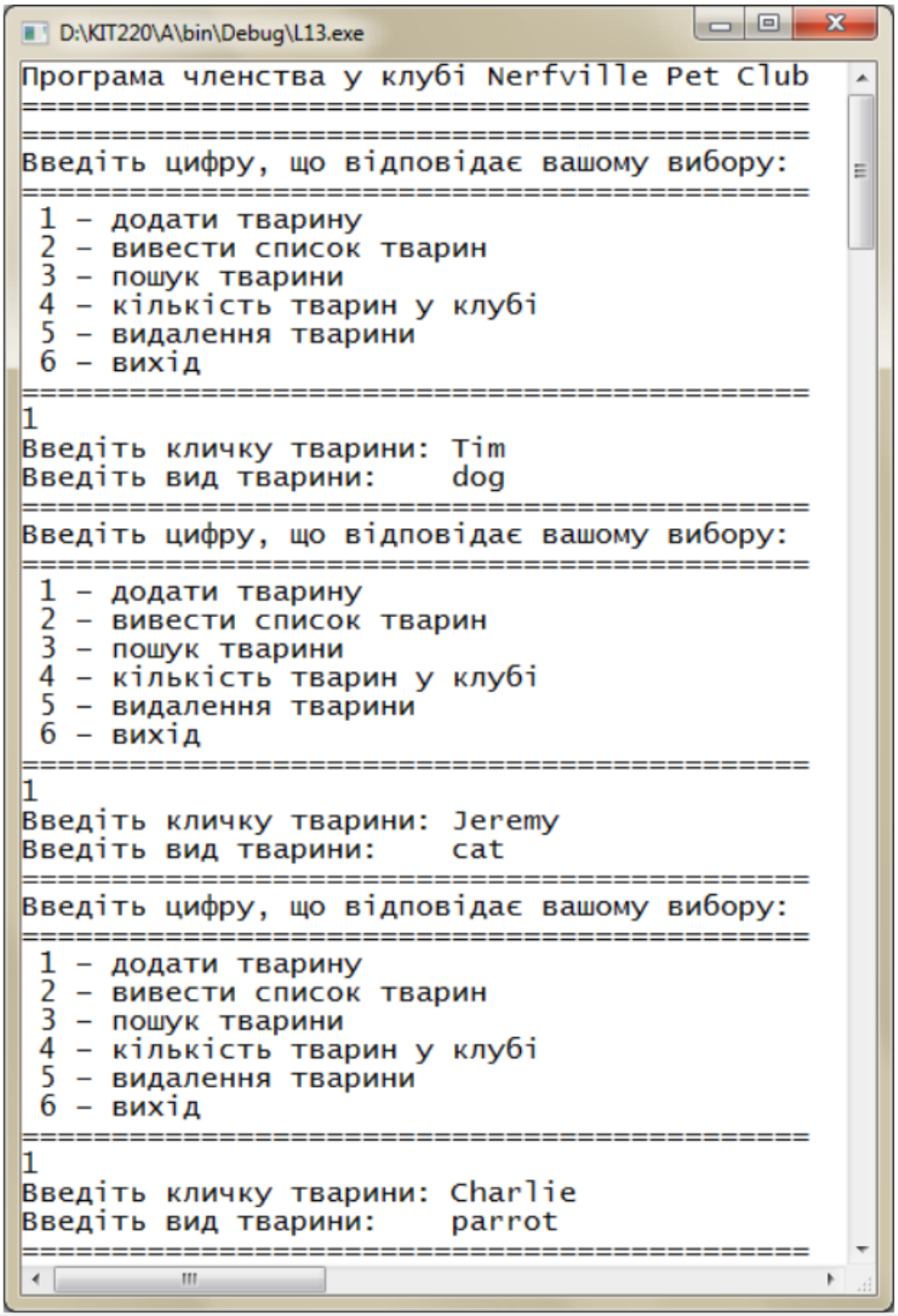


Рисунок 3.42 – Демонстрація функції додавання тварини до дерева

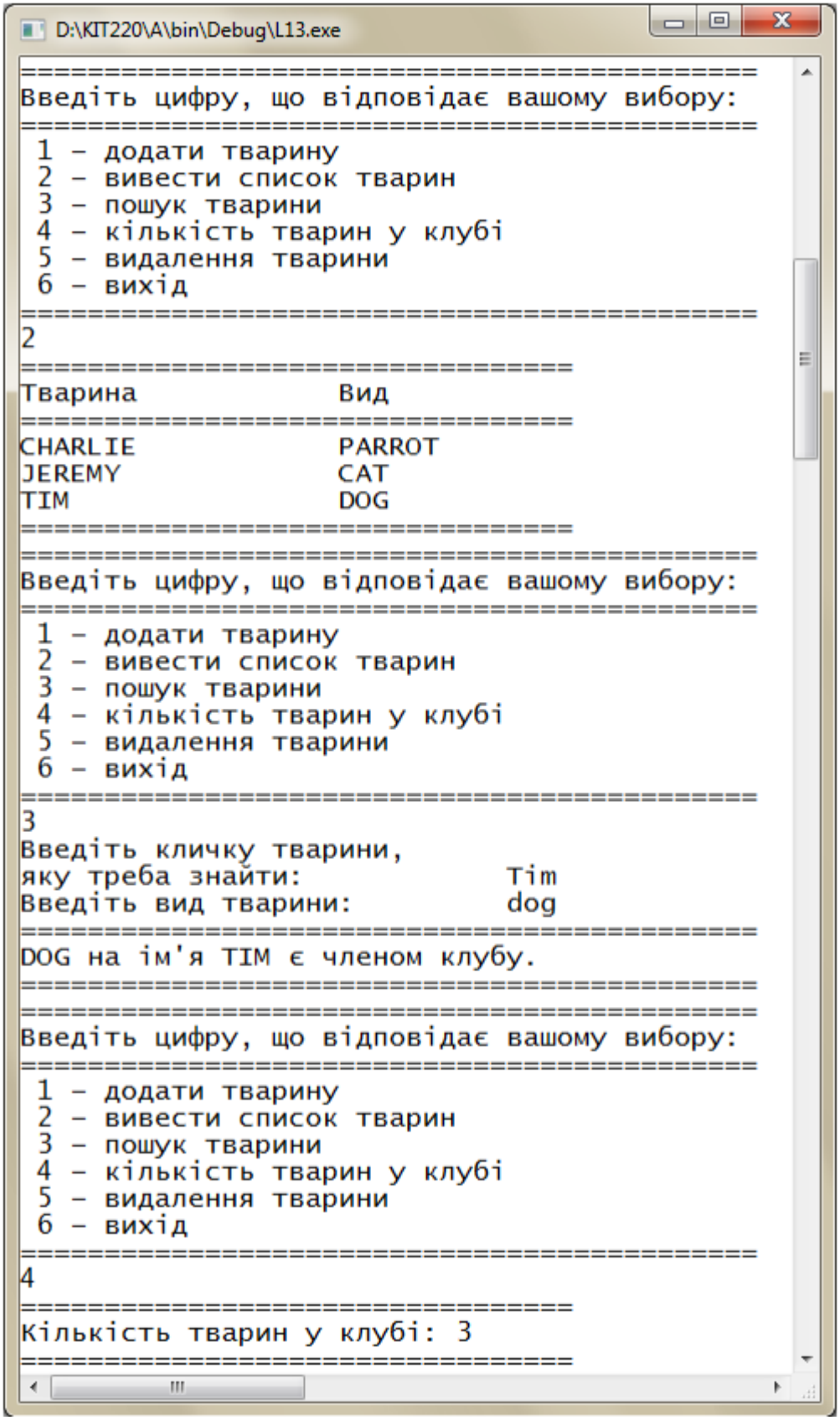


Рисунок 3.43 – Демонстрація функцій виведення списку тварин, пошуку тварини у дереві та виведення на екран кількості тварин

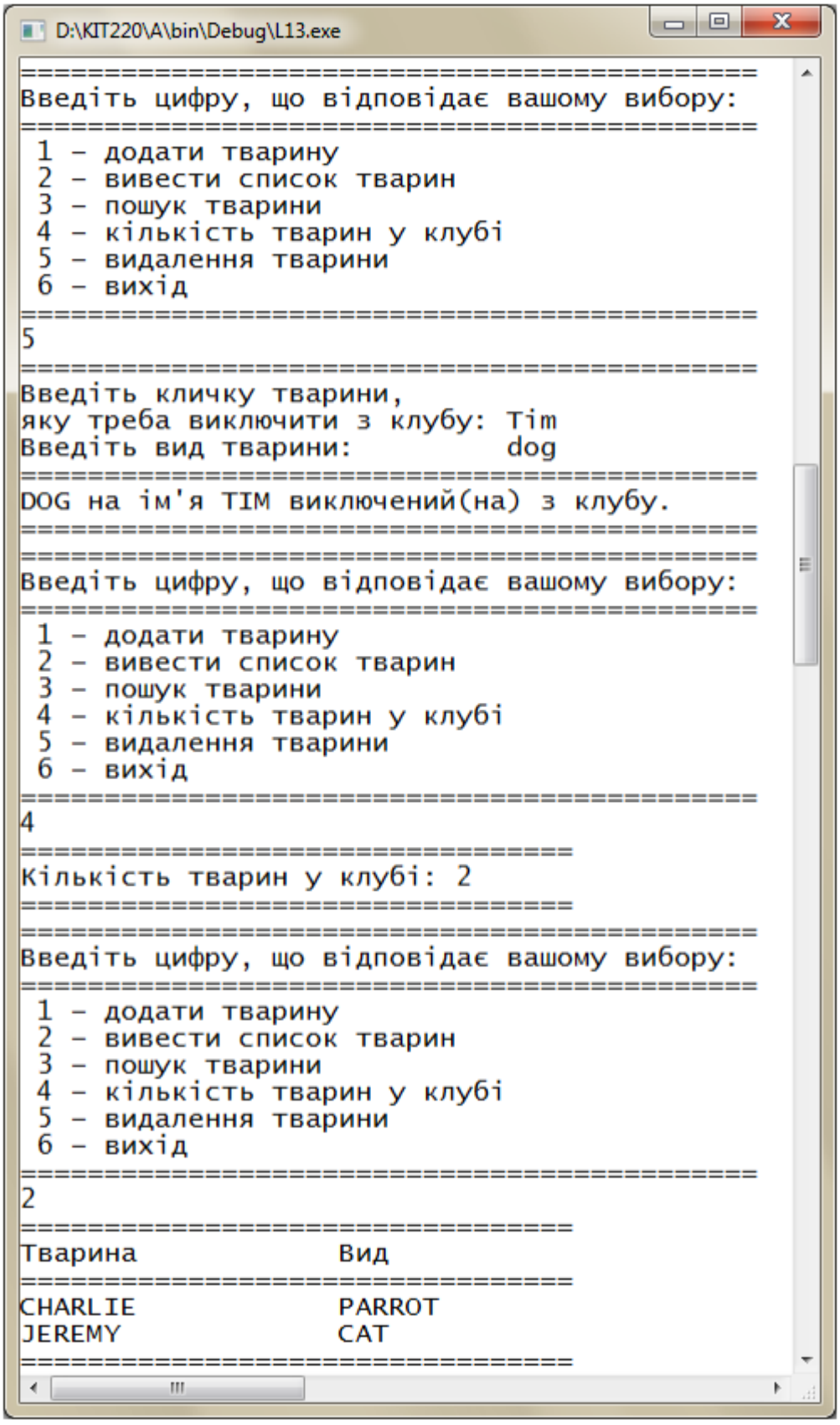


Рисунок 3.44 – Демонстрація функції видалення тварини з дерева

Контрольні запитання та завдання

1. Яку форму даних необхідно використовувати для зберігання інформації?

2. В чом полягає ідея відображеня зв’язаного списку?

3. Які дії передбачає створення списку?

4. Що ви розумієте під систематичним підходом до визначення типів? Що утворює тип?

5. Дайте визначення абстрактного типу даних (abstract data type – ADT).

6. З кількох частин складається розробка інтерфейсу ADT.

7. В чому полягає приховивавння даних? Наведіть приклад.

8. Наведіть переваги застовування ADT.

9. Що передбачає застосування абстрактних типів даних при програмуванні на мові С?

10. Дайте визначення поняття черга.

11. Чим обмежений у випадку використання списку розмір черги?

12. Які дії передбачає додавання елемента в чергу, видалення елемента на початку черги ?

13. Що перредбачає тестування черги?

14. Дайте визнгачення стеку. Які базові операції він підтримує?

15. Перелічить основні питання з реалізації стеку на мові С.

16. В якому випадку використовується динамічно зростаючий стек?

17. В чому полягає однопотокова реалізація стеку?

18. Що собою являє двозв’язний лінійний список і чим він відрізняється від однозв’язного лінійного списку?

19. Які основні операції можна робити з деком?

20. Що треба зробити для пошуку необхідного елемента в деку та за допомогою якої функції це можна зробити?

21. Яка операція використовується найпершою при роботі з деком?

22. Які способи реалізації деку вам відомі?

23. Чим відрізняються функції `listPrint()` і `listPrintRevers()` та для чого вони призначені?

24. Яке призначення має функція `swap()` та що вона приймає в якості параметрів?

Завдання для самостійного розв'язання

1. Напишіть С-програму, яка виконує створює в циклі зв'язний список, записує в нього 20 випадкових цілих чисел в діапазоні від 10 до 60 та виводить його на екран; додає число 5 після 5-го елемента списку та виводить результат на екран; додає наприкінці списку 40 випадкових чисел в діапазоні від 1 до 99 та виводить результат на екран; виводить на екран окремо голову та хвіст списку; виводить список після видалення голови та хвоста; видаляє 7-й елемент списку та виводить результат на екран

2. Напишіть С-програму, яка відсортовує зв'язний список, елементми якого є випадкові цілі числа від 34 до 89, за допомогою бульбашкового методу. Кількість елементів у списку повинна дорівнювати 11. Напрямок сортування за збільшенням. Виведіть на екран вхідний і відсортований списки.

3. Напишіть С-програму, яка відсортовує методом швидкого сортування двобічний зв'язний список, елементами якого є випадкові цілі числа від 25 до 78. Кількість елементів у списку повинна дорівнювати 25. Напрямок швидкого сортування сортування за зменшенням. Виведіть на екран вхідний і відсортований списки.

4. Напишіть С-програму, яка відсортовує методом злиття зв'язний список, елементами якого є цілі числа від 11 до 42. Кількість елементів у списку повинна дорівнювати 15. Напрямок сортування за збільшенням. Виведіть на екран вхідний і відсортований списки.

5. Напишіть С-програму, за домогою якої порівняйте час виконання сортування при використанні бульбашковго методу, метода швидкого сортування та метода злиття за зменшенням для списку, який містить 330 елементів цілого типу, що знаходяться в діапазоні від 23 до 2356. Зробіть висновки щодо найбільш продуктивного методу сортування.

6. Напишіть С-програму, яка демонструє роботу деку, реалізованого за допомогою списку, яка заповнює дек випадковими цілими числами в діапазоні від 15 до 37 за принципом: до голови деку заносяться парні числа, а до хвоста деку – непарні (кількість чисел дорівнює 11). Після занесення чергового числа необхідно виводити на екран вміст деку.

7. Напишіть С-програму, яка витягає з голови деку 5 чисел і виводить на екран вміст деку; витягає з хвоста деку 3 чисел і виводить на екран вміст деку; виводить на екран суму чисел, які на даний момент знаходяться в деку та додає до голови деку 4 випадкових чисел і виводить вміст деку на екран.

4. ПРЕПРОЦЕСОР І БІБЛІОТЕКА С

4.1. Препроцесор мови С

Мова **C** побудована на основі ключових слів, виразів, операторів, а також правил їх використання. Однак стандарт **C** не обмежується описом однієї лише мови. В ньому також визначено, що повинен робити препроцесор, встановлено, які функції формують стандартну бібліотеку **C**, і деталізовано, яким чином працюють ці функції.

Препроцесор, відповідно до своєї назви, аналізує програму до її компіляції. Дотримуючись вказаних директив, препроцесор замінює символічні скорочення в програмі сутностями, які вони представляють. За вашим запитом препроцесор може включати інші файли, і ви можете обирати, який код буде бачити компілятор. Препроцесору нічого не відомо про мову **C**. По суті він перетворює один текст на інший. Правда, такий опис не дає точного представлення про істинну користь і значимість препроцесора. Ви вже неодноразово зустрічали у попередніх програмах директиви **#define** і **#include**. Тепер можна об'єднати та розширити отримані раніше знання.

4.1.1. Перші кроки в трансляції програми

До того як передати керування препроцесору компілятор повинен провести програму через кілька етапів трансляції. Компілятор починає свою роботу з того, що встановлює відповідність символів початкового коду з вхідним набором символів. При цьому обробляються багатобайтні символи та триграфи – розширення символів, які забезпечують інтернаціональне застосування мови **C**.

У другу чергу компілятор виявляє всі входження зворотної скісної риски з подальшим символом нового рядка та видаляє їх. В результаті два фізичні рядки, такі як, наприклад,

```
printf("Це було \
чудово!\n");
```

перетворюються на один логічний рядок

```
printf("Це було чудово!\n");
```

Зверніть увагу, що в цьому контексті «символ нового рядка» означає символ, який згенерований натисненням клавіші **<Enter>** для переходу на наступний рядок у файлі початкового коду, а не символічне представлення **'\n'**.

Це необхідно для підготовки до попередньої обробки, оскільки препроцесор потребує, щоб вирази мали довжину, яка дорівнює одному логічному рядку, але один логічний рядок може поширюватися на декілька фізичних рядків.

Далі компілятор розбиває текст на послідовність препроцесорних лексем, а також на послідовність пробільних символів і коментарів. В базовій термінології лексеми являють собою групи, що відокремлюються одна від одної пробілами, табуляціями або розривами рядків. Слід зазначити, що кожен коментар замінюється одним символом пробілу. Таким чином, код `int /* це не схоже на пробіл */fox;` перетворюється на `int fox;` .

Крім того, в рамках реалізації компілятора може бути прийнято рішення замінювати кожну послідовність пробільних символів (крім символу нового рядка) одиночним пробілом. Нарешті, програма готова для етапу попередньої обробки, і препроцесор починає пошук своїх потенційних директив, що позначаються символом **'#'** на початку рядка.

4.1.2. Символічні константи: директива #define

Як і всі директиви препроцесора, директива **#define** починається з символу **'#'** на початку рядка. Стандарт **ANSI** і наступні стандарти дозволяють розміщення попереду символу **'#'** пробілів або табуляції, а також наявність пробілу між **'#'** і подальшою частиною директиви. Однак в попередніх версіях **C** зазвичай вимагалося, щоб директива починалася в крайній лівій позиції рядка, а пробіли між символом **'#'** і наступною частиною директиви не дозволялися. Директива може знаходитися в будь-якому місці файлу початкового коду, та її

визначення поширюється від цього місця до кінця файлу. В наших попередніх програмах ми інтенсивно використовували директиви для визначення символічних, або іменованих, констант. Однак, як незабаром буде показано, область застосування директив цим не обмежується. В наступній програмі продемонстровані деякі можливості і властивості директиви **#define**. Текст програми має наступний вид:

```
#include <stdio.h>
#include <windows.h>
// за бажанням можна використовувати коментар
#define TWO 2
// зворотна скісна риска переносить визначення
// на наступний рядок
#define OW "Логіка - останній притулок
позбавлених \ уяви. - Оскар Вайлд" #define FOUR
TWO * TWO
#define PX printf("X = %d\n", x)
#define FMT "X = %d\n"
 int
main(void)
{
    int x = TWO;
     SetConsoleOutputCP(1251);

    PX;      x = FOUR;
printf(FMT, x);
printf("%s\n", OW);
printf("TWO: OW\n");
return 0; }
```

Директива препроцесора поширюється до тих пір, поки не зустрінеться перший символ нового рядка після знаку '#'. Іншими словами, довжина директиви обмежена одним рядком. Однак, як вже згадувалося раніше, комбінації зворотної скісної риски та символу нового рядка видаляються до початку роботи препроцесора, тому директиву можна поширити на декілька фізичних рядків.

Проте, ці рядки утворюють один логічний рядок.

Кожен рядок **#define** (тобто логічний рядок) складається з трьох частин. Перша частина – це сама директива **#define**. Друга частина – це обране програмістом скорочення, що називається **ма́кросом**. Деякі макроси, як у

наведеному вище прикладі, являють собою значення. Вони називаються **об'єктними макросами**. В мові **C** існують ще функціональні макроси, про які мова піде пізніше.

Ім'я макросу не повинно містити пробілів. На макроси поширюються правила іменування змінних: дозволені тільки букви, цифри та символ підкреслювання `'_'`, а першим символом не повинна бути цифра. Третя частина (залишок рядка) називається списком заміни або тілом (рис. 4.1).

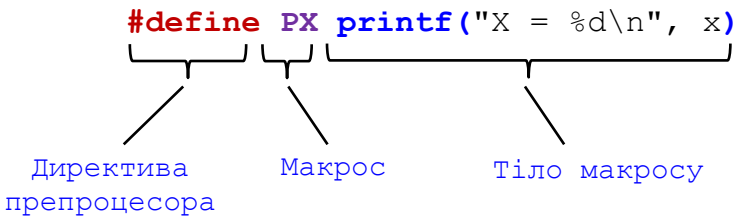


Рисунок 4.1 – Частини визначення об'єктного макросу

Коли препроцесор виявляє в програмі ім'я одного з макросів, він майже завжди замінює його на тіло. Як буде показано далі, з цього правила є одне виключення. Цей процес переходу від макросу до значення, що підставляється у підсумку, називається **розширенням**. Зверніть увагу, що в рядку **#define** можуть бути вказані стандартні коментарі. Необхідно пам'ятати, що до початку роботи препроцесору кожен коментар замінюється пробілом.

Результат роботи програми наведено на рис. 4.2.

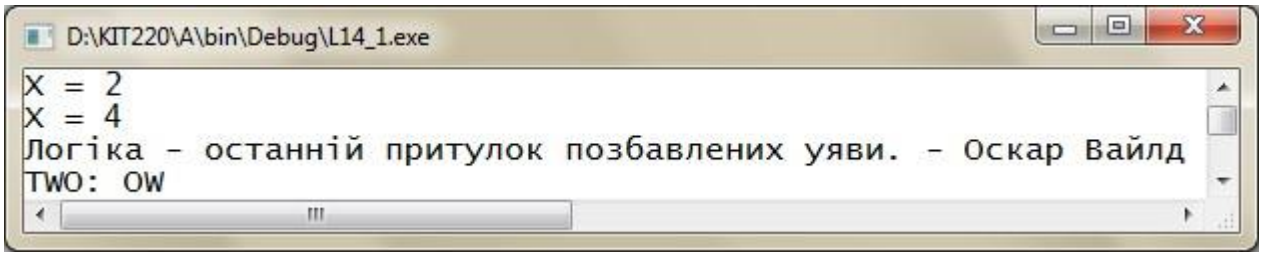


Рисунок 4.2 – Результат застосування директиви препроцесора **#define**

В розглянутій програмі оператор

```
int x = TWO;
```

перетворюється наступним чином:

```
int x = 2;
```

оскільки замість `TWO` було підставлено `2`. Потім оператор `PX;` набуває наступного вигляду:

```
printf("X = %d\n", x);
```

Підстановка здійснилася для всього оператора цілком. Це новий прийом, оскільки до сих пір ми використовували макроси тільки для представлення констант. В даному випадку ви бачите, що макрос може представляти будь-який рядок, навіть цілий вираз C. Однак відзначимо, що це константний рядок. Макрос `PX` буде виводити тільки значення змінної `x`.

В наступному рядку також демонструється новий прийом. Може здатися, що `FOUR` буде замінено на `4`, але насправді процес є дещо іншим.

Рядок `x = FOUR;`

перетворюється на

рядок `x = TWO *`

`TWO;`

який потім стає наступним рядком:

```
x = 2 * 2;
```

На цьому процес розширення макросу завершений. Справжнє множення відбувається не під час роботи препроцесора, а на етапі компіляції, оскільки компілятор C на цій стадії обчислює усі константні вирази (тобто вирази, які містять тільки константи). Препроцесор не виконує обчислень, а просто буквально здійснює вказані за допомогою директив підстановки.

Зверніть увагу, що визначення макросу може містити інші макроси, хоча деякі компілятори таку вкладеність макросів не підтримують. Наступний рядок `printf(FMT, x);` приводиться до виду:

```
printf("X = %d\n", x);
```

тому що FMT замінюється відповідним рядком. Такий підхід може виявитися зручним при наявності довгого керувального рядка, який доводиться застосовувати декілька разів. Замість цього можна було вдіяти наступним чином:

```
const char *fmt = "X = %d\n";
```

Потім можна використовувати fmt в якості керувального рядка для printf().

В наступному рядку OW замінюється відповідним рядком. Подвійні лапки перетворюють рядок заміщення символьною рядковою константою. Компілятор зберігає її в масиві з завершальним нульовим символом. Таким чином, директива

```
#define HAL 'Z'
```
визначає символьну константу, а директива

```
#define HAP "Z"
```
визначає символьний рядок: Z\0.

В цьому прикладі ми застосовували зворотну скісну риску безпосередньо перед кінцем рядка, поширюючи директиву на наступний рядок:

```
#define OW "Логіка - останній притулок позбавлених \
уяви. - Оскар Вайлд"
```

Зверніть увагу, що другий рядок вирівняний вліво. Якщо б директива мала вид:

```
#define OW "Логіка - останній притулок позбавлених
\     уяви. - Оскар Вайлд"
```
то вивід був би таким:

```
   Логіка - останній притулок позбавлених     уяви. - Оскар Вайлд"
```

Пробіли від початку рядка і до слова «уяви» вважаються частиною рядка. Зазвичай, де б препроцесор не виявив у програмі один з макросів, він замінює його літерально еквівалентним текстом заміни. Якщо рядок заміщення містить вкладені макроси, вони також замінюються. Єдиним виключенням під час заміни є ситуація, коли знайдений макрос розташований у подвійних дужках. Тому рядок `printf("TWO: OW\n");` виводить текст

```
TWO: OW
```
буквально, замість того, щоб вивести

```
2: Логіка - останній притулок позбавлених уяви. - Оскар Вайлд"
```

Для виводу вказаного рядка знадобиться наступний код:

```
printf("%d: %s\n", TWO, OW);
```

В цьому коді ім’я макросу знаходиться за межами подвійних дужок.

Коли ж слід використовувати символічні константи? Ви повинні їх застосовувати для більшості числових констант. Якщо число являє собою деяку константу, що приймає участь в обчисленнях, то символічне ім’я зробить її призначення більш зрозумілим. Якщо число є розміром масиву, то символічне ім’я спростить його майбутню зміну та коригування меж виконання циклів. Якщо число являє собою системний код, такий як **EOF**, то символічне ім’я збільшить ступінь мобільності програми: знадобиться змінити тільки одне визначення **EOF**. Мнемонічне значення, змінність і мобільність – усі ці характеристики пригортають до символічних констант особливу увагу.

Однак ключове слово **const**, яке тепер підтримується в **C**, забезпечує більш гнучкий спосіб створення констант. За допомогою **const** можна створювати глобальні та локальні константи, числові константи, константи у формі масивів і константи у вигляді структур. З іншого боку, константи-макроси можуть використовуватися для вказування розмірів стандартних масивів, а також

ініціалізуючих значень для величин **const**.

```
#define LIMIT 20
const int LIM = 50;
static int data1[LIMIT];            // допустимо
static int data2[LIM];              // не обов’язково повинно
                                    // бути допустимим
const int LIM2 = 2 * LIMIT;         // допустимо
const int LIM3 = 2 * LIM;           // не обов’язково повинно
// бути допустимим
```

Зверніть увагу на коментар «не обов’язково повинно бути допустимим». В мові **C** передбачається, що розмір масиву для неавтоматичних масивів задається цілочисловим константним виразом, тобто комбінацією цілочислових констант на зразок **5**, констант з перелічень і виразів **sizeof**. Значення, що оголошуються з застосуванням **const**, сюди не входять. В цьому відношенні **C** відрізняється від **C++**, де значення **const** можуть бути частиною константних виразів.

4.1.3. Лексеми препроцесора C

Формально тіло макросу повинно бути рядком лексем, а не рядком символів. Лексеми препроцесора C – це окремі слова в тілі визначення макросу. Вони відокремлюються одна від одної пробільними символами. Наприклад, визначення

```
#define FOUR 2*2
```

має одну лексему – послідовність `2*2`, але визначення

```
#define SIX 2 * 3
```

містить три лексеми: `2`, `*` і `3`.

Рядки символів і рядки лексем відрізняються в тому, як трактуються послідовності з множини пробілів. Розглянемо наступне визначення:

```
#define EIGHT 4     *     8
```

Препроцесор, який інтерпретує тіло макросу як рядок символів, замість `EIGHT` підставляє `4*8`. Тобто додаткові пробіли будуть частиною заміни, але препроцесор, який інтерпретує тіло як рядок лексем, замінить `EIGHT` трьома лексемами, що розділені одиночними пробілами: `4*8`. Іншими словами, інтерпретація у вигляді рядка символів трактує пробіли як частину тіла, а інтерпретація у вигляді рядка лексем вважає, що пробіли є роздільниками між лексемами всередині тіла. На практиці деякі компілятори C розглядають тіла макросів як рядки, а не як лексеми. Ця різниця має практичне значення тільки для більш складних випадків використання у порівнянні з наведеними тут.

До речі, в компіляторі C прийнята більш складна трактовка лексем у порівнянні з препроцесором. Компілятор розуміє правила мови C і не обов'язково потребує наявності пробілів для відокремлення лексем одна від одної. Наприклад, компілятор C буде інтерпретувати `2*2` як три лексеми, оскільки він з'ясує, що `2` є константою, а `*` – операцією.

4.1.4. Перевизначення констант

Припустимо, що ви визначили константу `LIMIT` як таку, що має значення `20`, і потім в тому ж самому файлі визначили її знову, але вже зі значенням `25`.

Такий процес називається перевизначенням константи. Політика перевизначення залежить від реалізації компілятора. Одні реалізації вважають перевизначення помилкою, якщо тільки нове визначення не збігається зі старим. Інші дозволяють перевизначення, можливо, виводячи попередження. В стандарті **ANSI** прийнятий перший варіант, що дозволяє перевизначення, тільки якщо нове визначення дублює попереднє.

Збіг визначень означає, що їх тіла повинні мати одні й ті ж самі лексеми в тому ж самому порядку. Тому наведені нижче визначення є еквівалентними:

```
#define SIX 2 * 3
#define SIX 2   *   3
```

Обидва визначення містять ті ж самі три лексеми, а надлишкові пробіли не є частиною тіла. Наступне визначення розглядається як таке, що відрізняється:

```
#define SIX 2*3
```

Воно містить тільки одну лексему, а не три, тому не збігається з попередніми визначеннями. Якщо ви бажаєте перевизначити макрос, застосовуйте директиву **#undef**, яка буде розглянута дещо пізніше.

Якщо треба перевизначити деякі константи, то для досягнення мети може бути простіше використовувати ключове слово **const** і правила області дії.

4.1.5. Використання аргументів у директиві #define

За допомогою аргументів можна створити **функціональні макроси**, що виглядають і діють переважним як звичайні функції. Макрос з аргументами дуже схожий на функцію, тому що аргументи розміщуються в круглих дужках. Визначення функціональних макросів мають один або більше аргументів в дужках, які потім будуть присутніми у виразі тіла заміни, як показано на рис. 4.3.

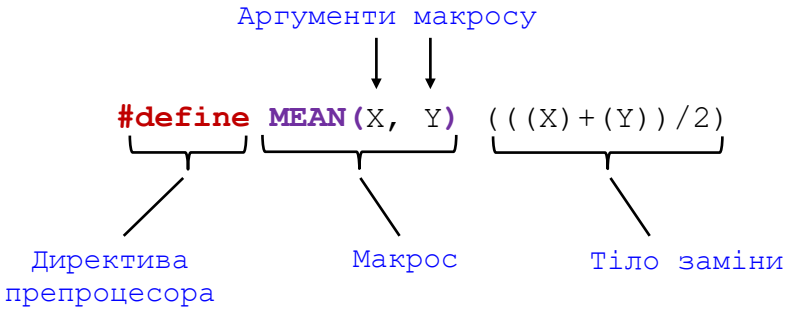


Рисунок 4.3 – Частини визначення функціонального макросу

Ось приклад визначення функціонального макросу:

```
#define SQUARE(X) X*X
```

Воно може застосовуватися в програмі наступним чином:

```
z = SQUARE(2);
```

Оператор виглядає схожим на виклик функції, хоча поведінка макросу не обов'язково буде ідентичною. В наступній програмі ілюструється використання макросів з аргументами.

```
#include <stdio.h>
#include <windows.h>
#define SQUARE(X) X*X
#define PR(X) printf("Результат: %3d\n", X)

int
main(void)
{
int X = 5;
int z;
SetConsoleOutputCP(1251);
printf("========================================\n");
printf("X = %d\n", X);
printf("========================================\n");
z = SQUARE(X);
printf("Обчислення SQUARE(X):     ");
PR(z);
z = SQUARE(2);
printf("Обчислення SQUARE(2):     ");
PR(z);
printf("Обчислення SQUARE(X+2):   ");
PR(SQUARE(X+2));
```

```
printf("Обчислення 100/SQUARE(2): ");
PR(100/SQUARE(2));
printf("========================================\n");
printf("X = %d\n", X);
printf("========================================\n");
printf("Обчислення SQUARE(++X):    ");
PR(SQUARE(++X));
printf("========================================\n");
printf("Після інкрементування X = %X\n", X);
printf("========================================\n");
return 0;
}
```

Макрос **SQUARE** має наступне визначення:

```
#define SQUARE(X) X*X
```

Тут **SQUARE** – ідентифікатор макросу, **X** в **SQUARE(X)** – аргумент макросу, а **X*X** – список заміни. Кожне входження **SQUARE(X)** в тексті програми замінюється на **X*X**. Відмінність даного прикладу від попередніх полягає у можливості використання в макросі будь-яких символів окрім **X**. Символ **X** у визначенні макросу замінюється символом, який вказаний при виклику макросу в програмі. Таким чином, **SQUARE(2)** замінюється на **2*2**, тому **X** дійсно грає роль аргументу.

Однак, як буде показано далі, аргумент макросу не працює в точності як аргумент функції. Нижче наведені результати виконання програми (рис. 4.4).

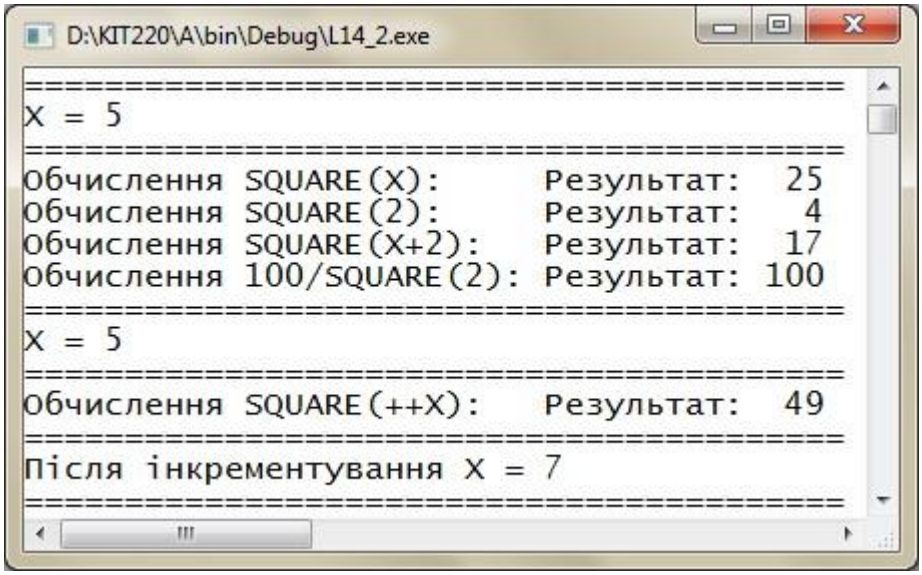


Рисунок 4.4 – Результат використання макросів з аргументами

Зверніть увагу, що деякі обчислення дають результат, який відрізняється від очікуваного. Насправді ваш компілятор може навіть видати не такий результат, як наведено в передостанньому рядку.

Перші два рядки цілком прогнозовані, але потім зустрічаються декілька дивних результатів. Згадайте, що `x` має значення `5`. Це може призвести до припущення, що `SQUARE(x+2)` повинно бути `7*7`, або `49`, проте виводиться `17` – просте число, але не квадрат! Причина такого виводу пов'язана з тим, що препроцесор не виконує обчислень, а просто замінює послідовність символів. Де б у визначенні не з'являлася `x`, препроцесор підставить замість `x` символи `x+2`.

Тому `X*X` приймає вигляд:

```
X+2*X+2
```

Єдиним множенням є `2*x`. Якщо `x` дорівнює `5`, вираз обчислюється наступним чином:

```
5 + 2 * 5 + 2 = 5 + 10 + 2 = 17
```

Цей приклад підкреслює важливу різницю між викликом функції і викликом макросу. При виклику функції їй передається значення аргументу під час виконання програми. При виклику макросу лексема аргументу передається в програму перед компіляцією. Це інший процес, що відбувається в інший час. Чи можна виправити визначення, щоб виклик `SQUARE(x+2)` видавав `49`? Звичайно.

Потрібні просто додаткові круглі дужки:

```
#define SQUARE(X) (X)*(X)
```

Тепер `SQUARE(X+2)` перетвориться на `(X + 2) * (X + 2)`, і ви отримаєте потрібне множення, оскільки круглі дужки залишаться в рядку.

Однак це не вирішує усіх проблем. Розглянемо події, які призводять до того, що наступний рядок `100/SQUARE(2)` у виводі перетворюється на рядок

```
100/2*2
```

Відповідно до пріоритетів операцій, вираз обчислюється зліва направо: `(100/2)*2`, або `50*2`, або `100`. Для усунення плутанини `SQUARE(X)` необхідно визначити наступним чином:

```
#define SQUARE(X) (X*X)
```

В результаті це дає `100/(2*2)`, що у підсумку обчислюється як `100/4`, або `25`. В наступному визначенні враховані помилки обох прикладів:

```
#define SQUARE(X) ((X)*(X))
```

З усього продемонстрованого можна зробити наступний висновок: треба застосовувати стільки круглих дужок, скільки необхідно для того, щоб забезпечити коректний порядок виконання операцій. Але навіть ці міри пересторогиі не рятують від помилки в останньому прикладі:

```
SQUARE(++X)
```

В результаті застосування макросу отримаємо:

```
++X*++X
```

Обчислення цього виразу призводить до ситуації, яка в стандарті називається **невизначеною поведінкою**. Через те, що вибір конкретного порядку виконання операцій залишений за розробниками реалізацій, деякі компілятори генерують множення `6*7` або `7*6`. Є компілятори, які можуть інкрементувати обидва операнди перед множенням, видаючи в результаті `7*7`, або `49`. В нашому випадку маємо саме цей результат. Тим не менш, в усіх цих випадках `x` починається зі значення `5` і закінчується значенням `7`, хоча код виглядає так, наче інкрементування відбувається тільки одного разу.

Найпростіше рішення цієї проблеми – уникати використання **++x** як аргументу макросу. Взагалі **краще не застосовувати в макросах операції інкременту та декременту**. Слід відзначити, що вираз **++x** буде працювати в якості аргументу функції, оскільки він обчислюється як значення `6`, яке потім передається функції.

4.1.6. Створення рядків з аргументів макросу: операція

Розглянемо наступний функціональний макрос:

```
#define PSQR(X) printf("Квадрат X дорівнює %d.\n", ((X)*(X)))
```

Припустимо, що цей макрос використовується наступним чином:

```
PSGR(8);
```

Тоді на консоль буде виведено:

```
Квадрат X дорівнює 64.
```

Зверніть увагу, що визначення **x** в рядку, обмеженому подвійними лапками, трактується як звичайний текст, а не лексема, яку можна замінити.

Припустимо, що ви бажаєте помістити аргумент макросу в рядок. Мова **C** дозволяє зробити це. Всередині частини функціонального макросу, що замінюється, символ **'#'** стає операцією препроцесора, яка перетворює лексеми на рядки. Нехай **x** є параметром макросу, тоді **#x** – це ім'я параметру, яке перетворене на рядок **"X"**. Такий процес називається перетворенням на рядок і демонструється в тексті наступної програми:

```
#include <stdio.h>
#include <windows.h>
#define PSQR(X) printf("Квадрат "#X" дорівнює %d.\n",((X)*(X)))
int main(void) {
int y = 5;

    SetConsoleOutputCP(1251);
    printf("y = %d\n", y);
    PSQR(y);
    PSQR(2 + 4);
    return 0;
}
```

Вивід на консоль має наступний вигляд (рис. 4.5).

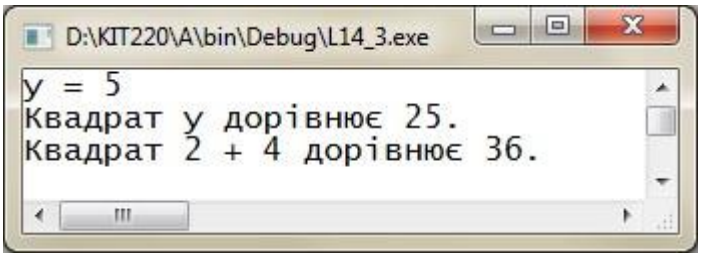


Рисунок 4.5 – Результат розміщення аргументу макросу в рядку

В першому виклику макросу **#x** замінюється рядком **"y"**, а в другому виклику замість **#x** підставляється **"2 + 4"**. Конкатенація рядків **ANSI C** потім

об’єднує ці рядки з іншими рядками в операторі **printf()** для отримання фінального рядка. Наприклад, перший виклик макросу дає наступний оператор:

```
printf("Квадрат " "y" " дорівнює %d.\n", ((y)*(y)));
```

Після цього конкатенація об’єднує три розташованих поруч рядки в один:

```
"Квадрат y дорівнює %d.\n"
```

4.1.7. Засіб препроцесору «злиття»: операція

Операція **##** може застосовуватися в замінній частині функціонального макросу. Додатково вона може використовуватися в замінній частині об’єктного макросу. Операція **##** об’єднує дві лексеми в одну. Припустимо, ви записали таке визначення:

```
#define XNAME(n) x ## n
```

Тоді макрос

```
XNAME(4)
```

буде розширений наступним чином:

```
x4
```

В наступній програмі цей і ще один макрос застосовуються для злиття лексем за допомогою операції **##**. Текст програми має наступний вигляд:

```
#include <stdio.h>
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n)
int main(void)
{
int XNAME(1) = 14;     // перетворюється на int x1 = 14;
int XNAME(2) = 20;     // перетворюється на int x2 = 20;
int x3 = 30;

PRINT_XN(1);  // перетворюється на printf("x1 = %d\n", x1);
PRINT_XN(2);   // перетворюється на printf("x2 = %d\n", x2);
PRINT_XN(3);   // перетворюється на printf("x3 = %d\n", x3);
return 0; }
```

Результат роботи програми наведено на рис. 4.6.

Зверніть увагу, що в макросі **PRINT_XN()** операція **#** використовується для об’єднання рядків, а операція **##** – для об’єднання лексем у новий ідентифікатор.

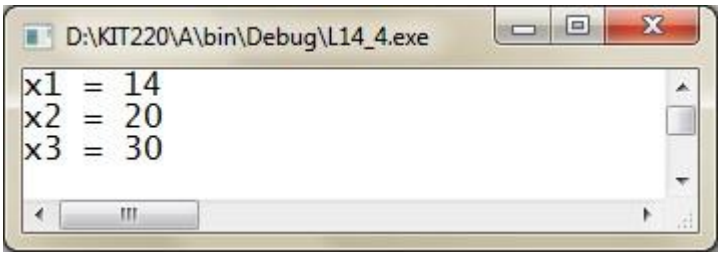


Рисунок 4.6 – Результат застосування макросів **#** і **##**

4.1.8. Макроси зі змінним числом аргументів: ... і __VA_ARGS__

Деякі функції, скажімо, **printf()**, приймають змінну кількість аргументів. Файл заголовку **stdvar.h**, що вже згадувався раніше, надає інструменти для створення функцій зі змінним числом аргументів, які визначається користувачем. В **C99/C11** те ж саме зроблено і для макросів.

Ідея полягає в тому, що останній аргумент у списку аргументів для визначення макросу може бути трьома крапками (...). Якщо це так, то в замінній частині може застосовуватися наперед визначений макрос **__VA_ARGS__**, який буде підставлений замість трьох крапок. Для прикладу розглянемо наступне визначення:

```
#define PR(...) printf(__VA_ARGS__)
```

Припустимо, що в програмі містяться наступні виклики макросу:

```
PR("Вітаю");
PR("вага = %d, доставка = $%.2f\n", wt, sp);
```

Для першого виклику **__VA_ARGS__** розширюється в один аргумент:

```
"Вітаю"
```

Для другого виклику він розширюється в три аргументи:

```
"вага = %d, доставка = $%.2f\n", wt, sp
```

Таким чином, підсумковий код виглядає так:

```
printf("Вітаю"); printf("вага = %d, доставка = $%.2f\n", wt, sp);
```

В наступній програмі наведено більш складний приклад, в якому використовується конкатенація рядків і операція #.

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
#define PR(X, ...) printf("Повідомлення " #X ": "__VA_ARGS__)
 int main(void)
{
double x = 48;
double y;

    SetConsoleOutputCP(1251);
    y = sqrt(x);
    PR(1, "x = %g\n", x);
    PR(2, "x = %.2f, y = %.4f\n", x, y);
return 0;
}
```

Результат роботи програми наведено на рис. 4.7.

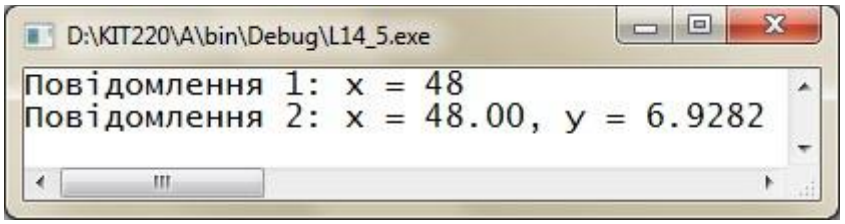


Рисунок 4.7 – Результат застосування макросу зі змінним числом аргументів

В першому виклику макросу **X** має значення **1**, так що **#X** стає **"1"**. В результаті отримуємо:

```
printf("Повідомлення" "1" ": " "x = %g\n", x);
```

Потім здійснюється конкатенація чотирьох рядків, скорочуючи виклик до наступного виду:

```
printf("Повідомлення 1: x = %g\n", x);
```

Не забувайте, що три крапки повинні бути останнім аргументом макросу. Наступне визначення є помилковим:

```
#define WRONG(X, ..., Y) #X #__VA_ARGS__ #y  // не працює
```

4.1.9. Вибір між макросом і функцією

Багато задач можуть бути вирішені за рахунок застосування макросу з аргументами або функції. Що ж саме краще використовувати? Тут немає будьяких строго визначених правил, але є декілька міркувань з цього приводу, які слід приймати до уваги.

Макроси є дещо складнішими у застосуванні, ніж звичайні функції, оскільки макроси можуть мати неочікувані побічні ефекти. Деякі компілятори обмежують визначення макросу одним рядком, і мабуть краще дотримуватися цього обмеження, навіть якщо у вашому компіляторі воно відсутнє.

Вибір між макросом і функцією пов'язаний з досягненням компромісу між швидкодією та розміром коду. Макрос генерує вбудований код, тобто в програму поміщається оператор. Якщо макрос використовується `20` разів, в програму вставляється `20` рядків коду. Коли `20` разів застосовується функція, в програмі все одно міститься лише одна копія її операторів, що зменшує розмір коду. З іншого боку, потік керування програми повинен переходити туди, де знаходиться функція, і потім повертатися в місце її виклику. Цей процес забирає більше часу, ніж виконання вбудованого коду.

Перевага макросів в тому, що вони не турбуються про типи змінних. Вони мають справу з рядками символів, а не справжніми значеннями. Таким чином, макрос `SQUARE(X)` може з однаковим успіхом використовуватися з типом `int` або типом `float`.

В стандарті `C99` з'явилася третя альтернатива – вбудовані функції, які будуть обговорюватися дещо пізніше. Програмісти зазвичай застосовують макроси для простих функцій, наприклад:

```
#define MAX(X, Y)  ((X) > (Y) ? (X) : (Y))
#define ABS(X)     ((X) < 0 ? -(X) : (X))
#define ISSIGN(X)  ((X) == '+' || (X) == '-' ? 1 : 0)
```

Останній макрос має значення `1`, або істину, якщо `X` є символом алгебраїчного знаку `'+'` або `'-'`.

Далі наведено декілька моментів, про які слід пам'ятати.

1) Ім’я макросу не повинне містити пробіли, але вони допускаються в рядку, що його заміщує. В **ANSI C** дозволені пробіли у списку аргументів.

2) Поміщайте в дужки кожен аргумент і визначення в цілому. Це гарантує коректне групування елементів у виразі наступного роду:

```
forks = 2 * MAX(guests + 3, last);
```

3) Використовуйте великі літери для імен функціональних макросів. Дана угода не так широко поширена, як застосування великих літер в іменах константних макросів. Проте, одна з важливих причин їх використання пов’язана з тим, що це нагадує вам про можливість побічних ефектів макросів.

4) Якщо ви маєте намір застосовувати макрос замість функції головним чином для прискорення роботи програми, спочатку спробуйте з’ясувати, чи забезпечить це помітний виграш. Макрос, який використовується в програмі лише один раз, не призведе до значного покращення швидкості її виконання. Макрос, що знаходиться всередині вкладеного циклу, є набагато кращим кандидатом для прискорення роботи програми. Багато систем пропонують профайлери (англ. **profilers**) програм, які допомагають виявляти фрагменти коду, що потребують найбільшого часу виконання.

Припустимо, що ви розробили декілька потрібних вам функціональних макросів. Чи повинні ви набирати їх кожного разу, коли пишеться нова програма? Звичайно ні, якщо ви будете пам’ятати про директиву #include, яка розглядається далі.

4.1.10. Включення файлів: директива #include

Коли препроцесор зустрічає директиву **#include**, він шукає файл з вказаним у директиві іменем і включає його вміст у поточний файл. Директива **#include** у файлі початкового коду замінюється текстом файлу, що включається. Це аналогічно вводу вмісту файлу, що включається, в тій самій позиції всередині початкового файлу. Існують два різновиди використання директиви **#include**:

```
#include <stdio.h>     // Ім’я файлу вказано в кутових дужках
#include "stuff.h"     // Ім’я файлу вказано в подвійних лапках
```

Кутові дужки повідомляють препроцесор про необхідність пошуку файлу в одному або декількох стандартних системних каталогах. Подвійні лапки говорять про те, що спочатку слід переглянути поточний каталог (або інший каталог, який вказаний разом з іменем файлу), а потім шукати в стандартних каталогах:

```
#include <stdio.h>        // Пошук в системних каталогах
#include "hot.h"          // Пошук в поточному робочому каталозі
#include "/usr/biff/p.h" // Пошук в каталозі /usr/biff
```

Інтегровані середовища розробки (IDE – Integrated Development Environment) також мають стандартне місце розташування або декілька таких місць для системних файлів заголовку. Багато таких IDE-середовищ надають опції меню для зазначення таких додаткових місць розташування, які повинні

переглядатися у випадку застосування кутових дужок. Як і в Unix, використання подвійних лапок означає пошук спочатку в локальному каталозі, але якому саме каталозі – залежить від компілятора. Деякі компілятори шукають в тому ж самому каталозі, де знаходиться початковий код, інші – в поточному робочому каталозі, а деякі – в каталозі, що містить файл проекту.

В ANSI C немає вимоги на суворе дотримання моделі каталогів для файлів, оскільки не всі обчислювальні системи організовані однаково. Взагалі метод, що застосовується для іменування файлів, залежить від системи, але використання кутових дужок і подвійних лапок – ні.

Навіщо ж включати файли? Причина полягає в тому, що вони містять інформацію, яка необхідна компілятору. Наприклад, файл **`stdio.h`** зазвичай містить визначення **`EOF`**, **`NULL`**, **`getchar()`** і **`putchar()`**. Два останніх визначені як функціональні макроси. Він також містить прототипи функцій вводу-виводу C.

Суфікс **`.h`** традиційно застосовується для файлів заголовку – файлів з інформацією, яка розміщується на початку програми. Файли заголовку часто містять оператори препроцесора. Деякі з них, наприклад, **`stdio.h`**, надаються системою, але ви можете створювати власні файли заголовків.

Включення крупного файлу заголовку не обов'язково призводить до значного збільшення розміру програми. Вміст файлів заголовку переважним чином є інформацією, яка використовується компілятором для генерації остаточного коду, а не матеріалом, що додається до цього коду.

Припустимо, що ви розробили структуру для зберігання імені та прізвища особи, а також написали функції для роботи з цією структурою. Усі можливі оголошення ви могли б зібрати разом всередині файлу заголовку **names_st.h**.

Приклад такого файлу має наступний вигляд:

```
#include <string.h>
#define SLEN 32

// оголошення структур
struct names_st
{
char first[SLEN];
char last[SLEN]; };

// визначення типів
typedef struct names_st names;

// прототипи функцій
void get_names(names *);
void show_names(const names *);
char *s_gets(char *st, int n);
```

Цей файл заголовку містить багато типових для таких файлів елементів: директиви **#define**, оголошення структур, оператори **typedef** і прототипи функцій. Зверніть увагу, що жоден з цих елементів не є виконуваним кодом. Вони являють собою інформацію, що застосовується компілятором при створенні виконуваного коду.

Показаний файл заголовку є досить простим. Зазвичай ви повинні використовувати **#ifndef** і **#define**, щоб захиститися від багаторазових включень файлу заголовку.

Виконуваний код зазвичай розміщується в файлі початкового коду, а не у файлі заголовку. В наступній програмі наведені визначення функцій, які відповідають прототипам функцій з файлу заголовку. В ній включається файл

заголовку, тому компілятор буде знати про тип **names_st**. Текст програми буде розташовуватися у файлі **names_st.c**.

```
#include <stdio.h>
#include "names_st.h"    // включення файлу заголовку

// визначення функцій void
get_names(names *pn)
{
printf("Введіть своє ім'я:      ");
s_gets(pn->first, SLEN);
printf("Введіть своє прізвище:  ");
s_gets(pn->last, SLEN);
}
 void show_names(const names *pn)
{
    printf("%s %s", pn->first, pn->last);
}

char *s_gets(char *st, int n)
{
    char *ret_val;     char *find;

    ret_val = fgets(st, n, stdin);     if(ret_val)
    {
        find = strchr(st, '\n'); // пошук символу нового рядка
        if(find)                 // якщо адреса не дорівнює нулю,
            *find = '\0';        // помістити туди нульовий символ
else
            while(getchar() != '\n')
                continue;        // відкинути залишок рядка
         }
         return
     ret_val; }
```

У функції **get_names()** застосовується функція **fgets()** (через **s_gets()**), щоб запобігти переповненню цільових масивів.

В наступній програмі використовується файл заголовку **names_st.h** і файл початкового коду **names_st.c**, які були розглянуті раніше. Текст програми буде розташовуватися у файлі **main.c**. Він має наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include "names_st.h"

int main(void)
{
```

```
    names candidate;
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    get_names(&candidate);
    printf("Ласкаво просимо до програми, шановний ");
show_names(&candidate);
printf("!\n");
return 0;
}
```

Результат роботи програми наведено на рис. 4.8.

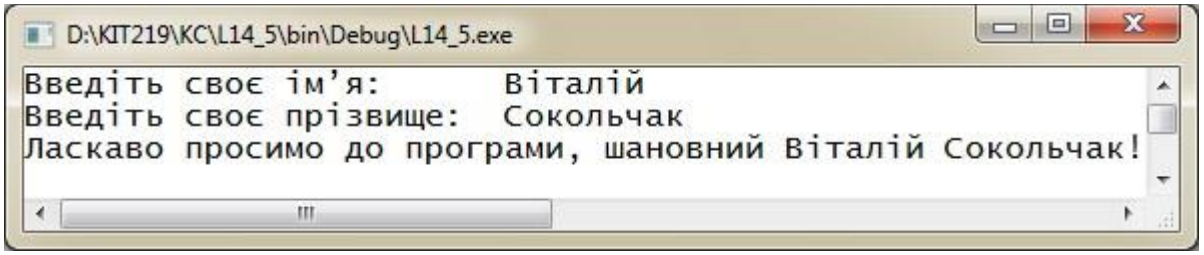


Рисунок 1.8 – Результат роботи програми з підключенням файлів заголовків

Зверніть увагу на наступні аспекти програми.

1) В обох файлах початкового коду застосовується структура **names_st**, тому вони обидва повинні включати файл заголовку **names_st.h**.

2) Необхідно компілювати та компонувати файли початкового коду **names_st.c** і **main.c**.

3) Оголошення та інші елементи подібного роду містяться в файлі заголовку **names_st.h**, а визначення функцій розміщені в файлі початкового коду **main.c**.

4.1.11. Випадки застосування файлів заголовку

Перегляд вмісту стандартних файлів заголовку допоможе отримати уяву про те, якого роду інформація в них знаходиться. Нижче наведено найбільш поширений вміст цих файлів:

1) **Символічні константи**. В типовому файлі `stdio.h`, наприклад,визначені константи `EOF`, `NULL` і `BUFSIZ` (розмір стандартного буфера вводу-виводу).

2) **Функціональні макроси**. Наприклад, функція `getchar()` зазвичай визначена як `getc(stdin)`, а `getc()` – в формі досить складного макросу. Файл заголовку `ctype.h`, як правило, містить визначення макросів для функцій, що працюють з символами.

3) **Визначення функцій**. Файл заголовку `string.h`, наприклад, містить оголошення для сімейства функцій обробки рядків. Відповідно до `ANSI C` і подальших стандартів, ці оголошення представлені у вигляді прототипів функцій.

4) **Визначення шаблонів структур**. Стандартні функції вводу-виводу використовують структуру `FILE`, яка містить інформацію про файл і зв'язаний з ним буфер. Оголошення цієї структури знаходиться у файлі `stdio.h`.

5) **Визначення типів**. Ви можете згадати, що стандартні функції вводувиводу застосовують аргумент типу вказівника на `FILE`. Зазвичай у файлі `stdio.h` використовується засіб `#define` або `typedef` для того, щоб ім'я `FILE` являло собою вказівник на структуру. Аналогічно, в файлах заголовку визначені типи `size_t` і `time_t`.

Багато хто з програмістів розробляють власні стандартні файли заголовків для застосування у своїх програмах. Це особливо корисно в ситуації, коли ви створюєте сімейство взаємопов'язаних функцій та/або структур.

Крім того, файли заголовків можна застосовувати для оголошення зовнішніх змінних, з якими спільно працюють декілька файлів. Це доцільно, наприклад, при розробці сімейства функцій, що спільно використовують змінну для повідомлення про деякий стан, такий як умова помилки. В такому випадку можна визначити змінну з зовнішнім зв'язуванням і областю дії на рівні файлу початкового коду, який містить оголошення функцій:

```
int status = 0;          // змінна з областю дії
                         // на рівні файлу початкового коду
```

Потім в файл заголовку, який зв'язаний з файлом початкового коду, можна помістити посилальне оголошення:

```
extern int status;      // в файлі заголовку
```

Цей код потім з'явиться в будь-якому файлі, де був включений даний файл заголовку, зробивши змінну доступною файлам, які працюють зі згаданим сімейством функцій. Крім того, шляхом включення це оголошення виявиться в файлі початкового коду функцій, однак в одному файлі допускається наявність визначального та посилального оголошень, якщо вони узгоджені за типом.

Ще одним кандидатом для включення до файлу заголовку є змінна або масив з областю дії на рівні файлу, внутрішнім зв'язуванням і кваліфікатором **const**. Частина **const** запобігає випадковим змінам, а частина **static** означає, що кожен файл, який включає цей заголовок, отримує власну копію констант. Це усуває необхідність в наявності одного файлу з визначеним оголошенням і решти файлів з посилальними оголошеннями.

Директиви **#include** і **#define** є такими засобами препроцесора C, які найбільш інтенсивно застосовуються. Інші директиви будемо розглядатися менш детально.

4.1.12. Інші директиви препроцесора

Програмістам часто доводиться створювати програми та бібліотечні пакети на мові C, які повинні працювати в різних середовищах. Види коду можуть варіюватися від середовища до середовища. Препроцесор пропонує декілька директив, що допомагають програмісту створювати код, який може переноситися з однієї системи на іншу за рахунок змін значень макросів **#define**. Директива **#undef** відміняє попереднє визначення **#define**. Директиви **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif** і **#endif** дозволяють вказувати різні варіанти коду, що підлягає компіляції. Директива **#line** дає можливість переустановлювати інформацію про рядки і файли, директива **#error** призначена для виводу повідомлень про помилки, а за допомогою директиви **#pragma** можна надавати інструкції компілятору.

Директива `#undef` відміняє задане визначення `#define`. Припустимо, що є наступне визначення:

```
#define LIMIT 400
```

Тоді директива

`#undef LIMIT` видалить це визначення. Потім `LIMIT` можна перевизначити, задавши нове значення. Відміна визначення `LIMIT` допустима навіть у випадку, якщо попереднє визначення не робилося. Якщо ви бажаєте використовувати деяке ім'я, але не переконані в тому, що воно не було визначено раніше, на всяк випадок його визначення можна відмінити.

Визначення з точки зору препроцесора. Стосовно того, що вважати ідентифікатором, препроцесор керується такими самими правилами, як і мова C: ідентифікатор може складатися тільки з букв верхнього і нижнього регістру, цифр і символу підкреслювання, а першим символом не може бути цифра. Коли препроцесор зустрічає в будь-якій директиві ідентифікатор, він вважає його визначеним або невизначеним. При цьому «визначений» означає, що ідентифікатор визначений препроцесором. Якщо ідентифікатор є іменем макросу, що був створений раніше директивою `#define` в тому ж самому файлі, і він не відмінявся за допомогою `#undef`, то ідентифікатор є визначеним. Якщо ідентифікатор – це не макрос, а, скажімо, змінна з областю дії на рівні файлу, то з точки зору препроцесора він не є визначеним.

Визначеним може бути об'єктний макрос, включаючи пустий макрос, або функціональний макрос:

```
#define LIMIT 1000          // ідентифікатор LIMIT – визначений
#define GOOD                // ідентифікатор GOOD – визначений
#define A(X) ((-(X))*(X)) // ідентифікатор A – визначений
int q;              // ідентифікатор q не є макросом,
                    //  тому він не є визначеним
#undef GOOD         // ідентифікатор GOOD – не визначений
```

Зверніть увагу, що область дії макросу `#define` починається з місця його оголошення у файлі і продовжується аж до відповідної директиви `#undef` або до кінця файлу, в залежності від того, що відбудеться першим. Крім того, майте на

увазі, що позиція директиви **#define** у файлі буде залежати від місця розташування директиви **#include**, якщо макрос потрапляє з файлу заголовку.

Декілька наперед заданих макросів, таких як **__DATE__** і **__FILE__**, завжди вважаються визначеними, причому їх визначення не може бути відмінено.

Умовна компіляція. Решту зі згаданих директив можна застосовувати для налаштування умовної компіляції. Це означає, що їх можна використовувати для повідомлення компілятору про те, приймати чи ігнорувати блоки інформації або коду відповідно до умов на етапі компіляції.

Директиви #ifdef, #else і #endif

Наступний короткий приклад прояснить, що робить умовна компіляція.

Погляньте на наступний код:

```
#ifdef MAVIS
    #include "horse.h"       // виконується, якщо ідентифікатор
                             // MAVIS визначений
    #define STABLES 5
#else
    #include "cow.h"         // виконується, якщо ідентифікатор
                             // MAVIS не визначений
    #define STABLES 15
#endif
```

В цьому коді були застосовані відступи, які були дозволені новими реалізаціями мови **C** і стандартом **ANSI**. У випадку більш старих реалізацій може доведеться вирівнювати вліво усі директиви або хоча б символи **#**:

```
#ifdef MAVIS
#    include "horse.h"       // виконується, якщо ідентифікатор
                             // MAVIS визначений
#    define STABLES 5
#else
#    include "cow.h"         // виконується, якщо ідентифікатор
                             // MAVIS не є визначеним
#    define STABLES 15
#endif
```

Директива **#ifdef** говорить про те, що у разі, коли наступний за нею ідентифікатор **MAVIS** був визначений препроцесором, необхідно обробити усі директиви і скомпілювати весь код **C** до наступної директиви **#else** або **#endif** в залежності від того, що зустрінеться раніше. Якщо передбачена директива **#else**, то треба обробити весь код між **#else** і **#endif**, коли ідентифікатор не

визначений. Форма **#ifdef...#else** багато в чому нагадує оператор **if...else** мови **C**. Основна відміна полягає в тому, що препроцесор не розпізнає фігурні дужки (**{}**) як метод позначення блоку, тому для позначення блоків директив використовуються директиви **#else** (якщо є) і **#endif** (повинна бути). Такі умовні структури можуть бути вкладеними. В наступній програмі показано, що ці директиви можна застосовувати також для позначення блоків операторів **C**.

```
#include <stdio.h>
#include <windows.h>
#define
JUST_CHECKING
#define LIMIT 4
int main(void)
{      int i;
int total = 0;
    SetConsoleOutputCP(1251);
for(i = 1; i <= LIMIT; i++)
    {
        total += 2 * i * i + 1;
#ifdef JUST_CHECKING          printf("i = %d, проміжна
сума = %2d\n", i, total); #endif
    }
printf("=========================\n");
printf("Результат = %d\n", total);
printf("=========================\n");
return 0; }
```

В результаті компіляції та виконання програми буде отриманий наступний екран (рис. 4.9):

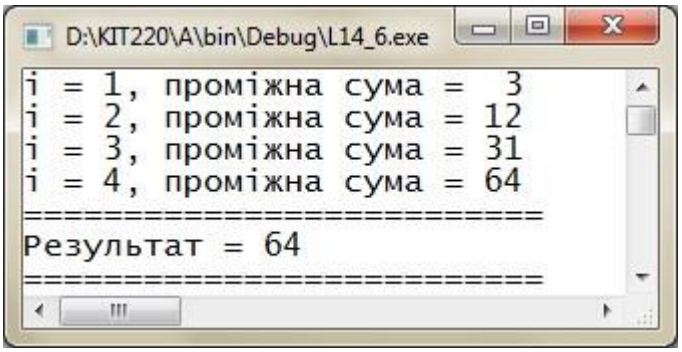


Рисунок 1.9 – Результат застосування умовної компіляції

Якщо опустити визначення **JUST_CHECKING** (або позначити його як коментар, або відмінити визначення за допомогою директиви **#undef**) і повторно

провести компіляцію програми, відобразиться тільки останній рядок. Таким прийомом можна користуватися, наприклад, при налагодженні програми.

Якщо визначити ідентифікатор **JUST_CHECKING** і задіяти його в умовних виборах за допомогою **#ifdef**, то компілятор буде включати програмний код для виводу проміжних значень з метою налагодження програми. Після налагодження визначення його можна видалити і повторно скомпілювати програму. Якщо в подальшому знову треба буде виводити проміжні значення, то можна знову вставити визначення і позбавити себе від необхідності повторно набирати усі додаткові оператори виводу.

Ще однією можливістю є застосування **#ifdef** для вибору альтернативних блоків коду, що прилаштовані до різних реалізацій **C**.

Директива #ifndef може використовуватися спільно з директивами **#else** та **#endif** таким самим способом, що й **#ifdef**. Директива **#ifndef** з'ясовує, чи не визначений ідентифікатор, який йде за нею. Вона являє собою інверсію директиви **#ifdef**. Ця директива часто застосовується для визначення константи, якщо вона ще не була визначена. Нижче наведено приклад.

```
// arrays.h
#ifndef SIZE
#define SIZE 100
#endif
```

Зазвичай така конструкція використовується для запобігання множинних визначень одного й того ж самого макросу при включенні декількох файлів заголовку, кожен з яких може містити визначення. В цьому випадку визначення в першому файлі заголовку стає активним, а наступні визначення в інших файлах заголовку ігноруються.

Розглянемо ще один випадок застосування директиви. Припустимо, що в заголовок файлу поміщений такий рядок:

```
#include "arrays.h"
```

В результаті константа **SIZE** буде визначена як **100**. Однак якщо помістити в заголовок файлу наступний код:

```
#define SIZE 10
```

`#include "arrays.h"` то `SIZE` встановлюється в `10`. В даному випадку `SIZE` визначається до обробки файлу `arrays.h`, тому рядок `#define SIZE 100` пропускається. Такий прийом можна використовувати, наприклад, під час тестування програми з застосуванням масиву меншого розміру. При коректній роботі програми, можна видалити оператор `#define SIZE 10` і провести повторну компіляцію. Тоді ніколи не доведеться думати про модифікацію самого файлу заголовку `arrays.h`.

Директива `#ifndef` часто використовується для запобігання багаторазового включення файлу. З цієї причини файли заголовку зазвичай містять наступні рядки:

```
// things.h
#ifndef THINGS_H_
#define THINGS_H_
// вміст файлу things.h
#endif
```

Припустимо, що цей файл певним чином був включений декілька разів. Коли препроцесор зустрічає перше включення даного файлу, ідентифікатор `THINGS_H_` не визначений, тому він визначається і обробляється частина файлу, що залишилася. При появі наступного включення того ж самого файлу ідентифікатор `THINGS_H_` вже є визначеним, тому решта частина файлу пропускається.

З’ясуємо, через що файл може бути включений декілька разів? Найбільш поширена причина полягає в тому, що багато файлів, що підключаються, містять директиви включення інших файлів, тому можна явно включити файл, в якому цей вказаний файл вже включено. Чому це є проблемою? Деякі елементи, що поміщаються в файли заголовків, такі як оголошення типів структур, можуть зустрічатися в файлі тільки один раз. Для запобігання багаторазового включення в стандартних файлах заголовку застосовується директива `#ifndef`. Одна з задач полягає в тому, щоб переконатися, що даний ідентифікатор не є визначеним в іншому місці. Постачальники бібліотек зазвичай вирішують її шляхом використання імені файлу в якості ідентифікатора, записуючи ім’я у верхньому регістрі, замінюючи крапки символами підкреслювання та додаючи символ

підкреслювання (або два) в якості префіксу і суфіксу. Якщо ви поглянете на текст файлу **stdio.h**, то можете виявити в ньому приблизно такий код:

```
#ifndef _STDIO_H
#define _STDIO_H
// вміст файлу
#endif
```

Ви можете зробити аналогічним чином. Проте, слід уникати застосування символу підкреслювання в якості префіксу, оскільки в стандарті вказано, що використання подібного роду є зарезервованим. Навряд чи у вас виникне бажання випадково визначити макрос, який конфліктує с будь-чим у стандартних файлах заголовків. В наступній програмі директива **#ifndef** використовується для захисту від багаторазового включення файлу заголовку.

```
// names.h - додавання захисту від багаторазового включення
#ifndef NAMES_H_
#define NAMES_H_
// константи
#define SLEN 32
// оголошення структур
struct names_st
{
    char first[SLEN];
char last[SLEN];
};
// визначення типів
typedef struct names_st names;
// прототипи функцій
void get_names(names *); void
show_names(const names *);
char *s_gets(char *st, int n);
#endif
```

Спробуємо провести тестування цього файлу заголовку за допомогою програми, що наведена нижче. Програма повинна працювати коректно, і не повинна успішно компілюватися, якщо видалити захист за допомогою **#ifndef**.

```
// main.c - двократне включення файлу заголовку
#include <stdio.h>
#include <windows.h>
#include "names.h"
#include "names.h"      // ненавмисне повторне включення

int main(void)
{
```

```
    names winner = { "Іван", "Іванов" };
    SetConsoleOutputCP(1251);
printf("Переможцем став %s %s.\n",winner.first, winner.last);
return 0;
}
```

Результат виконання програми наведено на рис. 4.10.

Рисунок 4.10 – Результат застосування умовної компіляції

Директива `#if` дуже схожа на звичайний оператор `if` мови **C**. За `#if` йде константний цілочисловий вираз, який вважається істинним, коли він має ненульове значення. У виразі можуть застосовуватися логічні операції та операції відношення:

```
#if SYS == 1
#include "ibra.h"
#endif
```

Для розширення комбінації `#if...#else` можна використовувати директиву `#elif`. Розглянемо наступний приклад:

```
#if SYS == 1
    #include "ibmpc.h"
#elif SYS = 2
    #include "vax.h"
#elif SYS == 3
    #include "mac.h"
#else
    #include "general.h"
#endif
```

Якщо наведені рядки застосовуються, скажімо, в системі `VAX`, ідентифікатор `VAX` повинен бути визначений десь раніше в цьому файлі за допомогою наступного рядка:

```
#define VAX
```

Однією з цілей використання засобів умовної компіляції є забезпечення можливості перенесення програми на інші платформи. За рахунок зміни декількох ключових визначень на початку файлу можна налаштовувати різні значення і включати певні файли заголовку для різних систем.

4.1.13. Наперед визначені макроси

В стандарті **C** описано декілька **наперед визначених макросів**, які наведені в табл. 4.1.

Таблиця 4.1 – Наперед визначені макроси

Макрос	Опис
`__DATE__`	Рядок символів у формі `"Ммм дд pppp"` – дата обробки препроцесором, наприклад, `Apr 27 2021`
`__FILE__`	Рядок символів – ім'я поточного файлу початкового коду
`__LINE__`	Цілочислова константа – номер рядка у поточному файлі початкового коду
`__STDC__`	Встановлений в `1` для зазначення того, що реалізація відповідає стандарту **C**
`__STDC_HOSTED__`	Встановлений в `1` для розміщеного середовища; в протилежному випадку – `0`
`__STDC_VERSION__`	Для **C99** встановлений в `199901L`; для **C11** встановлений в `201112L`
`__TIME__`	Час трансляції у формі `"гг:хх:сс"`, наприклад, `11:59:39`

Слід відзначити, що стандарт **C99** надає **наперед визначений ідентифікатор** `__func__`, який розширюється до рядкового представлення імені функції, що його містить. З цієї причини даний ідентифікатор повинен мати область дії в межах функції, в той час як макроси по суті мають область дії на

рівні файлу. Таким чином, **__func__** є наперед визначеним ідентифікатором мови **C**, а не наперед визначеним макросом.

В наступній програмі демонструються декілька наперед визначених ідентифікаторів і макросів в дії. Зверніть увагу, що деякі з них є нововведеннями стандарту **C99**, тому компілятори, що розроблені до появи цього стандарту, можуть їх не сприймати.

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

void why_me();
int main(void)
{
    SetConsoleOutputCP(1251);

printf("=======================================\n");
printf("Функція main()\n");
printf("=======================================\n");
printf("Файл:         %s\n", __FILE__);
printf("Дата:         %s\n", __DATE__);
printf("Час:          %s\n", __TIME__);
printf("Версія:             %ld\n", __STDC_VERSION__);
printf("Це рядок      %d\n", __LINE__);
printf("Це функція    %s\n", __func__);
why_me();
return 0;
}
void why_me()
{
printf("=======================================\n");
printf("Функція why_me()\n");
printf("=======================================\n");
printf("Це функція    %s\n", __func__);
printf("Це рядок      %d\n", __LINE__); }
```

Результат виконання програми наведено на рис. 4.11.

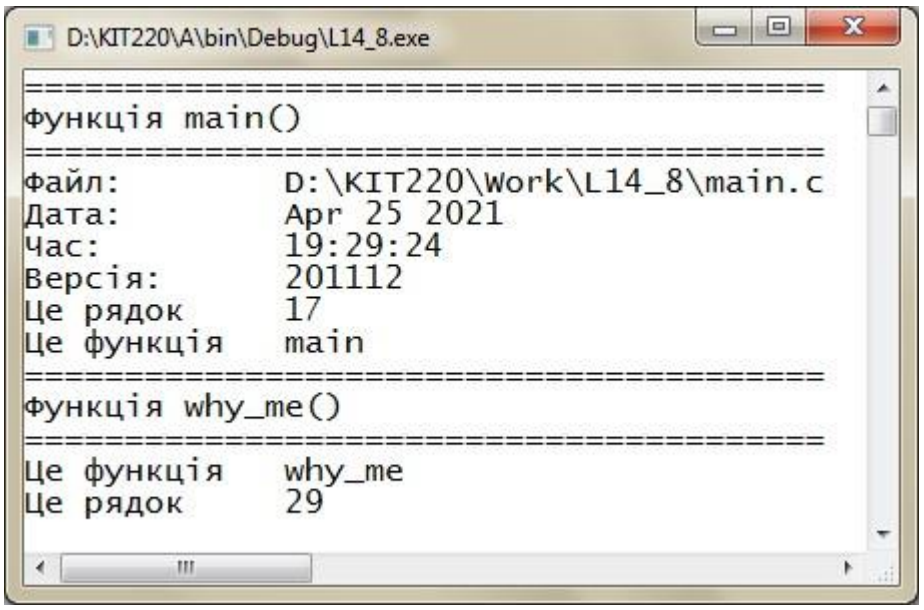


Рисунок 4.11 – Результат використання в програмі наперед визначених ідентифікаторів та макросів

Директива `#line` дозволяє перевстановлювати нумерацію рядків та ім'я файлу, що виводяться за допомогою макросів `__LINE__` і `__FILE__`. Директиву `#line` можна використовувати наступним чином:

```
#line 1000              // перевстановлює поточний номер
                        // рядка на 1000
#line 10 "cool.c"       // перевстановлює номер рядка на 10,
                        // а ім'я файлу - на cool.c
```

Директива `#error` змушує препроцесор видати повідомлення про помилку, яке містить у собі будь-який текст, що вказаний в директиві. Якщо це можливо, процес компіляції повинен призупинитися. Директиву можна застосовувати так:

```
#if __STDC_VERSION__ != 201112L
#error Невідповідність C11
#endif
```

Процес компіляції не буде відбуватися, коли компілятор використовує більш старий стандарт, і завершиться успішно, коли застосовується стандарт C11.

Директива `#pragma`**.** У сучасних компіляторів існує декілька налаштувань, які можна модифікувати за допомогою аргументів командного рядка або через

меню **IDE**середовища. Директива **#pragma** дозволяє поміщати інструкції для компілятора в початковий код. Наприклад, під час розробки стандарту **C99** на нього посилалися як на **C9X**, і в одному з компіляторів використовувалася наступна директива для включення підтримки цього стандарту:

```
#pragma c9x on
```

В загальному випадку кожен компілятор має власний набір вказівок. Вони можуть застосовуватися, наприклад, для керування об'ємом пам'яті, що виділяється під автоматичні змінні, для встановлення рівня суворості при перевірці помилок або для включення нестандартних мовних засобів. В стандарті **C99** надаються три стандартних вказівки технічної природи.

Крім того, стандарт **C99** підтримує операцію препроцесора **_Pragma()**. Вона перетворює рядок на звичайну вказівку компілятору.

Наприклад, операція

```
_Pragma("nonstandardtreatmenttypeB on")
```

є еквівалентом наступної вказівки:

```
#pragma nonstandardtreatmenttypeB on
```

Оскільки в цій операції не використовується символ #, вона може виступати в якості частини розширення макросу:

```
#define PRAGMA(X) _Pragma(#X)
#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE X)
```

Після цього можна застосовувати код на зразок того, що наведено нижче:

```
LIMRG(ON)
```

До речі, наступне визначення не працює, хоча виглядає цілком коректним:

```
#define LIMRG(X) _Pragma(STDC CX_LIMITED_RANGE #X)
```

Проблема полягає в тому, що воно покладається на конкатенацію рядків, але компілятор не виконує конкатенацію до тих пір, поки не завершиться робота препроцесора.

Оператор **_Pragma** виконує всю роботу щодо перетворення з рядків, тобто керувальні послідовності в рядку перетворюються на символи, що їх представляють. Таким чином, виклик операції

```
_Pragma("use_bool \"true \"false")
```

приймає наступний вигляд:

```
#pragma use_bool "true "false
```

4.1.14. Узагальнений вибір (C11)

Термін узагальнене програмування відноситься до коду, який не є специфічним для конкретного типу, але після зазначення типу може

транслюватися в код для цього типу. Наприклад, мова **C++** дозволяє створювати узагальнені алгоритми в формі шаблонів, які компілятор потім використовує при автоматичному створенні екземпляра коду для вказаного типу. В мові **C** немає нічого схожого на це. Тим не менш, в **C11** з'явився новий вид виразу, що називається **виразом узагальненого вибору**, який можна застосовувати для вибору значення на основі типу виразу, тобто базуючись на тому, чи є типом виразу **int**, **double** і т. д. Вираз узагальненого вибору – це не оператор препроцесора, але зазвичай він використовується як частина визначення макросу **#define**, що має певні риси узагальненого програмування.

Вираз узагальненого вибору виглядає наступним чином:

```
_Generic(X, int: 0, float: 1, double: 2, default: 3)
```

Тут **_Generic** – нове ключове слово **C11**. Круглі дужки після **_Generic** містять декілька елементів, розділених комами. Перший елемент являє собою вираз, а кожен з елементів, що залишилися, – це тип, за яким йде значення, на зразок **float**: **1**. Тип першого елемента відповідає одній з міток, і значенням усього виразу буде значення, яке вказано після мітки, що збіглася з типом.

Наприклад, припустимо, що **x** в наведеному вище виразі є змінною типу **int**. Тоді тип **x** відповідає мітці **int**, наслідком чого буде те, що увесь вираз

отримає значення `0`. Якщо тип не відповідає жодній з міток, значенням усього виразу стає те, що вказано після мітки `default`. Оператор узагальненого вибору схожий на оператор `switch` за виключенням того, що зіставлення з мітками проводиться для типу виразу, а не його значення.

Давайте розглянемо приклад об'єднання оператора узагальненого вибору з визначенням макросу:

```
#define MYTYPE(X) _Generic((X),
\    int   :  "int",          \
float  :  "float",            \
double :  "double",           \
default:  "other"           \
)
```

Згадайте, що макрос повинен бути визначений в одному логічному рядку, але за допомогою символу `'\'` один логічний рядок можна розбивати на декілька фізичних рядків. В даному випадку вираз узагальненого вибору оцінюється як рядок. Скажімо, виклик макросу `MYTYPE(5)` оцінюється як рядок `"int"`, оскільки тип значення `5` відповідає мітці `int:`. Програма для демонстрації даного макросу має наступний вигляд:

```
#include <stdio.h>
#define MYTYPE(X) _Generic((X), \
int    :  "int",              \
float  :  "float",            \
double :  "double",           \
default:  "other"             \
)
 int
main(void)
{
    int d = 5;
    printf("%s\n", MYTYPE(d));       // d має тип int
    printf("%s\n", MYTYPE(2.0*d));   // 2.0*d має тип double
printf("%s\n", MYTYPE(3L));      // 3L має тип long
printf("%s\n", MYTYPE(&d));      // &d має тип int *
return 0; }
```

Результат роботи програми наведено на рис. 4.12.

Рисунок 4.12 – Результат застосування виразу узагальненого вибору

При оцінюванні виразу узагальненого вибору програма не обчислює перший елемент. Вона тільки з'ясовує його тип. Єдиним виразом, який обчислюється, є те, що вказано у збіжній мітці.

Засіб `_Generic` можна застосовувати для визначення макросів, які діють подібно до функцій, що не залежать від типу («узагальнених»).

4.1.15. Вбудовані функції (C99)

Зазвичай з викликом функції пов'язані накладні витрати. Це означає, що підготовка виклику, передавання аргументів, перехід до коду функції і повернення потребують часу на виконання. Як ви вже бачили, макрос можна використовувати для вбудованого коду, тим самим уникаючи таких накладних витрат. В стандарті `C99` був запозичений у `C++` (але не повністю) інший підхід – **вбудовані функції**. Виходячи з назви, ви могли б очікувати, що вбудована функція замінює виклик функції вбудованим кодом, але це не так. В стандартах `C99` і `C11` насправді вказано: «перетворення функції на вбудовану передбачає, що її виклик буде настільки швидким, наскільки це можливо. Ступінь, до якого подібні припущення є ефективними, залежить від реалізації». Таким чином, перетворення функції на вбудовану може призвести до того, що компілятор замінить виклик функції вбудованим кодом та/або проведе оптимізаціє іншого роду або взагалі не буде робити жодних дій.

Існують різні способи створення визначень вбудованих функцій. В стандарті говориться про те, що функція з внутрішнім зв'язуванням може бути

зроблена вбудованою, і дане визначення для вбудованої функції повинне знаходитися в тому ж самому файлі, де функція застосовується. Тому простий підхід передбачає використання **специфікатора функції `inline`** поряд зі специфікатором класу зберігання **`static`**. Як правило, вбудовані функції визначаються до їх першого застосування у файлі, тому визначення діє також і в якості прототипу. Іншими словами, код буде виглядати приблизно так:

```
#include <stdio.h>
inline static void eatline()  // вбудоване визначення або
                              // прототип
{
    while(getchar() != '\n')
continue;
}

int main(void)
{
    ...
    eatline();                // виклик функції
    ...
}
```

Зустрівши вбудоване оголошення, компілятор може, наприклад, замінити виклик функції **`eatline()`** на її тіло. Це означає, що результат може бути такий, наче б ви замість цього написали наступний код:

```
#include <stdio.h>
inline static void eatline()   // вбудоване визначення або
                               // прототип
{
    while(getchar() != '\n')
continue;
}

int main(void)
{
    ...   while(getchar() != '\n')  // заміна виклику функції
              continue;
    ...
}
```

Оскільки вбудована функція не має окремого призначеного для неї блока коду, отримати її адресу неможливо. Крім того, вбудована функція може бути

прихованою при налагодженні програми.

Вбудована функція повинна бути короткою. Для великої за розміром функції час, що витрачається на її виклик, буде невеликим у порівнянні з часом виконання тіла функції, тому використання вбудованої функції не забезпечить істотної економії часу.

Для проведення оптимізацій щодо вбудованої функції компілятору треба мати інформацію про вміст визначення функції. Це означає, що визначення вбудованої функції повинне знаходитися в тому ж самому файлі, що і її виклик. З цієї причини вбудована функція зазвичай має внутрішнє зв'язування. Відповідно, якщо програма складається з декількох файлів, вбудоване визначення знадобиться помістити в кожен файл, який викликає функцію. Для досягнення такої умови простіше за все вказати визначення вбудованої функції у файлі заголовку і потім включати цей файл до тих файлів, де ця функція застосовується.

```
// eatline.h
#ifndef EATLINE_H_
#define EATLINE_H_
inline static void
eatline() {
    while(getchar() != '\n')
        continue;
}
#endif
```

Вбудована функція є виключенням з правила, яке не рекомендує поміщати виконуваний код до файлу заголовку. Оскільки вбудована функція має внутрішнє зв'язування, її визначення в декількох файлах не викликає проблем.

На відміну від **C++**, мова **C** дозволяє також змішувати вбудовані визначення з зовнішніми визначеннями (визначеннями функцій з зовнішнім зв'язуванням).

Наприклад, розглянемо програму, що складається з наступних трьох файлів:

```
// file1.c
...
inline static double square(double);
double square(double x) { return x * x;
}
```

```
int main(void)
{
    double q = square(1.3);
    ...
}

// file2.c
...
double square(double x) { return (int)(x * x); }
void spam(double v)
{
    double kv = square(v);
    ...
}

// file3.c
...
inline double square(double x) { return (int)(x * x + 0.5); }
void masp(double w)
{
    double kw = square(w);
    ...
}
```

Перший файл містить визначення `inline static`, як і раніше. Другий файл має визначення звичайної функції, звідси і наявність зовнішнього зв'язування. Третій файл має визначення `inline`, в якому не вказаний кваліфікатор `static`.

Що буде відбуватися? Функція `spam()` у файлі `file2.c` використовує визначення `square()` з цього файлу. Дане визначення, маючи зовнішнє зв'язування, є видимим для інших файлів, але `main()` у `file1.c` застосовує локальне визначення `static` функції `square()`. Оскільки це визначення також `inline`, то компілятор може (або ні) оптимізувати код, можливо, вбудувавши його. Нарешті, для файлу `file3.c` компілятор може вільно використовувати або вбудоване визначення з файлу `file3.c`, або визначення з зовнішнім зв'язуванням з файлу `file2.c` (або обидва!). Якщо ви не вкажете `static` у визначенні `inline`, як у файлі `file3.c`, то визначення `inline` розглядається в якості альтернативи, яка могла б застосовуватися замість зовнішнього визначення.

4.1.16. Специфікатор функції _Noreturn (C11)

Коли в стандарті **C99** з'явилося ключове слово **inline**, воно було єдиним прикладом специфікатора функції. Як ви пам'ятаєте, ключові слова **extern** і **static** називаються специфікаторами класу зберігання і можуть застосовуватися до об'єктів даних, а також до функцій. До стандарту **C11** був доданий другий **специфікатор функції** – **_Noreturn**, призначений для позначення функції, яка по завершенні не повертає керування до функції, що її викликала. Прикладом функції **_Noreturn** є **exit()**. Після звернення до неї функція, що її викликала, ніколи не поновить своє виконання. Зверніть увагу, що це відрізняється від типу **void**, що повертається. Типова функція **void** повертає керування функції, що викликає. Вона просто не надає їй будь-якого значення.

Мета застосування **_Noreturn** полягає в тому, щоб проінформувати користувача і компілятор, що конкретна функція не поверне керування програмі, що її викликала. Інформування користувача допомагає запобігти неправильному використанню функції, а позначення такого факту для компілятора може зробити можливим деякі оптимізації коду.

4.2. Бібліотека C

Спочатку офіційної бібліотеки **C** не існувало. Пізніше з'явився стандарт, який де-факто був оснований на реалізації **C** для **Unix**. Комітет **ANSI C**, у свою чергу, розробив офіційну стандартну бібліотеку, яка значною мірою базувалася на цьому стандарті де-факто. Враховуючи поширеність мови **C** в усьому світі, комітет потім вирішив перевизначити бібліотеку, щоб вона могла бути реалізована на великій кількості різних систем.

Ми з вами вже обговорювали деякі функції вводу-виводу, функції для обробки символів і функції для роботи з рядками з цієї бібліотеки. Сьогодні ми розглянемо ще декілька функцій з цієї бібліотеки, але спочатку поговоримо про те, як її використовувати.

4.2.1. Отримання доступу до бібліотеки C

Спосіб отримання доступу до бібліотеки `C` залежить від реалізації мови, тому вам необхідно ознайомитися з тим, наскільки більш загальні твердження можуть бути застосовані до вашої системи. По-перше, бібліотечні функції часто можна виявити в різних місцях. Наприклад, функція `getchar()` зазвичай визначена у вигляді макросу всередині файлу `stdio.h`, а функція `strlen()`, як правило, міститься у бібліотечному файлі `string.h`. По-друге, для різних систем передбачені різноманітні способи отримання доступу до цих функцій. Далі в загальному вигляді будуть представлені три можливості: автоматичний доступ, підключення файлів і підключення бібліотек.

Автоматичний доступ. В багатьох системах достатньо лише скомпілювати програму, оскільки найпоширеніші бібліотечні функції зроблені доступними автоматично.

Майте на увазі, що для функцій, які ви використовуєте, повинні бути оголошені їх типи. Зазвичай це можна зробити шляхом включення потрібного файлу заголовку. Файли, що підлягають включенню, описані в керівництвах користувача з бібліотечних функцій. Однак в деяких старих системах могла виникати необхідність у самостійному наборі оголошень функцій. В цьому випадку тип функції знову слід шукати в керівництві користувача.

Раніше імена файлів заголовків не були узгоджені між різними реалізаціями. Стандарт ANSI C групує бібліотечні функції в сімейства. Для кожного сімейства передбачено файл заголовку з прототипами функцій.

Підключення файлів. Якщо функція визначена у вигляді макросу, то за допомогою директиви `#include` можна включити файл, який містить її визначення. Часто схожі макроси збираються до файлу заголовку з відповідним іменем. Наприклад, з появою стандарту `ANSI C` компілятори `C` надходять з файлом `ctype.h`, що містить ряд макросів, які визначають природу символу: верхній регістр, цифра і т. п.

Підключення бібліотек. На певному етапі компіляції або компоновки програми може знадобитися вказати опцію бібліотеки. Навіть система, яка автоматично перевіряє свою стандартну бібліотеку, може мати інші бібліотеки функцій, що використовуються не так часто. Ці бібліотеки повинні викликатися явно з застосуванням опцій компілятора. Зверніть увагу, що даний процес відрізняється від включення файлу заголовку. Файл заголовку містить оголошення або прототипи функцій. Опція бібліотеки повідомляє системі де саме шукати код функцій.

4.2.2. Використання описів бібліотеки

Документацію по функціям можна знайти в декількох місцях. Система може мати он-лайн керівництво, а **IDE**-середовище часто має он-лайн довідку. Постачальники компіляторів **C** іноді надають керівництво користувача в друкованому вигляді, які містять опис бібліотечних функцій, або компакт-диск з аналогічним матеріалом. Чисельні видавництва випустили довідники функцій бібліотеки **C**. Одні з них мають загальну природу, а інші орієнтовані на певні реалізації мови **C**.

Під час перегляду документації важливо розуміти заголовки функцій. Опис з часом змінюється. Для прикладу розглянемо опис функції **fread()** однієї з бібліотек **C**:

Стандарт **ANSI C90** надає такий опис:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

Тип **size_t** визначений як цілочисловий тип без знаку, що повертається операцією **sizeof**. Зазвичай ним є тип **unsigned long**. Файл **stddef.h** містить визначення **typedef** або **#define** для **size_t**, як і декілька інших файлів, в тому числі **stdio.h**, зазвичай за рахунок включення **stddef.h**. Багато функцій, включаючи **fread()**, часто вбудовують операцію **sizeof** у вигляді частини

фактичного аргументу. Тип **size_t** забезпечує відповідність формального аргументу цьому загальному способу використання.

Крім того, в **ANSI C** застосовується вказівник на **void** в якості свого роду узагальненого вказівника для ситуацій, коли можуть використовуватися вказівники на різні типи даних.

Наприклад, першим аргументом функції **fread()** може бути вказівник на масив значень **double** або на деяку структуру. Якщо фактичний аргумент являє собою вказівник, скажімо, на масив з **20** значень **double**, а формальний аргумент є вказівником на **void**, то компілятор обере варіант для потрібного типу і не повідомить про невідповідність типів.

Нещодавно стандарти **C99/C11** впровадили в описи функцій нове ключове слово **restrict**:

```
#include <stdio.h>
size_t fread(void *restrict ptr, size_t size,
size_t nmemb, FILE *restrict stream);
```

А тепер перейдемо до огляду деяких специфічних функцій.

4.2.3. Бібліотека математичних функцій

Бібліотека математичних функцій містить багато зручних функцій такого роду. Їх оголошення або прототипи знаходяться в заголовному файлі **math.h**. В табл. 4.2 наведено декілька функцій, що оголошені в **math.h**. Зверніть увагу, що усі кути вимірюються в радіанах (один радіан складає **180/□ = 57.296** градусів).

Тригонометрія. Скористаємося бібліотекою математичних функцій для рішення типової задачі перетворення прямокутних координат в полярні (модуль і кут). Припустимо, що на квадратній сітці проведена лінія, протяжність якої складає **4** одиниці по горизонталі (значення **x**) і **3** одиниці по вертикалі (значення **y**).

Якими будуть довжина (модуль) і напрямок лінії? Відповідно до тригонометрії:

```
модуль = квадратний корінь (x² + y²);
кут = арктангенс (y/x).
```

Таблиця 4.2 – Деякі стандартні математичні функції **ANSI C**

Прототип	Опис
`double acos(double x)`	Повертає кут (від `0` до □ радіан), косинус якого дорівнює **x**
`double asin(double x)`	Повертає кут (від `-□/2` до `□/2` радіан), синус якого дорівнює **x**
`double atan(double x)`	Повертає кут (від `-□/2` до `□/2` радіан), тангенс якого дорівнює **x**
`double atan2(double y, double x)`	Повертає кут (від `-□` до `□` радіан), тангенс якого дорівнює **y/x**
`double cos(double x)`	Повертає косинус **x** (**x** в радіанах)
`double sin(double x)`	Повертає синус **x** (**x** в радіанах)
`double tan(double x)`	Повертає тангенс **x** (**x** в радіанах)
`double exp(double x)`	Повертає експоненціальну функцію **x** ( )
`double log(double x)`	Повертає натуральний логарифм **x**
`double log10(double x)`	Повертає логарифм **x** за основою `10`
`double pow(double x, double y)`	Повертає **x** в степені **y** ( )
`double sqrt(double x)`	Повертає квадратний корінь **x**
`double cbrt(double x)`	Повертає кубічний корінь **x**
`double ceil(double x)`	Повертає найменше ціле, яке не менше ніж **x**
`double fabs(double x)`	Повертає абсолютне значення **x**
`double floor(double x)`	Повертає найбільше ціле, яке не перевищує **x**

Бібліотека `math.h` надає функцію добування квадратного кореня та декілька функцій обчислення арктангенсу, тому ви можете відтворити дане рішення на мові **C**. Функція добування квадратного кореня – `sqrt()`, приймає аргумент `double` і повертає квадратний корінь аргументу також у вигляді значення `double`.

Функція `atan()` приймає аргумент `double` і повертає кут, значення тангенсу якого дорівнює цьому аргументу. Нажаль, функція `atan()` не враховує

квадрант вектора. Наприклад, якщо координати **x** і **y** вектору дорівнюють **-5** і **-5**, функція **atan()** дасть результат **45°**, оскільки **(-5)/(-5) = 1**. Той самий результат буде для вектору з координатами **5** і **5**. Іншими словами, функція **atan()** не розрізняє вектори, кути яких відрізняються на **180°**. Насправді функція **atan()** виводить результат в радіанах, а не в градусах. Ми обговоримо це перетворення пізніше.

На щастя, бібліотека **C** містить і функцію **atan2()**. Вона приймає два аргументи: значення **x** і **y**. Таким чином, функція спроможна аналізувати знаки координат і правильно визначати кут. На зразок **atan()**, функція **atan2()** повертає кут в радіанах.

Щоб перетворити радіани на градуси, помножте результуючий кут на **180** і поділіть на □. Обчислення значення кута □ можна доручити комп'ютеру, вказавши вираз **4 * atan(1)**. Усі описані дії продемонстровані в наступній програмі:

```
//============================================================
// Програма перетворює прямокутні координати на полярні
//============================================================
#include <stdio.h>
#include <windows.h>
#include <math.h>
#define RAD_TO_DEG (180/(4 * atan(1)))

typedef struct polar_v
{
double magnitude;
double angle; }
Polar_V;

typedef struct rect_v
{
double x;
double y;
} Rect_V;

Polar_V rect_to_polar(Rect_V);

int main(void)
{
    Rect_V input;
```

```
    Polar_V result;
    SetConsoleOutputCP(1251);
    puts("Введіть координати x і y. Для виходу введіть q.");
     while(scanf("%lf %lf", &input.x, &input.y) == 2)
    {
        result = rect_to_polar(input);
printf("Модуль = %0.2f, кут = %0.2f\n",
        result.magnitude, result.angle);
    }
    puts("Програма завершена.");
return 0;
}
 Polar_V rect_to_polar(Rect_V rv)
{
    Polar_V pv;
    pv.magnitude = sqrt(rv.x * rv.x + rv.y * rv.y);
if(pv.magnitude == 0)
     pv.angle = 0.0;
else
        pv.angle = RAD_TO_DEG * atan2(rv.y, rv.x);
return pv;
}
```

Результат роботи програми наведено на рис. 4.13.

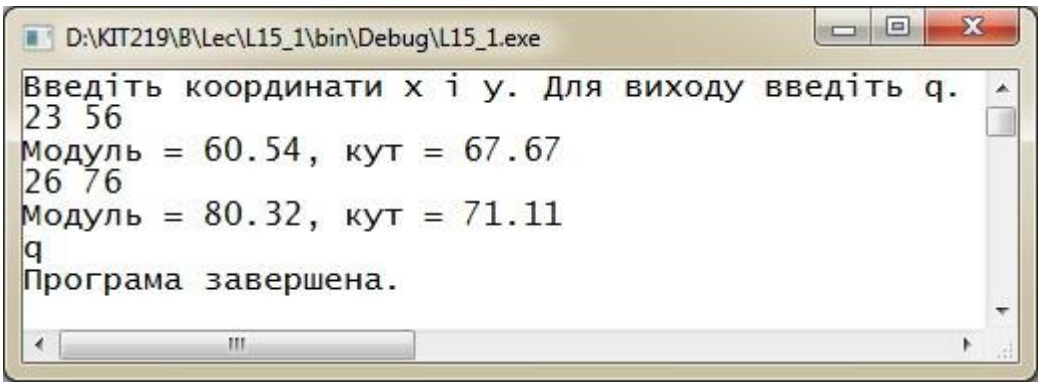


Рисунок 4.13 – Результат роботи програми для перетворення прямокутних координат на полярні

Варіанти типів. Базові математичні функції з рухомою крапкою приймають аргументи типу `double` і повертають значення типу `double`. Їм можна передавати аргументи типу `float` або `long double`, і функції як і раніше будуть працювати, оскільки аргументи вказаних типів перетворюються на тип

double. Це дуже зручно, але не обов'язково оптимально. Якщо подвійна точність не потрібна, то обчислення можуть виконуватися швидше, якщо застосовувати значення **float** з одинарною точністю. До того ж значення типу **long double** буде втрачати точність при передачі параметра типу **double**. Може навіть виявитися, що значення взагалі неможливо представити. Щоб вирішити такі потенційні проблеми, в стандарті **C** є версії стандартних функцій типу **float** і типу **long double**, що мають в імені суфікс **f** або **l** (заголовна буква **'L'**). Таким чином, **sqrtf()** – це версія типу **float** функції **sqrt()**, а **sqrtl()** – версія типу **long double** функції **sqrt()**.

Поява в стандарті **C11** виразу узагальненого вибору дозволяє визначати узагальнений макрос, який обирає найбільш прийнятну версію математичної функції на основі типу аргументу. В наступній програмі продемонстровані два різні підходи до визначення узагальнених макросів.

```
//=========================================================
// Визначення узагальнених макросів
//=========================================================
#include <stdio.h>
#include <math.h>
#define RAD_TO_DEG (180/(4 * atanl(1)))

// узагальнена функція добування квадратного кореня
#define SQRT(X) _Generic((X),          \
long double: sqrtl,                    \
default: sqrt,                         \
float: sqrtf)(X)

// узагальнена функція обчислення синусу кута,
// який заданий в градусах
#define SIN(X) _Generic((X),          \
long double: sinl((X)/RAD_TO_DEG),    \
default: sin((X)/RAD_TO_DEG),         \
float: sinf((X)/RAD_TO_DEG)           \
)
 int
main(void)
{
float x          = 45.0f;
double xx        = 45.0;
long double xxx = 45.0L;
```

```
long double y   = SQRT(x);
long double yy  = SQRT(xx);
long double yyy = SQRT(xxx);

    __mingw_printf("%.17Lf\n", y);     // відповідає float
    __mingw_printf("%.17Lf\n", yy);    // відповідає default
    __mingw_printf("%.17Lf\n", yyy); // відповідає long double
    int i = 45;
    yy  = SQRT(i);                     // відповідає default
    __mingw_printf("%.17Lf\n", yy);
    yyy = SIN(xxx);                  // відповідає long double
    __mingw_printf("%.17Lf\n", yyy);
return 0; }
```

Результат роботи програми наведено на рис. 4.14.

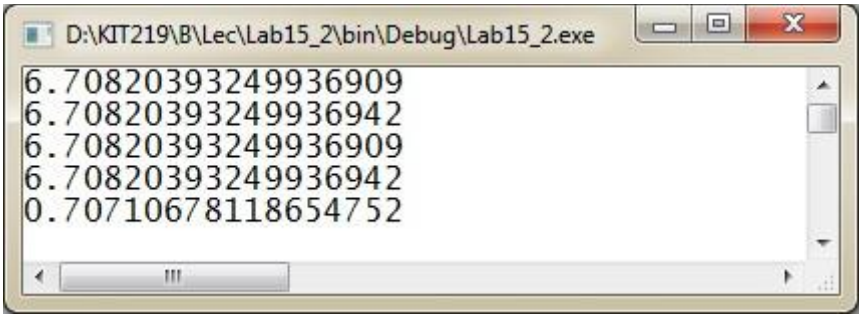


Рисунок 4.14 – Результат роботи програми для визначення узагальнених макросів

Як можна побачити, `SQRT(i)` має таке ж саме значення, що повертається, як і `SQRT(xx)`, оскільки типи обох аргументів (`int` і `double`) відповідає мітці `default`.

Яким же чином можна змусити макрос, що застосовує `_Generic`, діяти на зразок функції. У визначенні `SIN()` застосований, мабуть, найбільш очевидний підхід: кожне значення, яке позначене міткою, являє собою виклик функції, тому значенням виразу `_Generic` є окремий виклик функції, на зразок `sinf((X)/RAD_TO_DEG)`, з аргументом `SIN()`, що замінює `X`.

Визначення для `SQRT()` є більш елегантним. В ньому значення виразу `_Generic` – це ім'я функції, таке як `sinf`. Це ім'я функції замінюється на її адресу, тому значенням виразу `_Generic` буде вказівник на функцію. Однак за

повним виразом `_Generic` йде `(x)`, і комбінація `вказівник-на-функцію (аргумент)` викликає вказану функцію з заданим аргументом.

Тобто для `SIN()` виклик функції знаходиться всередині виразу узагальненого вибору, в той час як для `SQRT()` вираз узагальненого вибору оцінюється як вказівник, який потім застосовується для виклику функції.

4.2.4. Бібліотека tgmath.h (C99)

Стандарт `C99` пропонує файл заголовку `tgmath.h`, в якому визначені макроси узагальненого типу, що за своєю дією схожі на розглянуті в попередній програмі. Якщо будь-яка функція `math.h` визначена для кожного з трьох типів `float`, `double` і `long double`, то файл `tgmath.h` створює макрос узагальненого типу з тим самим іменем, що й у версії для `double`. Наприклад, він визначає макрос `sqrt()`, який перетворюється на функцію `sqrtf()`, `sqrt()` або `sqrtl()` в залежності від типу наданого аргументу. Іншими словами, макрос `sqrt()` веде себе на зразок макросу `SQRT()` з попередньої програми.

Якщо компілятор підтримує арифметику комплексних чисел, то для нього буде доступним файл `complex.h`, в якому оголошені комплексні аналоги математичних функцій. Наприклад, в цьому файлі оголошені функції `csqrtf()`, `csqrt()` і `csqrtl()`, які повертають комплексний квадратний корінь типу `float complex`, `double complex` і `long double complex`, відповідно. Коли така підтримка надається, макрос `sqrt()` з `tgmath.h` також може розгортатися в зв'язану функцію комплексного квадратного кореня.

Якщо ви бажаєте викликати, наприклад, функцію `sqrt()` замість макросу `sqrt()`, навіть незважаючи на те, що файл `tgmath.h` включено, можете помістити ім'я функції в круглі дужки:

```
#include <tgmath.h>
...
float x = 44.0;
double y;
y = sqrt(x);          // виклик макросу, відповідно sqrtf()
y = (sqrt)(x);        // виклик функції sqrt()
```

Код працює, оскільки ім'я функціонального макросу повинне супроводжуватися подальшою відкривальною круглою дужкою, що обходиться шляхом розміщення імені в круглих дужках. В протилежному випадку круглі дужки не впливають на вираз, що знаходиться всередині них, крім зміни порядку слідування операцій. Тому розміщення в дужках імені функції як і раніше призводить до її виклику. Крім того, для виклику функції `sqrt()` можна також використовувати `(*sqrt)()`.

Засіб виразів `_Generic`, який був доданий до стандарту **C11**, є простим способом реалізації макросів `tgmath.h`, не вдаючись до механізмів, які виходять за рамки стандарту **C**.

4.2.5. Бібліотека утиліт загального призначення

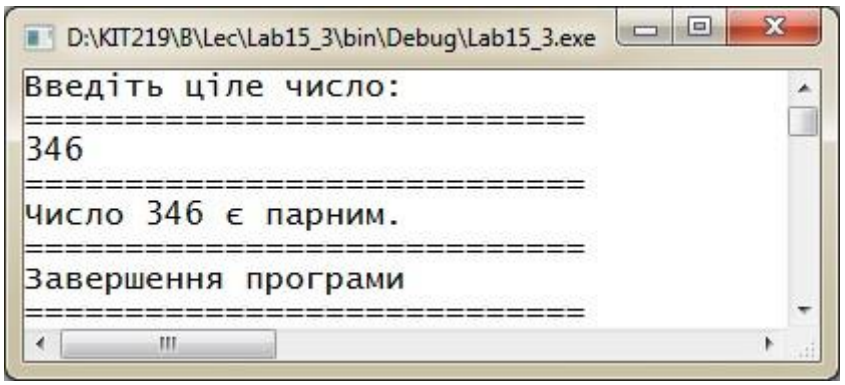
Бібліотека утиліт загального призначення містить велику кількість функцій, включаючи генератор випадкових чисел, функції для пошуку та сортування, функції для перетворення і функції для керування пам'яттю. Раніше ви вже перевіряли роботу функцій `rand()`, `srand()`, `malloc()` і `free()`. В **ANSI C** прототипи цих функцій знаходяться в заголовному файлі `stdlib.h`. Розглянемо деякі функції з цього файлу більш докладно.

Функції `exit()` і `atexit()`. Функція `exit()` вже застосовувалася явно в декількох прикладах. До того ж, функція `exit()` викликається автоматично при поверненні з `main()`. До стандарту **ANSI** додана пара цікавих можливостей, які ми ще не використовували. Найважливішою з них є можливість зазначення певних функцій, які повинні викликатися під час виконання `exit()`. Функція `atexit()` забезпечує це за рахунок реєстрації функцій, призначених для виклику при виконанні `exit()`.

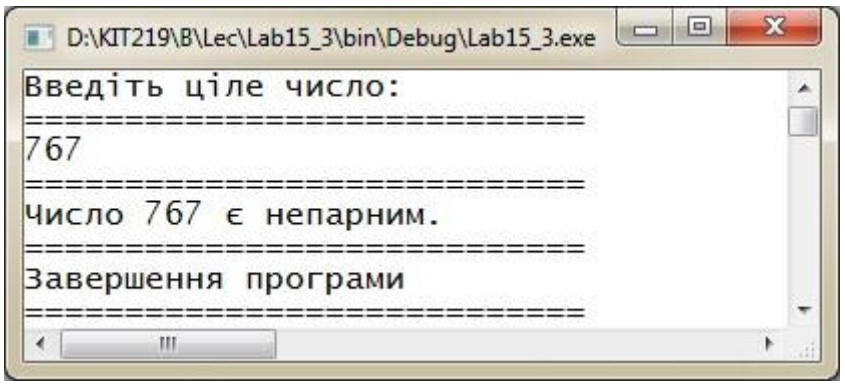
Функція `atexit()` приймає в якості аргументу вказівник на функцію.

```c
//=========================================================
// Приклад застосування функції atexit()
//=========================================================
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);

int main(void)
{
int n;
SetConsoleOutputCP(1251);
atexit(sign_off);     // реєстрація функції sign_off()
puts("Введіть ціле число:");
printf("============================\n");
if(scanf("%d", &n) != 1)
    {
        printf("============================\n");
        puts("Це не ціле число!");
        atexit(too_bad); // реєстрація функції too_bad()
printf("============================\n");
exit(EXIT_FAILURE);
    }
    printf("============================\n");
printf("Число %d є %s.\n", n,
          (n % 2 == 0)? "парним" : "непарним");
printf("============================\n");
return 0;
}

void sign_off(void)
{
    puts("Завершення програми");

printf("============================\n"); }

void too_bad(void)
{
    puts("Виникли певні непорозуміння");
}
```

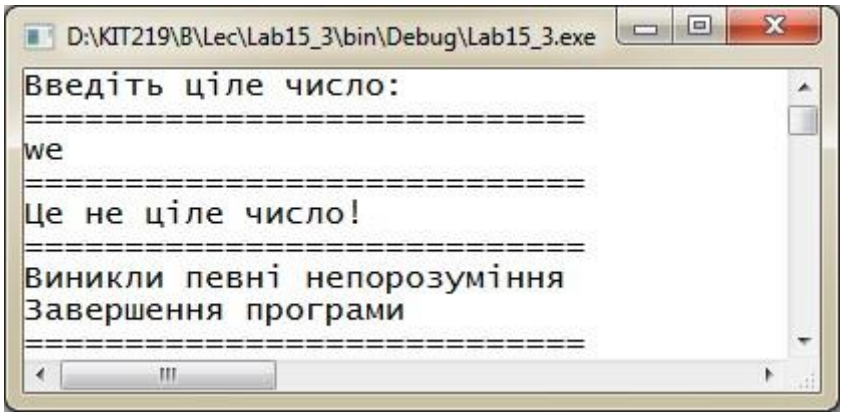
Результат роботи програми для трьох різних варіантів наведено на рис. 4.15.



а



б



в

Рисунок 4.15 – Результат роботи програми для застосування функції `atexit()` для трьох різних варіантів

Використання функції `atexit()`. Функція `atexit()` приймає вказівники на функції. Щоб використовувати її, просто необхідно передати адресу функції, яка повинна бути викликана при виході. Оскільки ім'я функції діє як адреса, коли застосовується в якості аргументу функції, для аргументу можна вказувати ім'я `sign_off` або `too_bad`. Потім `atexit()` реєструє цю

функцію у списку функцій, які призначені для виконання під час виклику `exit()`. Стандарт **ANSI** гарантує, що список може містити не менш ніж `32` функції. Кожна з них додається за допомогою окремого виклику `atexit()`. Коли функція `exit()`, нарешті, викликається, вона виконує ці функції, причому першою виконується функція, що була додана до списку останньою.

Зверніть увагу, що в результаті неприпустимого вводу користувача викликаються обидві функції – `sign_off()` і `too_bad()`, але у випадку вводу допустимого значення викликається тільки `sign_off()`. Справа в тому, що оператор `if` реєструє функцію `too_bad()` тільки для випадку недопустимого вводу. Крім того, першою була викликана функція, що зареєстрована останньою.

Давайте розглянемо два основних питання: застосування функції `atexit()` і аргументи функції `exit()`.

Функції, що реєструються `atexit()`, на зразок `sign_off()` і `too_bad()`, повинні мати тип `void` і не приймати аргументів. Зазвичай вони вирішують допоміжні задачі, такі як оновлення файлу моніторингу програми або переустановлення змінних середовища.

Зверніть увагу, що `sign_off()` викликається навіть у випадку, коли функція `exit()` не викликається явно. Причина полягає в тому, що `exit()` неявно викликається при завершенні `main()`.

Використання функції `exit()`. Коли `exit()` виконує функції, що вказані за допомогою `atexit()`, вона вживає власні кроки щодо очищення. Функція `exit()` скидає всі потоки виводу, закриває всі відкриті потоки і закриває тимчасові файли, що були створені в результаті звернень до стандартної функції вводу-виводу `tmpfile()`. Потім `exit()` повертає керування середовищу та, за можливістю, повідомляє середовищу стан завершення. Традиційно програми для **Unix** застосовували `0`, щоб повідомити про успішне завершення, і ненульове значення для повідомлення про відмову. Коди повернення **Unix** не обов'язково працюють з усіма системами, тому в **ANSI C** визначено макрос на ім'я `EXIT_FAILURE`, який може використовуватися на різних системах для позначення

відмови. Так само, в **ANSI C** визначено макрос **EXIT_SUCCESS** для повідомлення про успішне завершення, але **exit()** також приймає для цього значення **0**. В рамках стандарту **ANSI C** застосування **exit()** в нерекурсивній функції **main()** еквівалентно використанню ключового слова **return**. Проте, **exit()** також завершує програму, яка застосовується у функціях, що відрізняються від **main()**.

Функція **qsort().** Метод швидкого сортування входить до числа найбільш ефективних алгоритмів сортування, особливо у випадку великих масивів. Розроблений Чарльзом Ентоні Річардом Гоаром (англ. **Charles Antony Richard Hoare**) у 1962 році, цей алгоритм розділяє масиви на частини, що постійно зменшуються, поки не буде досягнуто рівня елемента. Спочатку масив поділяється на дві частини таким чином, що будь-яке значення в одній частині менше будь-якого значення в іншій частині. Цей процес продовжується аж до моменту, коли масив не стане повністю відсортованим.

Алгоритм швидкого сортування реалізований в **C** під іменем **qsort()**. Ця функція сортує масив об'єктів даних. Вона має наступний прототип **ANSI**:

```
void qsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));
```

Перший аргумент являє собою вказівник на початок масиву, що підлягає сортуванню. Стандарт **ANSI C** дозволяє приведення типу вказівника на дані до типу вказівника на **void**, таким чином, дозволяючи першому фактичному аргументу **qsort()** посилатися на масив будь-якого виду.

У другому аргументі передається кількість елементів, що підлягають сортуванню. Прототип перетворює це значення на тип **size_t**. Як вже декілька разів згадувалося, **size_t** є цілочисловим типом, який повертається операцією **sizeof** і визначений в стандартних файлах заголовку.

Через те, що функція **qsort()** перетворює свій перший аргумент на вказівник на тип **void**, вона втрачає інформацію про розмір кожного елемента масиву. Щоб компенсувати це, слід явно повідомити для **qsort()** розмір об'єкту даних. Саме для цього призначений третій аргумент. Наприклад, якщо ви

сортуєте масив типу `double`, то в третьому аргументі слід вказувати `sizeof(double)`.

Нарешті, `qsort()` потрібен вказівник на функцію, яка буде використовуватися для визначення порядку сортування. Функція порівняння повинна приймати два аргументи – вказівники на два елемента, що порівнюються. Вона повертає додатне ціле число, якщо перший елемент повинен йти за другим, нуль, якщо елементи є однаковими, та від'ємне ціле число, якщо другий елемент повинен йти за першим. Функція `qsort()` викликає цю функцію з передачею їй значень вказівників, які обчислює на основі іншої наведеної інформації.

Форма функції порівняння задана в останньому аргументі прототипу `qsort()`:

```
int (*compar)(const void *, const void *)
```

Тут видно, що останній аргумент `qsort()` являє собою вказівник на функцію, яка повертає значення `int` і приймає два аргументи, кожен з яких є вказівником на тип `const void`. Ці два вказівники посилаються на елементи, що порівнюються.

Наступна програма і подальші міркування ілюструють спосіб визначення функції порівняння та застосування функції `qsort()`. В програмі створюється масив випадкових значень з рухомою крапкою, який потім сортується. Програма використовує швидке сортування для впорядкування груп чисел.

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#define NUM 40
void fillarray(double ar[], int n);
void showarray(const double ar[], int n);
int mycomp(const void *p1, const void *p2);
int main(void)
{
double vals[NUM];
SetConsoleOuputCP(1251);
fillarray(vals, NUM);
puts("Список випадкових чисел:");
showarray(vals, NUM);
```

```
qsort(vals, NUN, sizeof(double), mycomp);
puts("\nВідсортований список:");
showarray(vals, NUM);
return 0;
}

void fillarray(double ar[], int n)
{
    int index;
    for(index = 0; index < n; index++)
        ar[index] = (double)rand()/((double)rand() + 0.1);
}
 void showarray(const double ar[], int n)
{
    int index;
    for(index = 0; index < n; index++)
    {
        printf("%9.4f ", ar[index]);
if(index % 6 == 5)
putchar('\n');
    }
    if(index % 6 !=0)
putchar('\n');
}

// сортування за зростанням
int mycomp(const void *p1, const void *p2)
{
    // для доступу до значень необхідно використовувати
    // вказівники на double
const double *a1 = (const double *) p1;
const double *a2 = (const double *) p2;
if(*a1 < *a2)
     return -1;
else if(*a1 == *a2)
      return 0;
else
    return 1;
}
```

Результат виконання програми наведено на рис. 4.16.

Розглянемо два ключових моменти: використання `qsort()` та визначення `mycomp()`.

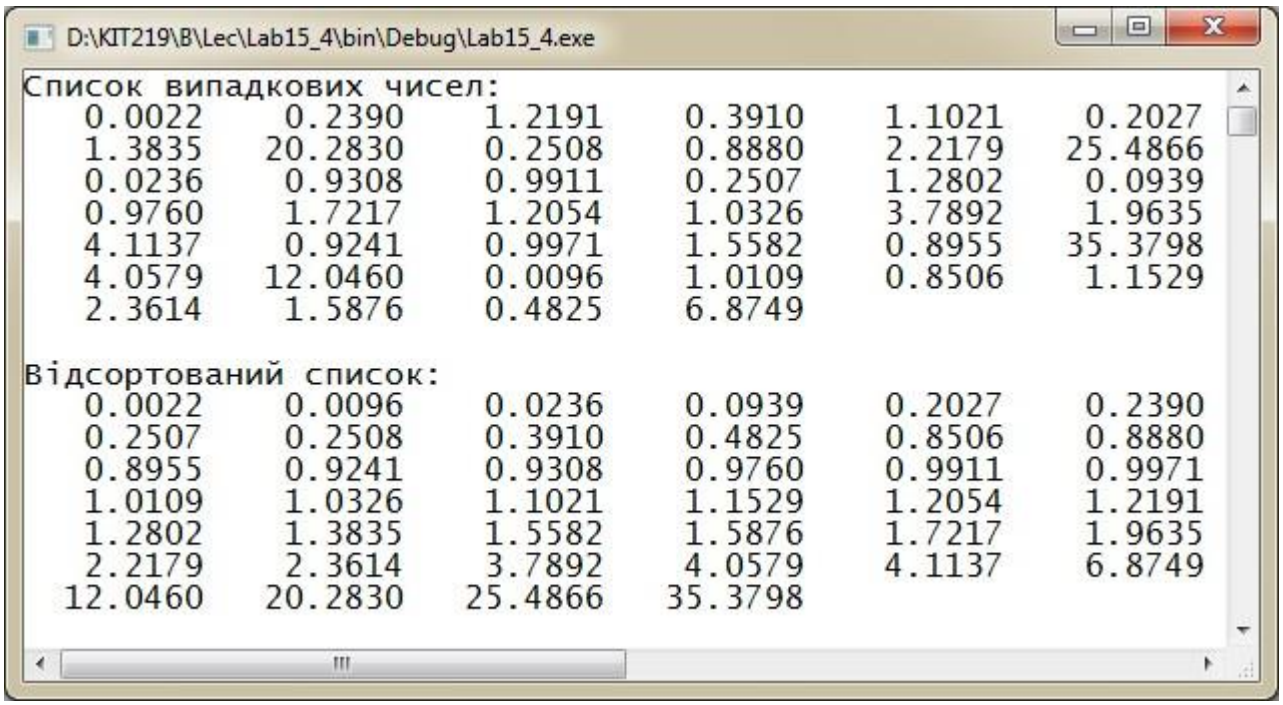


Рисунок 4.16 – Результат роботи програми для демонстрації швидкого сортування для упорядкування чисел за зростанням

Використання функції `qsort()`. Функція `qsort()` сортує масив об'єктів даних. Її прототип **ANSI** має наступний вигляд:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Перший аргумент – це вказівник на початок масиву, що підлягає сортуванню. В програмі застосовується фактичний аргумент `vals`, який являє собою ім'я масиву типу `double`. Таким чином, він є вказівником на перший елемент масиву. В прототипі **ANSI** для аргументу `vals` передбачено приведення до типу вказівника на `void`. Причина полягає в тому, що стандарт **ANSI C** дозволяє приводити будь-який тип вказівника на дані до типу вказівника на `void`, тим самим дозволяючи першому фактичному аргументу в `qsort()` посилатися на масив будь-якого виду.

У другому аргументі задається кількість елементів, що призначені для сортування. В розглянутій програмі це `N`, тобто число елементів масиву. Прототип перетворює це значення на тип `size_t`.

Третій аргумент – це розмір кожного елемента, в даному випадку `sizeof(double)`.

Останнім аргументом, **mycomp**, є адреса функції, яка повинна використовуватися для порівняння елементів.

Визначення функції `mycomp()`. Як вже згадувалося раніше, прототип `qsort()` встановлює форму функції порівняння:

```
int (*compar)(const void *, const void *);
```

Тут видно, що останній аргумент є вказівником на функцію, яка повертає значення `int` і приймає два аргументи. Кожен з цих аргументів являє собою вказівник на тип `const void`. Ми привели у відповідність до нього прототип функції `mycomp()`:

```
int mycomp(const void *p1, const void *p2);
```

Згадайте, що ім'я функції, яке передається в якості аргументу, є вказівником на неї, тому `mycomp` збігається з прототипом `compar`.

Функція `qsort()` передає у функцію порівняння адреси двох елементів, що порівнюються. В цій програмі змінним `p1` і `p2` присвоюються адреси двох значень типу `double`, що призначені для порівняння. Зверніть увагу, що перший аргумент в `qsort()` посилається на масив в цілому, а два аргументи функції порівняння посилаються на два елементи в масиві. Тут виникає проблема. Щоб порівняти значення, для яких доступними є тільки вказівники, ці вказівники необхідно розіменувати. Оскільки значення мають тип `double`, вказівники повинні бути розіменовані в тип `double`. Однак функції `qsort()` потрібні вказівники на тип `void`. Обійти проблему можна, оголосивши вказівники потрібного типу всередині функції та ініціалізувавши їх значеннями, які передаються в аргументах:

```
// сортування за зростанням
int mycomp(const void *p1, const void *p2)
{
    // для доступу до значень необхідно використовувати
    // вказівники на double
```

```
const double *a1 = (const double *) p1;
const double *a2 = (const double *) p2;
if(*a1 < *a2)
      return -1;
else if(*a1 == *a2)
    return 0;
else
    return 1;
}
```

Для досягнення універсальності в **qsort()** і в функції порівняння застосовуються вказівники на **void**. Як наслідок, функції доведеться явно повідомити розмір кожного елемента масиву, а всередині визначення функції порівняння перетворити аргументи типу «вказівник на **void**» на «вказівники на потрібний тип даних».

Давайте поглянемо на ще один приклад функції порівняння. Припустимо, що ми маємо наступні оголошення:

```
struct names
{
    char first[40];
    char last[40];
};
struct names staff[100];
```

Як повинен виглядати виклик **qsort()**? Відповідно до моделі, що реалізована в розглянутій раніше програмі, виклик міг би мати наступний вид:

```
qsort(staff, 100, sizeof(struct names), comp);
```

Тут **comp** являє собою ім’я функції порівняння. На що повинна бути схожою ця функція? Нехай необхідно виконати сортування за прізвищем, а потім за ім’ям.

Можна було б написати наступну функцію:

```
#include <string.h>

int comp(const void *p1, const void *p2) // обов’язкова форма
{
// отримання правильного типу вказівника
const struct names *ps1 = (const struct names *) p1;
const struct names *ps2 = (const struct names *) p2;
int res;
```

```
res = strcmp(ps1->last, ps2->last);    // порівняння прізвищ
if(res != 0)
     return res;
    else          // прізвища збігаються, тому порівнюємо імена
    return strcmp(ps1->first, ps2->first); }
```

В даній функції порівняння здійснюються за допомогою функції `strcmp()`, яка повертає значення, що задовольняють вимогам до функції порівняння. Зверніть увагу, що для застосування операції `->` потрібен вказівник на структуру.

4.2.6. Бібліотека тверджень

Бібліотека тверджень, що підтримується файлом заголовку `assert.h` – це невелика бібліотека, яка призначена для підтримки процесу налагодження програми. Вона складається з макросу `assert()`, який приймає в якості аргументу цілочисловий вираз. Якщо вираз оцінюється як хибний (не нульовий), макрос `assert()` виводить в стандартний потік помилок (`stderr`) повідомлення про помилку і викликає функцію `abort()`, яка припиняє виконання програми. Прототип функції `abort()` знаходиться у файлі заголовку `stdlib.h`. Ідея полягає в тому, щоб ідентифікувати критичні місця в програмі, де повинні бути істинними певні умови, і за допомогою оператора `assert()` завершувати програму, якщо одна з зазначених умов порушується. Зазвичай аргументом є вираз відношення або логічний вираз. Коли `assert()` припиняє виконання програми, то спочатку відображається тест, який не пройшов перевірку, а потім ім’я файлу, що містить цей тест, і номер рядка, де знаходиться тест.

Використання `assert()`.В наступній програмі наведено простий приклад застосування `assert()`. В ньому стверджується, що значення `z` повинне бути більшим за `0` або дорівнювати `0`, перш ніж ми будемо намагатися добувати з нього квадратний корінь. Крім того, помилково виконується віднімання значення замість його додавання, роблячи можливим отримання змінною `z` неприпустимого значення.

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <assert.h>
 int main(void)
{
    double x, y, z;
    SetConsoleOutputCP(1251);
    puts("Введіть пару чисел (0 0 для завершення): ");
while(scanf("%lf %lf", &x, &y) == 2
          && (x != 0 || y != 0))
    {
        z = x * x - y * y;       // повинно бути +
assert(z >= 0);
printf("Результатом є %f\n", sqrt(z));
puts("Введіть наступну пару чисел: ");
    }
    puts("Програма завершена.");
return 0;
}
```

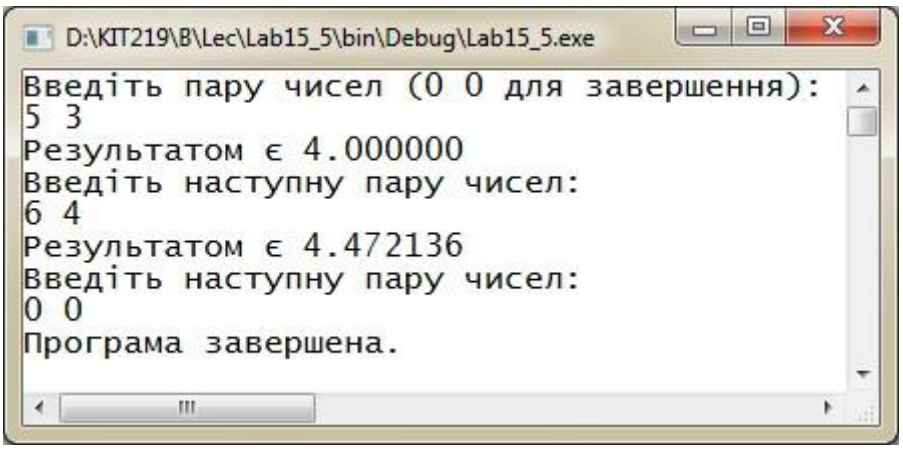
Результат виконання програми наведено на рис. 4.17 (а, б).

Точний текст в останньому рядку (рис. 4.17, б) залежить від компілятора. В повідомленні не говориться про умову **z >= 0**. Замість цього в ньому повідомляється про те, що відмовило твердження **z >= 0**.

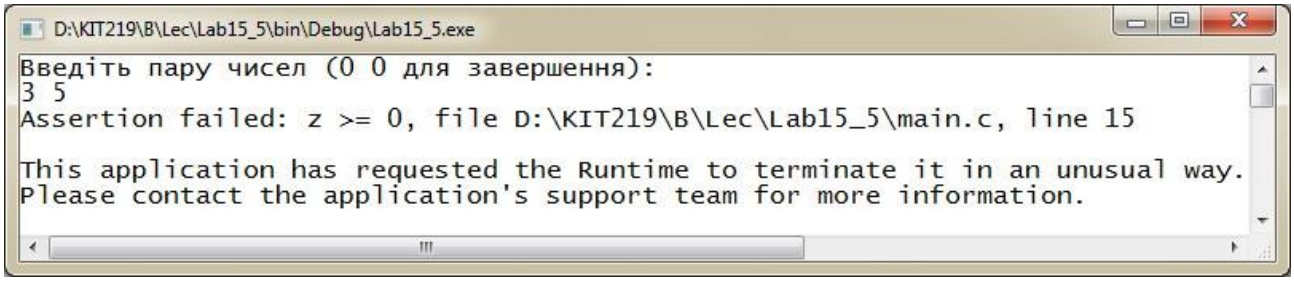
Щось схоже можна було отримати за допомогою оператора **if**:

```
if(z < 0)
{
    puts("z менше 0");
abort();

}
```

Проте, підхід з **assert()** має ряд переваг. Він автоматично ідентифікує файл і номер рядка, де виникла проблема. Нарешті, існує механізм включення та відключення макросу **assert()** без необхідності зміни коду. Якщо ви вважаєте, що вилучили помилки в програмі, розмістіть наступне визначення макросу **#define NDEBUG** перед місцем включення файлу **assert.h**, повторно скомпілюйте програму, і компілятор відключить у файлі усі оператори **assert()**. Якщо проблема виникне знову, можете видалити директиву **#define** (або закоментувати її) і провести повторну компіляцію. В результаті усі оператори **assert()** знову активізуються.



а



б

Рисунок 4.17 – Результат роботи програми для демонстрації застосування функції `assert()`

Ключове слово `_Static_assert` (C11). Вираз `assert()` перевіряється під час виконання. В **C11** з'явився новий засіб у формі оголошення `_Static_assert`, яке здійснює перевірку на етапі компіляції. Таким чином, `assert()` може призвести до переривання роботи програми, тоді як `_Static_assert` може стати причиною того, що програма не компілюється. Оголошення `_Static_assert` приймає два аргументи. Першим з них є цілочисловий константний вираз а другим – рядок. Якщо вираз є `0` (або `_False`), то компілятор відобразить рядок, і не буде компілювати програму. Розглянемо короткий приклад, після чого поглянемо на відмінності між `assert()` і `_Static_assert`. Текст програми має наступний вигляд:

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

```
_Static_assert(CHAR_BIT == 16,
               "Помилково передбачається 16-бітовий тип char");
int main(void)
{
    SetConsoleOutputCP(1251);
    puts("Тип char має 16 бітів.");
return 0;
}
```

Намагання проведення компіляції призведе до помилки. Якщо вираз `CHAR_BIT == 16` змінити на `CHAR_BIT == 8`, то програма успішно виконається (рис. 4.18).

Синтаксично `_Static_assert` трактується як оператор оголошення. Відповідно, на відміну від більшості різновидів операторів **C**, він може знаходитися або у функції, або (як в даному випадку) бути зовнішнім по відношенню до функції.

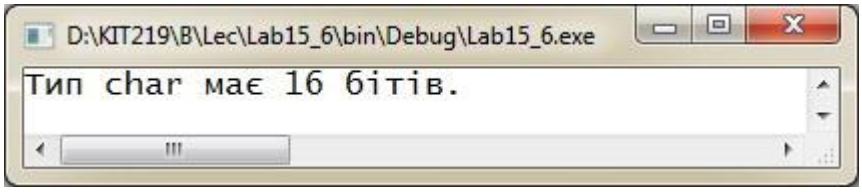


Рисунок 4.18 – Результат виконання програми для демонстрації `_Static_assert`

Вимога того, що першим аргументом в `_Static_assert` повинен бути цілочисловий константний вираз, гарантує можливість його оцінки на етапі компіляції.

У файлі заголовку `assert.h` ідентифікатор `static_assert` визначений як псевдонім для ключового слова `_Static_assert`. Це робить **C** більш сумісною з мовою **C++**, в якому для розглянутої можливості `static_assert` використовується в якості ключового слова.

4.2.7. Функції memcpy() і memmove() з бібліотеки string.h

Присвоювати один масив іншому неможливо, тому в таких випадках ми застосовували цикли для поелементного копіювання одного масиву в інший. Єдине виключення полягало в тому, що для символьних масивів ми використовували функції `strcpy()` і `strncpy()`. Функції `memcpy()` і `memmove()` пропонують майже такі самі послуги для інших видів масивів. Розглянемо прототипи цих функцій:

```
void *memcpy(void *restrict s1, const void *restrict s2,
size_t n); void *memmove(void *s1, const void *s2, size_t n);
```

Обидві функції копіюють `n` байтів з області, на яку вказує аргумент `s2`, в область, що вказана аргументом `s1`, і обидві повертають значення `s1`. Різниця між цими двома функціями, як зазначає ключове слово `restrict`, пов'язана з тим, що `memcpy()` дозволено вважати, що дві області пам'яті ніде не перекриваються одна з одною. Функція `memmove()` не робить такого припущення, тому копіювання відбувається таким чином, нібито усі байти спочатку поміщаються в тимчасовий буфер і тільки потім копіюються в область призначення. Що відбудеться, якщо застосувати `memcpy()` для областей, що перекриваються? В цьому випадку поведінка функції невизначена, тобто вона може як працювати, так і не працювати. Компілятор не забороняє використання функції `memcpy()`, коли це робити не слід, тому саме програміст несете відповідальність за забезпечення того, що області пам'яті не перекриваються.

Оскільки ці функції призначені для роботи з будь-яким типом даних, два їх аргументи мають тип вказівника на `void`. В `C` дозволено присвоювати вказівнику типу `void *` вказівник будь-якого типу. Зворотний бік такої гнучкості полягає в тому, що функції не здатні розпізнати, якого саме типу дані копіюються. Тому в них є третій аргумент, який задає кількість об'єктів, що копіюються. Зверніть увагу, що для масиву кількість байтів в загальному випадку не збігається з кількістю елементів. Таким чином, при копіюванні масиву з `10` значень `double` в якості третього аргументу слід застосовувати вираз `10*sizeof(double)`, а не `10`.

В наступній програмі показані приклади використання цих двох функцій. В ній припускається, що тип **double** має в два рази більший розмір, ніж **int**.

```
#include <stdio.h>
#include <windows.h>
#include <string.h> #define SIZE 10
void show_array(const int ar[], int n);

int main(void)
{
int values[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int target[SIZE];
double curious[SIZE/2] =
                { 2.0, 2.0e5, 2.0e10, 2.0e20, 5.0e30 };
    SetConsoleOutputCP(1251);
puts("Застосування функції memcpy():");
printf("================================================\n");
puts("Значення на вході:");
printf("================================================\n");
show_array(values, SIZE);
printf("================================================\n");
memcpy(target, values, SIZE * sizeof(int));
puts("Копія значень:");
printf("================================================\n");
show_array(target, SIZE);
printf("================================================\n\n");
puts("Застосування memmove() з областями, що "
         "перетинаються:");
printf("================================================\n");
memmove(values + 2, values, 5 * sizeof(int));
puts("Елементи 0-4 скопійовані в елементи 2-6:");
printf("================================================\n");
show_array(values, SIZE);
printf("================================================\n\n");
puts("Застосування memcpy() для копіювання double в int:");
memcpy(target, curious, (SIZE/2) * sizeof(double));
printf("================================================\n");
puts("5 значень double в 10 позиціях int:");
printf("================================================\n");
show_array(target, SIZE/2);     show_array(target + 5,
SIZE/2);
printf("================================================\n");
return 0;
}
void show_array(const int ar[], int n)
{
    for(int i = 0; i < n; i++)
        printf("%d ", ar[i]);
printf("\n");
}
```

Результат роботи програми наведено на рис. 4.19.

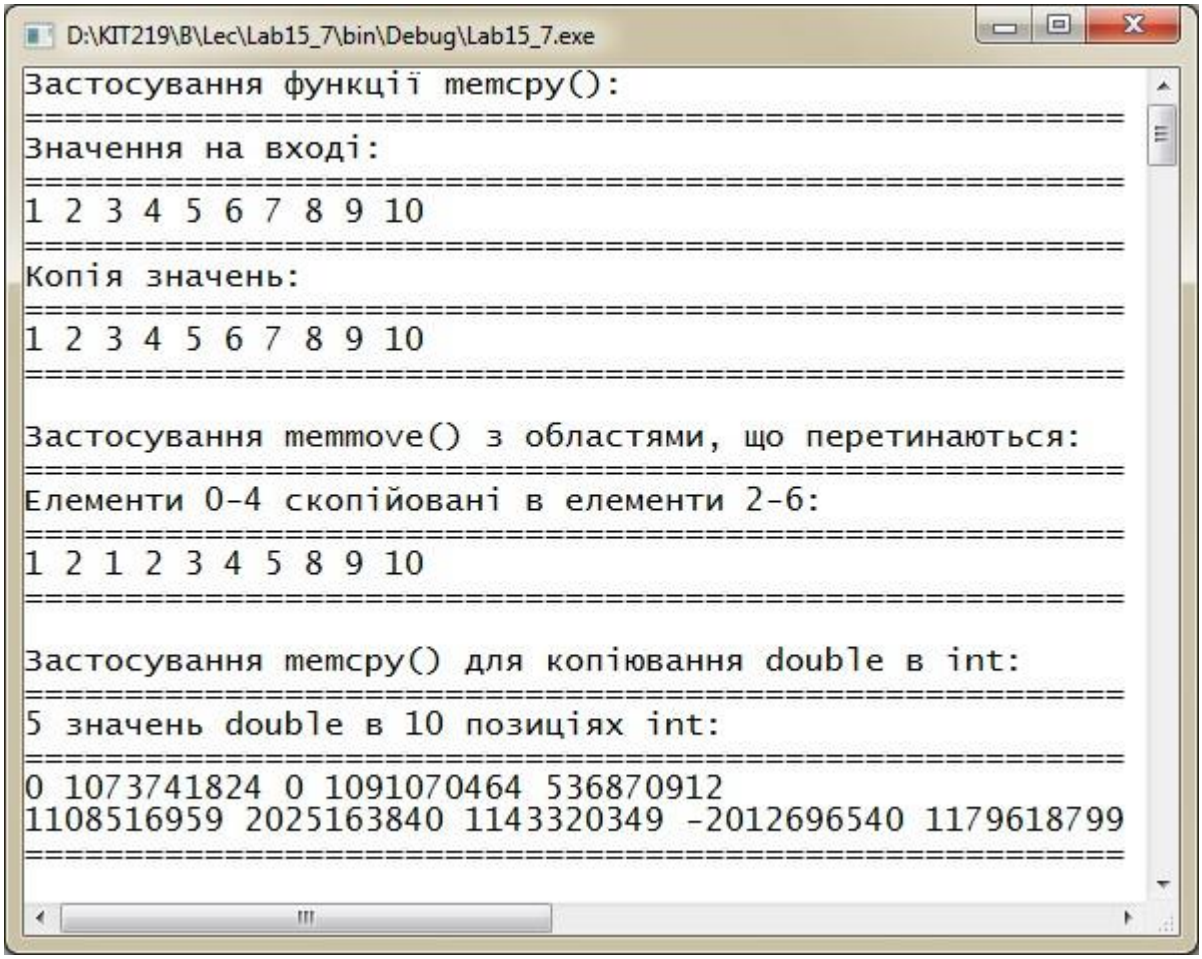


Рисунок 4.19– Результат виконання програми для демонстрації функцій **`memcpy()`** і **`memmove()`**

Останній виклик **`memcpy()`** копіює дані з масиву типу **`double`** до масиву типу **`int`**. Це демонструє той факт, що функція **`memcpy()`** нічого не знає, та й не бажає знати, про типи даних. Вона просто копіює байти з однієї області до іншої. Ви могли б, наприклад, копіювати байти зі структури до масиву символів. Крім того, жодного перетворення даних не відбувається. Якщо організувати цикл, що виконує поелементне присвоювання, то значення типу **`double`** будуть перетворені на тип **`int`**. В цьому випадку байти копіюються в тому вигляді, як вони є, і програма потім інтерпретує комбінації бітів таким чином, як ніби вони мали тип **`int`**.

4.2.8. Змінна кількість аргументів: файл stdarg.h

У попередньому розділі обговорювалися макроси зі змінним числом аргументів. Файл заголовку **stdarg.h** надає схожу можливість для функцій.

Однак використовувати її дещо складніше. Ви повинні виконати наступні дії.

1) Підготувати прототип функції, в якому застосовуються три крапки.

2) Створити у визначенні функції змінну типу **va_list**.

3) Використати макрос для ініціалізації цієї змінної списком аргументів.

4) Застосувати макрос для доступу до списку аргументів.

5) Використати макрос для очищення.

Давайте розглянемо ці дії більш докладно. Прототип для функції подібного роду повинен мати список, що містить, принаймні, один параметр, за яким йдуть три крапки:

```
void f1(int n, ...);                    // допустимо
int f2(const char *s, int k, ...);      // допустимо
char f3(char c1, ..., char c2);         // недопустимо, оскільки
                                        // три крапки знаходяться
                                        // не наприкінці списку
double f4(...);                         // недопустимо, параметри
                                        // відсутні
```

Крайній справа параметр, що передує трьом крапкам, відіграє спеціальну роль. Для його позначення в стандарті використовується термін **parmN**. В попередніх прикладах роль **parmN** відігравав параметр **n** в першому випадку і **k** – в другому. Фактичним аргументом, що передається цьому параметру, є кількість аргументів, які представлені розділом з трьох крапок. Наприклад, функцію **f1()**, прототип якої був визначений раніше, можна викликати наступним чином:

```
f1(2, 200, 400);                    // 2 додаткові аргументи
f1(4,  13, 117,  18,  23);          // 4 додаткові аргументи
```

Тип **va_list**, який оголошений у файлі заголовку **stdargs.h**, являє собою об’єкт даних, що застосовується для зберігання параметрів, які відповідають розділу з трьома крапками в списку параметрів. Початок визначення функції зі змінним числом аргументів виглядає приблизно таким чином:

```
double sum(int lim, ...)
{
   va__list ap; // оголошення об'єкту для зберігання аргументів
```

В цьому прикладі **lim** є параметром **parmN** і вказує кількість аргументів у списку змінних-аргументів.

Потім функція буде використовувати макрос **va_start()**, який також визначений в **stdargs.h**, для копіювання списку аргументів у змінну **va_list**. Макрос приймає два аргументи: змінну **va_list** і параметр **parmN**. Продовжуючи попередній приклад, змінна **va_list** названа **ap**, а параметру **parmN** призначено ім'я **lim**. Таким чином, виклик буде мати такий вигляд:

```
va_start(ap, lim);    // ініціалізація ap списком аргументів
```

На наступному етапі відбувається доступ до вмісту списку аргументів. Це передбачає застосування макросу – **va_arg()**, який приймає два аргументи: змінну типу **va_list** та ім'я типу. При першому виклику він повертає перший елемент списку, при наступному виклику – черговий елемент списку і т. д.

Наприклад, якщо б першим аргументом у списку був **double**, а другим – **int**, то ви могли б зробити наступним чином:

```
double tic;
int toe;
... tic = va_arg(ap, double); // отримуємо перший аргумент
toe = va_arg(ap, int);        // отримуємо другий аргумент
```

Але слід бути уважним. Тип аргументу насправді повинен відповідати специфікації. Якщо першим аргументом є **10.0**, попередній код для **tic** працює нормально. Однак якщо аргументом виявляється **10**, код може не працювати. Автоматичного перетворення **double** на **int**, що застосовувалося для операції присвоювання, в даному випадку не відбувається.

Нарешті, ви повинні провести очищення за допомогою макросу **va_end()**. Наприклад, може знадобитися звільнити пам'ять, яка була динамічно виділена для зберігання аргументів. Цей макрос приймає в якості аргументу змінну **va_list**:

```
va_end(ap);                    // очищення
```

Після цього змінна **ap** може виявитися непридатною до застосування до тих пір, поки ви не ініціалізуєте її повторно за допомогою макросу **va_start()**.

Оскільки макрос **va_arg()** не забезпечує копіювання попередніх аргументів для їх можливого відновлення, може виявитися доцільним зберігання копії змінної **va_list**. Для цього в стандарті **C99** передбачено макрос на ім'я **va_copy()**. Він приймає два аргументи типу **va_list** і копіює другий аргумент у перший:

```
va_list ap;
va_list apcopy;
double tic;
int toe;
    ...

va_start(ap, lim);     // ініціалізація ap списком аргументів
va_copy(apcopy, ap);   // робить apcopy копією ap
tic = va_arg(ap, double); // отримуємо перший аргумент
toe = va_arg(ap, int);    // отримуємо другий аргумент
```

На даному етапі як і раніше можна отримати перші два елемента з **apcopy**, незважаючи на те, що вони були видалені з **ap**.

В наступній програмі наведено приклад використання розглянутих можливостей для створення функції **sum()**, яка підсумовує змінну кількість аргументів.

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

double sum(int, ...);

int main(void)
{
    double s, t;
s = sum(3, 1.1, 2.5, 13.3);
t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);

    SetConsoleOutputCP(1251);
```

```
printf("Значення, що повертається "
           "sum(3, 1.1, 2.5, 13.3):                  %g\n", s);
   printf("Значення, що повертається "
           "sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1):  %g\n", t);
     return 0;
}
 double sum(int lim, ...)
{
    va_list ap;                          // оголошення об'єкту для
                                         // зберігання аргументів
double tot = 0;      int i;
    va_start(ap, lim);                   // ініціалізація ap списком
                                         // аргументів
      for(i = 0; i < lim; i++)
      tot += va_arg(ap, double); // доступ до кожного
                                 // елемента в списку  аргументів
    va_end(ap);                 // очищення
  return tot;
  }
```

Результат виконання програми наведено на рис. 4.20.

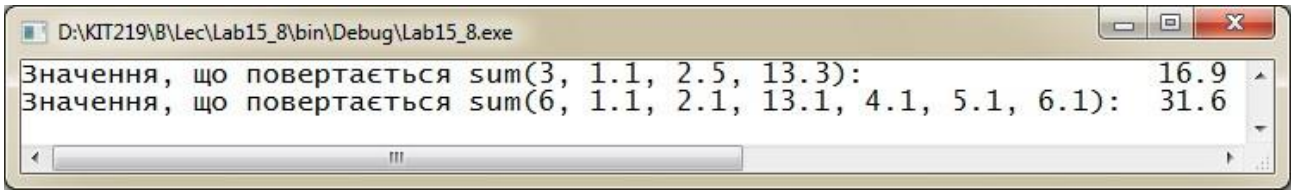


Рисунок 4.20 – Результат виконання програми для демонстрації можливостей використання функції зі змінним числом аргументів

Перевіривши обчислення, ви побачите, що функція **sum()** додала три числа при першому виклику і шість чисел – при другому.

В загальному випадку, функції зі змінним числом аргументів є складнішими у використанні на відміну від макросів такого роду, але мають більш широкий діапазон застосувань.

Стандарт не лише описує мову **C**. Він визначає пакет, що складається з: мови **C**, препроцесора **C** і стандартної бібліотеки **C**.

Препроцесор дозволяє виконати підготовчі дії перш ніж буде проведена компіляція тексту програми, вказуючи необхідні підстановки, обираючи рядки

коду, що підлягають компіляції, а також встановлюючи інші аспекти поведінки компілятора

Бібліотека `c` розширює можливості мови і надає готові рішення для багатьох задач програмування. Вона пропонує багато функцій, що призначені для сприяння в рішенні таких задач, як ввід, вивід, операції з файлами, керування пам'яттю, сортування та пошук, математичні обчислення, обробка рядків і багато інших.

Контрольні запитання та завдання

1. Для чого використовується препроцесор, та які функції він виконує?

2. Разом з якою директивою препроцесора застосовуються операції `##` і `#.` Що вони означають?

3. Які існують варіанти застосування директиви `#define`? Для чого вона призначена?

4. Яким чином використовують директиву `#include`? Які способи застосування цієї директиви ви знаєте?

5. Які директиви забезпечують умовну компіляцію?

6. Які наперед визначені макроси вам відомі?

7. Що дозволяє отримати наперед визначений ідентифікатор `__func__`?

8. Що препроцесор робить з коментарями?

9. Що треба робити у випадку, коли директива препроцесора #define не вміщується в один рядок?

Завдання для самостійного розв'язання

1. Напишіть С-програму, яка демонструє можливості використання аргументів у директиві `#define` при створенні функціональних макросів, можливості створення рядків з аргументів макросу за допомогою операції `#`, а також варіанти використання операції `##`.

2. Напишіть С-програму, яка демонструє можливості роботи з директивами препроцесора **#if**, **#else**, **#elif** і **#endif** , а також з директивами **#ifdef** і **#ifndef** для забезпечення умовної компіляції.

3. Напишіть С-програму, яка демонструє можливість застосування наперед визначених макросів: **__DATE__**, **__FILE__**, **__LINE__**, **__STDC__**, **__STDC_HOSTED__**, **__STDC_VERSION__**, **__TIME__**, а також наперед визначеного ідентифікатора **__func__**.

5. ВІЗУАЛІЗАЦІЯ ІНФОРМАЦІЇ

Стандарт **ANSI C** не визначає функцій для роботи з текстом або графікою в першу чергу тому, що існує велика кількість апаратних засобів, через які ускладнюється стандартизація. Тим не менш в цьому напрямку створена широка підтримка роботи з екраном і графікою.

Центральною концепцією для функцій роботи з текстом і графікою є концепція вікна, тобто активної області екрану, в межах якої здійснюється вивід даних. Вікно може бути розміром з цілий екран, а може мати визначені розміри. Існує дещо різна термінологія для текстової і графічної систем з метою зберігання цих систем окремо одна від одної. Текстові функції працюють з вікнами, в той час як графічна система використовує область перегляду. Однак концепція в обох випадках є однаковою. Уся інформація, що виводиться, знаходиться в активному вікні. Це означає, що частина зображення, яка розташована за межами вікна або області перегляду, автоматично відсікається.

При використанні графіки програма повинна в першу чергу ініціалізувати графічну систему за допомогою виклику функції `initgraph()`. По закінченні використання графіки необхідно викликати функцію `closegraph()`.

Важливо розуміти, що більшість текстових і графічних функцій стосується вікна або області перегляду. Наприклад, функція `gotoxy()` встановлює курсор в задану точку з координатами `(x, y)` по відношенню до вікна, а не до екрану.

Якщо екран функціонує в текстовому режимі, його лівий верхній кут розташований в точці з координатами `(1, 1)`. В графічному режимі лівий верхній кут являє собою точку з координатами `(0, 0)`.

5.1. Графічна бібліотека WinBGIm

В графічному режимі область виводу на екран (це може бути весь екран монітору або графічне вікно) являє собою прямокутну матрицю точок, колір кожної з яких можна змінювати окремо.

Для ефективного програмування графічного режиму доцільно використовувати різні графічні бібліотеки. Однією з них є бібліотека **WinBGIm** – аналог **WinBGI** фірми **Borland**. Ця бібліотека містить велику кількість графічних функцій. Передбачити запуск графічного режиму можна в будь-якому місці програми. Для цього потрібно включити у програму файл заголовку **graphics.h** і описати дві допоміжні змінні цілого типу:

```
#include <graphics.h>
...
int gd, gm;
```

Змінна **gd** (**graphic driver**) використовується для зберігання номера (коду) графічного драйвера, а **gm** (**graphic mode**) – графічного режиму. Імена змінних **gd** і **gm** можуть бути довільними. Перед ініціалізацією графіки необхідно присвоїти значення змінним **gd** і **gm**. Однак можна надати вибір найкращого драйвера і режиму функції **detectgraph()** автоматичного визначення параметрів графіки:

```
void detectgraph(int *graphdriver, int *graphmode);
```

Можливі значення номера графічного драйверу наведені в табл. 5.1. Частіше за все на персональних комп'ютерах застосовується графічний драйвер **VGA** з номером **9** разом з одним з трьох можливих режимів, які наведені в табл. 5.2.

У файлі заголовку **graphics.h** доступні драйвери визначені у переліченні **enum** graphics_drivers:

```
// Різні типи графічних драйверів з файлу graphics.h
enum graphics_drivers { DETECT, CGA, MCGA, EGA, EGA64,
                        EGAMONO, IBM8514, HERCMONO,
                        ATT400, VGA, PC3270 };
```

Таблиця 5.1 – Графічні драйвери

Назва драйверу	Номер
DETECT	0 (автоматичне визначення)
CGA	1
MCGA	2
EGA	3
ECGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Таблиця 5.2 – Режими VGA

Назва	Номер	Роздільність	Палітра
VGALO	0	640×200	16 кольорів
VGAMED	1	640×350	16 кольорів
VGAHI	2	640×480	16 кольорів

Можливі режими в файлі заголовку graphics.h визначені у переліченні

```
enum graphics_modes:
      // Різні режими для кожного драйверу
      enum graphics_modes { CGAC0, CGAC1, CGAC2, CGAC3, CGAHI,
                            MCGAC0 = 0, MCGAC1, MCGAC2, MCGAC3,
                            MCGAMED, MCGAHI,
                            EGALO = 0, EGAHI,
                            EGA64LO = 0, EGA64HI,
                            EGAMONOHI = 3,
                            HERCMONOHI = 0,
                            ATT400C0 = 0, ATT400C1, ATT400C2,
```

```
                    ATT400C3, ATT400MED, ATT400HI,
                    VGALO = 0, VGAMED, VGAHI,
                    PC3270HI = 0,
                    IBM8514LO = 0, IBM8514HI };
```

В цьому прототипі параметр **path** – це шлях до файлу графічного драйверу. При автоматичному визначенні параметрів у якості параметру **path** доцільно використовувати порожній рядок. При успішному виконанні функції **initgraph()** буде створене додаткове вікно графічного виводу програми. По закінченні роботи з графічним режимом його необхідно закрити за допомогою функції **closegraph()**, прототип якої має наступний вигляд:

```
void closegraph(int wid = ALL_WINDOWS);
```

Оскільки є можливість відкриття декількох графічних вікон, параметр функції **wid** дозволяє вказати, яке саме графічне вікно необхідно закрити. Цей параметр може приймати одне з двох значень: **CURRENT_WINDOW** (закриття поточного вікна) і **ALL_WINDOWS** (закриття усіх графічних вікон, використовується за замовчуванням), **NO_CURRENT_WINDOW** – закриття усіх вікон, крім поточного.

```
// Константи для функції initgraph()
#define CURRENT_WINDOW     -1
#define ALL_WINDOWS        -2
#define NO_CURRENT_WINDOW  -3
```

В наступному прикладі проводиться ініціалізація графічного режиму та його закриття після натиснення будь-якої клавіші.

```
#include <graphics.h>
int main(void)
{
   int gd, gm;
   detectgraph(&gd, &gm);
   initgraph(&gd, &gm, "");
   getch();
   closegraph();
   return 0;
}
```

Результат успішної роботи програми наведено на рис. 5.1.

Рисунок 5.1 – Результат роботи програми, що демонструє успішну ініціалізацію графічного режиму

Для кодування кольору в бібліотеці **WinBGIm** використовується нумерація кольорів цілими числами. Передбачені також символічні назви кольорів, що наведені в табл. 5.3. У файлі **graphics.h** вони описані за допомогою директиви перелічення **enum** colors:

```
// Стандартні кольори компанії Borland
#define MAXCOLORS       15      // від 0 до 15
enum colors { BLACK, BLUE, GREEN, CYAN, RED, MAGENTA,
              BROWN, LIGHTGRAY, DARKGRAY, LIGHTBLUE,
              LIGHTGREEN, LIGHTCYAN, LIGHTRED,
              LIGHTMAGENTA, YELLOW, WHITE };
```

5.2. Графічний курсор

Процес побудови зображень в графічному вікні спирається на систему координат з початком у лівому верхньому куті вікна. Вісь **x** спрямована вправо, вісь **y** – вниз. Аналогом текстового курсору в графічному режимі є невидимий поточний вказівник. Частіше за все рисування об’єктів виконується відносно поточного вказівника. Максимальне значення координат вказівника залежить

від розрізнення (кількості точок, що розміщується у вікні). Отримати максимально можливі координати можна за допомогою функцій:

```
int getmaxx(void);
int getmaxy(void);
```

Таблиця 5.3 – Символічні назви кольорів

Назва	Номер
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

Переміщення вказівника без прорисовування на екрані здійснюється за допомогою однієї з функцій:

```
void moveto(int x, int y);
void moverel(int dx, int dy);
```

Перша з них переміщує вказівник в зазначені координати, друга зміщує вказівник на величини **dx** і **dy** відносно поточної позиції.

Визначити поточні координати вказівника можна за допомогою функцій:

```
int getx(void);
int gety(void);
```

5.3.Рисування точок і ліній

В бібліотеці **WinBGIm** вивід точки здійснює функція `putpixel()`:

```
void putpixel(int x, int y, int color);
```

де **x**, **y** – координати розташування точки; **color** – колір.

Наприклад, команда

```
putpixel(getmaxx()/2, getmaxy()/2, GREEN);
```

виводить в центрі екрану зелену точку.

Для побудови відрізків прямих призначені функції `line()`, `lineto()` і `linerel()`.

Функція

```
void line(int x1, int y1, int x2, int y2);
```

рисує лінію з початком у точці з координатами (**x1**, **y1**) і кінцем у точці (**x2**, **y2**). Слід зазначити, що багато функцій рисування графічних примітивів, зокрема функція `line()`, не мають параметру встановлення кольору. В цьому випадку **колір задається перед рисуванням** об'єкту функцією

```
void setcolor(int color);
```

наприклад,

```
setcolor(RED);
line(0, 0, 100, 200);
```

Аналогічно функція

```
setbkcolor(int color);
```

визначає поточний колір фону.

Функції

```
int getcolor(void);
int getbkcolor(void);
```

дозволяють отримати номери поточного кольору вказівника і фону відповідно, а функція

```
unsigned getpixel(int x, int y);
```

визначає номер кольору точки вікна з вказаними координатами.

Функція

```
void lineto(int x, int y);
```

рисує лінію з точки поточного положення вказівника до точки з координатами `(x, y)`.

Функція

```
void linerel(int dx, int dy);
```

рисує лінію між точкою поточного положення вказівника `(x, y)` і точкою `(x + dx, y + dy)`.

Допускається встановлення стилю лінії за допомогою функції

```
void setlinestyle(int style, unsigned upattern, int thickness);
```

де параметр `style` визначає тип лінії, можливі значення якого описані у вигляді констант і наведені в табл. 5.4. У файлі `graphics.h` вони визначені за допомогою перелічення `enum line_styles`:

```
// Стандартні стилі ліній
enum line_styles { SOLID_LINE, DOTTED_LINE, CENTER_LINE,
                   DASHED_LINE, USERBIT_LINE };
```

Таблиця 5.4 – Стилі ліній

Назва	Номер	Опис
SOLID_LINE	0	суцільна
DOTTED_LINE	1	пунктирна
CENTER_LINE	2	штрих-пунктирна
DASHED_LINE	3	штрихова
USERBIT_LINE	4	стиль, що задається зразком

Параметр `thickness` визначає товщину лінії. Можливі значення товщини: `NORM_WIDTH`=1 (лінія товщиною `1` піксель) і `THICK_WIDTH`=3 (лінія товщиною `3` пікселі). У файлі `graphics.h` константи задаються за допомогою директив:

```
#define NORM_WIDTH  1
#define THICK_WIDTH 3
```

Параметр `upattern` вказує зразок стилю і використовується, якщо стиль лінії дорівнює `USERBIT_LINE` (в протилежному випадку `upattern=0`). Зразок задається в форматі двобайтового числа. В його двійковому представленні одиниці відповідають світлим ділянкам лінії, нулі – темним.

Зразок штрихової лінії може мати наступний вигляд:

```
1111000011110000 = 0xF0F0,
```

а пунктирної лінії як

```
1010101010101010 = 0xAAAA,
```

Наприклад:

```
setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
line(30, 30, 555, 400);
```

або

```
setlinestyle(USERBIT_LINE, 0x9898, THICK_WIDTH);
line(30, 30, 555, 400);
```

5.4. Рисування прямокутників

Прямокутник можна нарисувати, вказавши координати його двох характерних точок, наприклад лівого верхнього `(x1, y1)` і правого нижнього `(x2, y2)` кутів (рис. 5.2).

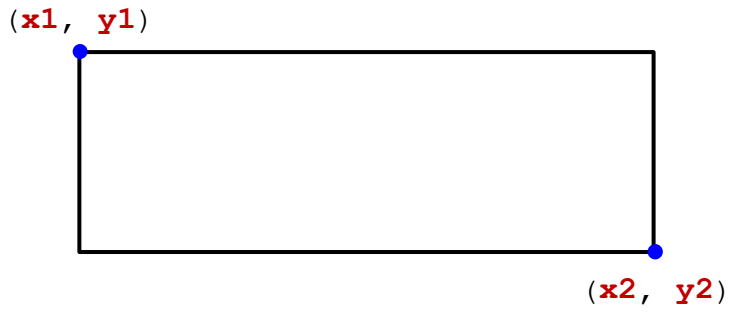


Рисунок 5.2 – Характерні точки для прямокутника

Функція

```
void rectangle(int x1, int y1, int x2, int y2);
```

рисує незафарбований прямокутник, а функція

```
void bar(int x1, int y1, int x2, int y2);
```

– зафарбований прямокутник.

Функція

```
void bar3d(int x1, int y1, int x2, int y2, int d, int b);
```

рисує тривимірний зафарбований прямокутник (паралелепіпед). Параметр **d** визначає глибину тривимірного контуру, а параметр **b** – визначає чи закрита верхня «кришка» паралелепіпеду, чи ні.

Для пояснення роботи функції **bar3d()** розглянемо наступну програму:

```
#include <graphics.h>
int main(void)
{
initwindow(600, 250);
bar3d( 50, 50, 200, 200, 50, 0);
bar3d(350, 50, 500, 200, 50, 1);
getch();
closegraph();
return 0;
}
```

Результат роботи програми наведено на рис. 5.3.

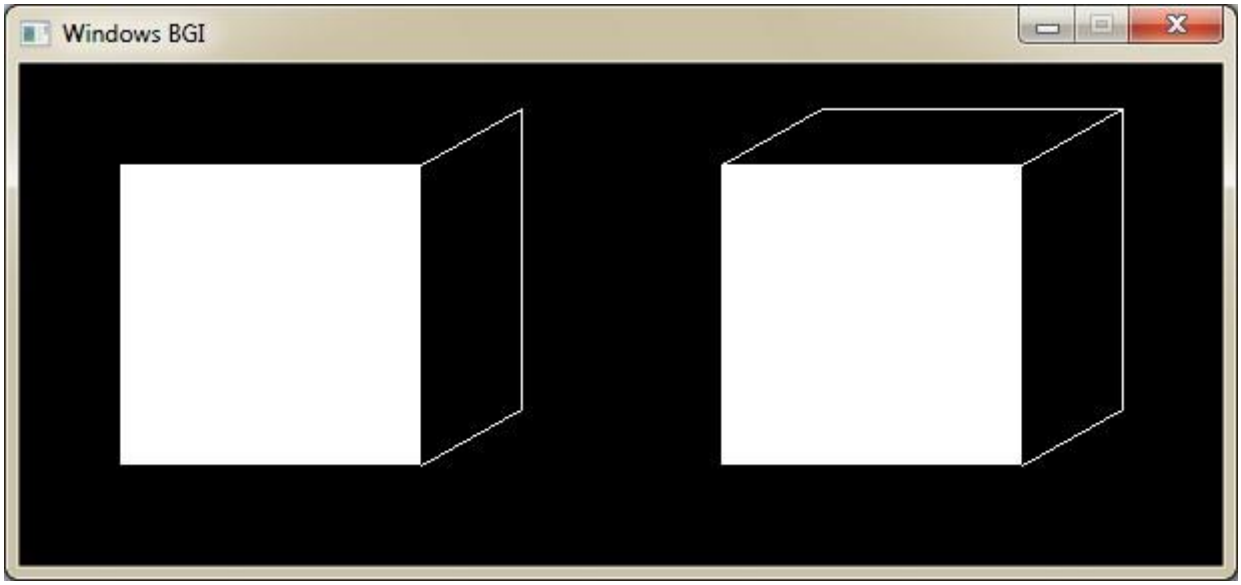


Рисунок 5.3 – Приклад використання функції **bar3d()**

5.5. Рисування окружностей та еліпсів

Рисування еліпсів та еліптичних дуг здійснює функція

```
void ellipse(int x, int y, int a, int b, int Rx, int Ry);
```

де **x**, **y** – координати центру еліпсу; **a**, **b** – початковий і кінцевий кути, що відраховується від горизонталі; **Rx**, **Ry** – радіуси в горизонтальному і вертикальному напрямках. Для побудови замкнутого еліпсу початковий і кінцевий кути встановлюють рівними **0** і **360** градусів. Окружність є окремим випадком еліпсу при однакових горизонтальному і вертикальному радіусах. Так, наприклад, оператор

```
ellipse(100, 100, 0, 360, 50, 50);
```

рисує в графічному вікні окружність.

Функція

```
void fillellipse(int x, int y, int Rx, int Ry);
```

рисує зафарбований еліпс.

Стиль і колір заливки визначаються функцією

```
void setfillstyle(int style, int color);
```

де **style** – стиль заливки; **color** – колір заливки. Можливі значення стилю заливки наведені в табл. 5.5.

У файлі **graphics.h** стандартні стилі заливки представлені переліченням **enum** fill_styles:

```
// Стандартні стилі заливки
enum fill_styles { EMPTY_FILL, SOLID_FILL, LINE_FILL,
                   LTSLASH_FILL, SLASH_FILL,
                   BKSLASH_FILL, LTBKSLASH_FILL,
HATCH_FILL,
                   XHATCH_FILL, INTERLEAVE_FILL,
                   WIDE_DOT_FILL, CLOSE_DOT_FILL, USER_FILL };
```

Таблиця 5.5 – Стилі заливки

Назва	Номер	Опис
EMPTY_FILL	0	суцільна (кольором фону)
SOLID_FILL	1	суцільна (заданим кольором)
лініями		
LINE_FILL	2	горизонтальними
LTSLASH_FILL	3	скісними /
SLASH_FILL	4	товстими скісними /
BKSLASH_FILL	5	товстими скісними \
LTBKSLASH_FILL	6	скісними \
клітинками		
HATCH_FILL	7	товстими лініями
XHATCH_FILL	8	товстими лініями під кутом 45°
INTERLEAVE_FILL	9	діагональною штриховкою
WIDE_DOT_FILL	10	рідкими точками
CLOSE_DOT_FILL	11	частими точками
USER_FILL	12	з використанням заданої маски

Текст програми для демонстрації застосування функції **setfillstyle()** має наступний вигляд:

```
#include <graphics.h>

int main(void)
{
    initwindow(450, 350);

    int k = 0;
setcolor(WHITE);
    for(int i = 0; i < 2; i++)
    {
        for(int j = 0; j < 6; j++)
        {
          ellipse(50 + (j*70), 50 + (i*70), 0, 360, 25, 25);
          setfillstyle(k++, YELLOW);
          floodfill(55 + (j * 70), 55 + (i * 70), WHITE);
```

```
            }
        }
        setcolor(GREEN);
        rectangle(100, 200, 200, 300);
        setfillstyle(LINE_FILL, BLUE);
        floodfill(150, 250, GREEN);
        getch();
        closegraph();
return 0;
}
```

Результат роботи програми наведено на рис. 5.4.

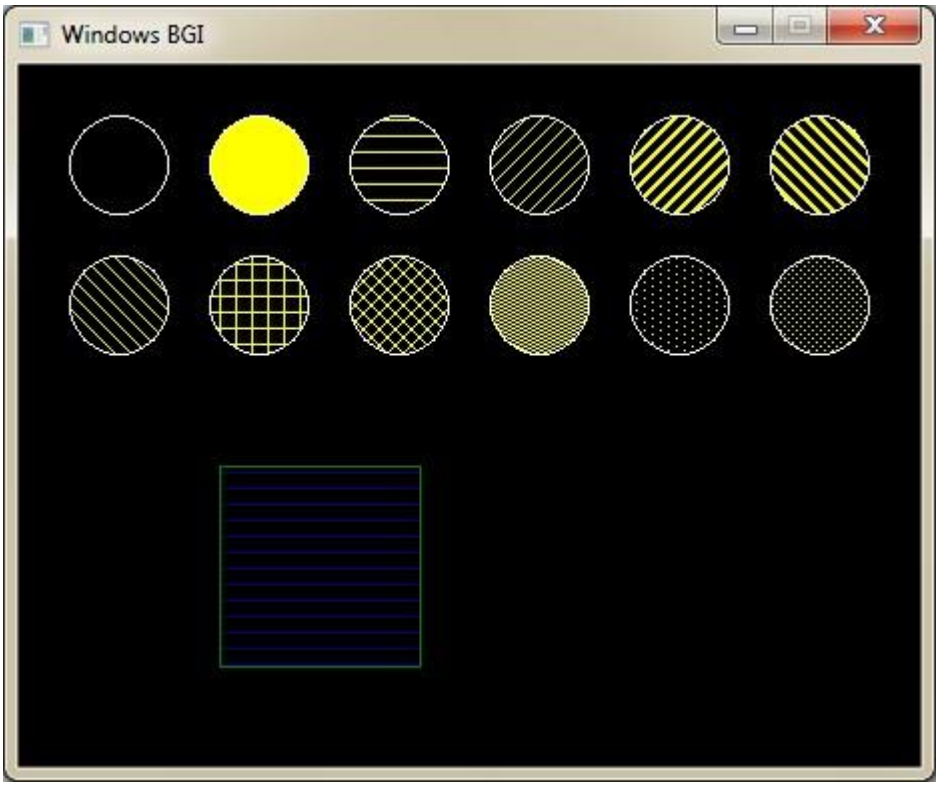


Рисунок 5.4 – Приклад використання функції **setfillstyle()**

5.6. Вивід тексту в графічному режимі

Для виводу на екран текстових надписів у графічному режимі використовуються функції **outtext()** і **outtextxy()**. Функція **void outtext(char *string);** виводить рядок тексту, починаючи з поточного положення вказівника.

Наприклад,

```
outtext("Для продовження натисніть Enter");
```

Якщо необхідно вказати точку для початку виводу, доцільно використовувати функцію

```
void outtextxy(int x, int y, char *string);
```

де **x**, **y** – координати точки для початку виводу тексту. Наприклад,

```
outtextxy(50, 100, "Для продовження натисніть Enter");
```

Слід зазначити, що для виводу на екран числових даних, необхідно спочатку перевести їх на рядки. Це можна зробити за допомогою функції

```
sprintf(char *s, "Форматний рядок", ...);
```

яка схожа на функцію **printf()**, тільки вивід відбувається в рядок **s**, який вказується в якості першого аргументу.

Розглянемо приклад застосування функції **outtextxy()**. Текст програми буде мати наступний вигляд:

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

int main(void)
{
  float x = 3.14159;
  char *s;

  initwindow(200, 100);
  sprintf(s, "%f", x);
  outtextxy(20, 40, s);
  getch();
  closegraph();
  return 0;
}
```

Результат роботи програми наведено на рис. 5.5.

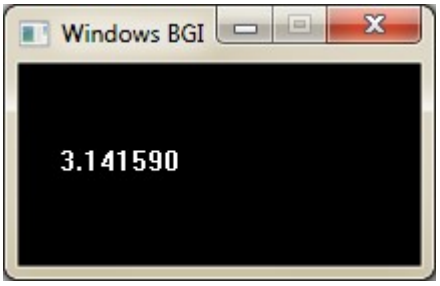


Рисунок 5.5 – Приклад використання функції **outtextxy()**

Функція

```
void settextstyle(int font, int direction, int charsize);
```

призначена для керування параметрами шрифту при виводі тексту в графічному режимі. В прототипі `font` – номер шрифту (за замовчуванням `DEFAULT_FONT=0`); `direction` – напрямок (горизонтальне `direction=0` або вертикальне `direction=1`); `charsize` – розмір символів, що виводяться (при `charsize=1` розмір букви `8×8` точок, при `charsize=2` розмір букви `16×16` точок. Можливі шрифти, які підтримуються бібліотекою, та їх номери наведені в табл. 5.6.

Таблиця 5.6 – Назви шрифтів, що підтримуються бібліотекою

Назва	Номер
DEFAULT_FONT	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SERIF_FONT	3
GOTHIC_FONT	4
SCRIPT_FONT	5
SIMPLEX_FONT	6
TRIPLEX_SCR_FONT	7
COMPLEX_FONT	8
EUROPEAN_FONT	9
BOLD_FONT	10

У файлі `graphics.h` визначені перелічення, що призначені для роботи з текстом у графічному режимі:

```
// Текстові режими
enum horiz { LEFT_TEXT, CENTER_TEXT, RIGHT_TEXT };
enum vertical { BOTTOM_TEXT, VCENTER_TEXT, TOP_TEXT };
enum font_names { DEFAULT_FONT, TRIPLEX_FONT, SMALL_FONT,
                  SANS_SERIF_FONT, GOTHIC_FONT, SCRIPT_FONT,
```

```
                    SIMPLEX_FONT, TRIPLEX_SCR_FONT,
                    COMPLEX_FONT, EUROPEAN_FONT, BOLD_FONT};
```

Очищення графічного вікна та встановлення вказівника в точку з координатами `(0, 0)` здійснює функція:

```
void cleardevice(void);
```

5.7. Побудова графіків функцій

В якості прикладу напишемо програму, яка рисує в графічному вікні графік функції **`f(x) = x * sin(x)`**. Включимо до програми заголовні файли

```
#include <graphics.h>
#include <math.h>
```

Головна функція **`main()`** починається з ініціалізації графічного режиму

```
int gd, gm;
detectgraph(&gd, &gm);
initgraph(&gd, &gm, "");
```

Далі визначаємо білий колір фону і очищуємо екран.

```
setbkcolor(WHITE);
cleardevice();
```

Будемо розрізняти екранну систему координат **`XOY`** з початком у лівому верхньому куту вікна і систему координат **`xoy`**, стосовно якої буде побудований графік (рис. 5.6). Центр системи **`xoy`** знаходиться в точці `(X0, Y0)` відносно системи координат **`XOY`**. Ось абсцис **`ox`** спрямуємо вправо, ось ординат **`oy`** – уверх, а також визначимо розміри одиничних відрізків **`edx`**, **`edy`**. Тоді для будь-якої точки з екранними координатами `(X1, Y1)` можна обчислити її координати в системі **`xoy`**

```
x = (X0 - X1) / edx,
y = (Y0 - Y1) / edy;
```

та навпаки, координати точок графіку `(x, y)` можна перерахувати на екранні координати:

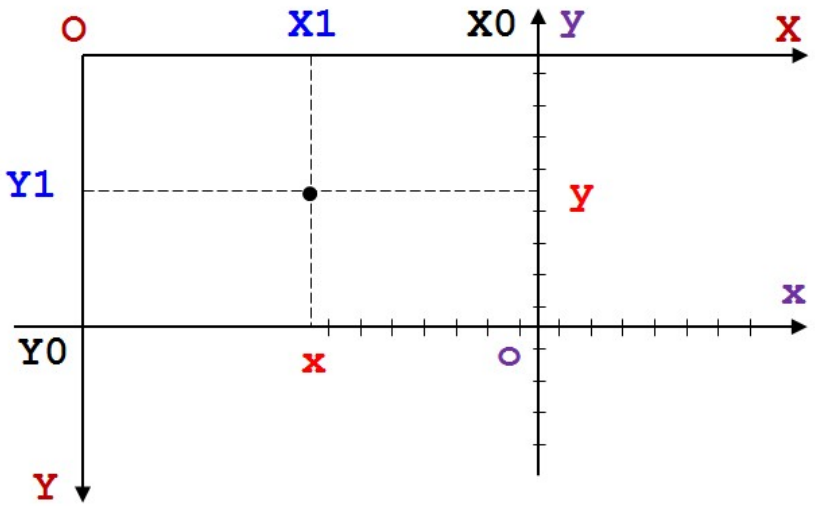


Рисунок 5.6 – Перетворення екранних координат

```
X1 = X0 - edx * x;
Y1 = Y0 - edy * y;
```

Далі необхідно нарисувати вісі координат, відносно яких буде побудований графік. Для цього обираємо екранні координати (**X0**, **Y0**) центру нової системи координат посередині екрану

```
int X0 = getmaxx()/2;
int Y0 = getmaxy()/2;
```

і рисуємо осі

```
line(0, Y0, getmaxx(), Y);
line(X0, 0, X0, getmaxy());
```

Перпендикулярно осям маленькими лініями зобразимо відліки з кроком, який дорівнює одиничним відрізкам

```
int edx = 20;
int edy = 15;
setcolor(LIGHTBLUE);
int k;
// відліки по осі x
for(k = 0; edx * k < X0; k++)
{
    line(X0 + k * edx, Y0 - 3, X0 + k * edx, Y0 + 3);
    line(X0 - k * edx, Y0 - 3, X0 - k * edx, Y0 + 3);
}
// відліки по осі y
for(k = 0; edy * k < Y0; k++)
```

```
{
    line(X0 - 3, Y0 + k * edy, X0 + 3, Y0 + k * edy);
    line(X0 - 3, Y0 - k * edy, X0 + 3, Y0 - k * edy);
}
```

Далі починаємо рисувати графік. Встановлюємо поточними рожевий колір і суцільний тип лінії товщиною **3** точки

```
setcolor(LIGHTRED);
setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
```

Оголошуємо змінні дійсного типу: **a**, **b** – межі області побудови по осі **ox**; **x**, **y** – поточні координати вказівника; **delta** – крок побудови по осі **ox**:

```
float a, b, x, y, delta;
// інтервал дискретизації
delta = 0.1;
// межі побудови по осі x
a = -15;
b = 15;
```

Початкова точка побудови графіка має координати **(a, f(a))**. Перемістимо до неї вказівник (графічний курсор).

```
  x = a;
  y = x * sin(x);
  moveto(X0 + edx * x, Y0 - edy * y);
```

Для побудови графіку будемо обчислювати координати наступної точки графіка **(x, f(x))** і рисувати лінію між поточним положенням вказівника і щойно визначеною точкою. Таким чином, графік буде являти собою ломану лінію, яка наближена до реального графіку.

```
// цикл побудови графіку
for(x = a; x <= b; x += delta)
{
  y = x * sin(x);
  lineto(X0 + edx * x, Y0 - edy * y);
}
```

Наприкінці програми у графічне вікно виводиться заголовок.

```
// встановлення стилю тексту
Settextstyle (SANS_SERIF_FONT, 0, 2);
// вивід заголовку
Outtextxy (180, 10, "Графік функції f(x)=x*sin(x)");
```

Нижче наведено повний текст програми побудови графіку.

```
#include <graphics.h>
#include <math.h>
#include <stdio.h>
int main(void)
{
    int gd, gm;
    detectgraph(&gd, &gm);
    initgraph(&gd, &gm, "");

// Встановлюємо білий колір для фону вікна
setbkcolor(WHITE);
// Очищуємо вікно кольором фону
cleardevice();
// Обчислюємо координати центру системи координат
int X0 = getmaxx()/2;
int Y0 = getmaxy()/2;

// Встановлюємо чорний колір для лінії
setcolor(BLACK);
// Рисуємо горизонтальну вісь
line(0, Y0, getmaxx(), Y0);
// Рисуємо вертикальну вісь
line(X0, 0, X0, getmaxy());
// Рисуємо відліки на осях
int edx = 20;
int edy = 15;
setcolor(LIGHTBLUE);
int k;
// по осі x
    for(k = 0; edx * k < X0; k++)
    {
         line(X0 + k * edx, Y0 - 3, X0 + k * edx, Y0 + 3);
         line(X0 - k * edx, Y0 - 3, X0 - k * edx, Y0 + 3);
    }
    // по осі y
    for(k = 0; edy * k < Y0; k++)
    {
          line(X0 - 3, Y0 + k * edy, X0 + 3, Y0 + k * edy);
          line(X0 - 3, Y0 - k * edy, X0 + 3, Y0 - k * edy);
    }
    setcolor(LIGHTRED);
    setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
  float a, b, x, y, delta;
// інтервал дискретизації
delta = 0.1;
// межі побудови по осі x
a = -15;
b = 15;
x = a;
```

```
y = x * sin(x);
moveto(X0 + edx * x, Y0 - edy * y);
 // цикл побудови графіку
for(x = a; x <= b; x += delta)
     {
          y = x * sin(x);
          lineto(X0 + edx * x, Y0 - edy * y);
     }
      setcolor(LIGHTBLUE);
     // встановлення стилю тексту
settextstyle(SANS_SERIF_FONT, 0, 2);
// вивід заголовку
outtextxy(180, 10, "Графік функції f(x)=x*sin(x)");

getch();
return 0;
}
```

Результат роботи програми наведено на рис. 5.7.

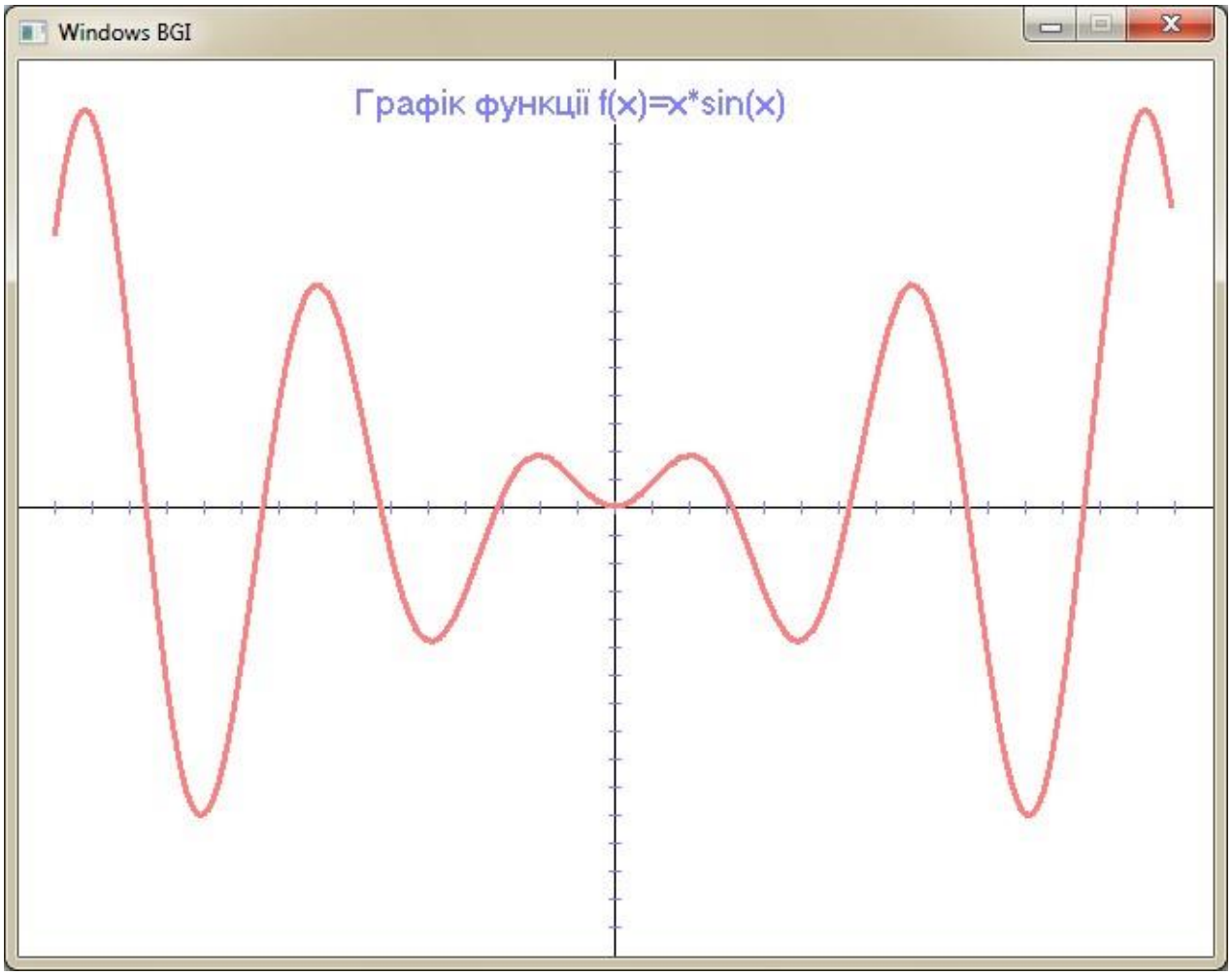


Рисунок 5.7 – Результат візуалізації графіку функції **f(x)=x*sin(x)**

5.8. Побудова кривих Гільберта

Криві Гільберта названі на честь німецького математика Давида Гільберта. Вперше вони були описані в 1891 році.

Криві Гільберта – це неперервні криві, що заповнюють простір. Вони також є фракталами, тобто вони є самоподібними. Якщо збільшити масштаб і уважно поглянути на частину кривої більш високого порядку, то можна побачити, що вона виглядає так само, як і сама крива.

Найпростіший спосіб зрозуміти, як будується крива Гільберта, наступний. Уявіть собі, що у вас є довгий фрагмент мотузки, і ви бажаєте розташувати його на плоскій сітці з квадратними комірками. Ваша мета полягає в тому, щоб мотузка перетинала сторони кожного квадрату сітки рівно один раз (рис. 5.8).

Не дозволяється, щоб мотузка перетинала сама себе.

Існує багато способів зробити це, однак криві Гільберта мають деякі додаткові цікаві властивості. Щоб їх зрозуміти, ми повинні уважніше поглянути на те, як вони будуються рекурсивно.

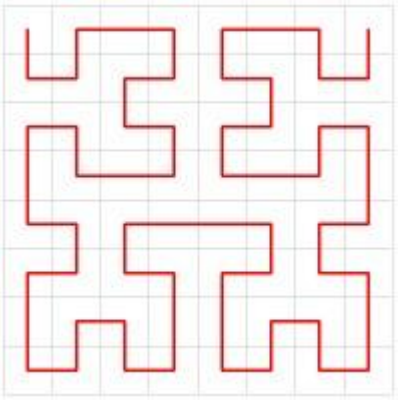


Рисунок 5.8 – Крива Гільберта, яка побудована для сітки 8×8

Основним елементом кривої Гільберта є П- подібний елемент (рис. 5.9).

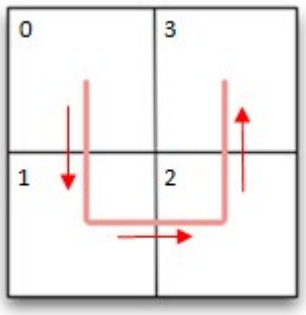


Рисунок 5.9 – Основний П-подібний елемент кривої Гільберта

На рис. 5.9 маємо сітку 2×2. Почнемо з лівого верхнього кута та проведемо мотузку через інші три квадрати сітки, закінчивши в правому верхньому куті.

Тепер уявіть, що ми подвоюємо розмір сітки, отримуючи сітку 4×4.

Ми можемо представити цю сітку 4×4 у вигляді набору 4=2×2 сіток, кожна з яких має розмір 2×2.

Ми бажаємо перетнути сітку більш високого рівня в такому порядку 0→ 1 → 2 → 3 (показано великими жовтими стрілками на рис. 1.10), а потім всередині цієї сітки будемо використовувати П-подібний шаблон обходу для кожної з сіток меншого рівня.

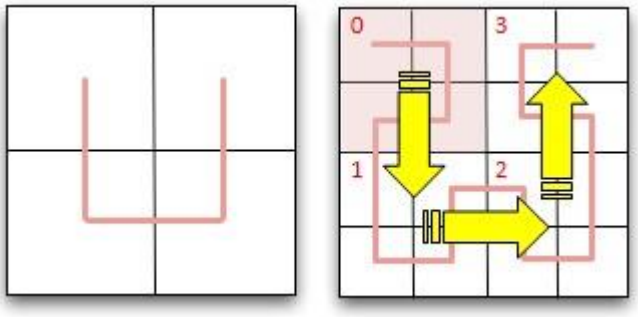


Рисунок 5.10 – П-подібний шаблон і порядок обходу квадратів

Результат обходу наведено на рис. 5.11. Можна помітити, що крива проходить через кожен квадрат сітки.

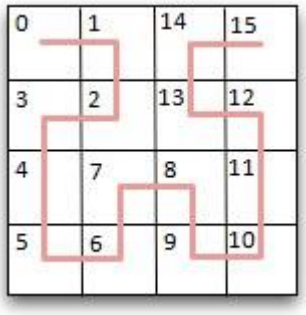


Рисунок 5.11 – Використання П-подібного шаблону обходу

Якщо повернутися до кривої на сітці `8×8` (рис. 5.8), то можна побачити що вона побудована з кривих для чотирьох сіток `4×4`, які розміщені за П-подібним зразком.

На рис. 5.12 наведені деякі криві Гільберта за зростанням порядку.

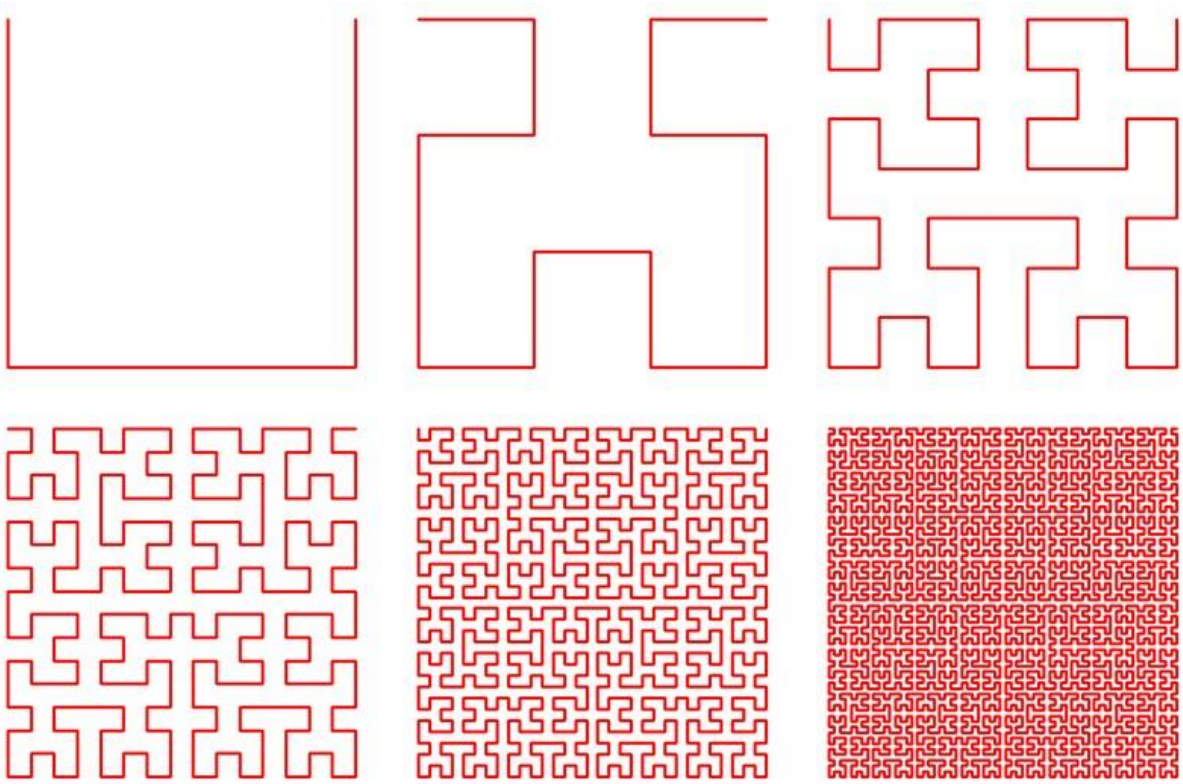


Рисунок 5.12 – Криві Гільберта за зростанням порядку (від 1-го до 6-го)

Розглянемо `C`-програму для рисування кривих Гільберта четвертого та п’ятого порядків. Порядок задається за допомогою змінної `p`, яка позначена

специфікатором **const**. Саме через це другий виклик функції **a()** здійснюється з параметром **5**. Алгоритм рисування є рекурсивним. Текст програми, що представлений чотирма функціями **a()**, **b()**, **c()** і **d()**, має наступний вигляд:

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

void a(int i);
void b(int i);
void c(int i);
void d(int i);
const int u = 10;         // довжина штриху
const int p =  4;         // порядок кривої
int i;
int main(void)
{
  int gdriver = DETECT, gmode, errorcode;
  initgraph(&gdriver, &gmode, "");
  errorcode = graphresult();
  if(errorcode != grOk)
 {
   printf("Помилка графіки: %s\n",
             grapherrormsg(errorcode));
   printf("Натисніть на будь-яку клавішу для виходу:\n");
   getch();
   exit(1);
 }
  moveto(50, 50);
  a(p);              // накреслити криву Гільберта 4-го порядку
  moveto(250, 50);
  a(5);              // накреслити криву Гільберта 5-го порядку

  getch();
  closegraph();
  return 0;
}
 void a(int i)
{     if(i > 0)
    {
      d(i-1);
      linerel(+u, 0);
      a(i-1);
      linerel(0, u);
      a(i-1);
      linerel(-u, 0);
      c(i-1);
    }
}
```

```
 void b(int i)
 {
  if(i > 0)
     {
       c(i-1);
       linerel(-u, 0);
       b(i-1);
       linerel(0, -u);
       b(i-1);
       linerel(u, 0);
       d(i-1);
     }
  }
 void c(int i)
{
    if(i > 0)
       {
         b(i-1);
         linerel(0, -u);
         c(i-1);
         linerel(-u, 0);
         c(i-1);
         linerel(0, u);
         a(i-1);
       }
}

void d(int i)
{
     if(i > 0)
     {
          a(i-1);
          linerel(0, u);
          d(i-1);
          linerel(u, 0);
          d(i-1);
          linerel(0, -u);
          b(i-1);
     }
}
```

Результат роботи програми наведено на рис.5.13.

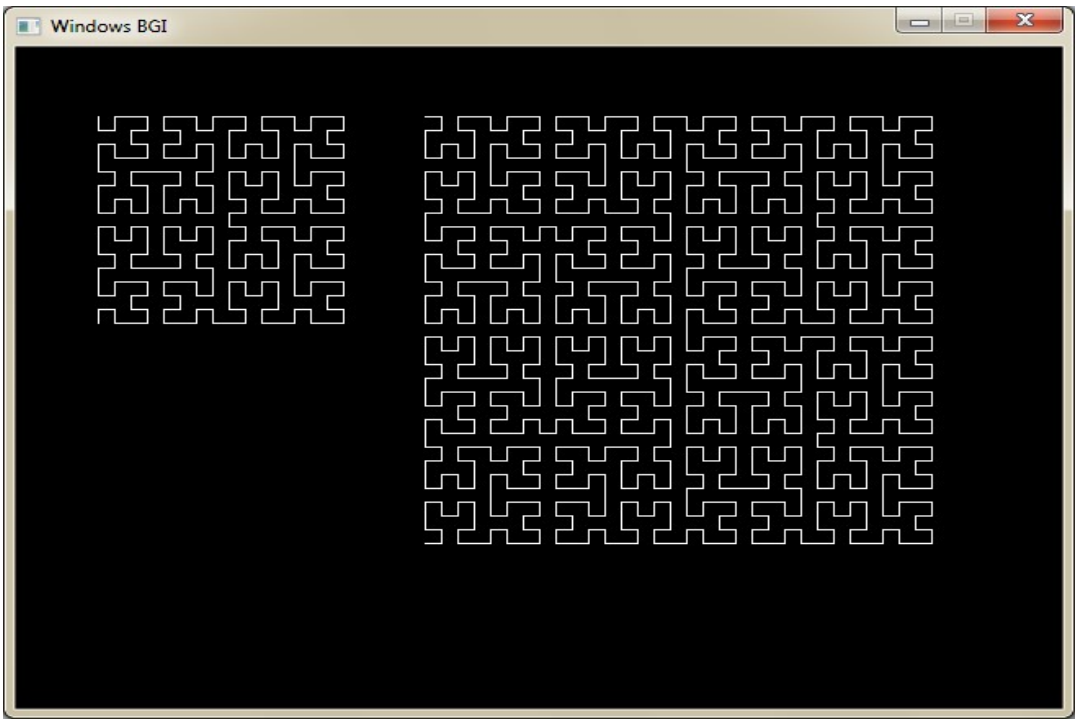


Рисунок 5.13 – Результат роботи програми для демонстрації кривих Гільберта четвертого та п’ятого порядків

Як можна побачити з рис. 5.13, криву Гільберта п’ятого порядку ми отримали вдвічі більшою за криву четвертого порядку. Спробуємо зробити таким чином, щоб ці криві були однакового розміру, як на рис. 5.12.

Для цього внесемо деякі зміни у попередню програму. По-перше позбудемося специфікатору `const` для змінної `u` для того, щоб мати змогу змінювати довжину штриха. По-друге, змінимо значення `u` на `12`, оскільки змінна має тип `int`, і , якщо ми поділимо `10` на `2`, то не отримаємо цілого результату.

По-третє, перед викликом функції `a(5)`, змінимо значення `u` на `6`.

Результат роботи програми в даному випадку наведено на рис. 5.14.

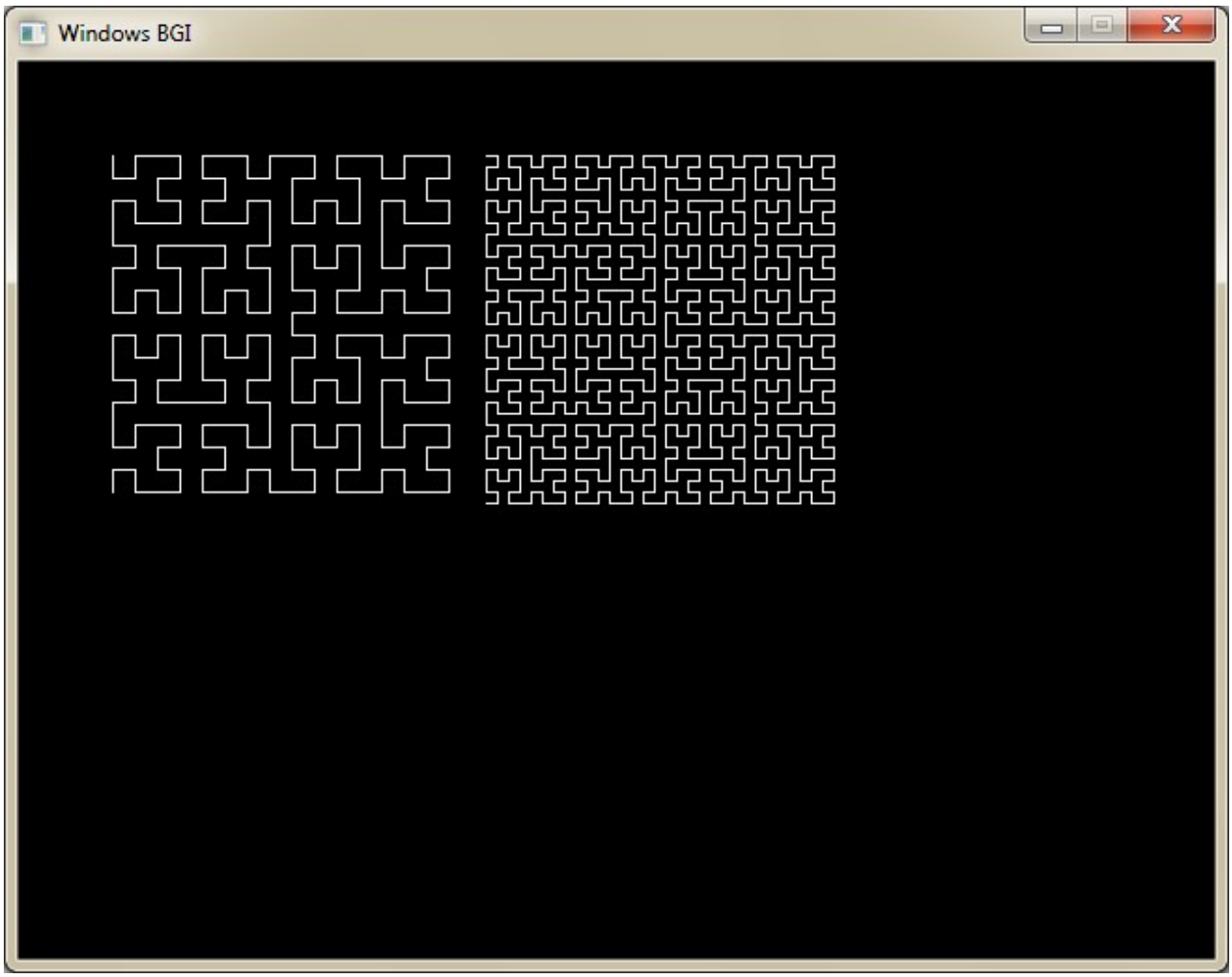


Рисунок 5.14 – Результат роботи програми для демонстрації кривих Гільберта четвертого та п’ятого порядків, що мають однаковий розмір

Контрольні запитання та завдання

1. Який файл заголовку необхідно підключити для роботи з графікою?

2. Що собою являє бібліотека WinBGIm? Як її налаштувати для роботи в середовищі Code::Blocks?

3. Які функції застосовуються для ініціалізації графічного режиму та закінчення роботи з графікою?

4. Яким чином можна створити графічне вікно для виводу графічної інформації?

5. За допомогою яких функцій можна отримати максимально можливі координати графічного вікна?

6. За допомогою яких функцій можна визначити поточні координати вказівника (графічного курсору)?

7. За допомогою яких функцій можна здійснювати переміщення вказівника без прорисовування на екрані?

8. Які функції забезпечують рисування точок і ліній у графічному вікні? Які додаткові функції при цьому слід застосовувати?

9. Які функції забезпечують рисування прямокутників у графічному вікні?

10. Які функції забезпечують рисування окружностей та еліпсів у графічному вікні?

11. Як вивести текст у графічному режимі? Що для цього треба зробити?

12. Яким чином можна забезпечити ефект анімації при рисуванні у графічному вікні?

Завдання для самостійного розв'язання

1. Напишіть С-програму, яка будує графік функції $f(x) = \ln(x^2 + 1)$ на інтервалі $[-2,2]$.

2. Створіть за допомогою С-програми двійкового дерева пошуку, що складається з наступних вузлів (порядок додавання вузлів повинен зберігатися, перший елемент – корінь): 38, 12, 6, 9, 91, 44, 27, 57, 39, 21, 88, 76, 33, 51, 30 2) та визначте в дереві максимальний і мінімальний елементи.

3. Напишіть С-програму, яка створює рисунок на Ваш розсуд. Фарбувати зображення можна (за бажанням). Позначте в правому нижньому куті графічного вікна свою групу та прізвище з ініціалами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Code::Blocks / Downloads / Binary releases. URL: https://www.codeblocks.org/downloads/binaries/ (дата звернення: 15.08.2023).

2. Brian W. Kernighan, Dennis M. Ritchi. C Programming Language, 2nd Edition. – Prentice Hall, 1988. – 278 p.

3. Slobodan Dmitrovic. Modern C for Absolute Beginners. A Friendly Introduction to the C Programming Language. 1st Ed. – Apress, 2021. – 346 p.

4. Thomas Mailund. Pointers in C Programming. A Modern Approach to Memory Management, Recursive Data Structures, Strings, and Arrays. – Apress, 2021. – 552 p.

5. Peter Prinz. C in a Nutshell: The Definitive Reference 2nd Edition. – O'Reilly, 2015. – 824 p.

6. Ben Klemens. 21st Century C: C Tips from the New School 2nd Edition. – O'Reilly, 2014. – 408 p.

7. Методичні вказівки до виконання лабораторних робіт з дисципліни «Програмування» для студентів денної та заочної форм навчання спеціальності 123 «Комп'ютерна інженерія». Ч. 2 / уклад. : Порошин С. М., Статкус А. В., Корольова Я. Ю., Онищенко В. В. – Харків : НТУ «ХПІ», 2021. – 183 с.

8. Методичні вказівки до виконання практичних робіт з дисципліни «Програмування» для студентів першого курсу денної та заочної форм навчання спеціальності 123 «Комп'ютерна інженерія». Частина 2 / уклад. : Онищенко В. В., Статкус А. В., Носик А. М., Корольова Я. Ю. – Харків: НТУ «ХПІ», 2021. – 168 с.

9. Allen I. Holub. Compiler design in C. – Prentice Hall, 1990. – 924 p.

11. Richard Reese. Understanding and Using C Pointers Core techniques for memory management. . – O'Reilly, 2014. – 226 p.

12. Robert C. Seacord. Effective C: An Introduction to Professional C Programming. – No Starch Press, 2020. – 272 p.

Навчальне видання

ПОРОШИН Сергій Михайлович,
НОСИК Андрій Михайлович

ПРОГРАМУВАННЯ
ЧАСТИНА 2

Навчальний посібник
для студентів усіх форм навчання за спеціальністю
123 «Комп'ютерна інженерія»

Відповідальний за випуск проф. Порошин С. М.

Роботу до видання рекомендував проф. Заполовський М. Й.

В авторській редакції

План 2024 р., поз. 31.

Підписано до друку 31.05 .2024 р.
Гарнітура Times New Roman

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Електронне видання